

Android Studio Hedgehog Essentials



Kotlin Edition



Payload
publishing

Android Studio Hedgehog Essentials

Kotlin Edition

Android Studio Hedgehog Essentials – Kotlin Edition

© 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



Find more books at <https://www.payloadbooks.com>.

Table of Contents

1. Introduction	1
1.1 Downloading the Code Samples	1
1.2 Feedback	1
1.3 Errata	2
2. Setting up an Android Studio Development Environment	3
2.1 System requirements	3
2.2 Downloading the Android Studio package	3
2.3 Installing Android Studio	4
2.3.1 Installation on Windows	4
2.3.2 Installation on macOS	4
2.3.3 Installation on Linux	5
2.4 The Android Studio setup wizard	5
2.5 Installing additional Android SDK packages	6
2.6 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
3. Creating an Example Android App in Android Studio	15
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
4. Creating an Android Virtual Device (AVD) in Android Studio	31
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 Removing the Device Frame.....	40
4.9 Summary	42
5. Using and Configuring the Android Studio AVD Emulator	43
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.....	49
5.9 Creating a Resizable Emulator.....	50
5.10 Summary	51
6. A Tour of the Android Studio User Interface	53
6.1 The Welcome Screen	53
6.2 The Menu Bar	54
6.3 The Main Window	55
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
7. Testing Android Studio Apps on a Physical Android Device.....	63
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
8. The Basics of the Android Studio Code Editor	71
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
9. An Overview of the Android Architecture	83
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
10. The Anatomy of an Android App	87
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
11. An Introduction to Kotlin	91
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
12. Kotlin Data Types, Variables, and Nullability	95
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
13. Kotlin Operators and Expressions	105
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
14. Kotlin Control Flow	113
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
15. An Overview of Kotlin Functions and Lambdas	121
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
16. The Basics of Object Oriented Programming in Kotlin	127
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
17. An Introduction to Kotlin Inheritance and Subclassing	137
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
18. An Overview of Android View Binding.....	143
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

19. Understanding Android Application and Activity Lifecycles	147
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
20. Handling Android Activity State Changes.....	153
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
21. Android Activity State Changes by Example	161
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
22. Saving and Restoring the State of an Android Activity	169
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
23. Understanding Android Views, View Groups and Layouts	173
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
24. A Guide to the Android Studio Layout Editor	177
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.....	185
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
25. A Guide to the Android ConstraintLayout.....	195
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
26. A Guide to Using ConstraintLayout in Android Studio	205
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
27. Working with ConstraintLayout Chains and Ratios in Android Studio	225
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
28. An Android Studio Layout Editor ConstraintLayout Tutorial	233
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
29. Manual XML Layout Design in Android Studio	241
29.1 Manually Creating an XML Layout.....	241
29.2 Manual XML vs. Visual Layout Design.....	244
29.3 Summary	244
30. Managing Constraints using Constraint Sets.....	245
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
31. An Android ConstraintSet Tutorial.....	251
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
32. A Guide to Using Apply Changes in Android Studio	259
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
33. An Overview and Example of Android Event Handling	263
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
34. Android Touch and Multi-touch Event Handling	269
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
35. Detecting Common Gestures Using the Android Gesture Detector Class	275
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
36. Implementing Custom Gesture and Pinch Recognition on Android	281
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
37. An Introduction to Android Fragments	291
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
38. Using Fragments in Android Studio - An Example.....	299
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
39. Modern Android App Architecture with Jetpack	309
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
40. An Android ViewModel Tutorial.....	315
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
41. An Android Jetpack LiveData Tutorial.....	321
41.1 LiveData - A Recap	321
41.2 Adding LiveData to the ViewModel.....	321
41.3 Implementing the Observer.....	323
41.4 Summary	324
42. An Overview of Android Jetpack Data Binding	325
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
43. An Android Jetpack Data Binding Tutorial.....	333
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	338
43.9 Adding a Listener Binding.....	339
43.10 Testing the App.....	339
43.11 Summary	339
44. An Android ViewModel Saved State Tutorial.....	341
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
45. Working with Android Lifecycle-Aware Components	345
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
46. An Android Jetpack Lifecycle Awareness Tutorial	349
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
47. An Overview of the Navigation Architecture Component	355
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
48. An Android Jetpack Navigation Component Tutorial	361
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
49. An Introduction to MotionLayout	375
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
50. An Android MotionLayout Editor.....	387
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
51. A MotionLayout KeyCycle Tutorial	401
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
52. Working with the Floating Action Button and Snackbar	413
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
53. Creating a Tabbed Interface using the TabLayout Component	419
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.....	428
54. Working with the RecyclerView and CardView Widgets.....	429
54.1 An Overview of the RecyclerView.....	429
54.2 An Overview of the CardView.....	431
54.3 Summary.....	432
55. An Android RecyclerView and CardView Tutorial.....	433
55.1 Creating the CardDemo Project.....	433
55.2 Modifying the Basic Views Activity Project.....	433
55.3 Designing the CardView Layout.....	434
55.4 Adding the RecyclerView.....	435
55.5 Adding the Image Files.....	435
55.6 Creating the RecyclerView Adapter.....	435
55.7 Initializing the RecyclerView Component.....	437
55.8 Testing the Application.....	438
55.9 Responding to Card Selections.....	439
55.10 Summary.....	440
56. Working with the AppBar and Collapsing Toolbar Layouts.....	441
56.1 The Anatomy of an AppBar.....	441
56.2 The Example Project.....	442
56.3 Coordinating the RecyclerView and Toolbar.....	442
56.4 Introducing the Collapsing Toolbar Layout.....	444
56.5 Changing the Title and Scrim Color.....	447
56.6 Summary.....	448
57. An Overview of Android Intents.....	449
57.1 An Overview of Intents.....	449
57.2 Explicit Intents.....	449
57.3 Returning Data from an Activity.....	450
57.4 Implicit Intents.....	451
57.5 Using Intent Filters.....	452
57.6 Automatic Link Verification.....	452
57.7 Manually Enabling Links.....	455
57.8 Checking Intent Availability.....	456
57.9 Summary.....	457
58. Android Explicit Intents – A Worked Example.....	459
58.1 Creating the Explicit Intent Example Application.....	459
58.2 Designing the User Interface Layout for MainActivity.....	459
58.3 Creating the Second Activity Class.....	460
58.4 Designing the User Interface Layout for SecondActivity.....	461
58.5 Reviewing the Application Manifest File.....	461
58.6 Creating the Intent.....	462
58.7 Extracting Intent Data.....	463
58.8 Launching SecondActivity as a Sub-Activity.....	464
58.9 Returning Data from a Sub-Activity.....	465
58.10 Testing the Application.....	465

58.11 Summary	465
59. Android Implicit Intents – A Worked Example	467
59.1 Creating the Android Studio Implicit Intent Example Project	467
59.2 Designing the User Interface	467
59.3 Creating the Implicit Intent	468
59.4 Adding a Second Matching Activity	469
59.5 Adding the Web View to the UI	469
59.6 Obtaining the Intent URL	470
59.7 Modifying the MyWebView Project Manifest File	471
59.8 Installing the MyWebView Package on a Device	472
59.9 Testing the Application	473
59.10 Manually Enabling the Link	473
59.11 Automatic Link Verification	475
59.12 Summary	477
60. Android Broadcast Intents and Broadcast Receivers	479
60.1 An Overview of Broadcast Intents	479
60.2 An Overview of Broadcast Receivers	480
60.3 Obtaining Results from a Broadcast	481
60.4 Sticky Broadcast Intents	481
60.5 The Broadcast Intent Example	481
60.6 Creating the Example Application	482
60.7 Creating and Sending the Broadcast Intent	482
60.8 Creating the Broadcast Receiver	483
60.9 Registering the Broadcast Receiver	484
60.10 Testing the Broadcast Example	485
60.11 Listening for System Broadcasts	485
60.12 Summary	485
61. An Introduction to Kotlin Coroutines	487
61.1 What are Coroutines?	487
61.2 Threads vs. Coroutines	487
61.3 Coroutine Scope	488
61.4 Suspend Functions	488
61.5 Coroutine Dispatchers	488
61.6 Coroutine Builders	489
61.7 Jobs	489
61.8 Coroutines – Suspending and Resuming	490
61.9 Returning Results from a Coroutine	491
61.10 Using withContext	491
61.11 Coroutine Channel Communication	493
61.12 Summary	494
62. An Android Kotlin Coroutines Tutorial	495
62.1 Creating the Coroutine Example Application	495
62.2 Adding Coroutine Support to the Project	495
62.3 Designing the User Interface	495
62.4 Implementing the SeekBar	497
62.5 Adding the Suspend Function	497
62.6 Implementing the launchCoroutines Method	498

Table of Contents

62.7 Testing the App.....	499
62.8 Summary	499
63. An Overview of Android Services.....	501
63.1 Intent Service	501
63.2 Bound Service.....	501
63.3 The Anatomy of a Service	502
63.4 Controlling Destroyed Service Restart Options.....	502
63.5 Declaring a Service in the Manifest File.....	502
63.6 Starting a Service Running on System Startup.....	503
63.7 Summary	504
64. Android Local Bound Services – A Worked Example	505
64.1 Understanding Bound Services	505
64.2 Bound Service Interaction Options	505
64.3 A Local Bound Service Example.....	505
64.4 Adding a Bound Service to the Project	506
64.5 Implementing the Binder	506
64.6 Binding the Client to the Service	508
64.7 Completing the Example.....	509
64.8 Testing the Application.....	510
64.9 Summary	511
65. Android Remote Bound Services – A Worked Example	513
65.1 Client to Remote Service Communication.....	513
65.2 Creating the Example Application.....	513
65.3 Designing the User Interface	513
65.4 Implementing the Remote Bound Service.....	513
65.5 Configuring a Remote Service in the Manifest File.....	515
65.6 Launching and Binding to the Remote Service.....	515
65.7 Sending a Message to the Remote Service	517
65.8 Summary	517
66. An Introduction to Kotlin Flow	519
66.1 Understanding Flows.....	519
66.2 Creating the Sample Project	519
66.3 Adding the Kotlin Lifecycle Library	520
66.4 Declaring a Flow.....	520
66.5 Emitting Flow Data	521
66.6 Collecting Flow Data	521
66.7 Adding a Flow Buffer.....	522
66.8 Transforming Data with Intermediaries	524
66.9 Terminal Flow Operators	525
66.10 Flow Flattening.....	526
66.11 Combining Multiple Flows	527
66.12 Hot and Cold Flows	528
66.13 StateFlow	529
66.14 SharedFlow.....	530
66.15 Summary	531
67. An Android SharedFlow Tutorial	533

67.1 About the Project	533
67.2 Creating the SharedFlowDemo Project.....	533
67.3 Designing the User Interface Layout	533
67.4 Adding the List Row Layout	533
67.5 Adding the RecyclerView Adapter.....	534
67.6 Adding the ViewModel	535
67.7 Configuring the ViewModelProvider.....	536
67.8 Collecting the Flow Values.....	537
67.9 Testing the SharedFlowDemo App	538
67.10 Handling Flows in the Background	538
67.11 Summary	541
68. An Overview of Android SQLite Databases	543
68.1 Understanding Database Tables	543
68.2 Introducing Database Schema	543
68.3 Columns and Data Types	543
68.4 Database Rows	544
68.5 Introducing Primary Keys	544
68.6 What is SQLite?	544
68.7 Structured Query Language (SQL).....	544
68.8 Trying SQLite on an Android Virtual Device (AVD)	545
68.9 The Android Room Persistence Library.....	546
68.10 Summary	547
69. The Android Room Persistence Library	549
69.1 Revisiting Modern App Architecture	549
69.2 Key Elements of Room Database Persistence.....	549
69.2.1 Repository	550
69.2.2 Room Database	550
69.2.3 Data Access Object (DAO)	550
69.2.4 Entities.....	550
69.2.5 SQLite Database	550
69.3 Understanding Entities.....	551
69.4 Data Access Objects.....	553
69.5 The Room Database.....	554
69.6 The Repository.....	555
69.7 In-Memory Databases.....	556
69.8 Database Inspector.....	556
69.9 Summary	557
70. An Android TableLayout and TableRow Tutorial	559
70.1 The TableLayout and TableRow Layout Views.....	559
70.2 Creating the Room Database Project	560
70.3 Converting to a LinearLayout.....	560
70.4 Adding the TableLayout to the User Interface.....	561
70.5 Configuring the TableRows	562
70.6 Adding the Button Bar to the Layout	563
70.7 Adding the RecyclerView.....	564
70.8 Adjusting the Layout Margins.....	565
70.9 Summary	565

71. An Android Room Database and Repository Tutorial.....	567
71.1 About the RoomDemo Project.....	567
71.2 Modifying the Build Configuration.....	567
71.3 Building the Entity.....	568
71.4 Creating the Data Access Object.....	569
71.5 Adding the Room Database.....	571
71.6 Adding the Repository	571
71.7 Adding the ViewModel	574
71.8 Creating the Product Item Layout	575
71.9 Adding the RecyclerView Adapter.....	576
71.10 Preparing the Main Activity	577
71.11 Adding the Button Listeners.....	578
71.12 Adding LiveData Observers	579
71.13 Initializing the RecyclerView.....	579
71.14 Testing the RoomDemo App.....	580
71.15 Using the Database Inspector.....	580
71.16 Summary.....	581
72. Video Playback on Android using the VideoView and MediaController Classes.....	583
72.1 Introducing the Android VideoView Class	583
72.2 Introducing the Android MediaController Class	584
72.3 Creating the Video Playback Example	584
72.4 Designing the VideoPlayer Layout	584
72.5 Downloading the Video File.....	585
72.6 Configuring the VideoView.....	585
72.7 Adding the MediaController to the Video View.....	587
72.8 Setting up the onPreparedListener	587
72.9 Summary	588
73. Android Picture-in-Picture Mode.....	589
73.1 Picture-in-Picture Features.....	589
73.2 Enabling Picture-in-Picture Mode.....	590
73.3 Configuring Picture-in-Picture Parameters	590
73.4 Entering Picture-in-Picture Mode.....	591
73.5 Detecting Picture-in-Picture Mode Changes.....	591
73.6 Adding Picture-in-Picture Actions.....	591
73.7 Summary	592
74. An Android Picture-in-Picture Tutorial.....	593
74.1 Adding Picture-in-Picture Support to the Manifest.....	593
74.2 Adding a Picture-in-Picture Button	593
74.3 Entering Picture-in-Picture Mode.....	594
74.4 Detecting Picture-in-Picture Mode Changes.....	595
74.5 Adding a Broadcast Receiver	595
74.6 Adding the PiP Action.....	596
74.7 Testing the Picture-in-Picture Action	599
74.8 Summary	599
75. Making Runtime Permission Requests in Android.....	601
75.1 Understanding Normal and Dangerous Permissions.....	601
75.2 Creating the Permissions Example Project.....	603

75.3 Checking for a Permission	603
75.4 Requesting Permission at Runtime.....	605
75.5 Providing a Rationale for the Permission Request	606
75.6 Testing the Permissions App.....	607
75.7 Summary	608
76. Android Audio Recording and Playback using MediaPlayer and MediaRecorder	609
76.1 Playing Audio	609
76.2 Recording Audio and Video using the MediaRecorder Class.....	610
76.3 About the Example Project	611
76.4 Creating the AudioApp Project.....	611
76.5 Designing the User Interface	611
76.6 Checking for Microphone Availability	612
76.7 Initializing the Activity.....	613
76.8 Implementing the recordAudio() Method.....	614
76.9 Implementing the stopAudio() Method.....	614
76.10 Implementing the playAudio() method.....	615
76.11 Configuring and Requesting Permissions	615
76.12 Testing the Application.....	617
76.13 Summary	617
77. An Android Notifications Tutorial	619
77.1 An Overview of Notifications.....	619
77.2 Creating the NotifyDemo Project.....	621
77.3 Designing the User Interface	621
77.4 Creating the Second Activity	621
77.5 Creating a Notification Channel	622
77.6 Requesting Notification Permission	623
77.7 Creating and Issuing a Notification	626
77.8 Launching an Activity from a Notification.....	628
77.9 Adding Actions to a Notification	630
77.10 Bundled Notifications.....	630
77.11 Summary	632
78. An Android Direct Reply Notification Tutorial	633
78.1 Creating the DirectReply Project.....	633
78.2 Designing the User Interface	633
78.3 Requesting Notification Permission	634
78.4 Creating the Notification Channel.....	635
78.5 Building the RemoteInput Object.....	636
78.6 Creating the PendingIntent.....	637
78.7 Creating the Reply Action.....	638
78.8 Receiving Direct Reply Input.....	639
78.9 Updating the Notification	640
78.10 Summary	641
79. Working with the Google Maps Android API in Android Studio	643
79.1 The Elements of the Google Maps Android API	643
79.2 Creating the Google Maps Project.....	644
79.3 Creating a Google Cloud Billing Account	644
79.4 Creating a New Google Cloud Project	645

79.5 Enabling the Google Maps SDK.....	646
79.6 Generating a Google Maps API Key.....	647
79.7 Adding the API Key to the Android Studio Project.....	648
79.8 Testing the Application.....	648
79.9 Understanding Geocoding and Reverse Geocoding	648
79.10 Adding a Map to an Application.....	650
79.11 Requesting Current Location Permission.....	650
79.12 Displaying the User's Current Location	652
79.13 Changing the Map Type.....	653
79.14 Displaying Map Controls to the User.....	654
79.15 Handling Map Gesture Interaction.....	654
79.15.1 Map Zooming Gestures.....	654
79.15.2 Map Scrolling/Panning Gestures	655
79.15.3 Map Tilt Gestures.....	655
79.15.4 Map Rotation Gestures.....	655
79.16 Creating Map Markers.....	655
79.17 Controlling the Map Camera	656
79.18 Summary.....	657
80. Printing with the Android Printing Framework	659
80.1 The Android Printing Architecture	659
80.2 The Print Service Plugins	659
80.3 Google Cloud Print.....	660
80.4 Printing to Google Drive.....	660
80.5 Save as PDF	661
80.6 Printing from Android Devices	661
80.7 Options for Building Print Support into Android Apps.....	662
80.7.1 Image Printing.....	662
80.7.2 Creating and Printing HTML Content	663
80.7.3 Printing a Web Page.....	664
80.7.4 Printing a Custom Document	665
80.8 Summary	665
81. An Android HTML and Web Content Printing Example	667
81.1 Creating the HTML Printing Example Application	667
81.2 Printing Dynamic HTML Content	667
81.3 Creating the Web Page Printing Example.....	670
81.4 Removing the Floating Action Button	670
81.5 Removing Navigation Features.....	670
81.6 Designing the User Interface Layout	671
81.7 Accessing the WebView from the Main Activity	672
81.8 Loading the Web Page into the WebView.....	672
81.9 Adding the Print Menu Option.....	673
81.10 Summary.....	675
82. A Guide to Android Custom Document Printing.....	677
82.1 An Overview of Android Custom Document Printing	677
82.1.1 Custom Print Adapters.....	677
82.2 Preparing the Custom Document Printing Project.....	678
82.3 Creating the Custom Print Adapter.....	679
82.4 Implementing the onLayout() Callback Method	680

82.5 Implementing the onWrite() Callback Method	683
82.6 Checking a Page is in Range	685
82.7 Drawing the Content on the Page Canvas	686
82.8 Starting the Print Job	688
82.9 Testing the Application.....	689
82.10 Summary	689
83. An Introduction to Android App Links.....	691
83.1 An Overview of Android App Links	691
83.2 App Link Intent Filters	691
83.3 Handling App Link Intents.....	692
83.4 Associating the App with a Website.....	692
83.5 Summary	693
84. An Android Studio App Links Tutorial	695
84.1 About the Example App	695
84.2 The Database Schema	695
84.3 Loading and Running the Project.....	695
84.4 Adding the URL Mapping.....	697
84.5 Adding the Intent Filter.....	700
84.6 Adding Intent Handling Code.....	700
84.7 Testing the App.....	703
84.8 Creating the Digital Asset Links File	703
84.9 Testing the App Link.....	704
84.10 Summary	704
85. An Android Biometric Authentication Tutorial.....	705
85.1 An Overview of Biometric Authentication.....	705
85.2 Creating the Biometric Authentication Project	705
85.3 Configuring Device Fingerprint Authentication	706
85.4 Adding the Biometric Permission to the Manifest File.....	706
85.5 Designing the User Interface	707
85.6 Adding a Toast Convenience Method	707
85.7 Checking the Security Settings.....	708
85.8 Configuring the Authentication Callbacks.....	709
85.9 Adding the CancellationSignal.....	710
85.10 Starting the Biometric Prompt	710
85.11 Testing the Project.....	711
85.12 Summary	712
86. Creating, Testing, and Uploading an Android App Bundle.....	713
86.1 The Release Preparation Process.....	713
86.2 Android App Bundles.....	713
86.3 Register for a Google Play Developer Console Account.....	714
86.4 Configuring the App in the Console	715
86.5 Enabling Google Play App Signing.....	716
86.6 Creating a Keystore File	716
86.7 Creating the Android App Bundle.....	717
86.8 Generating Test APK Files	719
86.9 Uploading the App Bundle to the Google Play Developer Console.....	720
86.10 Exploring the App Bundle	721

86.11 Managing Testers	722
86.12 Rolling the App Out for Testing.....	722
86.13 Uploading New App Bundle Revisions.....	723
86.14 Analyzing the App Bundle File	724
86.15 Summary	725
87. An Overview of Android In-App Billing	727
87.1 Preparing a Project for In-App Purchasing.....	727
87.2 Creating In-App Products and Subscriptions	727
87.3 Billing Client Initialization.....	728
87.4 Connecting to the Google Play Billing Library.....	729
87.5 Querying Available Products.....	729
87.6 Starting the Purchase Process.....	730
87.7 Completing the Purchase	730
87.8 Querying Previous Purchases.....	731
87.9 Summary	732
88. An Android In-App Purchasing Tutorial	733
88.1 About the In-App Purchasing Example Project.....	733
88.2 Creating the InAppPurchase Project	733
88.3 Adding Libraries to the Project	733
88.4 Designing the User Interface	734
88.5 Adding the App to the Google Play Store	734
88.6 Creating an In-App Product.....	735
88.7 Enabling License Testers	735
88.8 Initializing the Billing Client	736
88.9 Querying the Product.....	738
88.10 Launching the Purchase Flow	739
88.11 Handling Purchase Updates	739
88.12 Consuming the Product	740
88.13 Restoring a Previous Purchase	741
88.14 Testing the App.....	742
88.15 Troubleshooting	743
88.16 Summary	744
89. Working with Material Design 3 Theming	745
89.1 Material Design 2 vs. Material Design 3	745
89.2 Understanding Material Design Theming	745
89.3 Material Design 3 Theming	745
89.4 Building a Custom Theme.....	747
89.5 Summary	748
90. A Material Design 3 Theming and Dynamic Color Tutorial.....	749
90.1 Creating the ThemeDemo Project	749
90.2 Designing the User Interface	749
90.3 Building a New Theme	751
90.4 Adding the Theme to the Project	752
90.5 Enabling Dynamic Color Support	753
90.6 Previewing Dynamic Colors.....	754
90.7 Summary	755
91. Accessing Cloud Storage using the Android Storage Access Framework.....	757

91.1 The Storage Access Framework.....	757
91.2 Working with the Storage Access Framework.....	758
91.3 Filtering Picker File Listings.....	758
91.4 Handling Intent Results.....	759
91.5 Reading the Content of a File	759
91.6 Writing Content to a File	760
91.7 Deleting a File.....	761
91.8 Gaining Persistent Access to a File.....	761
91.9 Summary	761
92. An Android Storage Access Framework Example	763
92.1 About the Storage Access Framework Example.....	763
92.2 Creating the Storage Access Framework Example.....	763
92.3 Designing the User Interface	763
92.4 Adding the Activity Launchers.....	764
92.5 Creating a New Storage File.....	765
92.6 Saving to a Storage File.....	766
92.7 Opening and Reading a Storage File	767
92.8 Testing the Storage Access Application	768
92.9 Summary	770
93. An Android Studio Primary/Detail Flow Tutorial	771
93.1 The Primary/Detail Flow.....	771
93.2 Creating a Primary/Detail Flow Activity	772
93.3 Adding the Primary/Detail Flow Activity.....	772
93.4 Modifying the Primary/Detail Flow Template	773
93.5 Changing the Content Model.....	773
93.6 Changing the Detail Pane	775
93.7 Modifying the ItemDetailFragment Class	776
93.8 Modifying the ItemListFragment Class.....	777
93.9 Adding Manifest Permissions.....	777
93.10 Running the Application.....	778
93.11 Summary	778
94. An Overview of Gradle in Android Studio	779
94.1 An Overview of Gradle	779
94.2 Gradle and Android Studio	779
94.2.1 Sensible Defaults	779
94.2.2 Dependencies.....	779
94.2.3 Build Variants	780
94.2.4 Manifest Entries	780
94.2.5 APK Signing.....	780
94.2.6 ProGuard Support.....	780
94.3 The Property and Settings Gradle Build File.....	780
94.4 The Top-level Gradle Build File.....	781
94.5 Module Level Gradle Build Files.....	782
94.6 Configuring Signing Settings in the Build File.....	784
94.7 Running Gradle Tasks from the Command Line	785
94.8 Summary	786
Index.....	787

1. Introduction

Fully updated for Android Studio Hedgehog (2023.1.1) 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.payloadbooks.com/product/hedgehogkotlin/>

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 info@payloadbooks.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.payloadbooks.com/hedgehogkotlin>

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

3. Creating an Example Android App in Android Studio

The preceding chapters of this book have explained how 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:

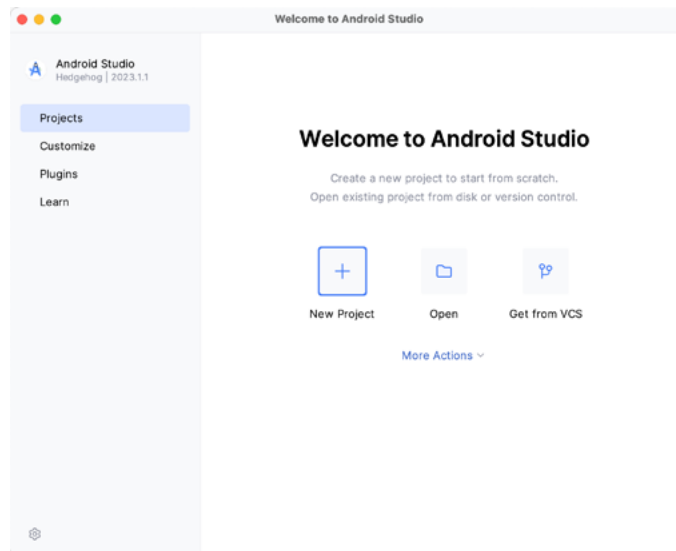


Figure 3-1

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.

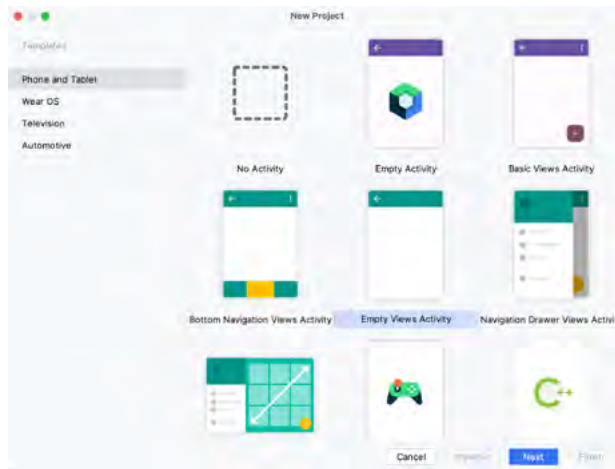


Figure 3-2

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:

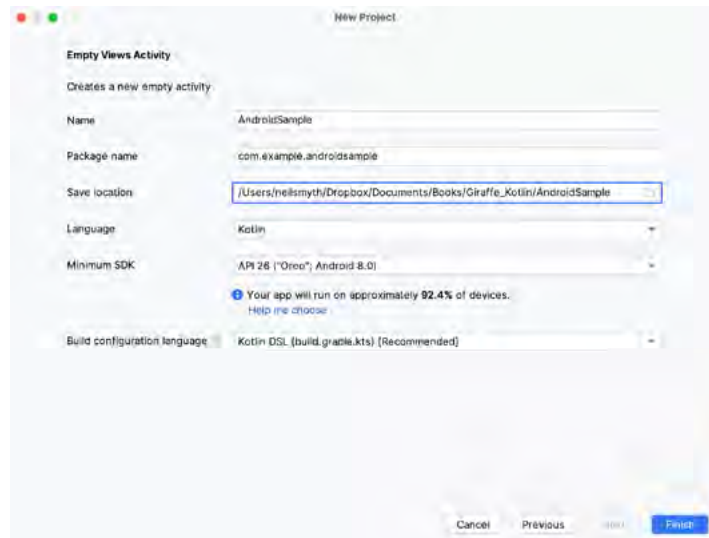


Figure 3-3

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 Hedgehog version. If your installation of Android Studio resembles Figure 3-4 below, then you will need to enable the new UI before proceeding:

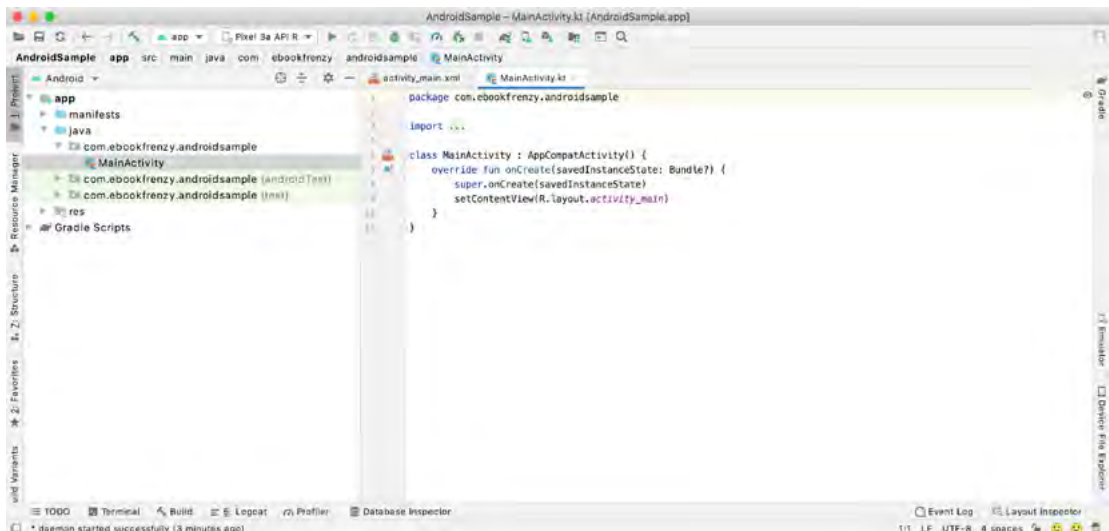


Figure 3-4

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:

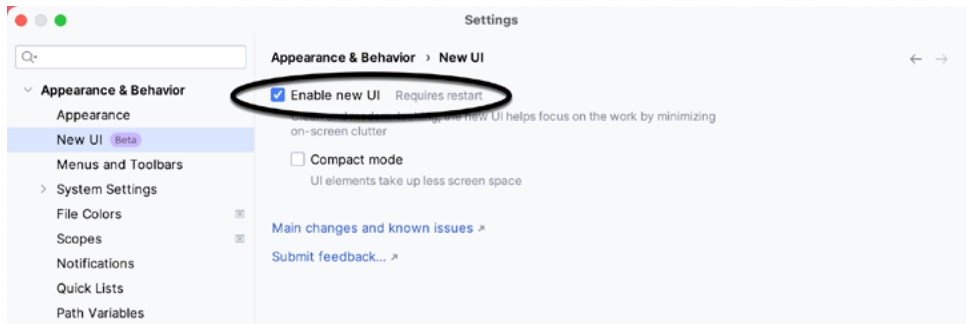


Figure 3-5

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:

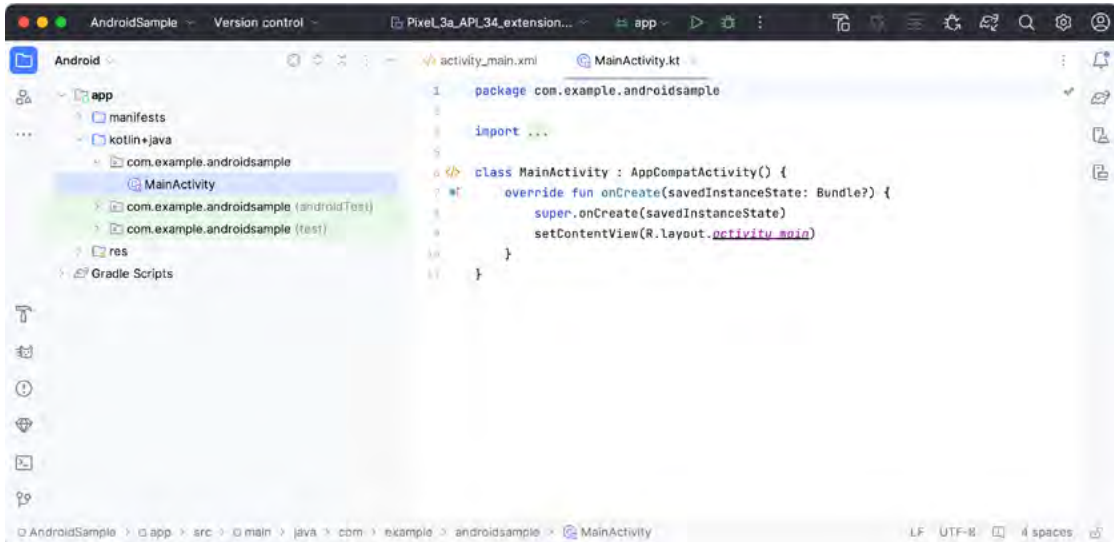


Figure 3-6

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:

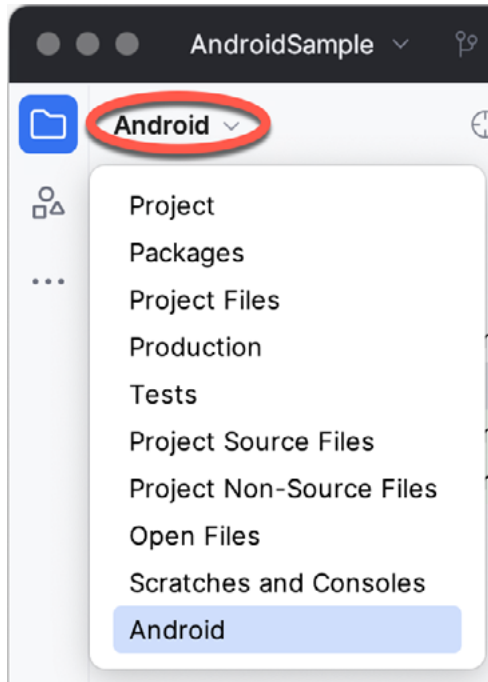


Figure 3-7

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:

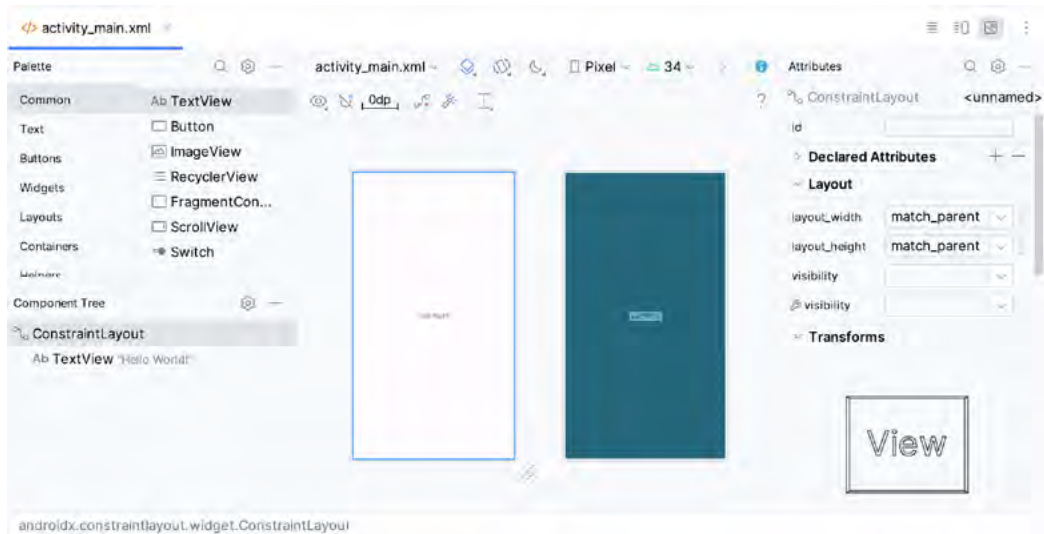




Figure 3-8

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

Creating an Example Android App in Android Studio

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:

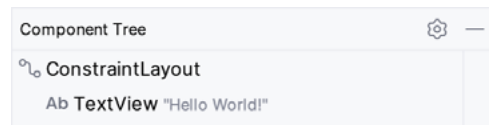


Figure 3-9

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.

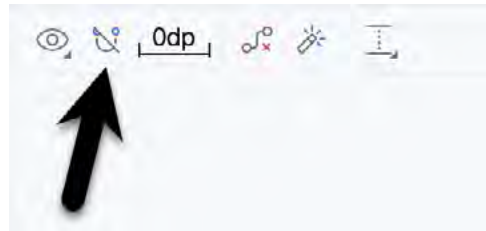


Figure 3-10

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:

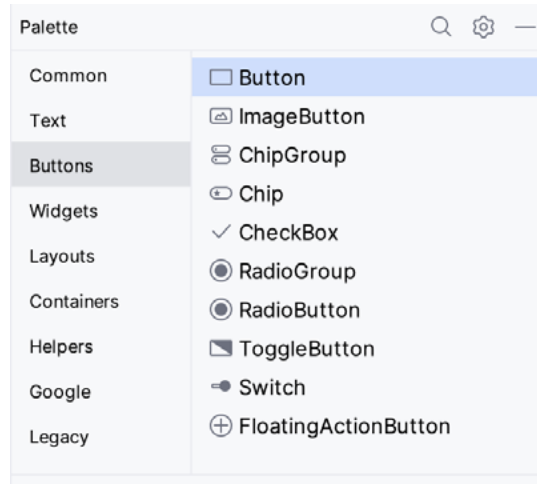


Figure 3-11

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:

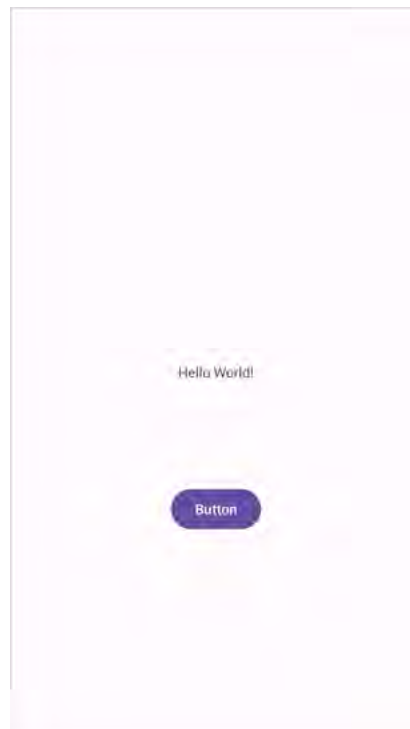


Figure 3-12

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:

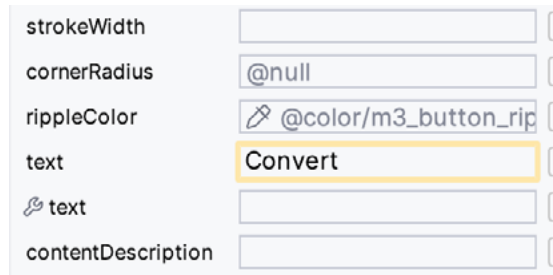


Figure 3-13

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:

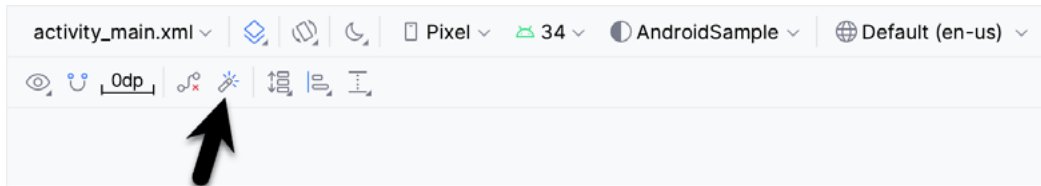


Figure 3-14

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:

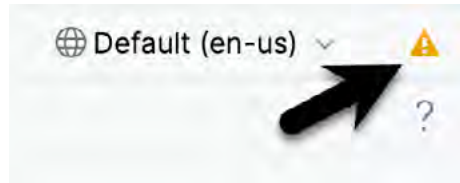


Figure 3-15

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

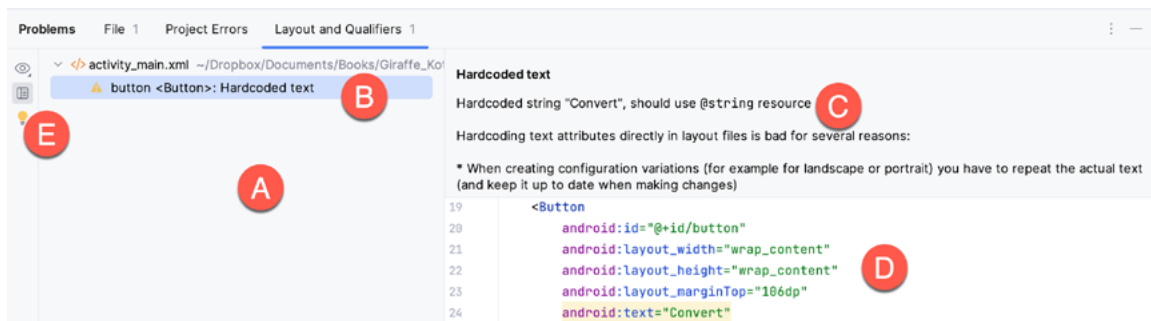


Figure 3-16

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:

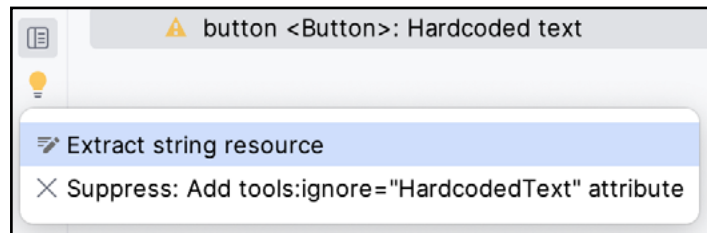


Figure 3-17

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:

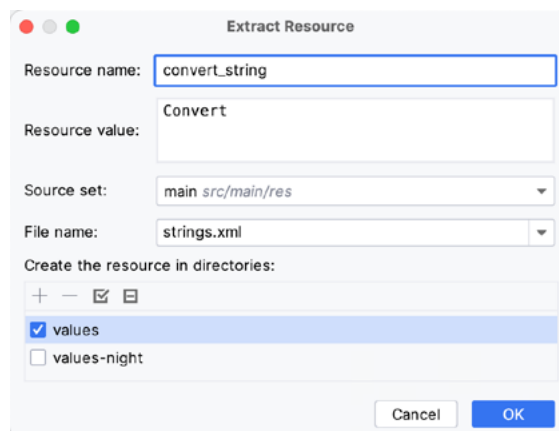


Figure 3-18

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:

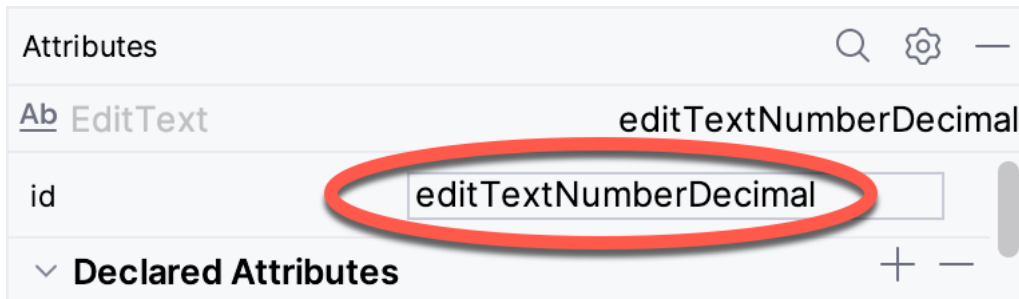


Figure 3-19

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:

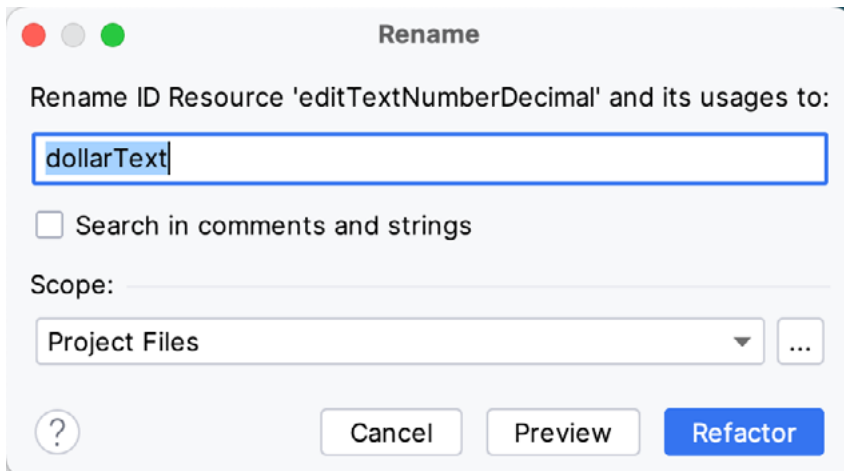


Figure 3-20

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:

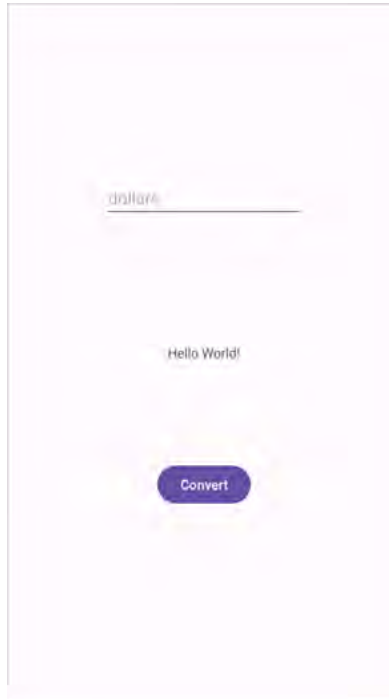


Figure 3-21

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 are the View Modes buttons marked A through C in Figure 3-22 below:

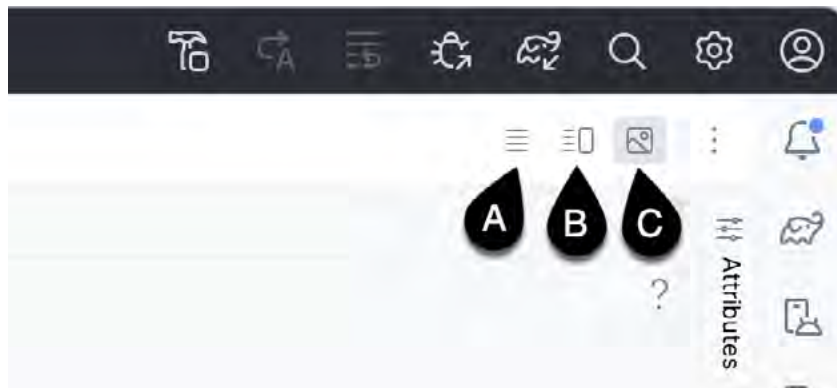


Figure 3-22

By default, the editor will be in *Design* mode (button C), whereby only the visual representation of the layout is displayed. In *Code* mode (A), the editor will display the XML for the layout, while in *Split* mode (B), both the layout and XML are displayed, as shown in Figure 3-23:

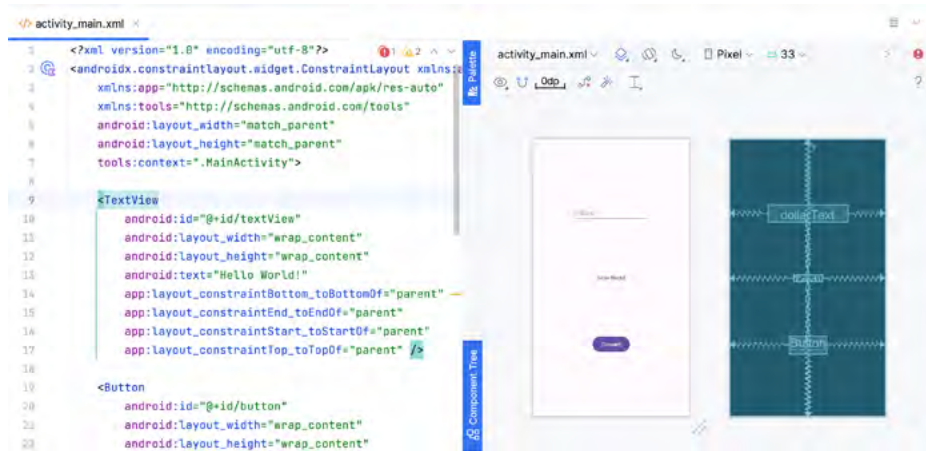


Figure 3-23

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:

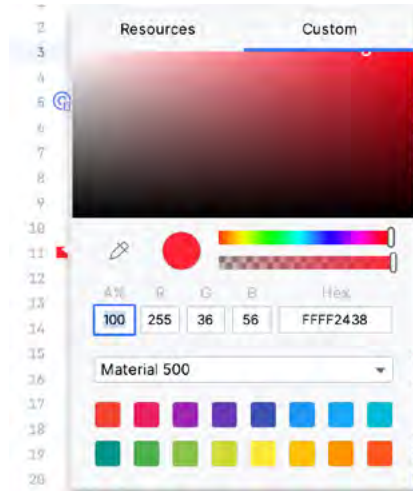


Figure 3-24

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="app_name">AndroidSample</string>
    <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:

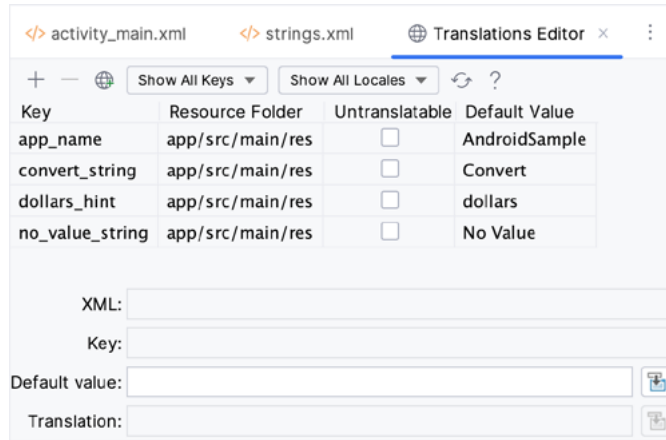


Figure 3-25

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:

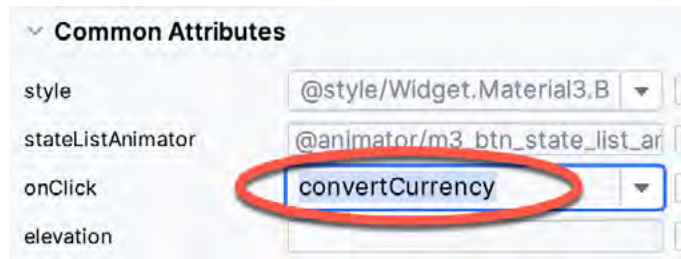


Figure 3-26

Next, double-click on the *MainActivity.kt* file in the Project tool window (*app* -> *kotlin+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.

5. Using and Configuring the Android Studio AVD Emulator

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

5.1 The Emulator Environment

When launched in standalone mode, the emulator displays an initial splash screen during the loading process. Once loaded, the main emulator window appears, containing a representation of the chosen device type (in the case of Figure 5-1, this is a Pixel 4 device):

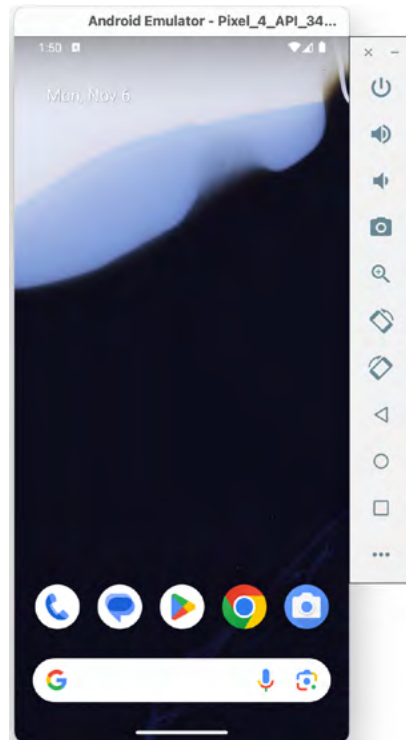


Figure 5-1

The toolbar positioned along the right-hand edge of the window provides quick access to the emulator controls and configuration options.

5.2 Emulator Toolbar Options

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

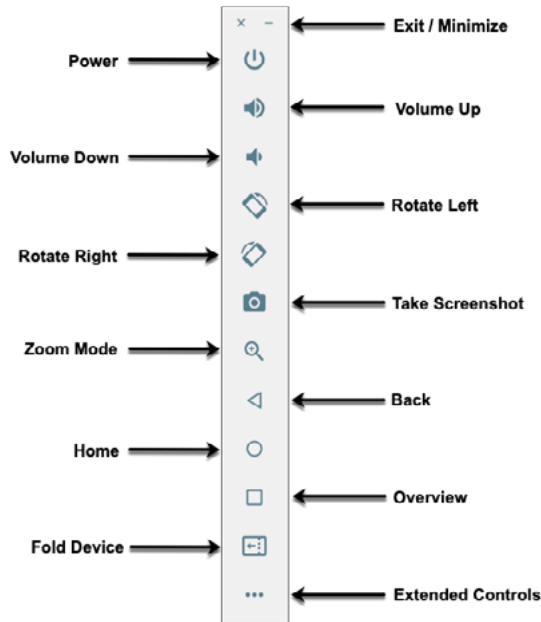


Figure 5-2

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

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

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

- **Fold Device** – Simulates the folding and unfolding of a foldable device. This option is only available if the emulator is running a foldable device system image.
- **Extended Controls** – Displays the extended controls panel, allowing for the configuration of options such as simulated location and telephony activity, battery strength, cellular network type, and fingerprint identification.

5.3 Working in Zoom Mode

The zoom button located in the emulator toolbar switches in and out of zoom mode. When zoom mode is active, the toolbar button is depressed, and the mouse pointer appears as a magnifying glass when hovering over the device screen. Clicking the left mouse button will cause the display to zoom in relative to the selected point on the screen, with repeated clicking increasing the zoom level. Conversely, clicking the right mouse button decreases the zoom level. Toggling the zoom button off reverts the display to the default size.

Clicking and dragging while in zoom mode will define a rectangular area into which the view will zoom when the mouse button is released.

While in zoom mode, the screen's visible area may be panned using the horizontal and vertical scrollbars located within the emulator window.

5.4 Resizing the Emulator Window

The emulator window's size (and the device's corresponding representation) can be changed at any time by enabling Zoom mode and clicking and dragging on any of the corners or sides of the window.

5.5 Extended Control Options

The extended controls toolbar button displays the panel illustrated in Figure 5-3. By default, the location settings will be displayed. Selecting a different category from the left-hand panel will display the corresponding group of controls:

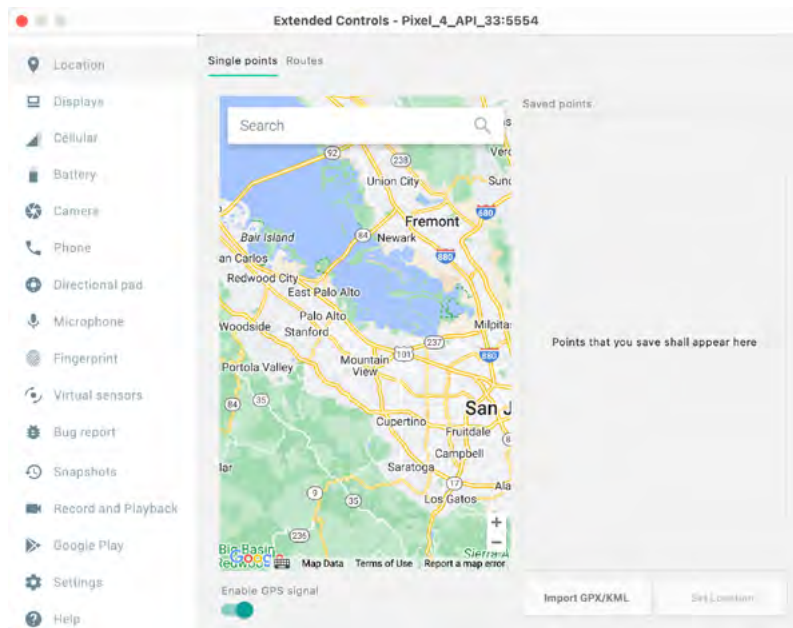


Figure 5-3

5.5.1 Location

The location controls allow simulated location information to be sent to the emulator as decimal or sexagesimal coordinates. Location information can take the form of a single location or a sequence of points representing the device's movement, the latter being provided via a file in either GPS Exchange (GPX) or Keyhole Markup Language (KML) format. Alternatively, the integrated Google Maps panel may be used to select single points or travel routes visually.

5.5.2 Displays

In addition to the main display shown within the emulator screen, the Displays option allows additional displays to be added running within the same Android instance. This can be useful for testing apps for dual-screen devices such as the Microsoft Surface Duo. These additional screens can be configured to be any required size and appear within the same emulator window as the main screen.

5.5.3 Cellular

The type of cellular connection being simulated can be changed within the cellular settings screen. Options are available to simulate different network types (CSM, EDGE, HSDPA, etc.) in addition to a range of voice and data scenarios, such as roaming and denied access.

5.5.4 Battery

Various battery state and charging conditions can be simulated on this panel of the extended controls screen, including battery charge level, battery health, and whether the AC charger is currently connected.

5.5.5 Camera

The emulator simulates a 3D scene when the camera is active. This takes the form of the interior of a virtual building through which you can navigate by holding down the Option key (Alt on Windows) while using the mouse pointer and keyboard keys when recording video or before taking a photo within the emulator. This extended configuration option allows different images to be uploaded for display within the virtual environment.

5.5.6 Phone

The phone extended controls provide two straightforward but helpful simulations within the emulator. The first option simulates an incoming call from a designated phone number. This can be particularly useful when testing how an app handles high-level interrupts.

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

5.5.7 Directional Pad

A directional pad (D-Pad) is an additional set of controls either built into an Android device or connected externally (such as a game controller) that provides directional controls (left, right, up, down). The directional pad settings allow D-Pad interaction to be simulated within the emulator.

5.5.8 Microphone

The microphone settings allow the microphone to be enabled and virtual headset and microphone connections to be simulated. A button is also provided to launch the Voice Assistant on the emulator.

5.5.9 Fingerprint

Many Android devices are now supplied with built-in fingerprint detection hardware. The AVD emulator makes it possible to test fingerprint authentication without the need to test apps on a physical device containing a fingerprint sensor. Details on configuring fingerprint testing within the emulator will be covered later in this chapter.

5.5.10 Virtual Sensors

The virtual sensors option allows the accelerometer and magnetometer to be simulated to emulate the effects of the physical motion of a device, such as rotation, movement, and tilting through yaw, pitch, and roll settings.

5.5.11 Snapshots

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

5.5.12 Record and Playback

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

5.5.13 Google Play

If the emulator is running a version of Android with Google Play Services installed, this option displays the current Google Play version. It also provides the option to update the emulator to the latest version.

5.5.14 Settings

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

5.5.15 Help

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

5.6 Working with Snapshots

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

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

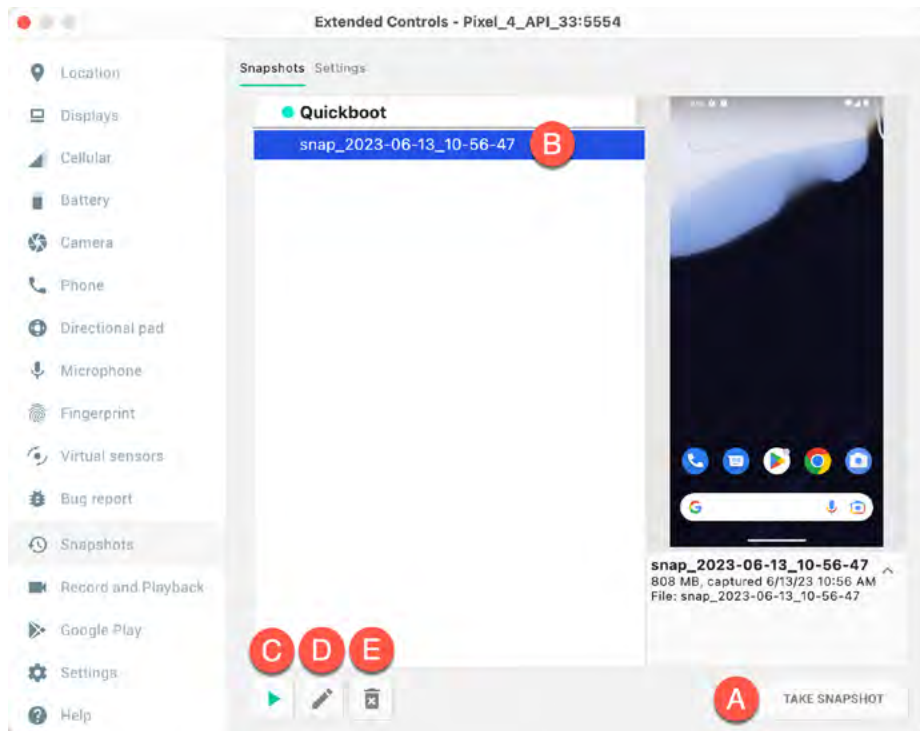


Figure 5-4

You can also choose whether to start an emulator using either a cold boot, the most recent quick-boot snapshot, or a previous snapshot by making a selection from the run target menu in the main toolbar, as illustrated in Figure 5-5:

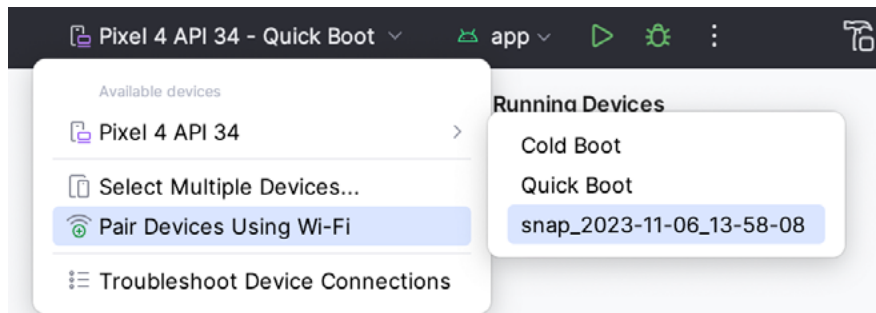


Figure 5-5

5.7 Configuring Fingerprint Emulation

The emulator allows up to 10 simulated fingerprints to be configured and used to test fingerprint authentication within Android apps. Configuring simulated fingerprints begins by launching the emulator, opening the Settings app, and selecting the Security option.

Within the Security settings screen, select the fingerprint option. On the resulting information screen, click on the *Next* button to proceed to the Fingerprint setup screen. Before fingerprint security can be enabled, a backup screen unlocking method (such as a PIN) must be configured. Enter and confirm a suitable PIN and complete the PIN entry process by accepting the default notifications option.

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

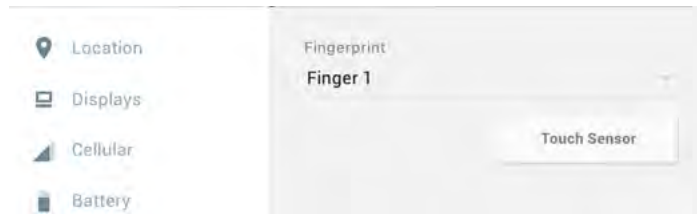


Figure 5-6

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

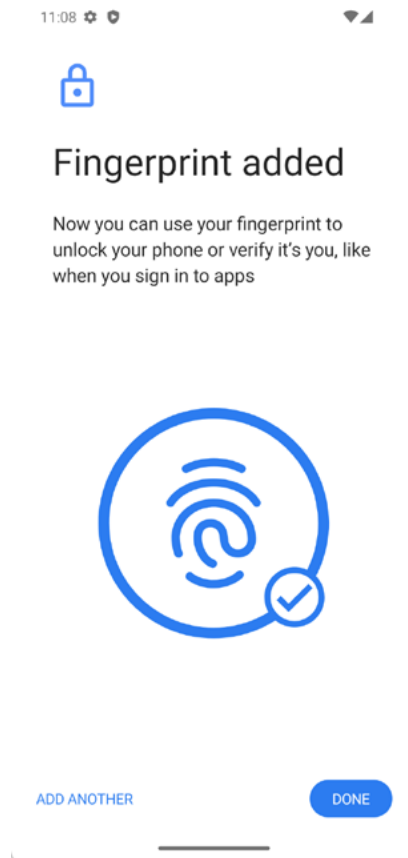


Figure 5-7

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

5.8 The Emulator in Tool Window Mode

As outlined in the previous chapter (*“Creating an Android Virtual Device (AVD) in Android Studio”*), Android Studio can be configured to launch the emulator in an embedded tool window so that it does not appear in a

separate window. When running in this mode, the same controls available in standalone mode are provided in the toolbar, as shown in Figure 5-8:

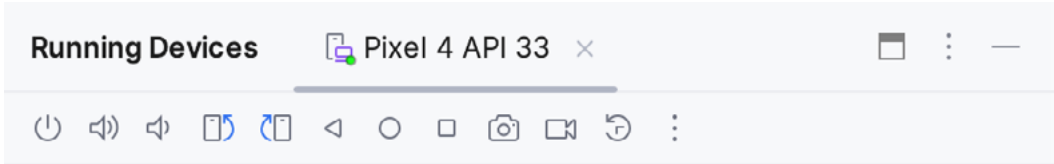


Figure 5-8

From left to right, these buttons perform the following tasks (details of which match those for standalone mode):

- Power
- Volume Up
- Volume Down
- Rotate Left
- Rotate Right
- Back
- Home
- Overview
- Screenshot
- Snapshots
- Extended Controls

5.9 Creating a Resizable Emulator

In addition to emulators configured to match specific Android device models, Android Studio also provides a resizable AVD that allows you to switch between phone, tablet, and foldable device sizes. To create a resizable emulator, open the Device Manager and click the ‘+’ toolbar button. Next, select the Resizable device definition illustrated in Figure 5-9, and follow the usual steps to create a new AVD:

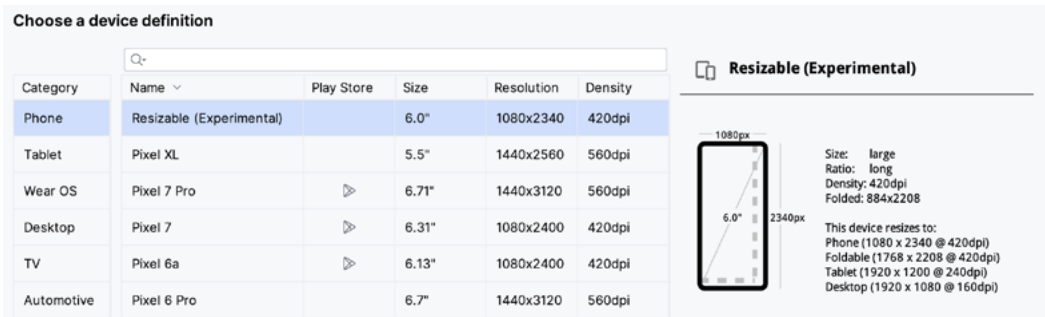


Figure 5-9

When you run an app on the new emulator within a tool window, the *Display mode* option will appear in the toolbar, allowing you to switch between emulator configurations as shown in Figure 5-10:

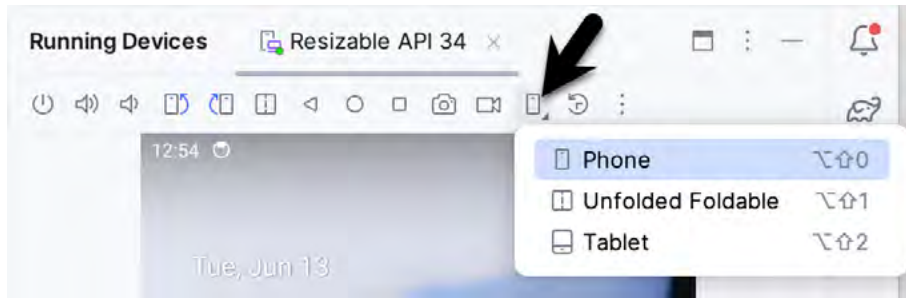


Figure 5-10

If the emulator is running in standalone mode, the Display mode option can be found in the side toolbar, as shown below:



Figure 5-11

5.10 Summary

Android Studio contains an Android Virtual Device emulator environment designed to make it easier to test applications without running them on a physical Android device. This chapter has provided a brief tour of the emulator and highlighted key features available to configure and customize the environment to simulate different testing conditions.

28. An Android Studio Layout Editor ConstraintLayout Tutorial

The easiest and most productive way to design a user interface for an Android application is to use the Android Studio Layout Editor tool. This chapter will provide an overview of how to create a ConstraintLayout-based user interface using this approach. The exercise included in this chapter will also be used as an opportunity to outline the creation of an activity starting with a “bare-bones” Android Studio project.

Having covered the use of the Android Studio Layout Editor, the chapter will also introduce the Layout Inspector tool.

28.1 An Android Studio Layout Editor Tool Example

The first step in this phase of the example is to create a new Android Studio project. Launch Android Studio and close any previously opened projects by selecting the *File -> Close Project* menu option.

Select the *New Project* option from the welcome screen, select the Empty Views Activity template, and click Next. Enter *LayoutSample* into the Name field and specify *com.ebookfrenzy.layoutsamle* 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.

28.2 Preparing the Layout Editor Environment

Locate and double-click on the *activity_main.xml* layout file in the *app -> res -> layout* folder to load it into the Layout Editor tool. Since this tutorial aims to gain experience with the use of constraints, turn off the Autoconnect feature using the button located in the Layout Editor toolbar. Once disabled, the button will appear with a line through it, as is the case in Figure 28-1:

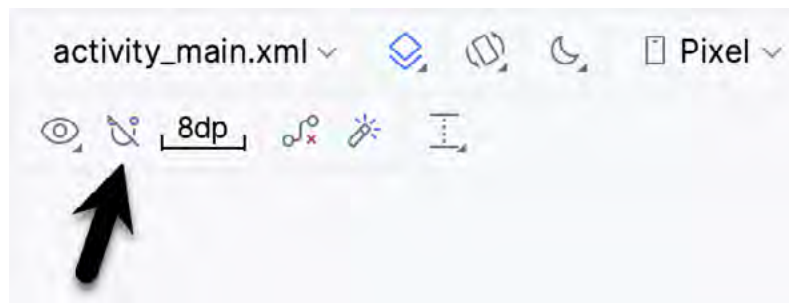


Figure 28-1

If the default margin value to the right of the Autoconnect button is not set to 8dp, click on it and select 8dp from the resulting panel.

The user interface design will also use the *ImageView* object to display an image. Before proceeding, this image should be added to the project, ready for use later in the chapter. This file is named *GalaxyS23.webp* and can be found in the *project_icons* folder of the sample code download available from the following URL:

<https://www.payloadbooks.com/product/hedgehogkotlin/>

Within Android Studio, display the Resource Manager tool window (*View -> Tool Windows -> Resource Manager*). Locate the *GalaxyS23.webp* image in the file system navigator for your operating system and drag and drop the image onto the Resource Manager tool window. In the resulting dialog, click *Next*, followed by the *Import* button, to add the image to the project. The image should now appear in the Resource Manager, as shown in Figure 28-2 below:

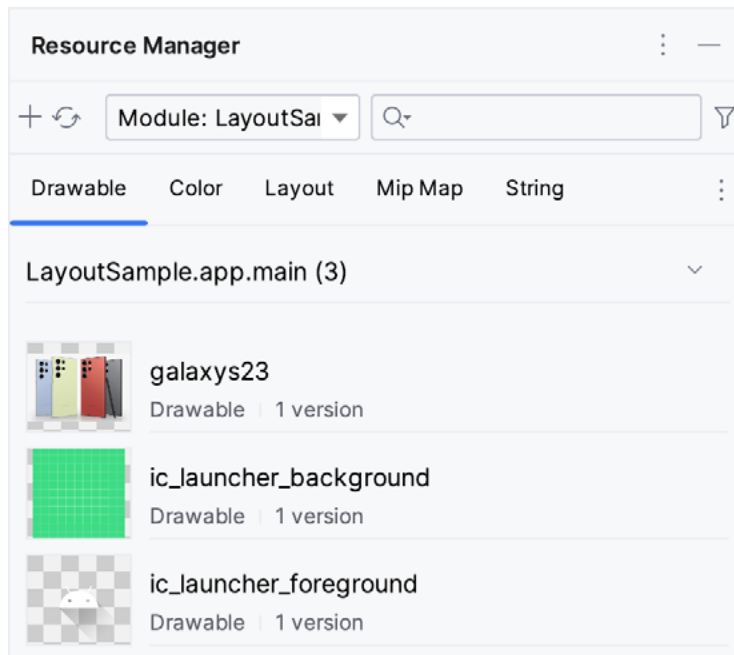


Figure 28-2

The image will also appear in the *res -> drawables* section of the Project tool window:

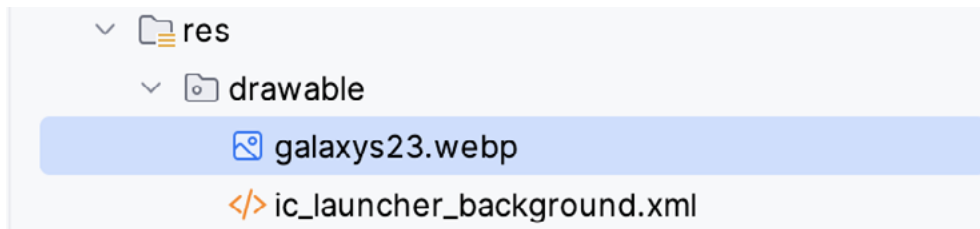


Figure 28-3

28.3 Adding the Widgets to the User Interface

From within the *Common* palette category, drag an *ImageView* object into the center of the display view. Note that horizontal and vertical dashed lines appear, indicating the center axes of the display. When centered, release the mouse button to drop the view into position. Once placed within the layout, the Resources dialog will appear, seeking the image to be displayed within the view. In the search bar at the top of the dialog, enter “galaxy” to locate the *galaxys6.png* resource, as illustrated in Figure 28-4.

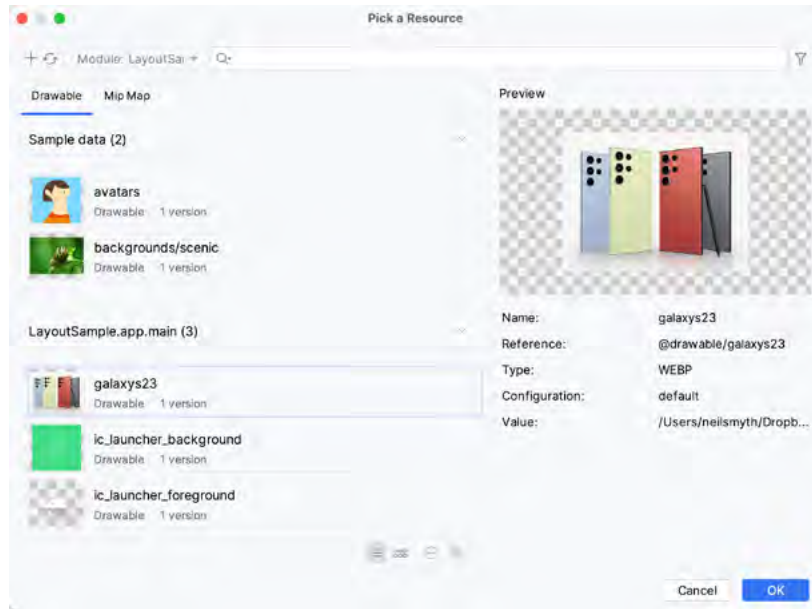


Figure 28-4

Select the image and click OK to assign it to the `ImageView` object. If necessary, adjust the size of the `ImageView` using the resize handles and reposition it in the center of the layout. At this point, the layout should match Figure 28-5:

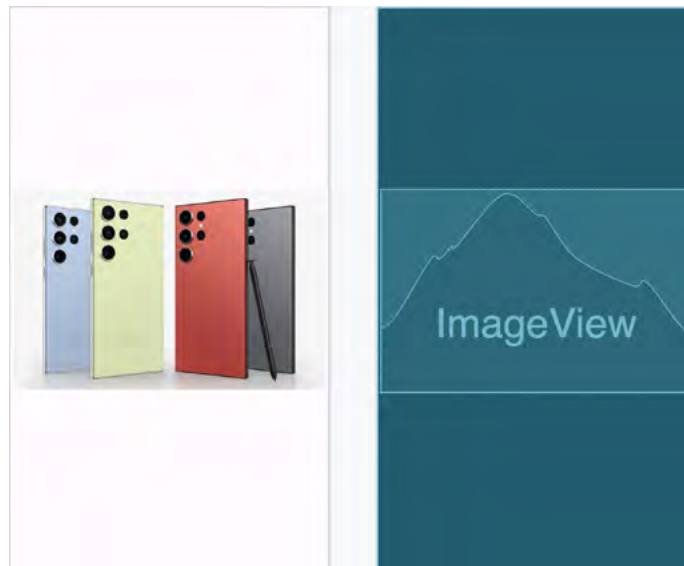


Figure 28-5

Click and drag a `TextView` object from the *Common* section of the palette and position it to appear above the `ImageView`, as illustrated in Figure 28-6.

Using the Attributes panel, unfold the *textAppearance* attribute entry in the Common Attributes section, change the *textSize* property to 24sp, the *textAlignment* setting to center, and the text to “Samsung Galaxy S23”.

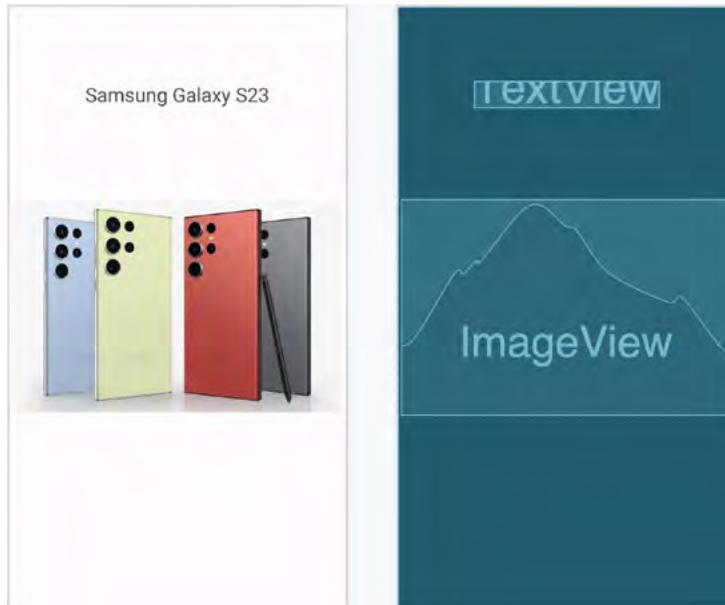


Figure 28-6

Next, add three Button widgets along the bottom of the layout and set the text attributes of these views to “Buy Now”, “Pricing”, and “Details”. The completed layout should now match Figure 28-7:



Figure 28-7

At this point, the widgets are not sufficiently constrained for the layout engine to be able to position and size the widgets at runtime. Were the app to run now, all of the widgets would be positioned in the top left-hand corner of the display.

With the widgets added to the layout, use the device rotation menu located in the Layout Editor toolbar (indicated by the arrow in Figure 28-8) to view the user interface in landscape orientation:

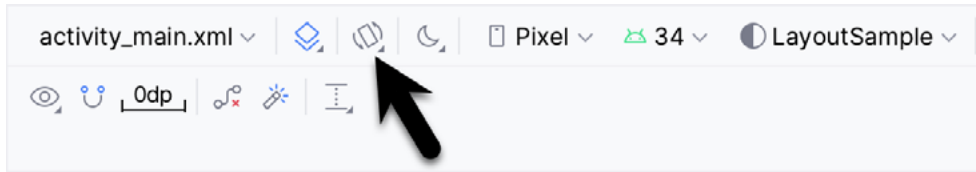


Figure 28-8

The absence of constraints results in a layout that fails to adapt to the change in device orientation, leaving the content off-center and with part of the image and all three buttons positioned beyond the screen's viewable area. Some work still needs to be done to make this a responsive user interface.

28.4 Adding the Constraints

Constraints are the key to creating layouts that adapt to device orientation changes and different screen sizes. Begin by rotating the layout back to portrait orientation and selecting the TextView widget above the ImageView. With the widget selected, establish constraints from the left, right and top sides of the TextView to the corresponding sides of the parent ConstraintLayout, as shown in Figure 28-9. Set the spacing on the top constraint to 16:

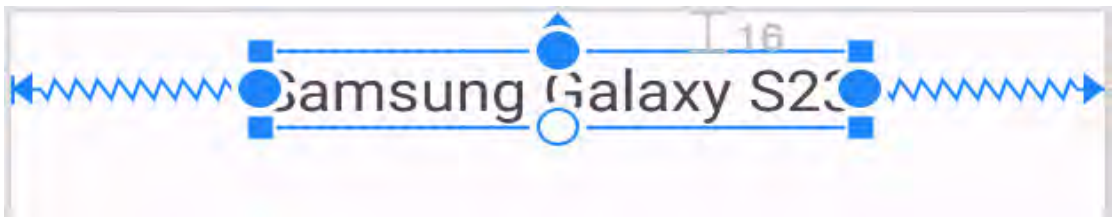


Figure 28-9

With the TextView widget constrained, select the ImageView instance and establish opposing constraints on the left and right sides, each connected to the corresponding sides of the parent layout. Next, establish a constraint connection from the top of the ImageView to the bottom of the TextView and from the bottom of the ImageView to the top of the center Button widget. If necessary, click and drag the ImageView to remain positioned in the vertical center of the layout.

With the ImageView still selected, use the Inspector in the attributes panel to change the top and bottom margins on the ImageView to 24 and 8, respectively, and to change both the widget height and width dimension properties to *match_constraint* so that the widget will resize to match the constraints. These settings will allow the layout engine to enlarge and reduce the size of the ImageView when necessary to accommodate layout changes:

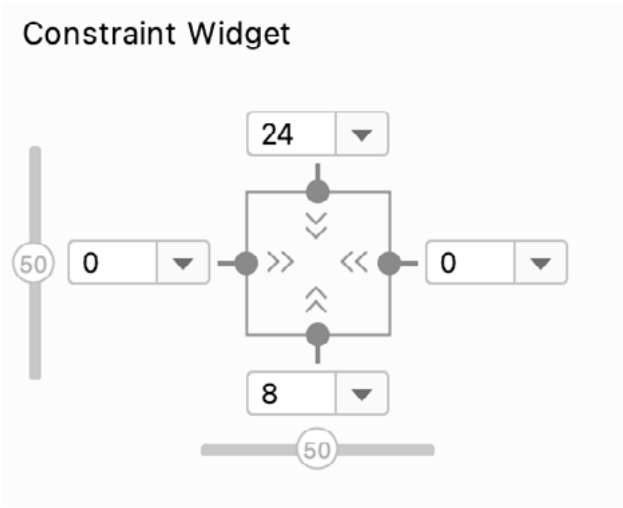


Figure 28-10

Figure 28-11 shows the currently implemented constraints for the ImageView relative to the other elements in the layout:



Figure 28-11

The final task is to add constraints to the three Button widgets. For this example, the buttons will be placed in a chain. Begin by turning on Autoconnect within the Layout Editor by clicking the toolbar button highlighted in Figure 28-1.

Next, click on the Buy Now button and then shift-click on the other two buttons to select all three. Right-click on the Buy Now button and select the *Chains -> Create Horizontal Chain* menu option from the resulting menu. By default, the chain will be displayed using the spread style, which is the correct behavior for this example.

Finally, establish a constraint between the bottom of the Buy Now button and the bottom of the layout with a margin of 8. Repeat this step for the remaining buttons.

On completion of these steps, the buttons should be constrained as outlined in Figure 28-12:

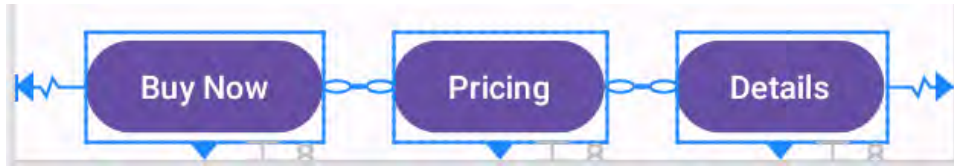


Figure 28-12

28.5 Testing the Layout

With the constraints added to the layout, rotate the screen into landscape orientation and verify that the layout adapts to accommodate the new screen dimensions.

While the Layout Editor tool provides a good visual environment in which to design user interface layouts, when it comes to testing, there is no substitute for testing the running app. Launch the app on a physical Android device or emulator session and verify that the user interface reflects the layout created in the Layout Editor. Figure 28-13, for example, shows the running app in landscape orientation:

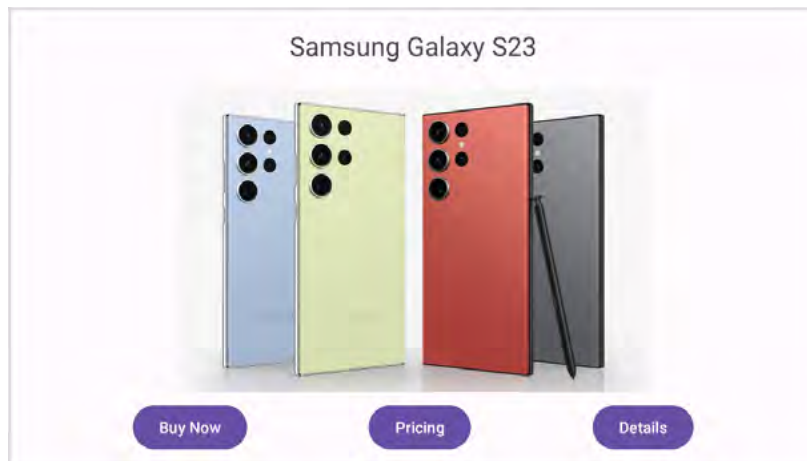


Figure 28-13

The user interface design is now complete. Designing a more complex user interface layout is a continuation of the steps outlined above. Drag and drop views onto the display, position, constrain and set properties as needed.

28.6 Using the Layout Inspector

The hierarchy of components comprising a user interface layout may be viewed using the Layout Inspector tool. The app must be running on a device or emulator running Android API 29 or later to access this information. Once the app is running, select the *Tools -> Layout Inspector* menu option, followed by the process to be inspected using the menu marked A in Figure 28-14 below).

Once the inspector loads, the leftmost panel (A) shows the hierarchy of components that make up the user

interface layout. The center panel (B) visually represents the layout design. Clicking on a widget in the visual layout will cause that item to highlight in the hierarchy list, making it easy to find where a visual component is situated relative to the overall layout hierarchy.

The right-most panel (marked C in Figure 28-14) contains all the property settings for the currently selected component, allowing for an in-depth analysis of the component's internal configuration. Where appropriate, the value cell will contain a link to the location of the property setting within the project source code.

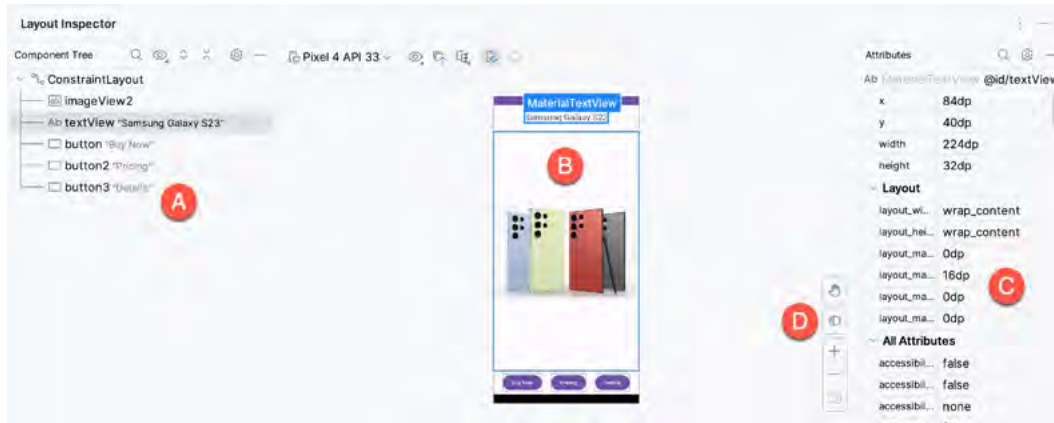


Figure 28-14

To view the layout in 3D, click on the button labeled D. This displays an “exploded” representation of the hierarchy so that it can be rotated and inspected. This can be useful for tasks such as identifying obscured views:

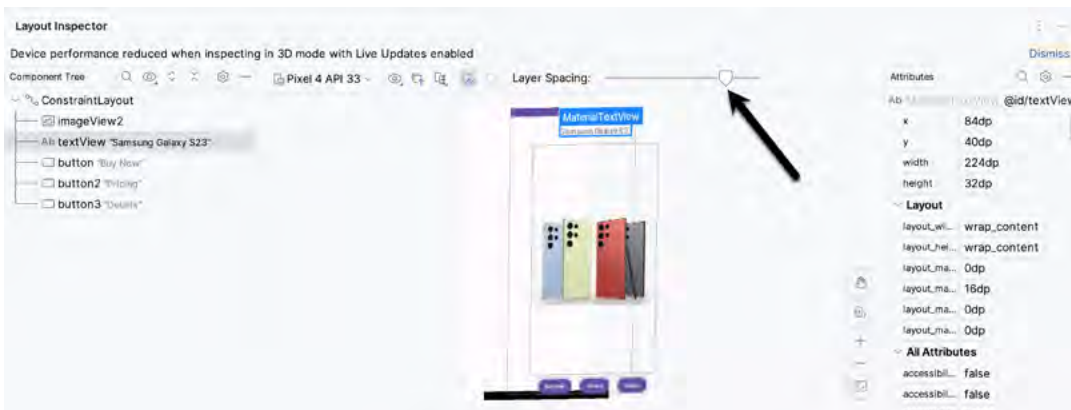


Figure 28-15

Click and drag the rendering to rotate it in three dimensions, using the slider indicated by the arrow above to increase the spacing between the layers. Click the button marked E again to return to the 2D view.

28.7 Summary

The Layout Editor tool in Android Studio has been tightly integrated with the ConstraintLayout class. This chapter has worked through creating an example user interface intended to outline how a ConstraintLayout-based user interface can be implemented using the Layout Editor tool to add widgets and set constraints. This chapter also introduced the Live Layout Inspector tool, which is useful for analyzing the structural composition of a user interface layout.

35. Detecting Common Gestures Using the Android Gesture Detector Class

The term “gesture” defines a contiguous sequence of interactions between the touch screen and the user. A typical gesture begins at the point that the screen is first touched and ends when the last finger or pointing device leaves the display surface. When correctly harnessed, gestures can be implemented to communicate between the user and the application. Swiping motions to turn the pages of an eBook or a pinching movement involving two touches to zoom in or out of an image are prime examples of how gestures can interact with an application.

The Android SDK provides mechanisms for the detection of both common and custom gestures within an application. Common gestures involve interactions such as a tap, double tap, long press, or a swiping motion in either a horizontal or a vertical direction (referred to in Android nomenclature as a *fling*).

This chapter explores using the Android GestureDetector class to detect common gestures performed on the display of an Android device. The next chapter, “*Implementing Custom Gesture and Pinch Recognition on Android*”, will cover detecting more complex, custom gestures such as circular motions and pinches.

35.1 Implementing Common Gesture Detection

When a user interacts with the display of an Android device, the *onTouchEvent()* method of the currently active application is called by the system and passed MotionEvent objects containing data about the user’s contact with the screen. This data can be interpreted to identify if the motion on the screen matches a common gesture such as a tap or a swipe. This can be achieved with minimal programming effort by using the Android GestureDetectorCompat class. This class is designed to receive motion event information from the application and trigger method calls based on the type of common gesture, if any, detected.

The basic steps in detecting common gestures are as follows:

1. Declaration of a class which implements the GestureDetector.OnGestureListener interface including the required *onFling()*, *onDown()*, *onScroll()*, *onShowPress()*, *onSingleTapUp()* and *onLongPress()* callback methods. Note that this can be either an entirely new or an enclosing activity class. If double-tap gesture detection is required, the class must also implement the GestureDetector.OnDoubleTapListener interface and include the corresponding *onDoubleTap()* method.
2. Creation of an instance of the Android GestureDetectorCompat class, passing through an instance of the class created in step 1 as an argument.
3. An optional call to the *setOnDoubleTapListener()* method of the GestureDetectorCompat instance to enable double tap detection if required.
4. Implementation of the *onTouchEvent()* callback method on the enclosing activity, which, in turn, must call the *onTouchEvent()* method of the GestureDetectorCompat instance, passing through the current motion event object as an argument to the method.

Detecting Common Gestures Using the Android Gesture Detector Class

Once implemented, the result is a set of methods within the application code that will be called when a gesture of a particular type is detected. The code within these methods can then be implemented to perform any tasks that need to be performed in response to the corresponding gesture.

In the remainder of this chapter, we will work through creating an example project intended to put the above steps into practice.

35.2 Creating an Example Gesture Detection Project

This project aims to detect the full range of common gestures currently supported by the `GestureDetectorCompat` class and to display status information to the user indicating the type of gesture that has been detected.

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 *CommonGestures* into the Name field and specify *com.ebookfrenzy.commongestures* 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.

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

Once the new project has been created, navigate to the *app -> res -> layout -> activity_main.xml* file in the Project tool window and double-click on it to load it into the Layout Editor tool.

Within the Layout Editor tool, select the “Hello, World!” `TextView` component and, in the Attributes tool window, enter *gestureStatusText* as the ID. Finally, set the `textSize` to 20sp and enable the bold `textStyle`:

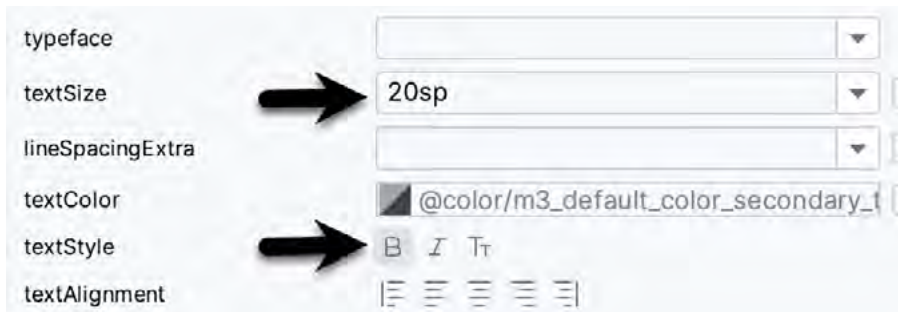


Figure 35-1

35.3 Implementing the Listener Class

As previously outlined, it is necessary to create a class that implements the `GestureDetector.OnGestureListener` interface and, if double tap detection is required, the `GestureDetector.OnDoubleTapListener` interface. While this can be an entirely new class, it is also perfectly valid to implement this within the current activity class. Therefore, we will modify the *MainActivity.kt* file to implement these listener interfaces for this example. Edit the *MainActivity.kt* file so that it reads as follows:

```
package com.ebookfrenzy.commongestures

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.GestureDetector
import android.view.MotionEvent
.
```

```

class MainActivity : AppCompatActivity(),
    GestureDetector.OnGestureListener, GestureDetector.OnDoubleTapListener
{
    .
    .

```

Declaring that the class implements the listener interfaces mandates that the corresponding methods also be implemented in the class:

```

class MainActivity : AppCompatActivity(),
    GestureDetector.OnGestureListener, GestureDetector.OnDoubleTapListener
{
    .
    .

    override fun onDown(event: MotionEvent): Boolean {
        binding.gestureStatusText.text = "onDown"
        return true
    }

    override fun onFling(event1: MotionEvent?, event2: MotionEvent,
        velocityX: Float, velocityY: Float): Boolean {
        binding.gestureStatusText.text = "onFling"
        return true
    }

    override fun onLongPress(event: MotionEvent) {
        binding.gestureStatusText.text = "onLongPress"
    }

    override fun onScroll(e1: MotionEvent?, e2: MotionEvent,
        distanceX: Float, distanceY: Float): Boolean {
        binding.gestureStatusText.text = "onScroll"
        return true
    }

    override fun onShowPress(event: MotionEvent) {
        binding.gestureStatusText.text = "onShowPress"
    }

    override fun onSingleTapUp(event: MotionEvent): Boolean {
        binding.gestureStatusText.text = "onSingleTapUp"
        return true
    }

    override fun onDoubleTap(event: MotionEvent): Boolean {
        binding.gestureStatusText.text = "onDoubleTap"
    }

```

```

        return true
    }

    override fun onDoubleTapEvent(event: MotionEvent): Boolean {
        binding.gestureStatusText.text = "onDoubleTapEvent"
        return true
    }

    override fun onSingleTapConfirmed(event: MotionEvent): Boolean {
        binding.gestureStatusText.text = "onSingleTapConfirmed"
        return true
    }
}

```

Note that many of these methods return *true*. This indicates to the Android Framework that the method has consumed the event and does not need to be passed to the next event handler in the stack.

35.4 Creating the GestureDetectorCompat Instance

With the activity class now updated to implement the listener interfaces, the next step is to create an instance of the GestureDetectorCompat class. Since this only needs to be performed once at the point that the activity is created, the best place for this code is in the *onCreate()* method. Since we also want to detect double taps, the code also needs to call the *setOnDoubleTapListener()* method of the GestureDetectorCompat instance:

```

.
.
import androidx.core.view.GestureDetectorCompat

class MainActivity : AppCompatActivity(), GestureDetector.OnGestureListener,
GestureDetector.OnDoubleTapListener
{
    private lateinit var binding: ActivityMainBinding
    var gDetector: GestureDetectorCompat? = null

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

        this.gDetector = GestureDetectorCompat(this, this)
        gDetector?.setOnDoubleTapListener(this)
    }
}
.
.

```

35.5 Implementing the onTouchEvent() Method

If the application were to be compiled and run at this point, nothing would happen if gestures were performed on the device display. This is because no code has been added to intercept touch events and to pass them through to the GestureDetectorCompat instance. To achieve this, it is necessary to override the *onTouchEvent()* method within

the activity class and implement it such that it calls the `onTouchEvent()` method of the `GestureDetectorCompat` instance. Remaining in the `MainActivity.kt` file, therefore, implement this method so that it reads as follows:

```
override fun onTouchEvent(event: MotionEvent): Boolean {
    this.gDetector?.onTouchEvent(event)
    // Be sure to call the superclass implementation
    return super.onTouchEvent(event)
}
```

35.6 Testing the Application

Compile and run the application on either a physical Android device or an AVD emulator. Once launched, experiment with swipes, presses, scrolling motions, and double and single taps. Note that the text view updates to reflect the events as illustrated in Figure 35-2:

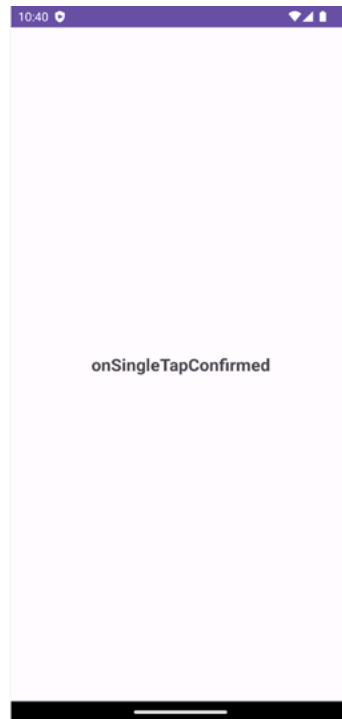


Figure 35-2

35.7 Summary

Any physical contact between the user and the touchscreen display of a device can be considered a “gesture”. Lacking the physical keyboard and mouse pointer of a traditional computer system, gestures are widely used as a method of interaction between the user and the application. While a gesture can comprise just about any sequence of motions, there is a widely used set of gestures with which users of touchscreen devices have become familiar. Some of these so-called “common gestures” can be easily detected within an application by using the Android Gesture Detector classes. In this chapter, the use of this technique has been outlined both in theory and through the implementation of an example project.

Having covered common gestures in this chapter, the next chapter will look at detecting a wider range of gesture types, including the ability to design and detect your own gestures.

37. An Introduction to Android Fragments

As you progress through the chapters of this book, it will become increasingly evident that many of the design concepts behind the Android system were conceived to promote the reuse of and interaction between the different elements that make up an application. One such area that will be explored in this chapter involves using Fragments.

This chapter will provide an overview of the basics of fragments in terms of what they are and how they can be created and used within applications. The next chapter will work through a tutorial designed to show fragments in action when developing applications in Android Studio, including the implementation of communication between fragments.

37.1 What is a Fragment?

A fragment is a self-contained, modular section of an application's user interface and corresponding behavior that can be embedded within an activity. Fragments can be assembled to create an activity during the application design phase and added to or removed from an activity during application runtime to create a dynamically changing user interface.

Fragments may only be used as part of an activity and cannot be instantiated as standalone application elements. However, a fragment can be considered a functional “sub-activity” with its own lifecycle similar to that of a full activity.

Fragments are stored in the form of XML layout files. They may be added to an activity by placing appropriate `<fragment>` elements in the activity's layout file or through code within the activity's class implementation.

37.2 Creating a Fragment

The two components that make up a fragment are an XML layout file and a corresponding Kotlin class. The XML layout file for a fragment takes the same format as a layout for any other activity layout and can contain any combination and complexity of layout managers and views. The following XML layout, for example, is for a fragment consisting of a `ConstraintLayout` with a red background containing a single `TextView` with a white foreground:

```
<?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:id="@+id/constraintLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/holo_red_dark"
    tools:context=".FragmentOne">
```

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="My First Fragment"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    android:textColor="@color/white"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The corresponding class to go with the layout must be a subclass of the Android *Fragment* class. This class should, at a minimum, override the *onCreateView()* method, which is responsible for loading the fragment layout. For example:

```
package com.example.myfragmentdemo

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment

class FragmentOne : Fragment() {

    private var _binding: FragmentTextBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTextBinding.inflate(inflater, container, false)
        return binding.root
    }
}
```

In addition to the *onCreateView()* method, the class may also override the standard lifecycle methods.

Once the fragment layout and class have been created, the fragment is ready to be used within application activities.

37.3 Adding a Fragment to an Activity using the Layout XML File

Fragments may be incorporated into an activity by writing Kotlin code or embedding the fragment into the activity's XML layout file. Regardless of the approach used, a key point to be aware of is that when the support library is being used for compatibility with older Android releases, any activities using fragments must be implemented as a subclass of *FragmentActivity* instead of the *AppCompatActivity* class:


```
package com.example.myFragmentDemo

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

class MainActivity : FragmentActivity() {
    .
    .
}
```

Fragments are embedded into activity layout files using the `FragmentManager` class. The following example layout embeds the fragment created in the previous section of this chapter into an activity layout:

```
<?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">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment2"
        android:name="com.ebookfrenzy.myfragmentdemo.FragmentOne"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="32dp"
        android:layout_marginEnd="32dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:layout="@layout/fragment_one" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

The key properties within the `<fragment>` element are *android:name*, which must reference the class associated with the fragment, and *tools:layout*, which must reference the XML resource file containing the fragment's layout.

Once added to the layout of an activity, fragments may be viewed and manipulated within the Android Studio Layout Editor tool. Figure 37-1, for example, shows the above layout with the embedded fragment within the Android Studio Layout Editor:



Figure 37-1

37.4 Adding and Managing Fragments in Code

The ease of adding a fragment to an activity via the activity's XML layout file comes at the cost of the activity not being able to remove the fragment at runtime. To achieve full dynamic control of fragments during runtime, those activities must be added via code. This has the advantage that the fragments can be added, removed, and even made to replace one another dynamically while the application is running.

When using code to manage fragments, the fragment will still consist of an XML layout file and a corresponding class. The difference comes when working with the fragment within the hosting activity. There is a standard sequence of steps when adding a fragment to an activity using code:

1. Create an instance of the fragment's class.
2. Pass any additional intent arguments through to the class instance.
3. Obtain a reference to the fragment manager instance.
4. Call the *beginTransaction()* method on the fragment manager instance. This returns a fragment transaction instance.
5. Call the *add()* method of the fragment transaction instance, passing through as arguments the resource ID of the view that is to contain the fragment and the fragment class instance.
6. Call the *commit()* method of the fragment transaction.

The following code, for example, adds a fragment defined by the `FragmentOne` class so that it appears in the container view with an ID of `LinearLayout1`:

```
val firstFragment = FragmentOne()
firstFragment.arguments = intent.extras
val transaction = fragmentManager.beginTransaction()
transaction.add(R.id.LinearLayout1, firstFragment)
transaction.commit()
```

The above code breaks down each step into a separate statement for clarity. The last four lines can, however, be abbreviated into a single line of code as follows:

```
supportFragmentManager.beginTransaction().add(
    R.id.LinearLayout1, firstFragment).commit()
```

Once added to a container, a fragment may subsequently be removed via a call to the *remove()* method of the fragment transaction instance, passing through a reference to the fragment instance that is to be removed:

```
transaction.remove(firstFragment)
```

Similarly, one fragment may be replaced with another by a call to the *replace()* method of the fragment transaction instance. This takes as arguments the ID of the view containing the fragment and an instance of the new fragment. The replaced fragment may also be placed on what is referred to as the *back* stack so that it can be quickly restored if the user navigates back to it. This is achieved by making a call to the *addToBackStack()* method of the fragment transaction object before making the *commit()* method call:

```
val secondFragment = FragmentTwo()
transaction.replace(R.id.LinearLayout1, secondFragment)
transaction.addToBackStack(null)
transaction.commit()
```

37.5 Handling Fragment Events

As previously discussed, a fragment is like a sub-activity with its layout, class, and lifecycle. The view components (such as buttons and text views) within a fragment can generate events like regular activity. This raises the question of which class receives an event from a view in a fragment, the fragment itself, or the activity in which the fragment is embedded. The answer to this question depends on how the event handler is declared.

In the chapter entitled “*An Overview and Example of Android Event Handling*”, two approaches to event handling were discussed. The first method involved configuring an event listener and callback method within the activity’s code. For example:

```
binding.button.setOnClickListener { // Code to be performed on button click }
```

In the case of intercepting click events, the second approach involved setting the *android:onClick* property within the XML layout file:

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="onClick"
    android:text="Click me" />
```

The general rule for events generated by a view in a fragment is that if the event listener were declared in the fragment class using the event listener and callback method approach, the event would be handled first by the fragment. However, if the *android:onClick* resource is used, the event will be passed directly to the activity containing the fragment.

37.6 Implementing Fragment Communication

Once one or more fragments are embedded within an activity, the chances are good that some form of communication will need to take place between the fragments and the activity and between one fragment and another. Good practice dictates that fragments do not communicate directly with one another. All communication should take place via the encapsulating activity.

An Introduction to Android Fragments

To communicate with a fragment, the activity must identify the fragment object via the ID assigned to it. Once this reference has been obtained, the activity can call the public methods of the fragment object.

Communicating in the other direction (from fragment to activity) is a little more complicated. In the first instance, the fragment must define a listener interface, which is then implemented within the activity class. For example, the following code declares a `ToolbarListener` interface on a fragment named `ToolbarFragment`. The code also declares a variable in which a reference to the activity will later be stored:

```
class ToolbarFragment : Fragment() {  
  
    var activityCallback: ToolbarFragment.ToolbarListener? = null  
  
    interface ToolbarListener {  
        fun onClick(fontsize: Int, text: String)  
    }  
    .  
    .  
}
```

The above code dictates that any class that implements the `ToolbarListener` interface must also implement a callback method named `onClick` which, in turn, accepts an integer and a `String` as arguments.

Next, the `onAttach()` method of the fragment class needs to be overridden and implemented. This method is called automatically by the Android system when the fragment has been initialized and associated with an activity. The method is passed a reference to the activity in which the fragment is contained. The method must store a local reference to this activity and verify that it implements the `ToolbarListener` interface:

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

Upon execution of this example, a reference to the activity will be stored in the local `activityCallback` variable, and an exception will be thrown if that activity does not implement the `ToolbarListener` interface.

The next step is to call the callback method of the activity from within the fragment. When and how this happens depends entirely on the circumstances under which the activity needs to be contacted by the fragment. The following code, for example, calls the callback method on the activity when a button is clicked:

```
override fun onClick(arg1: Int, arg2: String) {  
    activityCallback.onClick(arg1, arg2)  
}
```

All that remains is to modify the activity class to implement the `ToolbarListener` interface. For example:

```
class MainActivity : FragmentActivity(),  
    ToolbarFragment.ToolbarListener {
```

```

        override fun onClick(arg1: Int, arg2: String) {
            // Implement code for callback method
        }
    }
}

```

As we can see from the above code, the activity declares that it implements the `ToolbarListener` interface of the `ToolbarFragment` class and then proceeds to implement the `onClick()` method as required by the interface.

37.7 Summary

Fragments provide a powerful mechanism for creating reusable modules of user interface layout and application behavior, which, once created, can be embedded in activities. A fragment consists of a user interface layout file and a class. Fragments may be utilized in an activity by adding the fragment to the activity's layout file or writing code to manage the fragments at runtime. Fragments added to an activity in code can be removed and replaced dynamically at runtime. All communication between fragments should be performed via the activity within which the fragments are embedded.

Having covered the basics of fragments in this chapter, the next chapter will work through a tutorial designed to reinforce the techniques outlined in this chapter.

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.

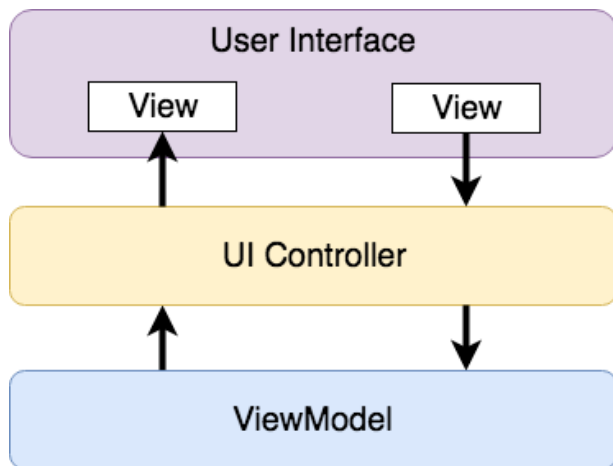


Figure 39-1

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.

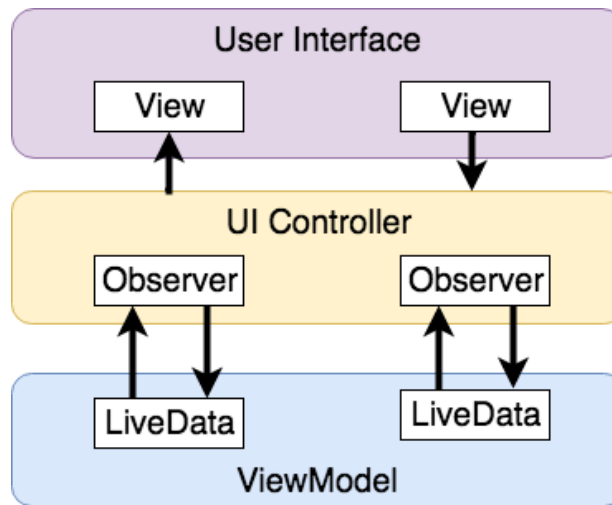


Figure 39-2

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.

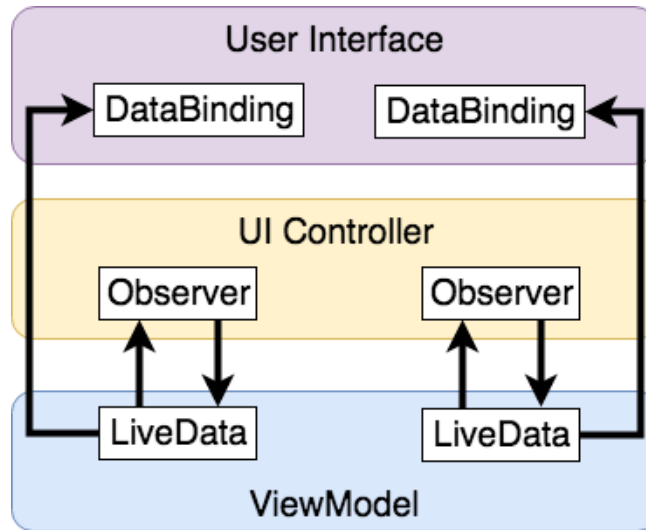


Figure 39-3

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.

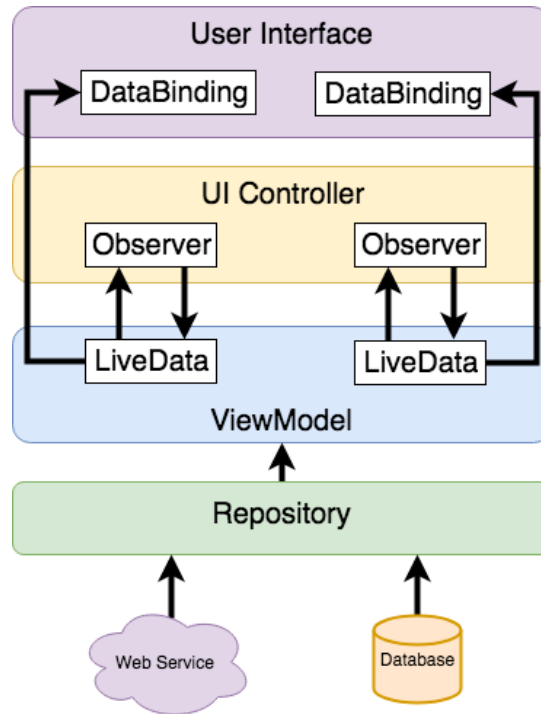


Figure 39-4

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.

47. An Overview of the Navigation Architecture Component

Very few Android apps today consist of just a single screen. In reality, most apps comprise multiple screens through which the user navigates using screen gestures, button clicks, and menu selections. Before the introduction of Android Jetpack, implementing navigation within an app was largely a manual coding process with no easy way to view and organize potentially complex navigation paths. However, this situation has improved considerably with the introduction of the Android Navigation Architecture Component combined with support for navigation graphs in Android Studio.

47.1 Understanding Navigation

Every app has a home screen that appears after the app has launched and after any splash screen has appeared (a splash screen being the app branding screen that appears temporarily while the app loads). The user will typically perform tasks from this home screen, resulting in other screens appearing. These screens will usually take the form of other activities and fragments within the app. For example, a messaging app may have a home screen listing current messages from which users can navigate to another screen to access a contact list or a settings screen. The contacts list screen, in turn, might allow the user to navigate to other screens where new users can be added or existing contacts updated. Graphically, the app's *navigation graph* might be represented as shown in Figure 47-1:

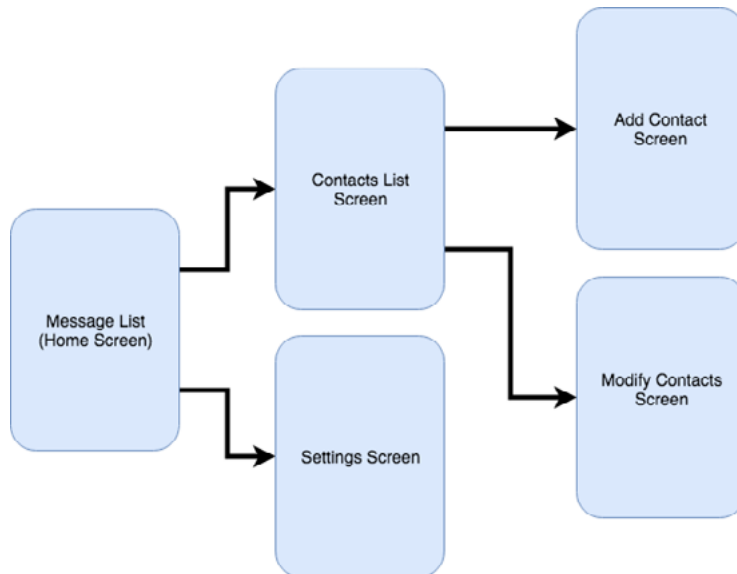


Figure 47-1

Each screen that makes up an app, including the home screen, is referred to as a *destination* and is usually a fragment or activity. The Android navigation architecture uses a *navigation stack* to track the user's path through the destinations within the app. When the app first launches, the home screen is the first destination placed onto the stack and becomes the *current destination*. When the user navigates to another destination, that screen

becomes the current destination and is *pushed* onto the stack above the home destination. As the user navigates to other screens, they are also pushed onto the stack. Figure 47-2, for example, shows the current state of the navigation stack for the hypothetical messaging app after the user has launched the app and is navigating to the “Add Contact” screen:

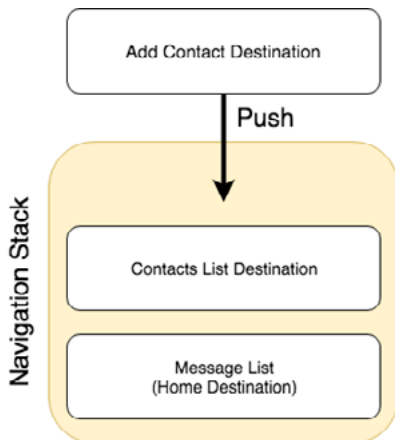


Figure 47-2

As the user navigates back through the screens using the system back button, each destination is *popped* off the stack until the home screen is once again the only destination on the stack. In Figure 47-3, the user has navigated back from the Add Contact screen, popping it off the stack and making the Contacts List screen the current destination:

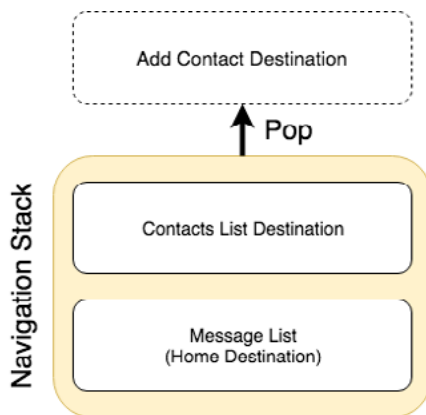


Figure 47-3

All of the work involved in navigating between destinations and managing the navigation stack is handled by a *navigation controller*, represented by the NavController class.

Adding navigation to an Android project using the Navigation Architecture Component is a straightforward process involving a navigation host, navigation graph, navigation actions, and minimal code writing to obtain a reference to, and interact with, the navigation controller instance.

47.2 Declaring a Navigation Host

A navigation host is a special fragment (NavHostFragment) embedded into the user interface layout of an activity and serves as a placeholder for the destinations through which the user will navigate. Figure 47-4, for example, shows a typical activity screen and highlights the area represented by the navigation host fragment:



Figure 47-4

A `NavHostFragment` can be placed into an activity layout within the Android Studio layout editor either by dragging and dropping an instance from the Containers section of the palette or by manually editing the XML as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout 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:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/demo_nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/navigation_graph" />

</FrameLayout>
```

The points of note in the above navigation host fragment element are the reference to the `NavHostFragment` in the *name* property, the setting of *defaultNavHost* to true, and the assignment of the file containing the navigation graph to the *navGraph* property.

When the activity launches, this navigation host fragment is replaced by the home destination designated in the navigation graph. As the user navigates through the app screens, the host fragment will be replaced by the appropriate fragment for the destination.

47.3 The Navigation Graph

A navigation graph is an XML file that contains the destinations that will be included in the app navigation. In addition to these destinations, the file contains navigation actions that define navigation between destinations and optional arguments for passing data from one destination to another. Android Studio includes a navigation graph editor that can be used to design graphs and implement actions either visually or by manually editing the XML.

Figure 47-5 shows the Android Studio navigation graph editor in Design mode:

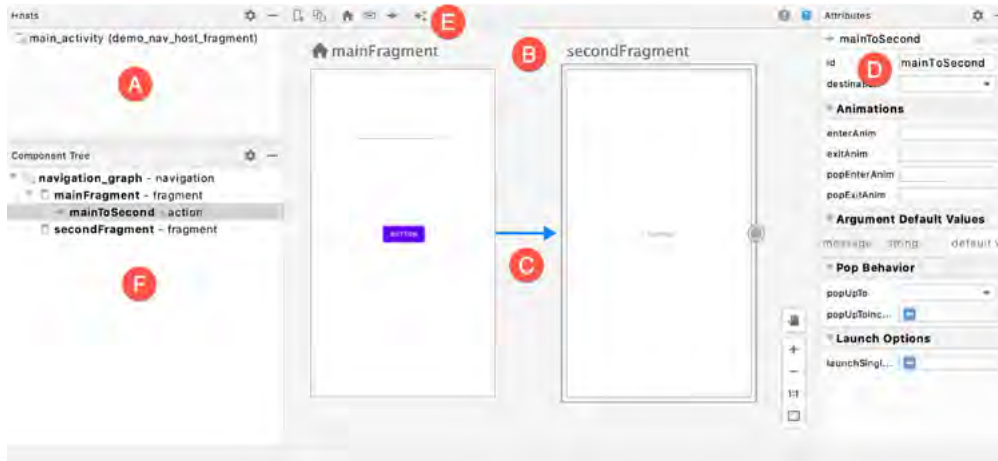


Figure 47-5

The destinations list (A) lists all destinations within the graph. Selecting a destination from the list will locate and select the corresponding destination in the graph (particularly useful for locating specific destinations in a large graph). The navigation graph panel (B) contains a dialog for each destination representing the user interface layout. In this example, this graph contains two destinations named `mainFragment` and `secondFragment`. Arrows between destinations (C) represent navigation action connections. Actions are added by hovering the mouse pointer over the edge of the origin until a circle appears, then clicking and dragging from the circle to the destination. The Attributes panel (D) allows the properties of the currently selected destination or action connection to be viewed and modified. In the above figure, the attributes for the action are displayed. New destinations are added by clicking on the button marked E and selecting options from a menu. Options are available to add existing fragments or activities as destinations or to create new blank fragment destinations. The Component Tree panel (F) provides a hierarchical overview of the navigation graph.

The underlying XML for the navigation graph can be viewed and modified by switching the editor into Code mode. The following XML listing represents the navigation graph for the destinations and action connection shown in Figure 47-5 above:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation 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:id="@+id/navigation_graph"
    app:startDestination="@id/mainFragment">

    <fragment
        android:id="@+id/mainFragment"
```



```

        android:name="com.ebookfrenzy.navigationdemo.ui.main.MainFragment"
        android:label="fragment_main"
        tools:layout="@layout/fragment_main" >
        <action
            android:id="@+id/mainToSecond"
            app:destination="@id/secondFragment" />
    </fragment>
    <fragment
        android:id="@+id/secondFragment"
        android:name="com.ebookfrenzy.navigationdemo.SecondFragment"
        android:label="fragment_second"
        tools:layout="@layout/fragment_second" >
    </fragment>
</navigation>

```

Navigation graphs can also be split over multiple files to improve organization and promote reuse. When structured in this way, *nested graphs* are embedded into *root graphs*. To create a nested graph, shift-click on the destinations to be nested, right-click over the first destination and select the *Move to Nested Graph -> New Graph* menu option. The nested graph will then appear as a new node in the graph. Double-click on the nested graph node to load the graph file into the editor to access the nested graph.

47.4 Accessing the Navigation Controller

Navigating from one destination to another usually occurs in response to an event within an app, such as a button click or menu selection. Before a navigation action can be triggered, the code must first obtain a reference to the navigation controller instance. This requires a call to the *findNavController()* method of the Navigation or NavHostFragment classes. The following code, for example, can be used to access the navigation controller of an activity. Note that for the code to work, the activity must contain a navigation host fragment:

```

val controller: NavController =
    Navigation.findNavController(activity, R.id.demo_nav_host_fragment)

```

In this case, the method call is passed a reference to the activity and the id of the NavHostFragment embedded in the activity's layout.

Alternatively, the navigation controller associated with any view may be identified by passing that view to the method:

```

val controller: NavController = Navigation.findNavController(button)

```

The final option finds the navigation controller for a fragment by calling the *findNavController()* method of the NavHostFragment class, passing through a reference to the fragment:

```

val controller: NavController = NavHostFragment.findNavController(fragment)

```

47.5 Triggering a Navigation Action

Once the navigation controller has been found, a navigation action is triggered by calling the controller's *navigate()* method and passing through the resource id of the action to be performed. For example:

```

controller.navigate(R.id.goToContactsList)

```

The id of the action is defined within the Attributes panel of the navigation graph editor when an action connection is selected.

47.6 Passing Arguments

Data may be passed from one destination to another during a navigation action by using arguments declared within the navigation graph file. An argument consists of a name, type, and an optional default value and may be added manually within the XML or using the Attributes panel when an action arrow or destination is selected within the graph. In Figure 47-6, for example, an integer argument named *contactsCount* has been declared with a default value of 0:

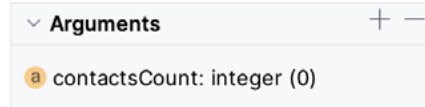


Figure 47-6

Once added, arguments are placed within the XML element of the receiving destination, for example:

```
<fragment
    android:id="@+id/secondFragment"
    android:name="com.ebookfrenzy.navigationdemo.SecondFragment"
    android:label="fragment_second"
    tools:layout="@layout/fragment_second" >
    <argument
        android:name="contactsCount"
        android:defaultValue=0
        app:type="integer" />
</fragment>
```

The Navigation Architecture Component provides two techniques for passing data between destinations. One approach involves placing the data into a Bundle object that is passed to the destination during an action, where it is then unbundled and the arguments extracted.

The main drawback to this particular approach is that it is not “type safe”. In other words, if the receiving destination treats an argument as a different type than it was declared (for example, treating a string as an integer) this error will not be caught by the compiler and will likely cause problems at runtime.

A better option, which is used in this book, is *safeargs*. Safeargs is a plugin for the Android Studio Gradle build system which automatically generates special classes that allow arguments to be passed in a type-safe way. The safeargs approach to argument passing will be described and demonstrated in the next chapter (“*An Android Jetpack Navigation Component Tutorial*”).

47.7 Summary

Navigation within the context of an Android app user interface refers to the ability of a user to move back and forth between different screens. Once time-consuming to implement and difficult to organize, Android Studio and the Navigation Architecture Component now make it easier to implement and manage navigation within Android app projects.

The different screens within an app are referred to as destinations and are usually represented by fragments or activities. All apps have a home destination, including the screen displayed when the app first loads. The content area of this layout is replaced by a navigation host fragment which is swapped out for other destination fragments as the user navigates the app. The navigation path is defined by the navigation graph file consisting of destinations and the actions that connect them together with any arguments to be passed between destinations. Navigation is handled by navigation controllers, which, in addition to managing the navigation stack, provide methods to initiate navigation actions from within app code.

49. An Introduction to MotionLayout

The MotionLayout class provides an easy way to add animation effects to the views of a user interface layout. This chapter will begin by providing an overview of MotionLayout and introduce the concepts of MotionScenes, Transitions, and Keyframes. Once these basics have been covered, the next two chapters (entitled “*An Android MotionLayout Editor Tutorial*” and “*A MotionLayout KeyCycle Tutorial*”) will provide additional detail and examples of MotionLayout animation in action through the creation of example projects.

49.1 An Overview of MotionLayout

MotionLayout is a layout container, the primary purpose of which is to animate the transition of views within a layout from one state to another. MotionLayout could, for example, animate the motion of an ImageView instance from the top left-hand corner of the screen to the bottom right-hand corner over a specified time. In addition to the position of a view, other attribute changes may also be animated, such as the color, size, or rotation angle. These state changes can also be interpolated (such that a view moves, rotates, and changes size throughout the animation).

The motion of a view using MotionLayout may be performed in a straight line between two points or implemented to follow a path comprising intermediate points at different positions between the start and end points. MotionLayout also supports using touches and swipes to initiate and control animation.

MotionLayout animations are declared entirely in XML and do not typically require writing code. These XML declarations may be implemented manually in the Android Studio code editor, visually using the MotionLayout editor, or combining both approaches.

49.2 MotionLayout

When implementing animation, the ConstraintLayout container typically used in a user interface must first be converted to a MotionLayout instance (a task which can be achieved by right-clicking on the ConstraintLayout in the layout editor and selecting the *Convert to MotionLayout* menu option). MotionLayout also requires at least version 2.0.0 of the ConstraintLayout library.

Unsurprisingly since it is a subclass of ConstraintLayout, MotionLayout supports all of the layout features of the ConstraintLayout. Therefore, a user interface layout can be similarly designed when using MotionLayout for views that do not require animation.

For views that are to be animated, two ConstraintSets are declared, defining the appearance and location of the view at the start and end of the animation. A *transition* declaration defines *keyframes* to apply additional effects to the target view between these start and end states and click and swipe handlers used to start and control the animation.

The start and end ConstraintSets and the transitions are declared within a MotionScene XML file.

49.3 MotionScene

As we have seen in earlier chapters, an XML layout file contains the information necessary to configure the appearance and layout behavior of the static views presented to the user, and this is still the case when using MotionLayout. For non-static views (in other words, the views that will be animated), those views are still declared within the layout file, but the start, end, and transition declarations related to those views are stored in a separate XML file referred to as the MotionScene file (so called because all of the declarations are defined

within a `MotionScene` element). This file is imported into the layout XML file and contains the start and end `ConstraintSets` and `Transition` declarations (a single file can contain multiple `ConstraintSet` pairs and `Transition` declarations, allowing different animations to be targeted to specific views within the user interface layout).

The following listing shows a template for a `MotionScene` file:

```
<?xml version="1.0" encoding="utf-8"?>
<MotionScene
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:motion="http://schemas.android.com/apk/res-auto">

    <Transition
        motion:constraintSetEnd="@+id/end"
        motion:constraintSetStart="@id/start"
        motion:duration="1000">
        <KeyFrameSet>
        </KeyFrameSet>
    </Transition>

    <ConstraintSet android:id="@+id/start">
    </ConstraintSet>

    <ConstraintSet android:id="@+id/end">
    </ConstraintSet>
</MotionScene>
```

In the above XML, `ConstraintSets` named *start* and *end* (though any name can be used) have been declared, which, at this point, are yet to contain any constraint elements. The `Transition` element defines that these `ConstraintSets` represent the animation start and end points and contain an empty `KeyFrameSet` element ready to be populated with additional animation keyframe entries. The `Transition` element also includes a millisecond duration property to control the running time of the animation.

`ConstraintSets` do not have to imply the motion of a view. It is possible to have the start and end sets declare the same location on the screen and then use the transition to animate other property changes, such as scale and rotation angle.

`ConstraintSets` do not have to imply the motion of a view. It is possible, for example, to have the start and end sets declare the same location on the screen and then use the transition to animate other property changes, such as scale and rotation angle.

49.4 Configuring ConstraintSets

The `ConstraintSets` in the `MotionScene` file allow the full set of `ConstraintLayout` settings to be applied to a view regarding positioning, sizing, and relation to the parent and other views. In addition, the following attributes may also be included within the `ConstraintSet` declarations:

- alpha
- visibility
- elevation
- rotation

- rotationX
- rotationY
- translationX
- translationY
- translationZ
- scaleX
- scaleY

For example, to rotate the view by 180° during the animation, the following could be declared within the start and end constraints:

```
<ConstraintSet android:id="@+id/start">
    <Constraint
        .
        .
        motion:layout_constraintStart_toStartOf="parent"
        android:rotation="0">
    </Constraint>
</ConstraintSet>

<ConstraintSet android:id="@+id/end">
    <Constraint
        .
        .
        motion:layout_constraintBottom_toBottomOf="parent"
        android:rotation="180">
    </Constraint>
</ConstraintSet>
```

The above changes tell MotionLayout that the view is to start at 0° and then, during the animation, rotate a full 180° before coming to rest upside-down.

49.5 Custom Attributes

In addition to the standard attributes listed above, it is possible to specify a range of *custom attributes* (declared using CustomAttribute). In fact, just about any property available on the view type can be specified as a custom attribute for inclusion in an animation. To identify the attribute's name, find the getter/setter name from the documentation for the target view class, remove the get/set prefix, and lower the case of the first remaining character. For example, to change the background color of a Button view in code, we might call the *setBackgroundColor()* setter method as follows:

```
myButton.setBackgroundColor(Color.RED)
```

When setting this attribute in a constraint set or keyframe, the attribute name will be *backgroundColor*. In addition to the attribute name, the value must also be declared using the appropriate type from the following list of options:

- **motion:customBoolean** - Boolean attribute values.

- **motion:customColorValue** - Color attribute values.
- **motion:customDimension** - Dimension attribute values.
- **motion:customFloatValue** - Floating point attribute values.
- **motion:customIntegerValue** - Integer attribute values.
- **motion:customStringValue** - String attribute values

For example, a color setting will need to be assigned using the *customColorValue* type :

```
<CustomAttribute
    motion:attributeName="backgroundColor"
    motion:customColorValue="#43CC76" />
```

The following excerpt from a MotionScene file, for example, declares start and end constraints for a view in addition to changing the background color from green to red:

```
.
.
<ConstraintSet android:id="@+id/start">
    <Constraint
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        motion:layout_editor_absoluteX="21dp"
        android:id="@+id/button"
        motion:layout_constraintTop_toTopOf="parent"
        motion:layout_constraintStart_toStartOf="parent" >
        <CustomAttribute
            motion:attributeName="backgroundColor"
            motion:customColorValue="#33CC33" />
    </Constraint>
</ConstraintSet>

<ConstraintSet android:id="@+id/end">
    <Constraint
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        motion:layout_editor_absoluteY="21dp"
        android:id="@+id/button"
        motion:layout_constraintEnd_toEndOf="parent"
        motion:layout_constraintBottom_toBottomOf="parent" >
        <CustomAttribute
            motion:attributeName="backgroundColor"
            motion:customColorValue="#F80A1F" />
    </Constraint>
</ConstraintSet>

.
.
```

49.6 Triggering an Animation

Without some event to tell MotionLayout to start the animation, none of the settings in the MotionScene file will affect the layout (except that the view will be positioned based on the setting in the start ConstraintSet).

The animation can be configured to start in response to either screen tap (OnClick) or swipe motion (OnSwipe) gesture. The OnClick handler causes the animation to start and run until completion, while OnSwipe will synchronize the animation to move back and forth along the timeline to match the touch motion. The OnSwipe handler will also respond to “flinging” motions on the screen. The OnSwipe handler also provides options to configure how the animation reacts to dragging in different directions and the side of the target view to which the swipe is to be anchored. This allows, for example, left-ward dragging motions to move a view in the corresponding direction while preventing an upward motion from causing a view to move sideways (unless, of course, that is the required behavior).

The OnSwipe and OnClick declarations are contained within the Transition element of a MotionScene file. In both cases, the view id must be specified. For example, to implement an OnSwipe handler responding to downward drag motions anchored to the bottom edge of a view named *button*, the following XML would be placed in the Transition element:

```
.
.
<Transition
    motion:constraintSetEnd="@+id/end"
    motion:constraintSetStart="@id/start"
    motion:duration="1000">
    <KeyFrameSet>
    </KeyFrameSet>
    <OnSwipe
        motion:touchAnchorId="@+id/button"
        motion:dragDirection="dragDown"
        motion:touchAnchorSide="bottom" />
</Transition>
.
.
```

Alternatively, to add an OnClick handler to the same button:

```
<OnClick motion:targetId="@id/button"
    motion:clickAction="toggle" />
```

In the above example, the action has been set to *toggle* mode. This mode and the other available options can be summarized as follows:

- **toggle** - Animates to the opposite state. For example, if the view is currently at the transition start point, it will transition to the end point, and vice versa.
- **jumpToStart** - Changes immediately to the start state without animation.
- **jumpToEnd** - Changes immediately to the end state without animation.
- **transitionToStart** - Transitions with animation to the start state.
- **transitionToEnd** - Transitions with animation to the end state.

49.7 Arc Motion

By default, a movement of view position will travel in a straight line between the start and end points. To change the motion to an arc path, use the *pathMotionArc* attribute as follows within the start constraint, configured with either a *startHorizontal* or *startVertical* setting to define whether the arc is to be concave or convex:

```
<ConstraintSet android:id="@+id/start">
    <Constraint
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        motion:layout_editor_absoluteX="21dp"
        android:id="@+id/button"
        motion:layout_constraintTop_toTopOf="parent"
        motion:layout_constraintStart_toStartOf="parent"
        motion:pathMotionArc="startVertical" >
```

Figure 49-1 illustrates *startVertical* and *startHorizontal* arcs in comparison to the default straight line motion:

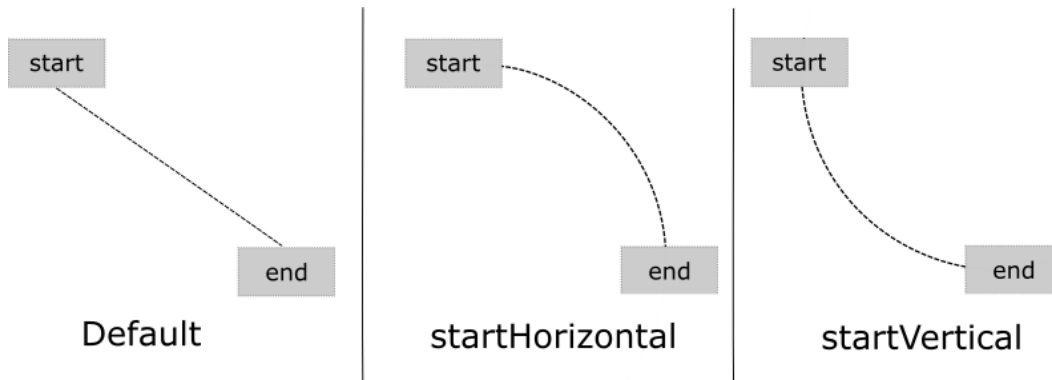


Figure 49-1

49.8 Keyframes

All of the *ConstraintSet* attributes outlined so far only apply to the start and end points of the animation. In other words, if the rotation property were set to 180° on the end point, the rotation would begin when the animation starts and complete when the end point is reached. It is not, therefore, possible to configure the rotation to reach the full 180° at a point 50% of the way through the animation and then rotate back to the original orientation by the end. Fortunately, this type of effect is available using Keyframes.

Keyframes are used to define intermediate points during the animation at which state changes are to occur. Keyframes could, for example, be declared such that the background color of a view is to have transitioned to blue at a point 50% of the way through the animation, green at the 75% point, and then back to the original color by the end of the animation. Keyframes are implemented within the *Transition* element of the *MotionScene* file embedded into the *KeyFrameSet* element.

MotionLayout supports several types of Keyframe which can be summarized as follows:

49.8.1 Attribute Keyframes

Attribute Keyframes (declared using *KeyAttribute*) allow view attributes to be changed at intermediate points in the animation timeline. *KeyAttribute* supports the attributes listed above for *ConstraintSets* combined with the ability to specify where the change will take effect in the animation timeline. For example, the following

Keyframe declaration will gradually cause the button view to double in size horizontally (scaleX) and vertically (scaleY), reaching full size at 50% through the timeline. For the remainder of the timeline, the view will decrease in size to its original dimensions:

```
<Transition
    motion:constraintSetEnd="@+id/end"
    motion:constraintSetStart="@+id/start"
    motion:duration="1000">
<KeyFrameSet>
    <KeyAttribute
        motion:motionTarget="@+id/button"
        motion:framePosition="50"
        android:scaleX="2.0" />
    <KeyAttribute
        motion:motionTarget="@+id/button"
        motion:framePosition="50"
        android:scaleY="2.0" />
</KeyFrameSet>
```

49.8.2 Position Keyframes

Position keyframes (KeyPosition) modify the path followed by a view as it moves between the start and end locations. By placing key positions at different points on the timeline, a path of just about any level of complexity can be applied to an animation. Positions are declared using x and y coordinates combined with the corresponding points in the transition timeline. These coordinates must be declared relative to one of the following coordinate systems:

- **parentRelative** - The x and y coordinates are relative to the parent container where the coordinates are specified as a percentage (represented as a value between 0.0 and 1.0):

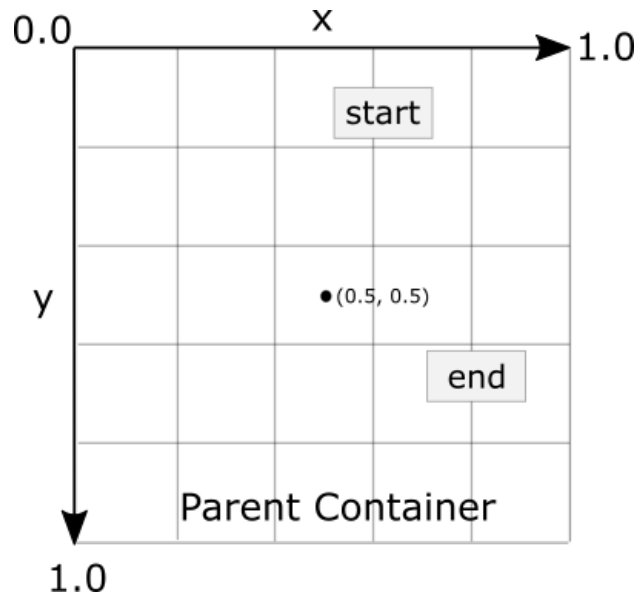


Figure 49-2

- **deltaRelative** - Instead of relative to the parent, the x and y coordinates are relative to the start and end

positions. For example, the start point is (0, 0) the end point (1, 1). Keep in mind that the x and y coordinates can be negative values):

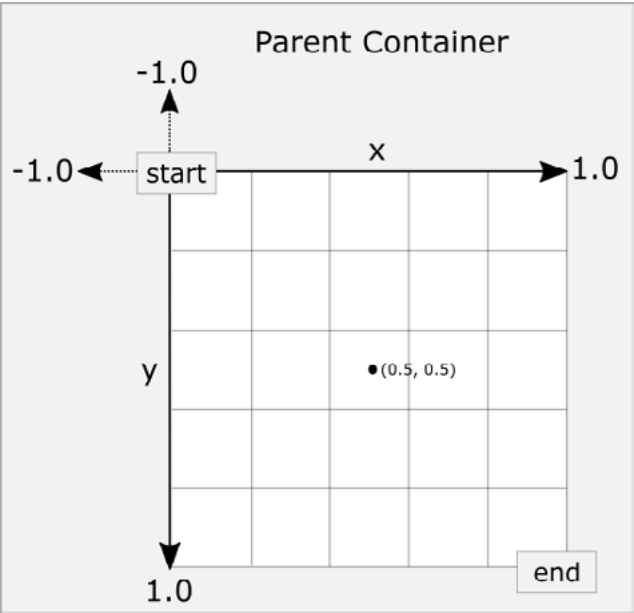


Figure 49-3

- **pathRelative** - The x and y coordinates are relative to the path, where the straight line between the start and end points serves as the graph's X-axis. Once again, coordinates are represented as a percentage (0.0 to 1.0). This is similar to the deltaRelative coordinate space but takes into consideration the angle of the path. Once again coordinates may be negative:

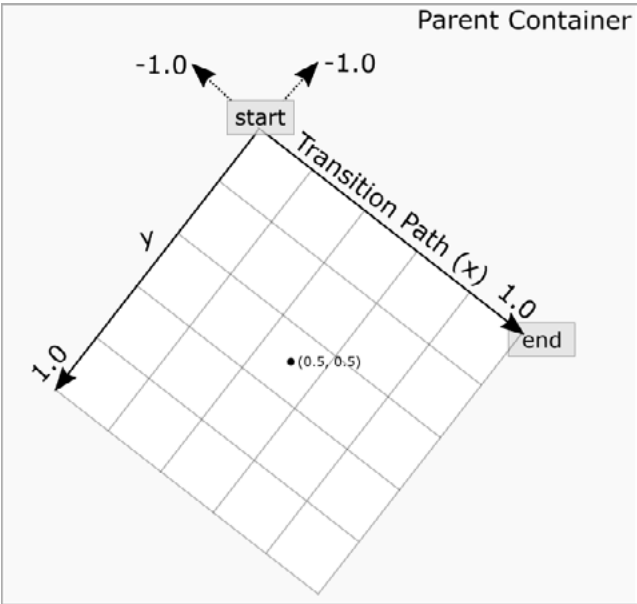


Figure 49-4

As an example, the following ConstraintSets declare start and end points on either side of a device screen. By

default, a view transition using these points would move in a straight line across the screen, as illustrated in Figure 49-5:



Figure 49-5

Suppose, however, that the view is required to follow a path similar to that shown in Figure 49-6 below:



Figure 49-6

To achieve this, keyframe position points could be declared within the transition as follows:

```
<KeyPosition
    motion:motionTarget="@+id/button"
    motion:framePosition="25"
    motion:keyPositionType="pathRelative"
    motion:percentY="0.3"
    motion:percentX="0.25"/>
```

```
<KeyPosition
    motion:motionTarget="@+id/button"
    motion:framePosition="75"
    motion:keyPositionType="pathRelative"
    motion:percentY="-0.3"
    motion:percentX="0.75"/>
```

The above elements create keyframe position points 25% and 75% through the path using the pathRelative coordinate system. The first position is placed at coordinates (0.25, 0.3) and the second at (0.75, -0.3). These position keyframes can be visualized as illustrated in Figure 49-7 below:

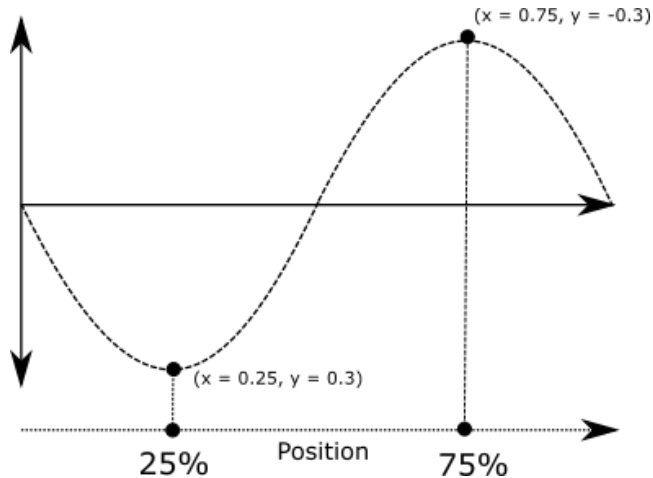


Figure 49-7

49.9 Time Linearity

Without additional settings, the animations outlined above will be performed at a constant speed. To vary the animation speed (for example, so that it accelerates and then decelerates), the transition easing attribute (`transitionEasing`) can be used within a `ConstraintSet` or `Keyframe`.

For complex easing requirements, the linearity can be defined by plotting points on a cubic Bézier curve, for example:

```
.
.
    motion:layout_constraintBottom_toBottomOf="parent"
    motion:transitionEasing="cubic(0.2, 0.7, 0.3, 1)"
    android:rotation="360">
.
.
```

If you are unfamiliar with Bézier curves, consider using the curve generator online at the following URL:

<https://cubic-bezier.com/>

For most requirements, however, easing can be specified using the built-in *standard*, *accelerate* and *decelerate* values:

```
.
.
    motion:layout_constraintBottom_toBottomOf="parent"
    motion:transitionEasing="decelerate"
    android:rotation="360">
.
.
```

49.10 KeyTrigger

The trigger keyframe (`KeyTrigger`) allows a method on a view to be called when the animation reaches a specified frame position within the animation timeline. This also takes into consideration the direction of the

animations. For example, different methods can be called depending on whether the animation runs forward or backward. Consider a button that is to be made visible when the animation moves beyond 20% of the timeline. The `KeyTrigger` would be implemented within the `KeyFrameSet` of the `Transition` element as follows using the `onPositiveCross` property:

```
.
.
    <KeyFrameSet>
        <KeyTrigger
            motion:framePosition="20"
            motion:onPositiveCross="show"
            motion:motionTarget="@id/button"/>
.
.
```

Similarly, if the same button is to be hidden when the animation is reversed and drops below 10%, a second key trigger could be added using the `onNegativeCross` property:

```
<KeyTrigger
    motion:framePosition="10"
    motion:onNegativeCross="show"
    motion:motionTarget="@id/button2"/>
```

If the animation is using toggle action, use the `onCross` property:

```
<KeyTrigger
    motion:framePosition="10"
    motion:onCross="show"
    motion:motionTarget="@id/button2"/>
```

49.11 Cycle and Time Cycle Keyframes

While position keyframes can be used to add intermediate state changes into the animation, this would quickly become cumbersome if large numbers of repetitive positions and changes needed to be implemented. For situations where state changes need to be performed repetitively with predictable changes, `MotionLayout` includes the `Cycle` and `Time Cycle` keyframes. The chapter entitled “*A MotionLayout KeyCycle Tutorial*” will cover this topic in detail.

49.12 Starting an Animation from Code

So far in this chapter, we have only looked at controlling an animation using the `OnSwipe` and `OnClick` handlers. It is also possible to start an animation from within code by calling methods on the `MotionLayout` instance. The following code, for example, runs the transition from start to end with a duration of 2000ms for a layout named `motionLayout`:

```
motionLayout.setTransitionDuration(2000)
motionLayout.transitionToEnd()
```

In the absence of additional settings, the start and end states used for the animation will be those declared in the `Transition` declaration of the `MotionScene` file. To use specific start and end constraint sets, reference them by id in a call to the `setTransition()` method of the `MotionLayout` instance:

```
motionLayout.setTransition(R.id.myStart, R.id.myEnd)
motionLayout.transitionToEnd()
```

An Introduction to MotionLayout

To monitor the state of an animation while it is running, add a transition listener to the MotionLayout instance as follows:

```
motionLayout.setTransitionListener(  
    object: MotionLayout.TransitionListener {  
  
        override fun onTransitionTrigger(motionLayout: MotionLayout?,  
            triggerId: Int, positive: Boolean, progress: Float) {  
            // Called when a trigger keyframe threshold is crossed  
        }  
  
        override fun onTransitionStarted(motionLayout: MotionLayout?,  
            startId: Int, endId: Int) {  
            // Called when the transition starts  
        }  
  
        override fun onTransitionChange(motionLayout: MotionLayout?,  
            startId: Int, endId: Int, progress: Float) {  
            // Called each time a property changes. Track progress value to find  
            // current position  
        }  
  
        override fun onTransitionCompleted(motionLayout: MotionLayout?,  
            currentId: Int) {  
            // Called when the transition is complete  
        }  
    })
```

49.13 Summary

MotionLayout is a subclass of ConstraintLayout designed specifically to add animation effects to the views in user interface layouts. MotionLayout works by animating the transition of a view between two states defined by start and end constraint sets. Additional animation effects may be added between these start and end points using keyframes.

Animations may be triggered via OnClick or OnSwipe handlers or programmatically via method calls on the MotionLayout instance.

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 `CoroutineScope` 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:

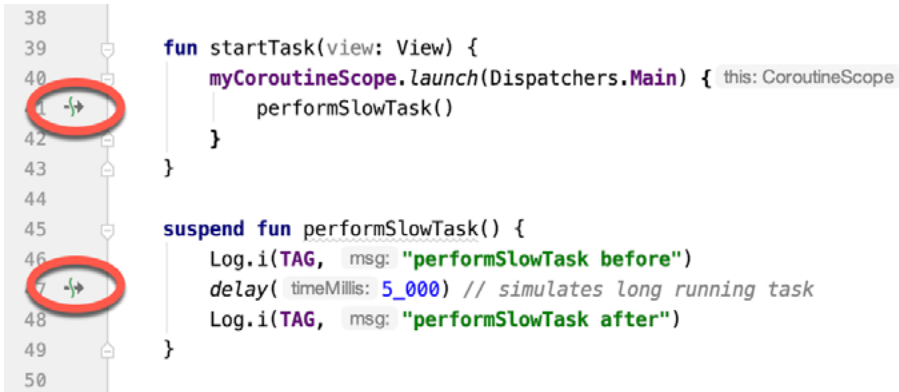


Figure 61-1

61.9 Returning Results from a Coroutine

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

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

```

fun startTask(view: View) {

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

```

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

```

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

```

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

61.10 Using withContext

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

designed to call multiple functions from within a suspend function:

```
fun startTask(view: View) {

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

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

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

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

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

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

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

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

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

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

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

Task 3 main

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

```
fun startTask(view: View) {

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

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

        return@withContext "Finished"
    }
}
```

61.11 Coroutine Channel Communication

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

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

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

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

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

An Introduction to Kotlin Coroutines

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

When executed, the following logcat output will be generated:

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

61.12 Summary

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

68. An Overview of Android SQLite Databases

Mobile applications that do not need to store at least some persistent data are few and far between. The use of databases is an essential aspect of most applications, ranging from almost entirely data-driven applications to those that need to store small amounts of data, such as the prevailing game score.

The importance of persistent data storage becomes even more evident when considering the transient lifecycle of the typical Android application. With the ever-present risk that the Android runtime system will terminate an application component to free up resources, a comprehensive data storage strategy to avoid data loss is a key factor in designing and implementing any application development strategy.

This chapter will cover the SQLite database management system bundled with the Android operating system and outline the Android SDK classes that facilitate persistent SQLite-based database storage within an Android application. Before delving into the specifics of SQLite in the context of Android development, however, a brief overview of databases and SQL will be covered.

68.1 Understanding Database Tables

Database *Tables* provide the most basic level of data structure in a database. Each database can contain multiple tables, each designed to hold information of a specific type. For example, a database may contain a *customer* table that contains the name, address, and telephone number of each of the customers of a particular business. The same database may also include a *products* table used to store the product descriptions with associated product codes for the items sold by the business.

Each table in a database is assigned a name that must be unique within that particular database. A table name, once assigned to a table in one database, may not be used for another table except within the context of another database.

68.2 Introducing Database Schema

Database Schemas define the characteristics of the data stored in a database table. For example, the table schema for a customer database table might define the customer name as a string of no more than 20 characters long and the customer phone number is a numerical data field of a certain format.

Schemas are also used to define the structure of entire databases and the relationship between the various tables in each database.

68.3 Columns and Data Types

It is helpful at this stage to begin viewing a database table as similar to a spreadsheet where data is stored in rows and columns.

Each column represents a data field in the corresponding table. For example, a table's name, address, and telephone data fields are all *columns*.

Each column, in turn, is defined to contain a certain type of data. Therefore, a column designed to store numbers would be defined as containing numerical data.

68.4 Database Rows

Each new record saved to a table is stored in a row. Each row, in turn, consists of the columns of data associated with the saved record.

Once again, consider the spreadsheet analogy described earlier in this chapter. Each entry in a customer table is equivalent to a row in a spreadsheet, and each column contains the data for each customer (name, address, telephone, etc.). When a new customer is added to the table, a new row is created, and the data for that customer is stored in the corresponding columns of the new row.

Rows are also sometimes referred to as *records* or *entries*, and these terms can generally be used interchangeably.

68.5 Introducing Primary Keys

Each database table should contain one or more columns that can be used to identify each row in the table uniquely. This is known in database terminology as the *Primary Key*. For example, a table may use a bank account number column as the primary key. Alternatively, a customer table may use the customer's social security number as the primary key.

Primary keys allow the database management system to uniquely identify a specific row in a table. Without a primary key, retrieving or deleting a specific row in a table would not be possible because there can be no certainty that the correct row has been selected. For example, suppose a table existed where the customer's last name had been defined as the primary key. Imagine the problem if more than one customer named "Smith" were recorded in the database. Without some guaranteed way to identify a specific row uniquely, ensuring the correct data was being accessed at any given time would be impossible.

Primary keys can comprise a single column or multiple columns in a table. To qualify as a single column primary key, no two rows can contain matching primary key values. When using multiple columns to construct a primary key, individual column values do not need to be unique, but all the columns' values combined must be unique.

68.6 What is SQLite?

SQLite is an embedded, relational database management system (RDBMS). Most relational databases (Oracle, SQL Server, and MySQL being prime examples) are standalone server processes that run independently and cooperate with applications requiring database access. SQLite is referred to as *embedded* because it is provided in the form of a library that is linked into applications. As such, there is no standalone database server running in the background. All database operations are handled internally within the application through calls to functions in the SQLite library.

The developers of SQLite have placed the technology into the public domain with the result that it is now a widely deployed database solution.

The developers of SQLite have placed the technology into the public domain with the result that it is now a widely deployed database solution.

SQLite is written in the C programming language, so the Android SDK provides a Java-based "wrapper" around the underlying database interface. This consists of classes that may be utilized within an application's Java or Kotlin code to create and manage SQLite-based databases.

For additional information about SQLite, refer to <https://www.sqlite.org>.

68.7 Structured Query Language (SQL)

Data is accessed in SQLite databases using a high-level language known as Structured Query Language. This is usually abbreviated to SQL and pronounced *sequel*. SQL is a standard language used by most relational database management systems. SQLite conforms mostly to the SQL-92 standard.

SQL is a straightforward and easy-to-use language designed specifically to enable the reading and writing of database data. Because SQL contains a small set of keywords, it can be learned quickly. In addition, SQL syntax is more or less identical between most DBMS implementations, so having learned SQL for one system, your skills will likely transfer to other database management systems.

While some basic SQL statements will be used within this chapter, a detailed overview of SQL is beyond the scope of this book. However, many other resources provide a far better overview of SQL than we could ever hope to provide in a single chapter here.

68.8 Trying SQLite on an Android Virtual Device (AVD)

For readers unfamiliar with databases and SQLite, diving right into creating an Android application that uses SQLite may seem intimidating. Fortunately, Android is shipped with SQLite pre-installed, including an interactive environment for issuing SQL commands from within an `adb` shell session connected to a running Android AVD emulator instance. This is a useful way to learn about SQLite and SQL and an invaluable tool for identifying problems with databases created by applications running in an emulator.

To launch an interactive SQLite session, begin by running an AVD session. This can be achieved within Android Studio by launching the Android Virtual Device Manager (*Tools -> AVD Manager*), selecting a previously configured AVD, and clicking on the start button.

Once the AVD is up and running, open a Terminal or Command-Prompt window and connect to the emulator using the `adb` command-line tool as follows (note that the `-e` flag directs the tool to look for an emulator with which to connect, rather than a physical device):

```
adb -e shell
```

Once connected, the shell environment will provide a command prompt at which commands may be entered. Begin by obtaining superuser privileges using the `su` command:

```
Generic_x86:/ su
root@android:/ #
```

If a message indicates that superuser privileges are not allowed, the AVD instance likely includes Google Play support. To resolve this, create a new AVD and, on the “Choose a device definition” screen, select a device that does not have a marker in the “Play Store” column.

The data in SQLite databases are stored in database files on the file system of the Android device on which the application is running. By default, the file system path for these database files is as follows:

```
/data/data/<package name>/databases/<database filename>.db
```

For example, if an application with the package name `com.example.MyDBApp` creates a database named `mydatabase.db`, the path to the file on the device would read as follows:

```
/data/data/com.example.MyDBApp/databases/mydatabase.db
```

For this exercise, therefore, change directory to `/data/data` within the `adb` shell and create a sub-directory hierarchy suitable for some SQLite experimentation:

```
cd /data/data
mkdir com.example.dbexample
cd com.example.dbexample
mkdir databases
cd databases
```

With a suitable location created for the database file, launch the interactive SQLite tool as follows:

```
root@android:/data/data/databases # sqlite3 ./mydatabase.db
```

An Overview of Android SQLite Databases

```
sqlite3 ./mydatabase.db
SQLite version 3.8.10.2 2015-05-20 18:17:19
Enter ".help" for usage hints.
sqlite>
```

At the *sqlite>* prompt, commands may be entered to perform tasks such as creating tables and inserting and retrieving data. For example, to create a new table in our database with fields to hold ID, name, address, and phone number fields, the following statement is required:

```
create table contacts (_id integer primary key autoincrement, name text, address
text, phone text);
```

Note that each row in a table should have a *primary key* that is unique to that row. In the above example, we have designated the ID field as the primary key, declared it as being of type *integer*, and asked SQLite to increment the number automatically each time a row is added. This is a common way to ensure that each row has a unique primary key. On most other platforms, the primary key's name choice is arbitrary. In the case of Android, however, the key must be named *_id* for the database to be fully accessible using all Android database-related classes. The remaining fields are each declared as being of type *text*.

To list the tables in the currently selected database, use the *.tables* statement:

```
sqlite> .tables
contacts
```

To insert records into the table:

```
sqlite> insert into contacts (name, address, phone) values ("Bill Smith", "123
Main Street, California", "123-555-2323");
sqlite> insert into contacts (name, address, phone) values ("Mike Parks", "10
Upping Street, Idaho", "444-444-1212");
```

To retrieve all rows from a table:

```
sqlite> select * from contacts;
1|Bill Smith|123 Main Street, California|123-555-2323
2|Mike Parks|10 Upping Street, Idaho|444-444-1212
```

To extract a row that meets specific criteria:

```
sqlite> select * from contacts where name="Mike Parks";
2|Mike Parks|10 Upping Street, Idaho|444-444-1212
```

To exit from the sqlite3 interactive environment:

```
sqlite> .exit
```

When running an Android application in the emulator environment, any database files will be created on the emulator's file system using the previously discussed path convention. This has the advantage that you can connect with adb, navigate to the location of the database file, load it into the sqlite3 interactive tool, and perform tasks on the data to identify possible problems occurring in the application code.

It is also important to note that while connecting with an adb shell to a physical Android device is possible, the shell is not granted sufficient privileges by default to create and manage SQLite databases. Therefore, database problem debugging is best performed using an AVD session.

68.9 The Android Room Persistence Library

As previously mentioned, SQLite is written in the C programming language, while Android applications are primarily developed using Java or Kotlin. To bridge this “language gap” in the past, the Android SDK included

a set of classes that provide a layer on top of the SQLite database management system. Although available in the SDK, use of these classes still involved writing a considerable amount of code and did not take advantage of the new architecture guidelines and features such as LiveData and lifecycle management. The Android Jetpack Architecture Components include the Room persistent library to address these shortcomings. This library provides a high-level interface on top of the SQLite database system, making it easy to store data locally on Android devices with minimal coding while also conforming to the recommendations for modern application architecture.

The next few chapters will provide an overview and tutorial on SQLite database management using the Room persistence library.

68.10 Summary

SQLite is a lightweight, embedded relational database management system included in the Android framework and provides a mechanism for implementing organized persistent data storage for Android applications. When combined with the Room persistence library, Android provides a modern way to implement data storage from within an Android app.

This chapter provided an overview of databases in general and SQLite in particular within the context of Android application development. The next chapters will provide an overview of the Room persistence library, after which we will work through the creation of an example application.

76. Android Audio Recording and Playback using MediaPlayer and MediaRecorder

This chapter will provide an overview of the MediaRecorder class and explain how this class can be used to record audio or video. The use of the MediaPlayer class to play back audio will also be covered. Having covered the basics, an example application will be created to demonstrate these techniques. In addition to looking at audio and video handling, this chapter will also touch on saving files to the SD card.

76.1 Playing Audio

In terms of audio playback, most implementations of Android support AAC LC/LTP, HE-AACv1 (AAC+), HE-AACv2 (enhanced AAC+), AMR-NB, AMR-WB, MP3, MIDI, Ogg Vorbis, and PCM/WAVE formats.

Audio playback can be performed using either the MediaPlayer or the AudioTrack classes. AudioTrack is a more advanced option that uses streaming audio buffers and provides greater control over the audio. The MediaPlayer class, on the other hand, provides an easier programming interface for implementing audio playback and will meet the needs of most audio requirements.

The MediaPlayer class has associated with it a range of methods that can be called by an application to perform certain tasks. A subset of some of the key methods of this class is as follows:

- **create()** – Called to create a new instance of the class, passing through the Uri of the audio to be played.
- **setDataSource()** – Sets the source from which the audio is to play.
- **prepare()** – Instructs the player to prepare to begin playback.
- **start()** – Starts the playback.
- **pause()** – Pauses the playback. Playback may be resumed via a call to the *resume()* method.
- **stop()** – Stops playback.
- **setVolume()** – Takes two floating-point arguments specifying the playback volume for the left and right channels.
- **resume()** – Resumes a previously paused playback session.
- **reset()** – Resets the state of the media player instance. Essentially sets the instance back to the uninitialized state. At a minimum, a reset player will need to have the data source set again, and the *prepare()* method called.
- **release()** – To be called when the player instance is no longer needed. This method ensures that any resources held by the player are released.

In a typical implementation, an application will instantiate an instance of the MediaPlayer class, set the source

of the audio to be played, and then call *prepare()* followed by *start()*. For example:

```
val mediaPlayer = MediaPlayer()
```

```
mediaPlayer?.setDataSource("https://www.yourcompany.com/myaudio.mp3")
```

```
mediaPlayer?.prepare()
```

```
mediaPlayer?.start()
```

76.2 Recording Audio and Video using the MediaRecorder Class

As with audio playback, recording can be performed using several different techniques. One option is to use the MediaRecorder class, which, as with the MediaPlayer class, provides several methods that are used to record audio:

- **setAudioSource()** – Specifies the audio source to be recorded (typically, this will be MediaRecorder.AudioSource.MIC for the device microphone).
- **setVideoSource()** – Specifies the source of the video to be recorded (for example MediaRecorder.VideoSource.CAMERA).
- **setOutputFormat()** – Specifies the format into which the recorded audio or video is to be stored (for example MediaRecorder.OutputFormat.AAC_ADTS).
- **setAudioEncoder()** – Specifies the audio encoder for the recorded audio (for example MediaRecorder.AudioEncoder.AAC).
- **setOutputFile()** – Configures the path to the file into which the recorded audio or video will be stored.
- **prepare()** – Prepares the MediaRecorder instance to begin recording.
- **start()** – Begins the recording process.
- **stop()** – Stops the recording process. Once a recorder has been stopped, it must be completely reconfigured and prepared before restarting.
- **reset()** – Resets the recorder. The instance will need to be completely reconfigured and prepared before being restarted.
- **release()** – Should be called when the recorder instance is no longer needed. This method ensures that all resources held by the instance are released.

A typical implementation using this class will set the source, output, encoding format, and output file. Calls will then be made to the *prepare()* and *start()* methods. The *stop()* method will then be called when the recording ends, followed by the *reset()* method. When the application no longer needs the recorder instance, a call to the *release()* method is recommended:

```
val mediaRecorder = MediaRecorder(context)
```

```
mediaRecorder?.setAudioSource(MediaRecorder.AudioSource.MIC)
```

```
mediaRecorder?.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP)
```

```
mediaRecorder?.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB)
```

```
mediaRecorder?.setOutputFile(audioFilePath)
```

```
mediaRecorder?.prepare()
```

```
mediaRecorder?.start()
```

```
.
```

```
mediaRecorder?.stop()
mediaRecorder?.reset()
mediaRecorder?.release()
```

To record audio, the manifest file for the application must include the `android.permission.RECORD_AUDIO` permission:

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

As outlined in the chapter entitled “*Making Runtime Permission Requests in Android*”, access to the microphone falls into the category of dangerous permissions. To support Android 6, therefore, a specific request for microphone access must also be made when the application launches, the steps for which will be covered later in this chapter.

76.3 About the Example Project

The remainder of this chapter will create an example application to demonstrate the use of the `MediaPlayer` and `MediaRecorder` classes to implement the recording and playback of audio on an Android device.

When developing applications that use specific hardware features, the microphone being a case in point, it is important to check the feature’s availability before attempting to access it in the application code. The application created in this chapter will, therefore, also include code to detect the presence of a microphone on the device.

Once completed, this application will provide a straightforward interface allowing the user to record and play audio. The recorded audio will be stored within an audio file on the device. That being the case, this tutorial will also briefly explore the mechanism for using SD Card storage.

76.4 Creating the AudioApp 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 *AudioApp* into the *Name* field and specify *com.ebookfrenzy.audioapp* as the package name. Before clicking on the *Finish* button, change the *Minimum API level* setting to *API 31: Android 12.0* and the *Language* menu to *Kotlin*. Add view binding support to the project using the steps outlined in section 18.8 *Migrating a Project to View Binding*.

76.5 Designing the User Interface

Once the new project has been created, select the *activity_main.xml* file from the *Project* tool window, and with the *Layout Editor* tool in *Design* mode, select the “Hello World!” *TextView* and delete it from the layout.

Drag and drop three *Button* views onto the layout. The positioning of the buttons is not paramount to this example, though Figure 76-1 shows a suggested layout using a vertical chain.

Configure the buttons to display string resources that read *Play*, *Record*, and *Stop* and give them view IDs of *playButton*, *recordButton*, and *stopButton*, respectively.

Select the *Play* button and, within the *Attributes* panel, configure the *onClick* property to call a method named *playAudio* when selected by the user. Repeat these steps to configure the remaining buttons to call methods named *recordAudio* and *stopAudio*, respectively.

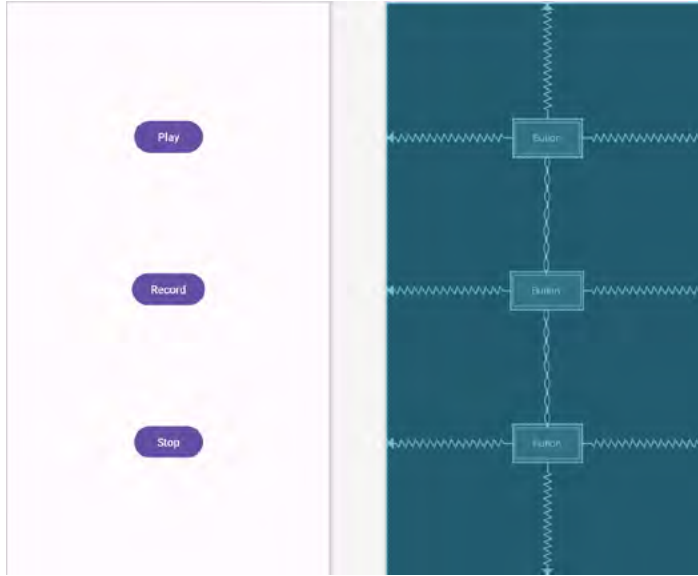


Figure 76-1

76.6 Checking for Microphone Availability

Attempting to record audio on a device without a microphone will cause the Android system to throw an exception. It is vital, therefore, that the code checks for the presence of a microphone before making such an attempt. There are several ways of doing this, including checking for the physical presence of the device. An easier approach that is more likely to work on different Android devices is to ask the Android system if it has a package installed for a particular *feature*. This involves creating an instance of the Android PackageManager class and then calling the object's *hasSystemFeature()* method. *PackageManager.FEATURE_MICROPHONE* is the feature of interest in this case.

For this example, we will create a method named *hasMicrophone()* that may be called upon to check for the presence of a microphone. Within the Project tool window, locate and double-click on the *MainActivity.kt* file and modify it to add this method:

```
package com.ebookfrenzy.audioapp
.
.
import android.content.pm.PackageManager

class MainActivity : AppCompatActivity() {
.
.
    private fun hasMicrophone(): Boolean {
        val pmanager = this.packageManager
        return pmanager.hasSystemFeature(
            PackageManager.FEATURE_MICROPHONE)
    }
}
```


76.7 Initializing the Activity

The next step is to modify the activity to perform several initialization tasks. Remaining within the *MainActivity.kt* file, modify the code as follows:

```
.
.
import android.media.MediaRecorder
import android.os.Environment
import android.view.View
import android.media.MediaPlayer

import java.io.File
.
.
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
    private var mediaRecorder: MediaRecorder? = null
    private var mediaPlayer: MediaPlayer? = null

    private var audioFilePath: String? = null
    private var isRecording = false

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

    private fun audioSetup() {

        if (!hasMicrophone()) {
            binding.stopButton.isEnabled = false
            binding.playButton.isEnabled = false
            binding.recordButton.isEnabled = false
        } else {
            binding.playButton.isEnabled = false
            binding.stopButton.isEnabled = false
        }

        val audioFile = File(this.filesDir, "myaudio.3gp")
        audioFilePath = audioFile.absolutePath
    }
.
.
```

```
}
```

The added code calls *hasMicrophone()* method to ascertain whether the device includes a microphone. If it does not, all the buttons are disabled; otherwise, only the Stop and Play buttons are disabled.

The next line of code needs a little more explanation:

```
val audioFile = File(this.filesDir, "myaudio.3gp")
audioFilePath = audioFile.absolutePath
```

This code creates a new file named *myaudio.3gp* within the app's internal storage to store the audio recording.

76.8 Implementing the recordAudio() Method

The *recordAudio()* method will be called when the user touches the Record button. This method will need to turn the appropriate buttons on and off and configure the MediaRecorder instance with information about the source of the audio, the output format and encoding, and the file's location into which the audio is to be stored. Finally, the *prepare()* and *start()* methods of the MediaRecorder object will need to be called. Combined, these requirements result in the following method implementation in the *MainActivity.kt* file:

```
fun recordAudio(view: View) {
    isRecording = true
    binding.stopButton.isEnabled = true
    binding.playButton.isEnabled = false
    binding.recordButton.isEnabled = false

    try {
        mediaRecorder = MediaRecorder(this)
        mediaRecorder?.setAudioSource(MediaRecorder.AudioSource.MIC)
        mediaRecorder?.setOutputFormat(
            MediaRecorder.OutputFormat.THREE_GPP)
        mediaRecorder?.setOutputFile(audioFilePath)
        mediaRecorder?.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB)
        mediaRecorder?.prepare()
    } catch (e: Exception) {
        e.printStackTrace()
    }
    mediaRecorder?.start()
}
```

76.9 Implementing the stopAudio() Method

The *stopAudio()* method enables the Play button, turning off the Stop button, and then stopping and resetting the MediaRecorder instance. The code to achieve this reads as outlined in the following listing and should be added to the *MainActivity.kt* file:

```
fun stopAudio(view: View) {

    binding.stopButton.isEnabled = false
    binding.playButton.isEnabled = true

    if (isRecording) {
        binding.recordButton.isEnabled = false
```

```

        mediaRecorder?.stop()
        mediaRecorder?.release()
        mediaRecorder = null
        isRecording = false
    } else {
        mediaPlayer?.release()
        mediaPlayer = null
        binding.recordButton.isEnabled = true
    }
}

```

76.10 Implementing the playAudio() method

The *playAudio()* method will create a new MediaPlayer instance, assign the audio file located on the SD card as the data source and then prepare and start the playback:

```

fun playAudio(view: View) {
    binding.playButton.isEnabled = false
    binding.recordButton.isEnabled = false
    binding.stopButton.isEnabled = true

    mediaPlayer = MediaPlayer()
    mediaPlayer?.setDataSource(audioFilePath)
    mediaPlayer?.prepare()
    mediaPlayer?.start()
}

```

76.11 Configuring and Requesting Permissions

Before testing the application, the appropriate permissions must be requested within the manifest file for the application. Specifically, the application will require permission to access the microphone. Within the Project tool window, locate and double-click on the *AndroidManifest.xml* file to load it into the editor and modify the XML to add the permission tags:

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.RECORD_AUDIO" />

    <application
.
.

```

The above steps will be adequate to ensure that the user enables microphone access permission when the app is installed on devices running versions of Android predating Android 6.0. Microphone access is categorized in Android as being a dangerous permission because it allows the app to compromise the user's privacy. For the example app to function on Android 6 or later devices, code needs to be added to request permission at app runtime.

Edit the *MainActivity.kt* file and begin by adding some additional import directives and a constant to act as request identification codes for the permissions being requested:

```
.
.
import android.Manifest
import android.widget.Toast
import androidx.core.app.ActivityCompat
import androidx.core.content.ContextCompat
.
.
class MainActivity : AppCompatActivity() {
.
.
    private val RECORD_REQUEST_CODE = 101
.
.
}
```

Next, a method needs to be added to the class, the purpose of which is to take as arguments the permission to be requested and the corresponding request identification code. Remaining with the *MainActivity.kt* class file, implement this method as follows:

```
private fun requestPermission(permissionType: String, requestCode: Int) {
    val permission = ContextCompat.checkSelfPermission(this,
        permissionType)

    if (permission != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
            arrayOf(permissionType), requestCode
        )
    }
}
```

Using the steps outlined in the “*Making Runtime Permission Requests in Android*” chapter of this book, the above method verifies that the specified permission has not already been granted before making the request, passing through the identification code as an argument.

When the request has been handled, the *onRequestPermissionsResult()* method will be called on the activity, passing through the identification code and the request results. The next step, therefore, is to implement this method within the *MainActivity.kt* file as follows:

```
override fun onRequestPermissionsResult(requestCode: Int,
    permissions: Array<String>, grantResults: IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)

    when (requestCode) {
        RECORD_REQUEST_CODE -> {
            if (grantResults.isEmpty() || grantResults[0]
                != PackageManager.PERMISSION_GRANTED
            ) {

                binding.recordButton.isEnabled = false
            }
        }
    }
}
```

```

        Toast.makeText (
            this,
            "Record permission required",
            Toast.LENGTH_LONG
        ).show()
    }
}
}
}

```

The above code checks the request identifier code to identify which permission request has returned before checking whether or not the corresponding permission was granted. If permission is denied, a message is displayed to the user indicating that the app will not function and the record button is disabled.

Before testing the app, all that remains is to call the newly added *requestPermission()* method for microphone access when the app launches. Remaining in the *MainActivity.kt* file, modify the *audioSetup()* method as follows:

```

private fun audioSetup() {
    .
    .

    audioFilePath = audioFile.absolutePath

    requestPermission(Manifest.permission.RECORD_AUDIO,
        RECORD_REQUEST_CODE)
}

```

76.12 Testing the Application

Compile and run the application on an Android device containing a microphone, allow microphone access, and tap the Record button. After recording, touch Stop followed by Play. At this point, the recorded audio should play back through the device speakers.

76.13 Summary

The Android SDK provides several mechanisms to implement audio recording and playback. This chapter has looked at two of these: the MediaPlayer and MediaRecorder classes. Having covered the theory of using these techniques, this chapter worked through creating an example application designed to record and then play back audio. While working with audio in Android, this chapter also looked at the steps involved in ensuring that the device on which the application is running has a microphone before attempting to record audio.

89. Working with Material Design 3 Theming

The appearance of an Android app is intended to conform to a set of guidelines defined by Material Design. Google developed Material Design to provide a level of design consistency between different apps while also allowing app developers to include their own branding in terms of color, typography, and shape choices (a concept referred to as Material theming). In addition to design guidelines, Material Design also includes a set of UI components for use when designing user interface layouts, many of which we have used throughout this book.

This chapter will provide an overview of how theming works within an Android Studio project and explore how the default design configurations provided for newly created projects can be modified to meet your branding requirements.

89.1 Material Design 2 vs. Material Design 3

Before beginning, it is important to note that Google is transitioning from Material Design 2 to Material Design 3 and that Android Studio Hedgehog projects default to Material Design 3. Material Design 3 provides the basis for Material You, a feature introduced in Android 12 that allows an app to automatically adjust theme elements to complement preferences configured by the user on the device. For example, dynamic color support provided by Material Design 3 allows the colors used in apps to adapt automatically to match the user's wallpaper selection.

89.2 Understanding Material Design Theming

We know that Android app user interfaces are created by assembling components such as layouts, text fields, and buttons. These components appear using default colors unless we specifically override a color attribute in the XML layout resource file or by writing code. The project's theme defines these default colors. The theme consists of a set of color slots (declared in *themes.xml* files) which are assigned color values (declared in the *colors.xml* file). Each UI component is programmed internally to use theme color slots as the default color for specific attributes (such as the foreground and background colors of the Text widget). It follows, therefore, that we can change the application-wide theme of an app by changing the colors assigned to specific theme slots. When the app runs, the new default colors will be used for all widgets when the user interface is rendered.

89.3 Material Design 3 Theming

Before exploring Material Design 3, we must consider how it is used in an Android Studio project. The theme used by an application project is declared as a property of the *application* element within the *AndroidManifest.xml* file, for example:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
```

Working with Material Design 3 Theming

```
    android:supportsRtl="true"
    android:theme="@style/Theme.MyDemoApp"
    tools:targetApi="31">
    <activity
```

.

.

As previously discussed, all of the files associated with the project theme are contained within the *colors.xml* and *themes.xml* files located in the *res -> values* folder, as shown in Figure 89-1:

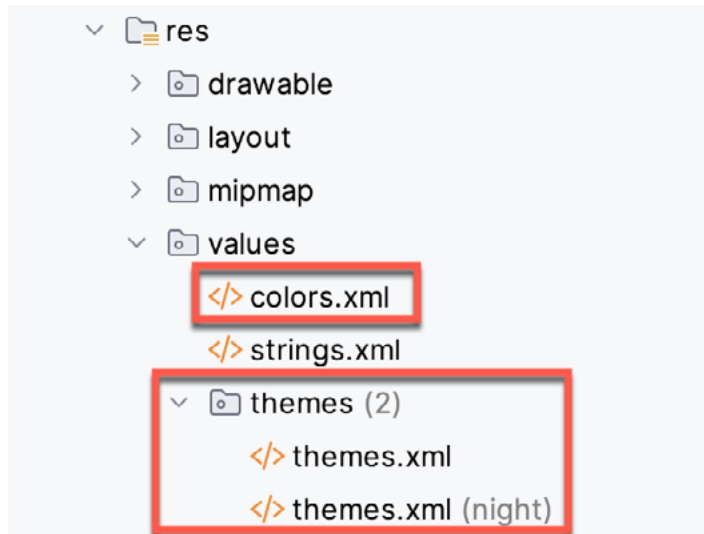


Figure 89-1

The theme itself is declared in the two *themes.xml* files located in the *themes* folder. These resource files declare different color palettes containing Material Theme color slots for use when the device is in light or dark (night) mode. Note that the style name property in each file must match that referenced in the *AndroidManifest.xml* file, for example:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Base.Theme.MyDemoApp" parent="Theme.Material3.DayNight.
NoActionBar">
        <!-- Customize your light theme here. -->
        <!-- <item name="colorPrimary">@color/my_light_primary</item> -->
    </style>

    <style name="Theme.MyDemoApp" parent="Base.Theme.MyDemoApp" />
</resources>
```

These color slots (also referred to as *color attributes*) are used by the Material components to set colors when they are rendered on the screen. For example, the *colorPrimary* color slot is used as the background color for the Material Button component.

Color slots in MD3 are grouped as Primary, Secondary, Tertiary, Error, Background, and Surface. These slots are further divided into pairs consisting of a *base color* and an “on” *base color*. This generally translates to the background and foreground colors of a Material component.

The particular group used for coloring will differ between widgets. A Material Button widget, for example, will use the *colorPrimary* base color for the background color and *colorOnPrimary* for its content (i.e., the text or icon it displays). The FloatingActionButton component, on the other hand, uses *colorPrimaryContainer* as the background color and *colorOnPrimaryContainer* for the foreground. The correct group for a specific widget type can usually be identified quickly by changing color settings in the theme files and reviewing the rendering in the layout editor.

Suppose that we need to change *colorPrimary* to red. We achieve this by adding a new entry to the *colors.xml* file for the red color and then assigning it to the *colorPrimary* slot in the *themes.xml* file. The *colorPrimary* slot in an MD3 theme night, therefore, read as follows:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Base.Theme.MyDemoApp" parent="Theme.Material3.DayNight.NoActionBar">
        <item name="colorPrimary">@color/my_bright_primary</item>
    </style>

    <style name="Theme.MyDemoApp" parent="Base.Theme.MyDemoApp" />
</resources>
```

This color is then declared in the *colors.xml* file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    .
    .
    <color name="my_bright_primary">#FC0505</color>
</resources>
```

89.4 Building a Custom Theme

As we have seen, the coding work in implementing a theme is relatively simple. The difficult part, however, is often choosing complementary colors to make up the theme. Fortunately, Google has developed a tool that makes it easy to design custom color themes for your apps. This tool is called the Material Theme Builder and is available at:

<https://m3.material.io/theme-builder#/custom>

On the custom screen (Figure 89-2), make a color selection for the primary color key (A) by clicking on the color circle to display the color selection dialog. Once a color has been selected, the preview (B) will change to reflect the recommended colors for all MD3 color slots, along with example app interfaces and widgets. In addition, you can override the generated colors for the Secondary, Tertiary, and Neutral slots by clicking on the corresponding color circles to display the color selection dialog.

The area marked B displays example app interfaces, light and dark color scheme charts, and widgets that update to preview your color selections. Since the panel is longer than the typical browser window, you must scroll down to see all the information.

To incorporate the theme into your design, click the Export button (C) and select the Android View (XML) option. Once downloaded, the *colors.xml* and *themes.xml* files can be used to replace the existing files in your project. Note that the theme name in the two exported *themes.xml* files must be changed to match your project.

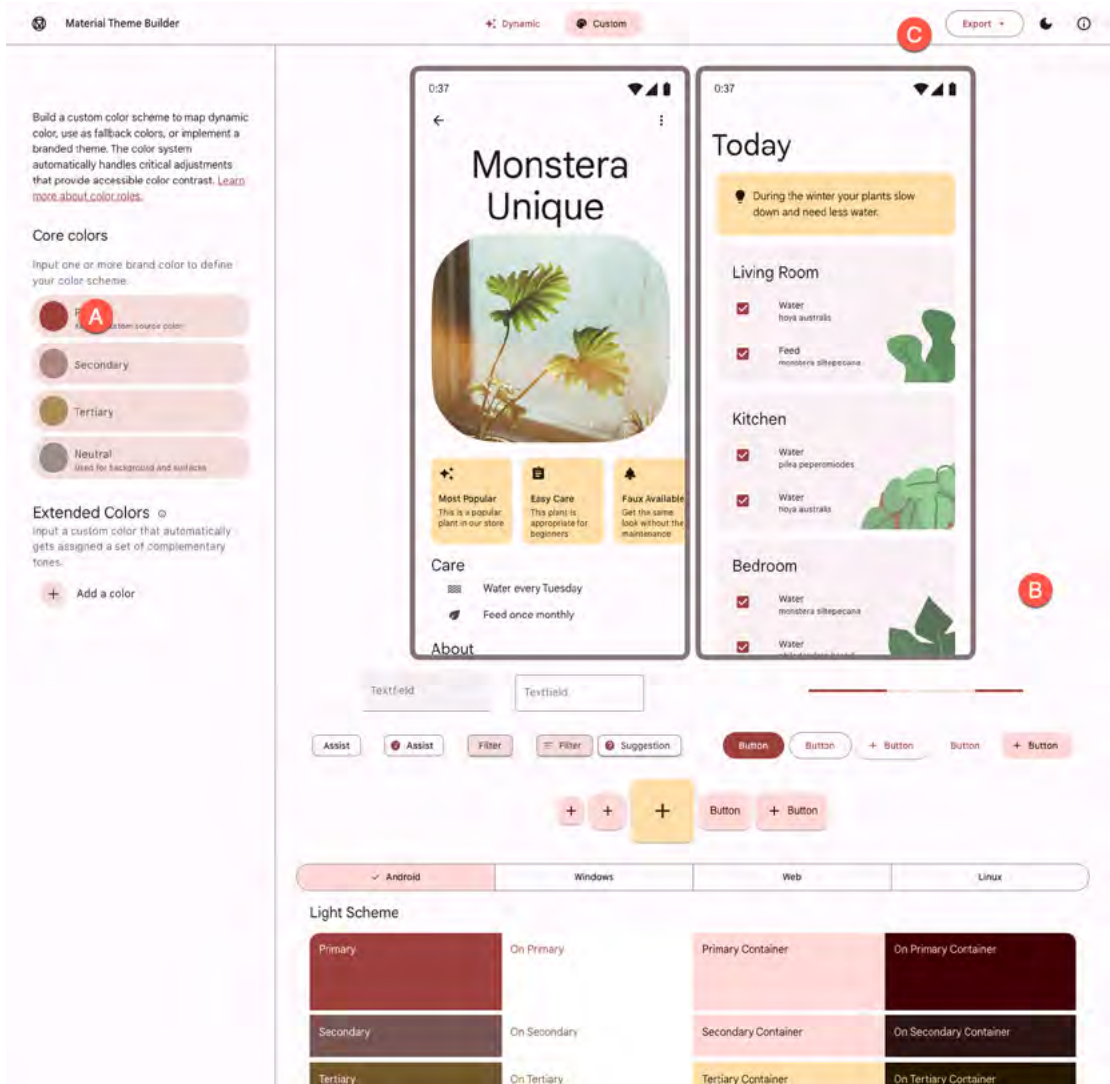


Figure 89-2

89.5 Summary

Material Design provides guidelines and components defining how Android apps appear. Individual branding can be applied to an app by designing themes that specify the colors, fonts, and shapes used when displaying the app. Google recently introduced Material Design 3, which replaces Material Design 2 and supports the new features of Material You, including dynamic colors. Google also provides the Material Theme Builder for designing your own themes, which eases the task of choosing complementary theme colors. Once this tool has been used to design a theme, the corresponding files can be exported and used within an Android Studio project.

Index

Symbols

?. 101
 <application> 502
 <fragment> 293
 <fragment> element 293
 <receiver> 480
 <service> 502, 508, 515
 :: operator 103
 .well-known folder 453, 476, 692

A

AbsoluteLayout 174
 ACCESS_COARSE_LOCATION permission 602
 ACCESS_FINE_LOCATION permission 602
 acknowledgePurchase() method 731
 ACTION_CREATE_DOCUMENT 765
 ACTION_CREATE_INTENT 766
 ACTION_DOWN 270
 ACTION_MOVE 270
 ACTION_OPEN_DOCUMENT intent 758
 ACTION_POINTER_DOWN 270
 ACTION_POINTER_UP 270
 ACTION_UP 270
 ACTION_VIEW 471
 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 450
 state change example 161

 state changes 153
 states 150
 Visible Lifetime 157
 Activity Lifecycle 149
 Activity Manager 86
 ActivityResultLauncher 451
 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 479
 addMarker() method 655
 addView() method 245
 ADD_VOICEMAIL permission 602
 android
 exported 503
 gestureColor 286
 layout_behavior property 443
 onClick 295
 process 503, 515
 uncertainGestureColor 286
 Android
 Activity 87
 architecture 83
 events 263
 intents 88

Index

- onClick Resource 263
- runtime 84
- SDK Packages 6
- android.app 84
- Android Architecture Components 309
- android.content 84
- android.content.Intent 449
- 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 485
- android.intent.action.BOOT_COMPLETED 503
- android.intent.action.MAIN 471
- android.intent.category.LAUNCHER 471
- 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 611
- 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 Storage Access Framework 758
- 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 780
- API Key 647
- APK analyzer 724
- APK file 717
- APK File
 - analyzing 724
- APK Signing 780
- APK Wizard dialog 716
- App Architecture
 - modern 309
- AppBar
 - anatomy of 441
- appbar_scrolling_view_behavior 443

- App Bundles 713
 - creating 717
 - overview 713
 - revisions 723
 - uploading 720
- 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 700
 - Digital Asset Links file 692, 453
 - Intent Filter Handling 700
 - Intent Filters 691
 - Intent Handling 692
 - Testing 704
 - URL Mapping 697
- App Links 691
 - auto verification 452
 - autoVerify 453
 - overview 691
- 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 753
- Architecture Components 309
- ART 84
- as 103
- as? 103
- asFlow() builder 520
- assetlinks.json , 692, 453
- asSharedFlow() 530
- asStateFlow() 529
- async 489

- Attribute Keyframes 380
- Audio
 - supported formats 609
- Audio Playback 609
- Audio Recording 609
- Autoconnect Mode 207
- Automatic Link Verification 452, 475
- autoVerify 453, 700
- AVD
 - cold boot 48
 - command-line creation 31
 - creation 31
 - device frame 40
 - Display mode 50
 - 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 732
 - acknowledgePurchase() method 731
 - consumeAsync() method 731
 - getPurchaseState() method 730
 - initialization 728, 736
 - launchBillingFlow() method 730
 - queryProductDetailsAsync() method 729
 - queryPurchasesAsync() method 732
- BillingResult 743
 - getDebugMessage() 743
- Binding Expressions 329

Index

- one-way 329
- two-way 330
- BIND_JOB_SERVICE permission 503
- bindService() method 501, 505, 509
- Biometric Authentication 705
 - callbacks 709
 - overview 705
 - tutorial 705
- Biometric Prompt 710
- BitmapFactory 760
- Bitwise AND 109
- Bitwise Inversion 108
- Bitwise Left Shift 110
- Bitwise OR 109
- Bitwise Right Shift 110
- Bitwise XOR 109
- black activity 16
- Blank template 177
- Blueprint view 205
- BODY_SENSORS permission 602
- Boolean 96
- Bound Service 501, 505
 - adding to a project 506
 - Implementing the Binder 506
 - Interaction options 505
- BoundService class 507
- Broadcast Intent 479
 - example 481
 - overview 88, 479
 - sending 482
 - Sticky 481
- Broadcast Receiver 479
 - adding to manifest file 484
 - creation 483
 - overview 88, 480
- BroadcastReceiver class 480
- BroadcastReceiver superclass 483
- BufferedReader object 768
- buffer() operator 523
- Build tool window 58
- Build Variants , 58
 - tool window 58

- Bundle class 170
- Bundled Notifications 630

C

- Calendar permissions 602
- CALL_PHONE permission 602
- CAMERA permission 602
- Camera permissions 602
- CameraUpdateFactory class
 - methods 656
- cancelAndJoin() 489
- cancelChildren() 489
- CancellationSignal 710
- Canvas class 686
- CardView
 - layout file 431
 - responding to selection of 439
- CardView class 431
- CATEGORY_OPENABLE 758
- C/C++ Libraries 85
- Chain bias 228
- chain head 198
- chains 198
- Chains
 - creation of 225
- Chain style
 - changing 227
- chain styles 198
- Char 96
- CheckBox 173
- checkSelfPermission() method 606
- Circle class 643
- Code completion 74
- Code 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 76
- Code Reformatting 79
- code samples
 - download 1
- cold boot 48
- Cold flows 528
- CollapsingToolBarLayout
 - example 444
 - introduction 444
 - parallax mode 444
 - pin mode 444
 - setting scrim color 447
 - setting title 447
 - with image 444
- collectLatest() operator 522
- Color class 687
- COLOR_MODE_COLOR 662, 682
- COLOR_MODE_MONOCHROME 662, 682
- combine() operator 527
- Common Gestures 275
 - detection 275
- Communicating Sequential Processes 487
- Companion Objects 133
- Component tree 20
- conflate() operator 522
- 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

Index

- createHorizontalChain() method 247
- createVerticalChain() method 247
- guidelines 248
- removeFromHorizontalChain() method 248
- removeFromVerticalChain() method 248
- removing constraints 248
- rotation 249
- 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 731
- ConsumeParams 741
- Contacts permissions 602
- container view 173
- Content Provider 86
 - overview 89
- Context class 89
- CoordinatorLayout 174, 443
- Coroutine Builders 489
 - async 489
 - coroutineScope 489
 - launch 489
 - runBlocking 489
 - supervisorScope 489
 - withContext 489
- Coroutine Dispatchers 488
- Coroutines 487, 519
 - adding libraries 495
 - channel communication 493
 - GlobalScope 488
 - returning results 491
 - Suspend Functions 488
 - suspending 490
 - tutorial 495
 - ViewModelScope 488
 - vs. Threads 487
- coroutineScope 489
- Coroutine Scope 488
- createPrintDocumentAdapter() method 677
- Custom Accessors 131
- Custom Attribute 377
- Custom Document Printing 665, 677
- Custom Gesture
 - recognition 281
- Custom Print Adapter
 - implementation 679
- Custom Print Adapters 677
- Custom Theme
 - building 747
- Cycle Editor 405
- Cycle Keyframe 385
- Cycle Keyframes
 - overview 401

D

- dangerous permissions
 - list of 602
- Dark Theme 36
 - enable on device 36
- Data Access Object (DAO) 550
- Database Inspector 556, 580
 - live updates 580
 - SQL query 580
- Database Rows 544
- Database Schema 543
- Database Tables 543
- 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 453, 475
- 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 692, 453, 453
- Direct Reply Input 639
- Dispatchers.Default 489
- Dispatchers.IO 488
- Dispatchers.Main 488
- document provider 757
- dp 241
- DROP_LATEST 530
- DROP_OLDEST 530
- Dynamic Colors
 - applyToActivitiesIfAvailable() method 753
 - enabling in Android 753
- 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 49
 - Virtual Sensors 47
 - zoom 44
- enablePendingPurchases() method 731
- enabling ADB support 63
- Escape Sequences 97
- ettings.gradle file 780
- Event Handling 263
 - example 264
- Event Listener 265
- Event Listeners 264
- Events
 - consuming 267
- explicit
 - intent 88
- explicit intent 449
- Explicit Intent 449
- Extended Control
 - options 45

Index

F

Files

- switching between 72

filter() operator 524

findPointerIndex() method 270

findViewById() 143

Fingerprint

- emulation 48

Fingerprint authentication

- device configuration 706

- permission 706

- steps to implement 705

Fingerprint Authentication

- overview 705

- tutorial 705

FLAG_INCLUDE_STOPPED_PACKAGES 479

flatMapConcat() operator 527

flatMapMerge() operator 527

flexible space area 441

Float 96

floating action button 16, 178

- changing appearance of 416

- margins 414

- removing 179

- sizes 414

Flow 519

- asFlow() builder 520

- asSharedFlow() 530

- asStateFlow() 529

- backgroundn handling 538

- buffering 522

- buffer() operator 523

- cold 528

- collect() 521

- collecting data 521

- collectLatest() operator 522

- combine() operator 527

- conflate() operator 522

- declaring 520

- emit() 521

- emitting data 521

- filter() operator 524

- flatMapConcat() operator 527

- flatMapMerge() operator 527

- flattening 526

- flowOf() builder 520

- flow of flows 526

- fold() operator 526

- hot 528

- intermediate operators 524

- library requirements 520

- map() operator 524

- MutableSharedFlow 530

- MutableStateFlow 529

- onEach() operator 528

- reduce() operator 525, 526

- repeatOnLifecycle 540

- SharedFlow 530

- single() operator 522

- StateFlow 529

- terminal flow operators 525

- transform() operator 525

- try/finally 522

- zip() operator 527

flowOf() builder 520

flow of flows 526

Flow operators 524

Flows

- combining 527

- Introduction to 519

Foldable Devices 158

- multi-resume 158

Foreground Process 148

Forward-geocoding 649

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 650
- Geocoding 648
- 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 602
- getAction() method 485
- getDebugMessage() 743
- getFromLocation() method 650
- getId() method 246
- getIntent() method 450
- getPointerCount() method 270
- getPointerId() method 270
- getPurchaseState() method 730
- getService() method 509
- GlobalScope 488
- GNU/Linux 84

- Google Cloud
 - billing account 644
 - new project 645
- Google Cloud Print 660
- Google Drive 758
 - printing to 660
- GoogleMap 643
 - map types 653
- GoogleMap.MAP_TYPE_HYBRID 653
- GoogleMap.MAP_TYPE_NONE 653
- GoogleMap.MAP_TYPE_NORMAL 653
- GoogleMap.MAP_TYPE_SATELLITE 653
- GoogleMap.MAP_TYPE_TERRAIN 653
- Google Maps Android API 643
 - Controlling the Map Camera 656
 - displaying controls 654
 - Map Markers 655
 - overview 643
- Google Maps SDK 643
 - API Key 647
 - Credentials 647
 - enabling 646
 - Maps SDK for Android 647
- Google Play App Signing 716
- Google Play Console 734
 - Creating an in-app product 734
 - License Testers 735
- Google Play Developer Console 714
- Gradle
 - APK signing settings 784
 - Build Variants 780
 - command line tasks 785
 - dependencies 779
 - Manifest Entries 780
 - overview 779
 - sensible defaults 779
- Gradle Build File
 - top level 781
- Gradle Build Files
 - module level 782
- gradle.properties file 780
- GridLayout 174

Index

GridLayoutManager 429

H

Handler class 514

Higher-order Functions 125

Hot flows 528

HP Print Services Plugin 659

HTML printing 663

HTML Printing

 example 667

I

IBinder 501, 507

IBinder object 505, 514

Image Printing 662

Immutable Variables 98

implicit

 intent 88

implicit intent 449

Implicit Intent 451

Implicit Intents

 example 467

importance hierarchy 147

in 241

INAPP 732

In-App Products 727

In-App Purchasing 733

 acknowledgePurchase() method 731

 BillingClient 728

 BillingResult 743

 consumeAsync() method 731

 ConsumeParams 741

 Consuming purchases 740

 enablePendingPurchases() method 731

 getPurchaseState() method 730

 launchBillingFlow() method 730

 Libraries 733

 newBuilder() method 728

 onBillingServiceDisconnected() callback 737

 onBillingServiceDisconnected() method 729

 onBillingSetupFinished() listener 737

 onProductDetailsResponse() callback 738

Overview 727

ProductDetail 730

ProductDetails 738

products 727

ProductType 732

Purchase Flow 739

PurchaseResponseListener 732

PurchasesUpdatedListener 730

PurchaseUpdatedListener 739

purchase updates 739

queryProductDetailsAsync() 738

queryProductDetailsAsync() method 729

queryPurchasesAsync() 741

queryPurchasesAsync() method 732

runOnUiThread() 739

subscriptions 727

tutorial 733

Initializer Blocks 131

In-Memory Database 556

Inner Classes 132

IntelliJ IDEA 91

Intent 88

 explicit 88

 implicit 88

Intent Availability

 checking for 456

Intent.CATEGORY_OPENABLE 766

Intent Filters 452

 App Link 691

Intents 449

 ActivityResultLauncher 451

 overview 449

 registerForActivityResult() 451, 464

Intent Service 501

Intent URL 470

intermediate flow operators 524

is 103

isInitialized property 103

J

Java

 convert to Kotlin 91

- Java Native Interface 85
- JetBrains 91
- Jetpack 309
 - overview 309
- JobIntentService 501
 - BIND_JOB_SERVICE permission 503
 - onHandleWork() method 501
- join() 489

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 716
- KeyTimeCycle 401
- keytool 453
- 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 519
 - 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

Index

- 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
- 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 489
- launchBillingFlow() method 730
- layout_collapseMode
 - parallax 446
 - pin 446
- layout_constraintDimentionRatio 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 185
 - 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 683
- Layouts 173
- layout_scrollFlags
 - enterAlwaysCollapsed mode 443
 - enterAlways mode 443
 - exitUntilCollapsed mode 443
 - scroll mode 443
- Layout Validation 192
- let Function 101
- libc 85
- License Testers 735
- Lifecycle
 - awareness 345
 - components 312
 - observers 346
 - owners 345
 - states and events 346
 - tutorial 349
- Lifecycle-Aware Components 345
- Lifecycle library 520
- Lifecycle Methods 155
- Lifecycle Observer 349
 - creating a 349
- Lifecycle Owner
 - creating a 351
- Lifecycles
 - modern 312
- Lifecycle.State.CREATED 540
- Lifecycle.State.DESTROYED 540
- Lifecycle.State.INITIALIZED 540
- Lifecycle.State.RESUMED 540
- Lifecycle.State.STARTED 540
- LinearLayout 174
- LinearLayoutManager 429
- LinearLayoutManager layout 437

- Linux Kernel 84
- list devices 63
- LiveData 310, 321
 - adding to ViewModel 321
 - observer 323
 - tutorial 321
- Live Templates 80
- Local Bound Service 505
 - example 505
- Local Functions 122
- Location Manager 86
- Location permission 602
- Logcat
 - tool window 57
- LogCat
 - enabling 165
- M**
- MANAGE_EXTERNAL_STORAGE 603
 - adb enabling 603
 - testing 603
- Manifest File
 - permissions 471
- map() operator 524
- Maps 643
- MapView 643
 - adding to a layout 650
- Marker class 643
- Master/Detail Flow
 - creation 772
 - two pane mode 771
- match_parent properties 241
- Material design 413
- Material Design 2 745
- Material Design 2 Theming 745
- Material Design 3 745
- Material Theme Builder 747
- Material You 745
- measureTimeMillis() function 523
- MediaController
 - adding to VideoView instance 587
- MediaController class 584
 - methods 584
- MediaPlayer class 609
 - methods 609
- MediaRecorder class 609
 - methods 610
 - recording audio 610
- Memory Indicator 73
- Messenger object 514
- Microphone
 - checking for availability 612
- Microphone permissions 602
- 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

Index

moveCamera() method 656

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 530

MutableStateFlow 529

Mutable Variables 98

My Location Layer 643

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 728

normal permissions 601

Notification

 adding actions 630

 Direct Reply Input 639

 issuing a basic 626

 launch activity from a 628

 PendingIntent 636

 Reply Action 638

 updating direct reply 640

Notifications

 bundled 630

 overview 619

Notifications Manager 86

Not-Null Assertion 101

Nullable Type 100

O

Observer

 implementing a LiveData 323

onAttach() method 296

onBillingServiceDisconnected() callback 737

onBillingServiceDisconnected() method 729

onBillingSetupFinished() listener 737

onBind() method 502, 505

onBindViewHolder() method 437

OnClick 379

onClickListener 264, 265, 268

onClick() method 263

onCreateContextMenuListener 264

onCreate() method 148, 155, 502

onCreateView() method 156

onDestroy() method 156, 502

onDoubleTap() method 275

onDown() method 275

onEach() operator 528

onFling() method 275

onFocusChangeListener 264

OnFragmentInteractionListener

 implementation 369

onGesturePerformed() method 281

onHandleWork() method 502

onKeyListener 264

onLayoutFailed() method 683

onLayoutFinished() method 683

onLongClickListener 264

onLongPress() method 275

onMapReady() method 652

onPageFinished() callback 668

- onPause() method 156
- onProductDetailsResponse() callback 738
- onReceive() method 148, 480, 481, 483
- onRequestPermissionsResult() method 605, 616, 624, 634
- 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 505, 508, 515
- onServiceDisconnected() method 505, 508, 515
- onShowPress() method 275
- onSingleTapUp() method 275
- onStartCommand() method 502
- onStart() method 155
- onStop() method 156
- onTouchEvent() method 275, 287
- onTouchListener 264
- onTouch() method 270
- onViewCreated() method 156
- onViewStatusRestored() method 156
- openFileDescriptor() method 758
- OpenJDK 3

P

- Package Explorer 18
- Package Manager 86
- PackageManager class 612
- PackageManager.FEATURE_MICROPHONE 612
- PackageManager.PERMISSION_DENIED 603
- PackageManager.PERMISSION_GRANTED 603
- Package Name 16
- Packed chain 199, 228
- PageRange 684, 685
- Paint class 687
- parentRelative 381
- parent view 175
- pathRelative 382

- Paused state 150
- PdfDocument 665
- PdfDocument.Page 677, 684
- PendingIntent class 636
- Permission
 - checking for 603
- permissions
 - normal 601
- Persistent State 155
- Phone permissions 602
- picker 757
- Pinch Gesture
 - detection 287
 - example 287
- Pinch Gesture Recognition 281
- Position Keyframes 381
- POST_NOTIFICATIONS permission 602, 634
- Primary Constructor 128
- PrintAttributes 682
- PrintDocumentAdapter 665, 677
- Printing
 - color 662
 - monochrome 662
- Printing framework
 - architecture 659
- Printing Framework 659
- Print Job
 - starting 688
- PrintManager service 669
- Problems
 - tool window 58
- process
 - priority 147
 - state 147
- PROCESS_OUTGOING_CALLS permission 602
- Process States 147
- ProductDetail 730
- ProductDetails 738
- ProductType 732
- Profiler
 - tool window 58
- ProgressBar 173

Index

proguard-rules.pro file 784
ProGuard Support 780
Project Name 16
Project tool window 18, 57
pt 241
PurchaseResponseListener 732
PurchasesUpdatedListener 730
PurchaseUpdatedListener 739
putExtra() method 449, 479
px 242

Q

queryProductDetailsAsync() 738
queryProductDetailsAsync() method 729
queryPurchaseHistoryAsync() method 732
queryPurchasesAsync() 741
queryPurchasesAsync() method 732
quickboot snapshot 48
Quick Documentation 79

R

RadioButton 173
Range Operator 108
ratios 229
READ_CALENDAR permission 602
READ_CALL_LOG permission 602
READ_CONTACTS permission 602
READ_EXTERNAL_STORAGE permission 603
READ_PHONE_STATE permission 602
READ_SMS permission 602
RECEIVE_MMS permission 602
RECEIVE_SMS permission 602
RECEIVE_WAP_PUSH permission 602
Recent Files Navigation 60
RECORD_AUDIO permission 602
Recording Audio
 permission 611
RecyclerView 429
 adding to layout file 430
 LayoutManager 429
 initializing 437
 LinearLayoutManager 429

 StaggeredLayoutManager 429
RecyclerView Adapter
 creation of 435
RecyclerView.Adapter 430, 436
 getItemCount() method 430
 onBindViewHolder() method 430
 onCreateViewHolder() method 430
RecyclerView.ViewHolder
 getAdapterPosition() method 440
reduce() operator 525, 526
registerForActivityResult() 451
registerForActivityResult() method 450, 464
registerReceiver() method 481
RelativeLayout 174
releasePersistableUriPermission() method 761
Release Preparation 713
Remote Bound Service 513
 client communication 513
 implementation 513
 manifest file declaration 515
RemoteInput.Builder() method 636
RemoteInput Object 636
Remote Service
 launching and binding 515
 sending a message 517
repeatOnLifecycle 540
Repository
 tutorial 567
Repository Modules 312
Resizable Emulator 50
Resource
 string creation 23
Resource File 25
Resource Management 147
Resource Manager , 57
result receiver 481
Reverse-geocoding 649
Reverse Geocoding 648
Room
 Data Access Object (DAO) 550
 entities 550, 551
 In-Memory Database 556

- Repository 550
- Room Database 550
 - tutorial 567
- Room Database Persistence 549
- Room Persistence Library 546, 549
- root element 173
- root view 175
- Run
 - tool window 57
- runBlocking 489
- Running Devices
 - tool window 69
- runOnUiThread() 739

S

- safeargs 371
- 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 479, 481
- sendOrderedBroadcast() method 479, 481
- SEND_SMS permission 602
- sendStickyBroadcast() method 479
- Sensor permissions 602
- Service
 - anatomy 502
 - launch at system start 503
 - manifest file entry 502
 - overview 88
 - run in separate process 503
- ServiceConnection class 515
- Service Process 148
- Service Restart Options 502
- setAudioEncoder() method 610
- setAudioSource() method 610
- setBackgroundColor() 246
- setCompassEnabled() method 654
- setContentView() method 245, 251
- setId() method 246
- setMyLocationButtonEnabled() method 654
- setOnClickListener() method 263, 265
- setOnDoubleTapListener() method 275, 278
- setOutputFile() method 610
- setOutputFormat() method 610
- setResult() method 451
- setText() method 172
- settings.gradle.kts file 780
- setTransition() 385
- setVideoSource() method 610
- SHA-256 certificate fingerprint 453
- SharedFlow 530, 533
 - backgroundn handling 538
 - DROP_LATEST 530
 - DROP_OLDEST 530
 - in ViewModel 535
 - repeatOnLifecycle 540
 - SUSPEND 531
 - tutorial 533
- shouldOverrideUrlLoading() method 668
- SimpleOnScaleGestureListener 287
- SimpleOnScaleGestureListener class 288
- single() operator 522
- SMS permissions 602
- Snackbar 413, 414, 415
- Snapshots
 - emulator 47
- sp 241
- Spread chain 198
- Spread inside 228
- Spread inside chain 198
- SQL 544

Index

SQLite 543

- AVD command-line use 545

- Columns and Data Types 543

- overview 544

- Primary keys 544

StaggeredGridLayoutManager 429

startActivity() method 449

startForeground() method 148

START_NOT_STICKY 502

START_REDELIVER_INTENT 502

START_STICKY 502

State

- restoring 172

State Change

- handling 151

StateFlow 529

Statement Completion 76

Status Bar Widgets 73

- Memory Indicator 73

Sticky Broadcast Intents 481

Stopped state 150

Storage Access Framework 757

- ACTION_CREATE_DOCUMENT 758

- ACTION_OPEN_DOCUMENT 758

- deleting a file 761

- example 763

- file creation 765

- file filtering 758

- file reading 759

- file writing 760

- intents 758

- MIME Types 759

- Persistent Access 761

- picker 757

Storage permissions 603

String 96

StringBuilder object 768

strings.xml file 27

Structure

- tool window 58

Structured Query Language 544

Structure tool window 58

SUBS 732

subscriptions 727

supervisorScope 489

SupportMapFragment class 643

SUSPEND 531

Suspend Functions 488

Switcher 60

System Broadcasts 485

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, 559

TableRow 559

Telephony Manager 86

Templates

- blank vs. empty 177

Terminal

- tool window 58

terminal flow operators 525

Theme

- building a custom 747

Theming 745

- tutorial 749

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 525
- try/finally 522
- Type Annotations 99
- Type Casting 103
- Type Checking 103
- Type Inference 99

U

- UiSettings class 643
- unbindService() method 501
- unregisterReceiver() method 481
- upload key 716
- URL Mapping 697
- USB connection issues
 - resolving 66
- USE_BIOMETRIC 706
- user interface state 155
- USE_SIP permission 602

V

- Video Playback 583
- VideoView class 583
 - methods 583
 - supported formats 583
- view bindings
 - enabling 144
 - using 144
- View class
 - setting properties 252
- view conversion 189
- ViewGroup 173
- View Groups 173
- View Hierarchy 175
- ViewHolder class 430
 - sample implementation 436
- 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 488
- 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 663, 668
- WebView view 469
- 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 489, 491
- wrap_content properties 243
- WRITE_CALENDAR permission 602
- WRITE_CALL_LOG permission 602
- WRITE_CONTACTS permission 602
- WRITE_EXTERNAL_STORAGE permission 603

X

- XML Layout File
 - manual creation 241
 - vs. Java Code 245

Z

Index

zip() operator 527