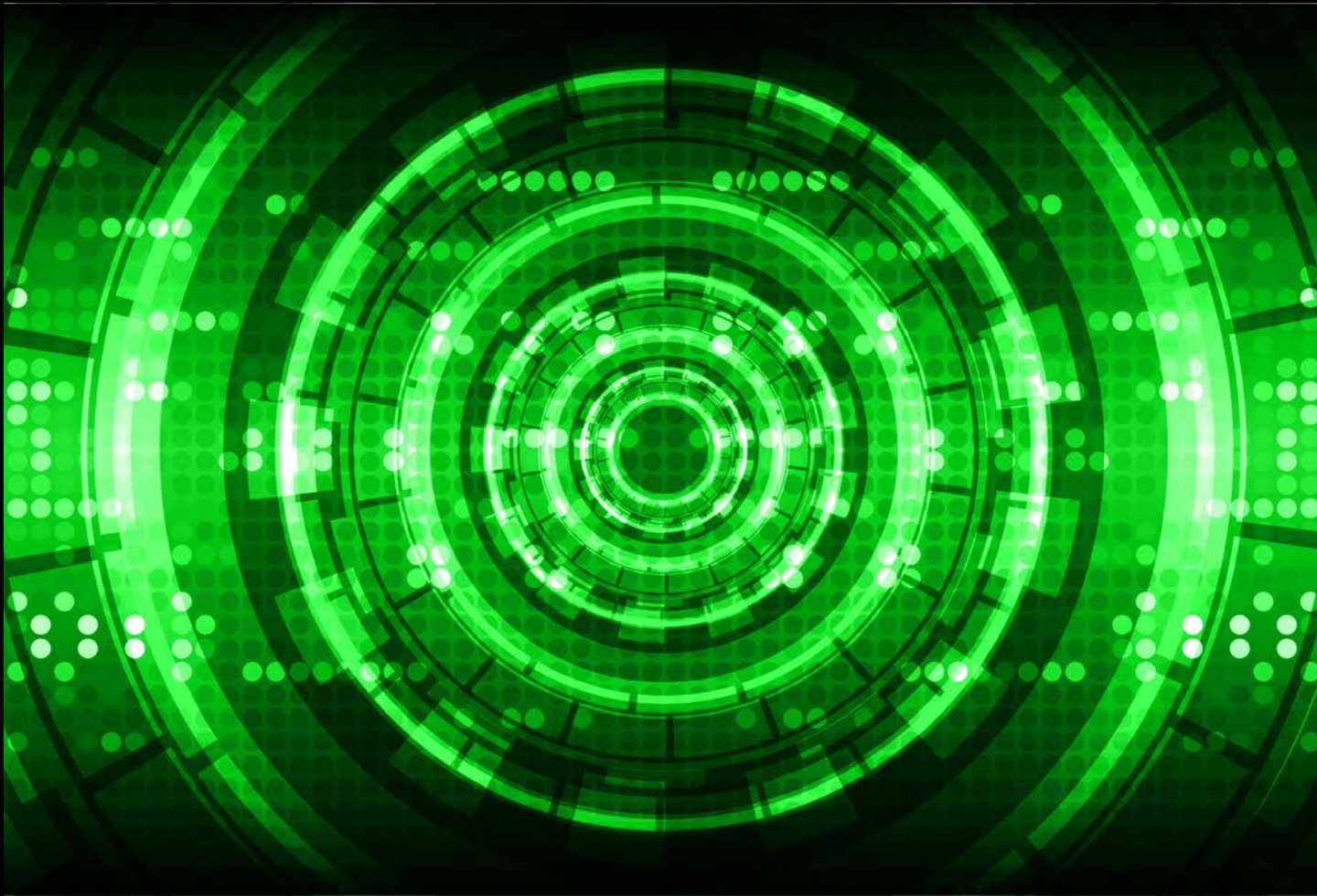


Android Studio Iguana Essentials



Java Edition

Android Studio Iguana Essentials

Java Edition

Android Studio Iguana Essentials – Java Edition

ISBN: 978-1-951442-89-7

© 2024 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
1.4 Authors Wanted	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

Table of Contents

4.7 Running the Emulator in a Separate Window.....	37
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	52
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	54
6.4 The Tool Windows	56
6.5 The Tool Window Menus.....	59
6.6 Android Studio Keyboard Shortcuts	59
6.7 Switcher and Recent Files Navigation	60
6.8 Changing the Android Studio Theme	61
6.9 Summary	62
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	77
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 Hardware Abstraction Layer.....	84
9.4 Android Runtime – ART.....	84
9.5 Android Libraries.....	84
9.5.1 C/C++ Libraries	85
9.6 Application Framework.....	86
9.7 Applications	86
9.8 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 Overview of Android View Binding.....	91
11.1 Find View by Id	91
11.2 View Binding	91
11.3 Converting the AndroidSample project.....	92
11.4 Enabling View Binding.....	92
11.5 Using View Binding	92

Table of Contents

11.6 Choosing an Option	94
11.7 View Binding in the Book Examples	94
11.8 Migrating a Project to View Binding	94
11.9 Summary	95
12. Understanding Android Application and Activity Lifecycles	97
12.1 Android Applications and Resource Management	97
12.2 Android Process States	97
12.2.1 Foreground Process	98
12.2.2 Visible Process	98
12.2.3 Service Process	98
12.2.4 Background Process	98
12.2.5 Empty Process	99
12.3 Inter-Process Dependencies	99
12.4 The Activity Lifecycle	99
12.5 The Activity Stack	99
12.6 Activity States	100
12.7 Configuration Changes	100
12.8 Handling State Change	101
12.9 Summary	101
13. Handling Android Activity State Changes	103
13.1 New vs. Old Lifecycle Techniques	103
13.2 The Activity and Fragment Classes	103
13.3 Dynamic State vs. Persistent State	105
13.4 The Android Lifecycle Methods	106
13.5 Lifetimes	107
13.6 Foldable Devices and Multi-Resume	108
13.7 Disabling Configuration Change Restarts	108
13.8 Lifecycle Method Limitations	109
13.9 Summary	109
14. Android Activity State Changes by Example	111
14.1 Creating the State Change Example Project	111
14.2 Designing the User Interface	112
14.3 Overriding the Activity Lifecycle Methods	113
14.4 Filtering the Logcat Panel	115
14.5 Running the Application	117
14.6 Experimenting with the Activity	117
14.7 Summary	118
15. Saving and Restoring the State of an Android Activity	119
15.1 Saving Dynamic State	119
15.2 Default Saving of User Interface State	119
15.3 The Bundle Class	120
15.4 Saving the State	121
15.5 Restoring the State	122
15.6 Testing the Application	122
15.7 Summary	122

16. Understanding Android Views, View Groups and Layouts	125
16.1 Designing for Different Android Devices	125
16.2 Views and View Groups	125
16.3 Android Layout Managers	125
16.4 The View Hierarchy	127
16.5 Creating User Interfaces	128
16.6 Summary	128
17. A Guide to the Android Studio Layout Editor Tool	129
17.1 Basic vs. Empty Views Activity Templates	129
17.2 The Android Studio Layout Editor	133
17.3 Design Mode	133
17.4 The Palette	134
17.5 Design Mode and Layout Views	135
17.6 Night Mode	136
17.7 Code Mode	136
17.8 Split Mode	137
17.9 Setting Attributes	137
17.10 Transforms	139
17.11 Tools Visibility Toggles	140
17.12 Converting Views	141
17.13 Displaying Sample Data	142
17.14 Creating a Custom Device Definition	143
17.15 Changing the Current Device	143
17.16 Layout Validation	144
17.17 Summary	145
18. A Guide to the Android ConstraintLayout	147
18.1 How ConstraintLayout Works	147
18.1.1 Constraints	147
18.1.2 Margins	148
18.1.3 Opposing Constraints	148
18.1.4 Constraint Bias	149
18.1.5 Chains	150
18.1.6 Chain Styles	150
18.2 Baseline Alignment	151
18.3 Configuring Widget Dimensions	151
18.4 Guideline Helper	152
18.5 Group Helper	152
18.6 Barrier Helper	152
18.7 Flow Helper	154
18.8 Ratios	155
18.9 ConstraintLayout Advantages	155
18.10 ConstraintLayout Availability	156
18.11 Summary	156
19. A Guide to Using ConstraintLayout in Android Studio	157
19.1 Design and Layout Views	157
19.2 Autoconnect Mode	159

Table of Contents

19.3 Inference Mode.....	159
19.4 Manipulating Constraints Manually.....	159
19.5 Adding Constraints in the Inspector	161
19.6 Viewing Constraints in the Attributes Window.....	161
19.7 Deleting Constraints.....	162
19.8 Adjusting Constraint Bias	163
19.9 Understanding ConstraintLayout Margins.....	163
19.10 The Importance of Opposing Constraints and Bias	165
19.11 Configuring Widget Dimensions.....	167
19.12 Design Time Tools Positioning.....	168
19.13 Adding Guidelines	169
19.14 Adding Barriers	171
19.15 Adding a Group.....	172
19.16 Working with the Flow Helper.....	173
19.17 Widget Group Alignment and Distribution.....	173
19.18 Converting other Layouts to ConstraintLayout.....	175
19.19 Summary	175
20. Working with ConstraintLayout Chains and Ratios in Android Studio	177
20.1 Creating a Chain.....	177
20.2 Changing the Chain Style	179
20.3 Spread Inside Chain Style.....	180
20.4 Packed Chain Style.....	180
20.5 Packed Chain Style with Bias.....	180
20.6 Weighted Chain.....	180
20.7 Working with Ratios	181
20.8 Summary	183
21. An Android Studio Layout Editor ConstraintLayout Tutorial	185
21.1 An Android Studio Layout Editor Tool Example	185
21.2 Preparing the Layout Editor Environment	185
21.3 Adding the Widgets to the User Interface.....	186
21.4 Adding the Constraints	189
21.5 Testing the Layout.....	191
21.6 Using the Layout Inspector	191
21.7 Summary	192
22. Manual XML Layout Design in Android Studio	193
22.1 Manually Creating an XML Layout	193
22.2 Manual XML vs. Visual Layout Design.....	196
22.3 Summary	196
23. Managing Constraints using Constraint Sets.....	197
23.1 Java Code vs. XML Layout Files.....	197
23.2 Creating Views.....	197
23.3 View Attributes.....	198
23.4 Constraint Sets.....	198
23.4.1 Establishing Connections.....	198
23.4.2 Applying Constraints to a Layout	198
23.4.3 Parent Constraint Connections.....	198

23.4.4 Sizing Constraints	199
23.4.5 Constraint Bias	199
23.4.6 Alignment Constraints	199
23.4.7 Copying and Applying Constraint Sets	199
23.4.8 ConstraintLayout Chains	199
23.4.9 Guidelines	200
23.4.10 Removing Constraints	200
23.4.11 Scaling	200
23.4.12 Rotation	201
23.5 Summary	201
24. An Android ConstraintSet Tutorial.....	203
24.1 Creating the Example Project in Android Studio	203
24.2 Adding Views to an Activity	203
24.3 Setting View Attributes	205
24.4 Creating View IDs	205
24.5 Configuring the Constraint Set	206
24.6 Adding the EditText View	207
24.7 Converting Density Independent Pixels (dp) to Pixels (px)	208
24.8 Summary	210
25. A Guide to Using Apply Changes in Android Studio	211
25.1 Introducing Apply Changes	211
25.2 Understanding Apply Changes Options	211
25.3 Using Apply Changes	212
25.4 Configuring Apply Changes Fallback Settings	213
25.5 An Apply Changes Tutorial	213
25.6 Using Apply Code Changes	213
25.7 Using Apply Changes and Restart Activity	214
25.8 Using Run App	214
25.9 Summary	214
26. An Overview and Example of Android Event Handling	215
26.1 Understanding Android Events	215
26.2 Using the android:onClick Resource	215
26.3 Event Listeners and Callback Methods	216
26.4 An Event Handling Example	216
26.5 Designing the User Interface	217
26.6 The Event Listener and Callback Method	217
26.7 Consuming Events	219
26.8 Summary	220
27. Android Touch and Multi-touch Event Handling	221
27.1 Intercepting Touch Events	221
27.2 The MotionEvent Object	221
27.3 Understanding Touch Actions	222
27.4 Handling Multiple Touches	222
27.5 An Example Multi-Touch Application	222
27.6 Designing the Activity User Interface	223
27.7 Implementing the Touch Event Listener	223

Table of Contents

27.8 Running the Example Application.....	226
27.9 Summary	227
28. Detecting Common Gestures Using the Android Gesture Detector Class	229
28.1 Implementing Common Gesture Detection.....	229
28.2 Creating an Example Gesture Detection Project	230
28.3 Implementing the Listener Class.....	230
28.4 Creating the GestureDetectorCompat Instance.....	232
28.5 Implementing the onTouchEvent() Method.....	233
28.6 Testing the Application.....	233
28.7 Summary	234
29. Implementing Custom Gesture and Pinch Recognition on Android	235
29.1 The Android Gesture Builder Application.....	235
29.2 The GestureOverlayView Class	235
29.3 Detecting Gestures	235
29.4 Identifying Specific Gestures	235
29.5 Installing and Running the Gesture Builder Application	235
29.6 Creating a Gestures File	236
29.7 Creating the Example Project.....	236
29.8 Extracting the Gestures File from the SD Card	236
29.9 Adding the Gestures File to the Project	237
29.10 Designing the User Interface	237
29.11 Loading the Gestures File	238
29.12 Registering the Event Listener.....	239
29.13 Implementing the onGesturePerformed Method.....	239
29.14 Testing the Application.....	240
29.15 Configuring the GestureOverlayView.....	240
29.16 Intercepting Gestures.....	241
29.17 Detecting Pinch Gestures.....	241
29.18 A Pinch Gesture Example Project.....	241
29.19 Summary	243
30. An Introduction to Android Fragments.....	245
30.1 What is a Fragment?	245
30.2 Creating a Fragment	245
30.3 Adding a Fragment to an Activity using the Layout XML File.....	246
30.4 Adding and Managing Fragments in Code	248
30.5 Handling Fragment Events	249
30.6 Implementing Fragment Communication.....	250
30.7 Summary	251
31. Using Fragments in Android Studio - An Example.....	253
31.1 About the Example Fragment Application	253
31.2 Creating the Example Project.....	253
31.3 Creating the First Fragment Layout.....	253
31.4 Migrating a Fragment to View Binding	255
31.5 Adding the Second Fragment.....	256
31.6 Adding the Fragments to the Activity	257
31.7 Making the Toolbar Fragment Talk to the Activity	258

31.8 Making the Activity Talk to the Text Fragment	261
31.9 Testing the Application.....	262
31.10 Summary	263
32. Modern Android App Architecture with Jetpack	265
32.1 What is Android Jetpack?	265
32.2 The “Old” Architecture.....	265
32.3 Modern Android Architecture	265
32.4 The ViewModel Component	266
32.5 The LiveData Component.....	266
32.6 ViewModel Saved State.....	267
32.7 LiveData and Data Binding.....	267
32.8 Android Lifecycles	268
32.9 Repository Modules.....	268
32.10 Summary	269
33. An Android ViewModel Tutorial.....	271
33.1 About the Project	271
33.2 Creating the ViewModel Example Project.....	271
33.3 Removing Unwanted Project Elements.....	271
33.4 Designing the Fragment Layout.....	272
33.5 Implementing the View Model.....	273
33.6 Associating the Fragment with the View Model.....	274
33.7 Modifying the Fragment	275
33.8 Accessing the ViewModel Data.....	276
33.9 Testing the Project.....	276
33.10 Summary	277
34. An Android Jetpack LiveData Tutorial.....	279
34.1 LiveData - A Recap	279
34.2 Adding LiveData to the ViewModel.....	279
34.3 Implementing the Observer.....	281
34.4 Summary	283
35. An Overview of Android Jetpack Data Binding	285
35.1 An Overview of Data Binding.....	285
35.2 The Key Components of Data Binding	285
35.2.1 The Project Build Configuration.....	285
35.2.2 The Data Binding Layout File.....	286
35.2.3 The Layout File Data Element	287
35.2.4 The Binding Classes	288
35.2.5 Data Binding Variable Configuration.....	288
35.2.6 Binding Expressions (One-Way).....	289
35.2.7 Binding Expressions (Two-Way).....	290
35.2.8 Event and Listener Bindings	290
35.3 Summary	291
36. An Android Jetpack Data Binding Tutorial.....	293
36.1 Removing the Redundant Code.....	293
36.2 Enabling Data Binding	294

Table of Contents

36.3 Adding the Layout Element.....	295
36.4 Adding the Data Element to Layout File.....	296
36.5 Working with the Binding Class	296
36.6 Assigning the ViewModel Instance to the Data Binding Variable	297
36.7 Adding Binding Expressions	298
36.8 Adding the Conversion Method	299
36.9 Adding a Listener Binding.....	299
36.10 Testing the App.....	299
36.11 Summary.....	300
37. An Android ViewModel Saved State Tutorial.....	301
37.1 Understanding ViewModel State Saving.....	301
37.2 Implementing ViewModel State Saving	301
37.3 Saving and Restoring State	303
37.4 Adding Saved State Support to the ViewModelDemo Project.....	303
37.5 Summary	304
38. Working with Android Lifecycle-Aware Components	305
38.1 Lifecycle Awareness	305
38.2 Lifecycle Owners	305
38.3 Lifecycle Observers	306
38.4 Lifecycle States and Events.....	307
38.5 Summary	308
39. An Android Jetpack Lifecycle Awareness Tutorial	309
39.1 Creating the Example Lifecycle Project.....	309
39.2 Creating a Lifecycle Observer.....	309
39.3 Adding the Observer	311
39.4 Testing the Observer	311
39.5 Creating a Lifecycle Owner.....	311
39.6 Testing the Custom Lifecycle Owner.....	313
39.7 Summary	313
40. An Overview of the Navigation Architecture Component.....	315
40.1 Understanding Navigation	315
40.2 Declaring a Navigation Host.....	316
40.3 The Navigation Graph	318
40.4 Accessing the Navigation Controller	319
40.5 Triggering a Navigation Action	319
40.6 Passing Arguments.....	320
40.7 Summary	320
41. An Android Jetpack Navigation Component Tutorial	321
41.1 Creating the NavigationDemo Project.....	321
41.2 Adding Navigation to the Build Configuration.....	321
41.3 Creating the Navigation Graph Resource File.....	322
41.4 Declaring a Navigation Host.....	323
41.5 Adding Navigation Destinations.....	324
41.6 Designing the Destination Fragment Layouts.....	326
41.7 Adding an Action to the Navigation Graph.....	328

41.8 Implement the OnFragmentManagerInteractionListener	329
41.9 Adding View Binding Support to the Destination Fragments	330
41.10 Triggering the Action	331
41.11 Passing Data Using Safeargs	332
41.12 Summary	335
42. An Introduction to MotionLayout.....	337
42.1 An Overview of MotionLayout	337
42.2 MotionLayout	337
42.3 MotionScene	337
42.4 Configuring ConstraintSets	338
42.5 Custom Attributes	339
42.6 Triggering an Animation.....	341
42.7 Arc Motion.....	342
42.8 Keyframes.....	342
42.8.1 Attribute Keyframes.....	342
42.8.2 Position Keyframes	343
42.9 Time Linearity	346
42.10 KeyTrigger.....	346
42.11 Cycle and Time Cycle Keyframes	347
42.12 Starting an Animation from Code.....	347
42.13 Summary	348
43. An Android MotionLayout Editor.....	349
43.1 Creating the MotionLayoutDemo Project	349
43.2 ConstraintLayout to MotionLayout Conversion	349
43.3 Configuring Start and End Constraints	351
43.4 Previewing the MotionLayout Animation	354
43.5 Adding an OnClick Gesture	354
43.6 Adding an Attribute Keyframe to the Transition.....	356
43.7 Adding a CustomAttribute to a Transition.....	358
43.8 Adding Position Keyframes	360
43.9 Summary	362
44. A MotionLayout KeyCycle Tutorial	363
44.1 An Overview of Cycle Keyframes	363
44.2 Using the Cycle Editor.....	367
44.3 Creating the KeyCycleDemo Project.....	368
44.4 Configuring the Start and End Constraints.....	368
44.5 Creating the Cycles	370
44.6 Previewing the Animation	372
44.7 Adding the KeyFrameSet to the MotionScene	372
44.8 Summary	374
45. Working with the Floating Action Button and Snackbar	375
45.1 The Material Design.....	375
45.2 The Design Library	375
45.3 The Floating Action Button (FAB)	375
45.4 The Snackbar.....	376
45.5 Creating the Example Project.....	377

Table of Contents

45.6 Reviewing the Project	377
45.7 Removing Navigation Features.....	378
45.8 Changing the Floating Action Button	378
45.9 Adding an Action to the Snackbar	380
45.10 Summary	380
46. Creating a Tabbed Interface using the TabLayout Component	381
46.1 An Introduction to the ViewPager2	381
46.2 An Overview of the TabLayout Component	381
46.3 Creating the TabLayoutDemo Project.....	382
46.4 Creating the First Fragment.....	383
46.5 Duplicating the Fragments.....	384
46.6 Adding the TabLayout and ViewPager2.....	385
46.7 Performing the Initialization Tasks.....	387
46.8 Testing the Application.....	389
46.9 Customizing the TabLayout.....	389
46.10 Summary	391
47. Working with the RecyclerView and CardView Widgets	393
47.1 An Overview of the RecyclerView	393
47.2 An Overview of the CardView	395
47.3 Summary	396
48. An Android RecyclerView and CardView Tutorial	397
48.1 Creating the CardDemo Project.....	397
48.2 Modifying the Basic Views Activity Project	397
48.3 Designing the CardView Layout.....	398
48.4 Adding the RecyclerView.....	399
48.5 Adding the Image Files.....	399
48.6 Creating the RecyclerView Adapter.....	400
48.7 Initializing the RecyclerView Component.....	402
48.8 Testing the Application.....	403
48.9 Responding to Card Selections.....	403
48.10 Summary	405
49. A Layout Editor Sample Data Tutorial	407
49.1 Adding Sample Data to a Project	407
49.2 Using Custom Sample Data	411
49.3 Summary	414
50. Working with the AppBar and Collapsing Toolbar Layouts	415
50.1 The Anatomy of an AppBar	415
50.2 The Example Project	416
50.3 Coordinating the RecyclerView and Toolbar	416
50.4 Introducing the Collapsing Toolbar Layout	418
50.5 Changing the Title and Scrim Color	421
50.6 Summary	422
51. An Android Studio Primary/Detail Flow Tutorial	423
51.1 The Primary/Detail Flow.....	423

51.2 Creating a Primary/Detail Flow Activity	424
51.3 Adding the Primary/Detail Flow Activity.....	424
51.4 Modifying the Primary/Detail Flow Template	425
51.5 Changing the Content Model	425
51.6 Changing the Detail Pane	427
51.7 Modifying the ItemDetailFragment Class	428
51.8 Modifying the ItemListFragment Class.....	429
51.9 Adding Manifest Permissions.....	430
51.10 Running the Application.....	430
51.11 Summary	430
52. An Overview of Android Services.....	431
52.1 Intent Service	431
52.2 Bound Service.....	431
52.3 The Anatomy of a Service	432
52.4 Controlling Destroyed Service Restart Options.....	432
52.5 Declaring a Service in the Manifest File.....	432
52.6 Starting a Service Running on System Startup.....	433
52.7 Summary	434
53. An Overview of Android Intents	435
53.1 An Overview of Intents	435
53.2 Explicit Intents.....	435
53.3 Returning Data from an Activity	436
53.4 Implicit Intents	437
53.5 Using Intent Filters.....	438
53.6 Automatic Link Verification	438
53.7 Manually Enabling Links	441
53.8 Checking Intent Availability	442
53.9 Summary	443
54. Android Explicit Intents – A Worked Example	445
54.1 Creating the Explicit Intent Example Application.....	445
54.2 Designing the User Interface Layout for MainActivity.....	445
54.3 Creating the Second Activity Class.....	446
54.4 Designing the User Interface Layout for SecondActivity	447
54.5 Reviewing the Application Manifest File	447
54.6 Creating the Intent	448
54.7 Extracting Intent Data	449
54.8 Launching SecondActivity as a Sub-Activity.....	450
54.9 Returning Data from a Sub-Activity.....	451
54.10 Testing the Application.....	451
54.11 Summary	452
55. Android Implicit Intents – A Worked Example	453
55.1 Creating the Android Studio Implicit Intent Example Project	453
55.2 Designing the User Interface	453
55.3 Creating the Implicit Intent	454
55.4 Adding a Second Matching Activity.....	454
55.5 Adding the Web View to the UI.....	455

Table of Contents

55.6 Obtaining the Intent URL	455
55.7 Modifying the MyWebView Project Manifest File	457
55.8 Installing the MyWebView Package on a Device	458
55.9 Testing the Application.....	459
55.10 Manually Enabling the Link	459
55.11 Automatic Link Verification	461
55.12 Summary	463
56. Android Broadcast Intents and Broadcast Receivers	465
56.1 An Overview of Broadcast Intents	465
56.2 An Overview of Broadcast Receivers	466
56.3 Obtaining Results from a Broadcast	467
56.4 Sticky Broadcast Intents	467
56.5 The Broadcast Intent Example.....	468
56.6 Creating the Example Application	468
56.7 Creating and Sending the Broadcast Intent.....	468
56.8 Creating the Broadcast Receiver	469
56.9 Registering the Broadcast Receiver.....	470
56.10 Testing the Broadcast Example	471
56.11 Listening for System Broadcasts.....	471
56.12 Summary	472
57. Android Local Bound Services – A Worked Example	473
57.1 Understanding Bound Services	473
57.2 Bound Service Interaction Options	473
57.3 A Local Bound Service Example.....	473
57.4 Adding a Bound Service to the Project	474
57.5 Implementing the Binder	474
57.6 Binding the Client to the Service	477
57.7 Completing the Example.....	478
57.8 Testing the Application.....	479
57.9 Summary	479
58. Android Remote Bound Services – A Worked Example	481
58.1 Client to Remote Service Communication.....	481
58.2 Creating the Example Application	481
58.3 Designing the User Interface	481
58.4 Implementing the Remote Bound Service.....	481
58.5 Configuring a Remote Service in the Manifest File.....	483
58.6 Launching and Binding to the Remote Service.....	484
58.7 Sending a Message to the Remote Service	485
58.8 Summary	486
59. An Overview of Java Threads, Handlers and Executors	487
59.1 The Application Main Thread.....	487
59.2 Thread Handlers	487
59.3 A Threading Example	487
59.4 Building the App	488
59.5 Creating a New Thread.....	489
59.6 Implementing a Thread Handler.....	490

59.7 Passing a Message to the Handler	492
59.8 Java Executor Concurrency	492
59.9 Working with Runnable Tasks.....	493
59.10 Shutting down an Executor Service.....	494
59.11 Working with Callable Tasks and Futures	494
59.12 Handling a Future Result	496
59.13 Scheduling Tasks	497
59.14 Summary	498
60. Making Runtime Permission Requests in Android.....	499
60.1 Understanding Normal and Dangerous Permissions.....	499
60.2 Creating the Permissions Example Project.....	501
60.3 Checking for a Permission	501
60.4 Requesting Permission at Runtime.....	503
60.5 Providing a Rationale for the Permission Request	504
60.6 Testing the Permissions App.....	506
60.7 Summary	506
61. An Android Notifications Tutorial	507
61.1 An Overview of Notifications.....	507
61.2 Creating the NotifyDemo Project	509
61.3 Designing the User Interface	509
61.4 Creating the Second Activity	509
61.5 Creating a Notification Channel	510
61.6 Requesting Notification Permission	511
61.7 Creating and Issuing a Notification	514
61.8 Launching an Activity from a Notification.....	516
61.9 Adding Actions to a Notification	518
61.10 Bundled Notifications.....	519
61.11 Summary	521
62. An Android Direct Reply Notification Tutorial	523
62.1 Creating the DirectReply Project.....	523
62.2 Designing the User Interface	523
62.3 Requesting Notification Permission	524
62.4 Creating the Notification Channel.....	525
62.5 Building the RemoteInput Object.....	526
62.6 Creating the PendingIntent.....	527
62.7 Creating the Reply Action.....	528
62.8 Receiving Direct Reply Input.....	530
62.9 Updating the Notification	531
62.10 Summary	533
63. Foldable Devices and Multi-Window Support	535
63.1 Foldables and Multi-Window Support.....	535
63.2 Using a Foldable Emulator	536
63.3 Entering Multi-Window Mode	537
63.4 Enabling and using Freeform Support	538
63.5 Checking for Freeform Support	538
63.6 Enabling Multi-Window Support in an App	538

Table of Contents

63.7 Specifying Multi-Window Attributes	539
63.8 Detecting Multi-Window Mode in an Activity	540
63.9 Receiving Multi-Window Notifications	540
63.10 Launching an Activity in Multi-Window Mode	541
63.11 Configuring Freeform Activity Size and Position.....	541
63.12 Summary.....	542
64. An Overview of Android SQLite Databases	543
64.1 Understanding Database Tables	543
64.2 Introducing Database Schema	543
64.3 Columns and Data Types	543
64.4 Database Rows	544
64.5 Introducing Primary Keys	544
64.6 What is SQLite?	544
64.7 Structured Query Language (SQL)	544
64.8 Trying SQLite on an Android Virtual Device (AVD)	545
64.9 Android SQLite Classes.....	546
64.9.1 Cursor	547
64.9.2 SQLiteDatabase	547
64.9.3 SQLiteOpenHelper	547
64.9.4 ContentValues.....	548
64.10 The Android Room Persistence Library.....	548
64.11 Summary.....	548
65. An Android SQLite Database Tutorial	549
65.1 About the Database Example.....	549
65.2 Creating the SQLDemo Project.....	549
65.3 Designing the User interface	549
65.4 Creating the Data Model.....	550
65.5 Implementing the Data Handler.....	551
65.6 The Add Handler Method.....	553
65.7 The Query Handler Method	553
65.8 The Delete Handler Method	554
65.9 Implementing the Activity Event Methods.....	554
65.10 Testing the Application.....	556
65.11 Summary.....	556
66. Understanding Android Content Providers.....	557
66.1 What is a Content Provider?.....	557
66.2 The Content Provider	557
66.2.1 onCreate()	557
66.2.2 query()	557
66.2.3 insert()	557
66.2.4 update()	558
66.2.5 delete()	558
66.2.6 getType()	558
66.3 The Content URI	558
66.4 The Content Resolver	558
66.5 The <provider> Manifest Element	559

66.6 Summary	559
67. An Android Content Provider Tutorial	561
67.1 Copying the SQLDemo Project.....	561
67.2 Adding the Content Provider Package	561
67.3 Creating the Content Provider Class	562
67.4 Constructing the Authority and Content URI	563
67.5 Implementing URI Matching in the Content Provider.....	564
67.6 Implementing the Content Provider onCreate() Method	565
67.7 Implementing the Content Provider insert() Method	565
67.8 Implementing the Content Provider query() Method	566
67.9 Implementing the Content Provider update() Method	567
67.10 Implementing the Content Provider delete() Method.....	569
67.11 Declaring the Content Provider in the Manifest File	570
67.12 Modifying the Database Handler.....	571
67.13 Summary	573
68. An Android Content Provider Client Tutorial.....	575
68.1 Creating the SQLDemoClient Project.....	575
68.2 Designing the User interface	575
68.3 Accessing the Content Provider	575
68.4 Adding the Query Permission.....	576
68.5 Testing the Project.....	577
68.6 Summary	577
69. The Android Room Persistence Library	579
69.1 Revisiting Modern App Architecture	579
69.2 Key Elements of Room Database Persistence.....	579
69.2.1 Repository	580
69.2.2 Room Database	580
69.2.3 Data Access Object (DAO)	580
69.2.4 Entities.....	580
69.2.5 SQLite Database	580
69.3 Understanding Entities.....	581
69.4 Data Access Objects.....	584
69.5 The Room Database	585
69.6 The Repository.....	586
69.7 In-Memory Databases.....	587
69.8 Database Inspector.....	587
69.9 Summary	587
70. An Android TableLayout and TableRow Tutorial	589
70.1 The TableLayout and TableRow Layout Views	589
70.2 Creating the Room Database Project	590
70.3 Converting to a LinearLayout.....	590
70.4 Adding the TableLayout to the User Interface.....	591
70.5 Configuring the TableRows	592
70.6 Adding the Button Bar to the Layout	593
70.7 Adding the RecyclerView.....	594
70.8 Adjusting the Layout Margins.....	595

Table of Contents

70.9 Summary	595
71. An Android Room Database and Repository Tutorial.....	597
71.1 About the RoomDemo Project.....	597
71.2 Modifying the Build Configuration.....	597
71.3 Building the Entity	598
71.4 Creating the Data Access Object.....	600
71.5 Adding the Room Database.....	601
71.6 Adding the Repository	601
71.7 Adding the ViewModel	604
71.8 Creating the Product Item Layout	605
71.9 Adding the RecyclerView Adapter.....	606
71.10 Preparing the Main Activity	607
71.11 Adding the Button Listeners.....	608
71.12 Adding LiveData Observers	609
71.13 Initializing the RecyclerView.....	610
71.14 Testing the RoomDemo App.....	610
71.15 Using the Database Inspector.....	610
71.16 Summary.....	611
72. Accessing Cloud Storage using the Android Storage Access Framework.....	613
72.1 The Storage Access Framework.....	613
72.2 Working with the Storage Access Framework.....	614
72.3 Filtering Picker File Listings.....	614
72.4 Handling Intent Results.....	615
72.5 Reading the Content of a File	615
72.6 Writing Content to a File	616
72.7 Deleting a File.....	617
72.8 Gaining Persistent Access to a File.....	617
72.9 Summary	617
73. An Android Storage Access Framework Example	619
73.1 About the Storage Access Framework Example.....	619
73.2 Creating the Storage Access Framework Example.....	619
73.3 Designing the User Interface	619
73.4 Adding the Activity Launchers.....	620
73.5 Creating a New Storage File.....	622
73.6 Saving to a Storage File.....	622
73.7 Opening and Reading a Storage File	624
73.8 Testing the Storage Access Application	625
73.9 Summary	626
74. Video Playback on Android using the VideoView and MediaController Classes	627
74.1 Introducing the Android VideoView Class	627
74.2 Introducing the Android MediaController Class	628
74.3 Creating the Video Playback Example	628
74.4 Designing the VideoPlayer Layout	628
74.5 Downloading the Video File.....	629
74.6 Configuring the VideoView.....	629
74.7 Adding the MediaController to the Video View.....	631

74.8 Setting up the onPreparedListener	631
74.9 Summary	632
75. Android Picture-in-Picture Mode.....	633
75.1 Picture-in-Picture Features.....	633
75.2 Enabling Picture-in-Picture Mode.....	634
75.3 Configuring Picture-in-Picture Parameters	634
75.4 Entering Picture-in-Picture Mode	635
75.5 Detecting Picture-in-Picture Mode Changes	635
75.6 Adding Picture-in-Picture Actions.....	636
75.7 Summary	636
76. An Android Picture-in-Picture Tutorial.....	639
76.1 Adding Picture-in-Picture Support to the Manifest	639
76.2 Adding a Picture-in-Picture Button	639
76.3 Entering Picture-in-Picture Mode	640
76.4 Detecting Picture-in-Picture Mode Changes	641
76.5 Adding a Broadcast Receiver	641
76.6 Adding the PiP Action.....	642
76.7 Testing the Picture-in-Picture Action	645
76.8 Summary	646
77. Android Audio Recording and Playback using MediaPlayer and MediaRecorder	647
77.1 Playing Audio	647
77.2 Recording Audio and Video using the MediaRecorder Class.....	648
77.3 About the Example Project	649
77.4 Creating the AudioApp Project.....	649
77.5 Designing the User Interface	649
77.6 Checking for Microphone Availability	650
77.7 Initializing the Activity.....	651
77.8 Implementing the recordAudio() Method.....	652
77.9 Implementing the stopAudio() Method.....	652
77.10 Implementing the playAudio() method.....	653
77.11 Configuring and Requesting Permissions	653
77.12 Testing the Application.....	655
77.13 Summary	656
78. Working with the Google Maps Android API in Android Studio	657
78.1 The Elements of the Google Maps Android API	657
78.2 Creating the Google Maps Project	658
78.3 Creating a Google Cloud Billing Account	658
78.4 Creating a New Google Cloud Project	659
78.5 Enabling the Google Maps SDK.....	660
78.6 Generating a Google Maps API Key.....	661
78.7 Adding the API Key to the Android Studio Project	662
78.8 Testing the Application.....	662
78.9 Understanding Geocoding and Reverse Geocoding.....	662
78.10 Adding a Map to an Application	664
78.11 Requesting Current Location Permission.....	664
78.12 Displaying the User's Current Location	666

Table of Contents

78.13 Changing the Map Type	667
78.14 Displaying Map Controls to the User	668
78.15 Handling Map Gesture Interaction	669
78.15.1 Map Zooming Gestures	669
78.15.2 Map Scrolling/Panning Gestures	669
78.15.3 Map Tilt Gestures	669
78.15.4 Map Rotation Gestures	669
78.16 Creating Map Markers	670
78.17 Controlling the Map Camera	671
78.18 Summary	672
79. Printing with the Android Printing Framework	673
79.1 The Android Printing Architecture	673
79.2 The Print Service Plugins	673
79.3 Google Cloud Print	674
79.4 Printing to Google Drive	674
79.5 Save as PDF	675
79.6 Printing from Android Devices	675
79.7 Options for Building Print Support into Android Apps	676
79.7.1 Image Printing	676
79.7.2 Creating and Printing HTML Content	677
79.7.3 Printing a Web Page	678
79.7.4 Printing a Custom Document	679
79.8 Summary	679
80. An Android HTML and Web Content Printing Example	681
80.1 Creating the HTML Printing Example Application	681
80.2 Printing Dynamic HTML Content	681
80.3 Creating the Web Page Printing Example	684
80.4 Removing the Floating Action Button	684
80.5 Removing Navigation Features	684
80.6 Designing the User Interface Layout	686
80.7 Accessing the WebView from the Main Activity	686
80.8 Loading the Web Page into the WebView	687
80.9 Adding the Print Menu Option	688
80.10 Summary	690
81. A Guide to Android Custom Document Printing	691
81.1 An Overview of Android Custom Document Printing	691
81.1.1 Custom Print Adapters	691
81.2 Preparing the Custom Document Printing Project	692
81.3 Creating the Custom Print Adapter	693
81.4 Implementing the onLayout() Callback Method	694
81.5 Implementing the onWrite() Callback Method	697
81.6 Checking a Page is in Range	699
81.7 Drawing the Content on the Page Canvas	700
81.8 Starting the Print Job	702
81.9 Testing the Application	703
81.10 Summary	703

82. An Introduction to Android App Links.....	705
82.1 An Overview of Android App Links	705
82.2 App Link Intent Filters	705
82.3 Handling App Link Intents	706
82.4 Associating the App with a Website.....	706
82.5 Summary	707
83. An Android Studio App Links Tutorial	709
83.1 About the Example App	709
83.2 The Database Schema	709
83.3 Loading and Running the Project	709
83.4 Adding the URL Mapping.....	711
83.5 Adding the Intent Filter.....	714
83.6 Adding Intent Handling Code.....	714
83.7 Testing the App.....	717
83.8 Creating the Digital Asset Links File	717
83.9 Testing the App Link.....	718
83.10 Summary	718
84. An Android Biometric Authentication Tutorial.....	719
84.1 An Overview of Biometric Authentication.....	719
84.2 Creating the Biometric Authentication Project	719
84.3 Configuring Device Fingerprint Authentication	720
84.4 Adding the Biometric Permission to the Manifest File.....	720
84.5 Designing the User Interface	721
84.6 Adding a Toast Convenience Method	721
84.7 Checking the Security Settings.....	722
84.8 Configuring the Authentication Callbacks.....	723
84.9 Adding the CancellationSignal.....	724
84.10 Starting the Biometric Prompt	724
84.11 Testing the Project.....	725
84.12 Summary	726
85. Creating, Testing, and Uploading an Android App Bundle.....	727
85.1 The Release Preparation Process	727
85.2 Android App Bundles.....	727
85.3 Register for a Google Play Developer Console Account.....	728
85.4 Configuring the App in the Console	729
85.5 Enabling Google Play App Signing.....	730
85.6 Creating a Keystore File	730
85.7 Creating the Android App Bundle.....	731
85.8 Generating Test APK Files	733
85.9 Uploading the App Bundle to the Google Play Developer Console.....	734
85.10 Exploring the App Bundle	735
85.11 Managing Testers	736
85.12 Rolling the App Out for Testing.....	736
85.13 Uploading New App Bundle Revisions.....	737
85.14 Analyzing the App Bundle File	738
85.15 Summary	739

86. An Overview of Android In-App Billing	741
86.1 Preparing a Project for In-App Purchasing.....	741
86.2 Creating In-App Products and Subscriptions	741
86.3 Billing Client Initialization.....	742
86.4 Connecting to the Google Play Billing Library.....	743
86.5 Querying Available Products.....	744
86.6 Starting the Purchase Process.....	744
86.7 Completing the Purchase	745
86.8 Querying Previous Purchases.....	746
86.9 Summary	747
87. An Android In-App Purchasing Tutorial	749
87.1 About the In-App Purchasing Example Project.....	749
87.2 Creating the InAppPurchase Project.....	749
87.3 Adding Libraries to the Project	749
87.4 Designing the User Interface	750
87.5 Adding the App to the Google Play Store	751
87.6 Creating an In-App Product	751
87.7 Enabling License Testers	752
87.8 Initializing the Billing Client	753
87.9 Querying the Product.....	754
87.10 Launching the Purchase Flow	756
87.11 Handling Purchase Updates	756
87.12 Consuming the Product.....	757
87.13 Restoring a Previous Purchase	758
87.14 Testing the App.....	759
87.15 Troubleshooting	760
87.16 Summary.....	760
88. Creating and Managing Overflow Menus on Android.....	761
88.1 The Overflow Menu	761
88.2 Creating an Overflow Menu	761
88.3 Displaying an Overflow Menu.....	762
88.4 Responding to Menu Item Selections.....	762
88.5 Creating Checkable Item Groups.....	763
88.6 Menus and the Android Studio Menu Editor.....	764
88.7 Creating the Example Project.....	765
88.8 Designing the Menu.....	765
88.9 Modifying the onOptionsItemSelected() Method.....	767
88.10 Testing the Application.....	768
88.11 Summary.....	769
89. Working with Material Design 3 Theming	771
89.1 Material Design 2 vs. Material Design 3	771
89.2 Understanding Material Design Theming	771
89.3 Material Design 3 Theming	771
89.4 Building a Custom Theme.....	773
89.5 Summary	774
90. A Material Design 3 Theming and Dynamic Color Tutorial.....	775

90.1 Creating the ThemeDemo Project	775
90.2 Designing the User Interface	775
90.3 Building a New Theme	777
90.4 Adding the Theme to the Project	778
90.5 Enabling Dynamic Color Support	779
90.6 Previewing Dynamic Colors.....	780
90.7 Summary	781
91. An Overview of Gradle in Android Studio	783
91.1 An Overview of Gradle	783
91.2 Gradle and Android Studio	783
91.2.1 Sensible Defaults	783
91.2.2 Dependencies.....	783
91.2.3 Build Variants	784
91.2.4 Manifest Entries	784
91.2.5 APK Signing.....	784
91.2.6 ProGuard Support.....	784
91.3 The Property and Settings Gradle Build File	784
91.4 The Top-level Gradle Build File.....	785
91.5 Module Level Gradle Build Files.....	786
91.6 Configuring Signing Settings in the Build File.....	788
91.7 Running Gradle Tasks from the Command Line	789
91.8 Summary	790
Index	791

1. Introduction

This book, fully updated for Android Studio Iguana (2023.2.1) and the new UI, teaches you how to develop Android-based applications using the Java programming language..

This book begins with the basics and outlines how to set up an Android development and testing environment, followed by an overview of areas such as tool windows, the code editor, and the Layout Editor tool. An introduction to the architecture of Android is followed by an in-depth look at the design of Android applications and user interfaces using the Android Studio environment.

Chapters also cover the Android Architecture Components, including view models, lifecycle management, Room database access, content providers, 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 Java 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/iguanajava>

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/iguanajava>

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.

1.4 Authors Wanted

Payload Publishing is looking for authors.

Are you an aspiring author with a book idea in mind? When you publish with us, you'll receive our full support every step of the way. We offer guidance and technical and editorial assistance to help you bring your book to life. Once your book is completed, we will publish and market it worldwide through our distribution and channel partnerships while paying you higher royalties than traditional publishers.

Find out more at:

<https://www.payloadbooks.com/authors-wanted>

or email us at:

authors@payloadbooks.com

2. Setting up an Android Studio Development Environment

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

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

2.1 System requirements

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

- Windows 8/10/11 64-bit
- macOS 10.14 or later running on Intel or Apple silicon
- Chrome OS device with Intel i5 or higher
- Linux systems with version 2.31 or later of the GNU C Library (glibc)
- Minimum of 8GB of RAM
- Approximately 8GB of available disk space
- 1280 x 800 minimum screen resolution

2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio Iguana 2023.2.1 using the Android API 34 SDK (UpsideDownCake), which, at the time of writing, are the latest stable releases.

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

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

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

If this page provides instructions for downloading a newer version of Android Studio, there may be differences between this book and the software. A web search for “Android Studio Iguana” should provide the option to download the older version if these differences become a problem. Alternatively, visit the following web page to find Android Studio Iguana 2023.2.1 in the archives:

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

2.3 Installing Android Studio

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

2.3.1 Installation on Windows

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

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

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

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

2.3.2 Installation on macOS

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

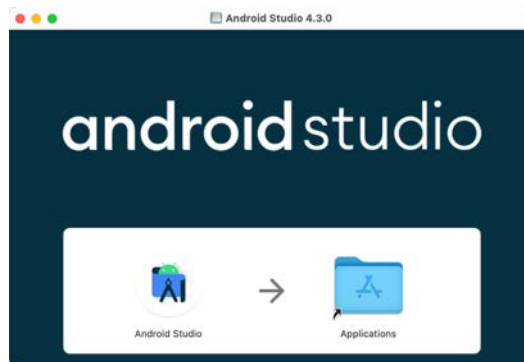


Figure 2-1

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

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

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

2.3.3 Installation on Linux

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

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

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

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

```
./studio.sh
```

2.4 The Android Studio setup wizard

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

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

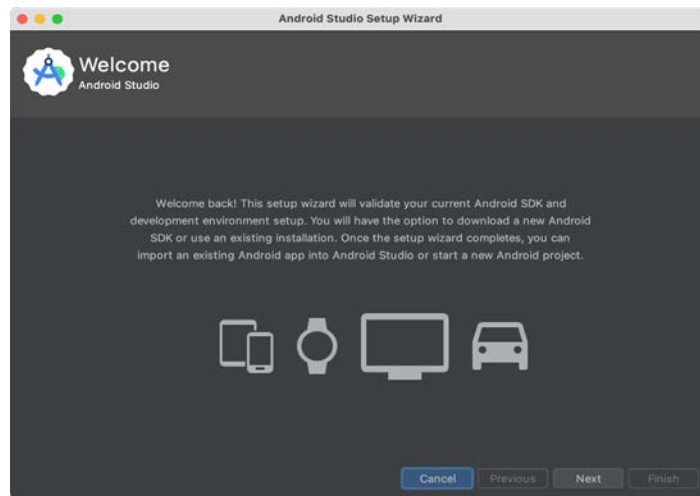


Figure 2-2

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

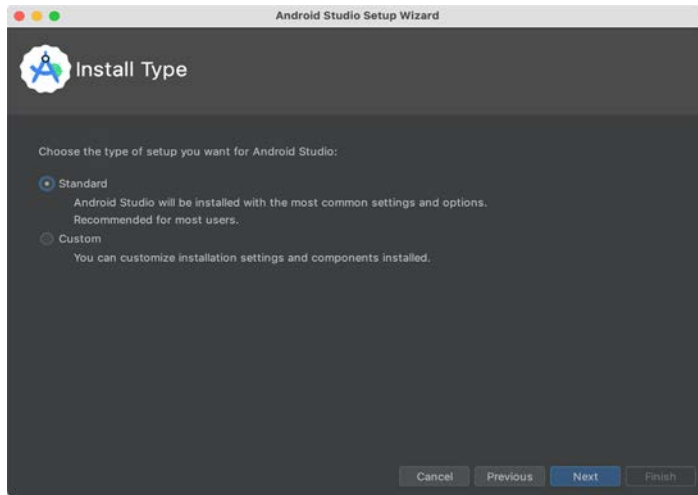


Figure 2-3

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

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

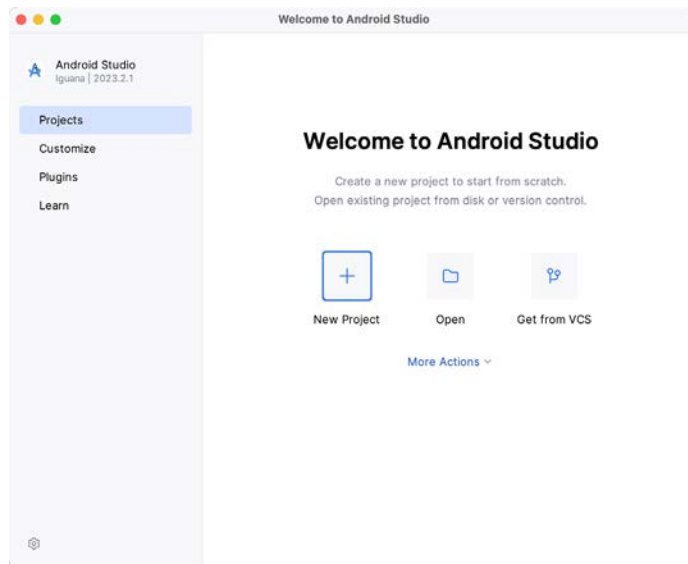


Figure 2-4

2.5 Installing additional Android SDK packages

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

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

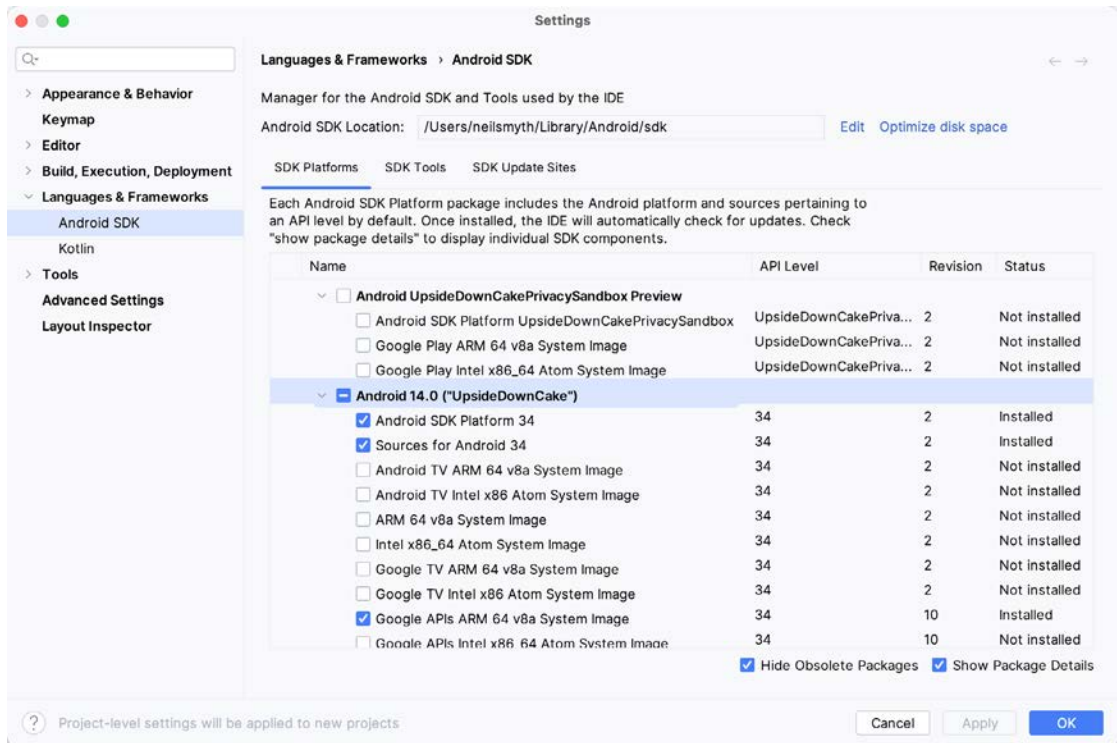


Figure 2-5

Google pairs each release of Android Studio with a maximum supported Application Programming Interface (API) level of the Android SDK. In the case of Android Studio Iguana, this is Android UpsideDownCake (API Level 34). This information can be confirmed using the following link:

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

Immediately after installing Android Studio for the first time, it is likely that only the latest supported version of the Android SDK has been installed. To install older versions of the Android SDK, select the checkboxes corresponding to the versions and click the *Apply* button. The rest of this book assumes that the Android UpsideDownCake (API Level 34) SDK is installed.

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

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

Setting up an Android Studio Development Environment

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

Figure 2-6

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

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



Figure 2-7

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

- Android SDK Build-tools
- Android Emulator
- Android SDK Platform-tools
- Google Play Services
- Intel x86 Emulator Accelerator (HAXM installer)*
- Google USB Driver (Windows only)
- Layout Inspector image server for API 31-34

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

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

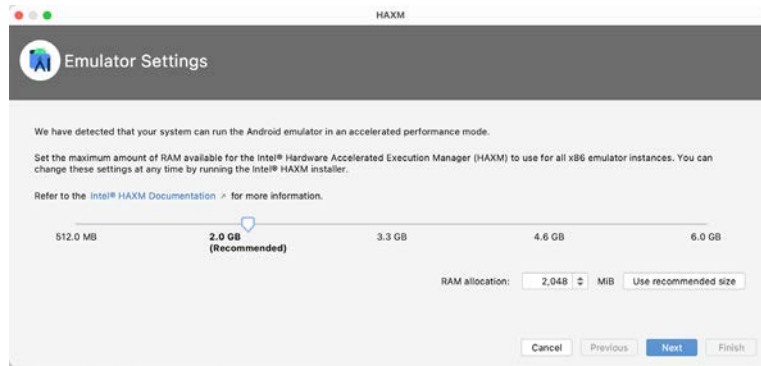


Figure 2-8

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

2.6 Installing the Android SDK Command-line Tools

Android Studio includes tools that allow some tasks to be performed from your operating system command line. To install these tools on your system, open the SDK Manager, select the SDK Tools tab, and locate the *Android SDK Command-line Tools (latest)* package as shown in Figure 2-9:

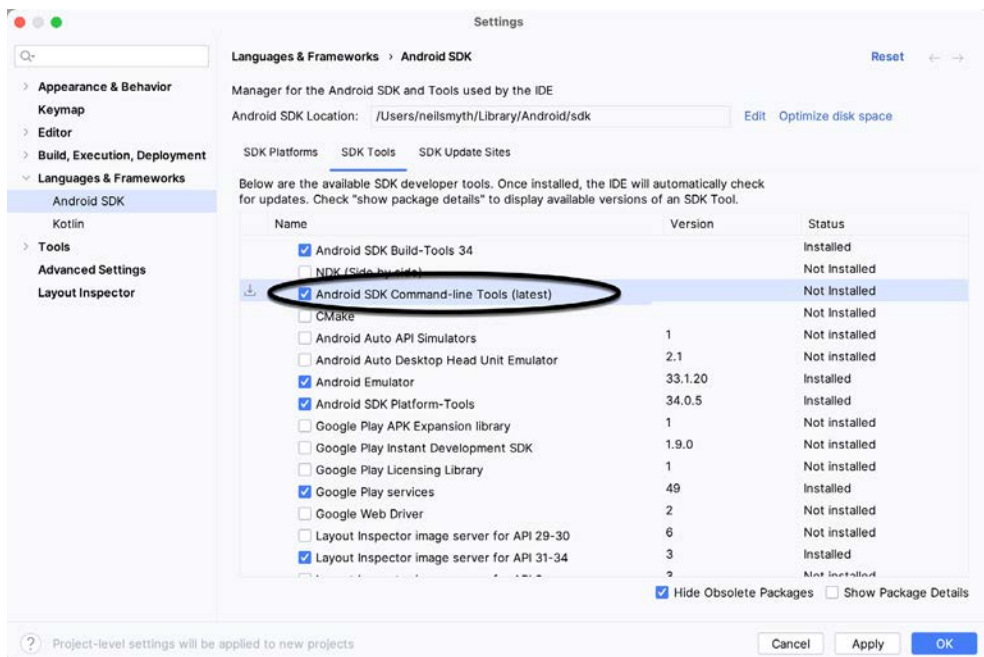


Figure 2-9

If the command-line tools package is not already installed, enable it and click *Apply*, followed by *OK* to complete the installation. When the installation completes, click *Finish* and close the SDK Manager dialog.

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

Setting up an Android Studio Development Environment

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

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

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



Figure 2-10

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

2.6.1 Windows 8.1

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

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

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

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

```
echo %Path%
```

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

```
adb
```

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

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

```
avdmanager
```

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

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

2.6.2 Windows 10

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

2.6.3 Windows 11

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

2.6.4 Linux

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

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

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

2.6.5 macOS

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

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

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

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

2.7 Android Studio memory management

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

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

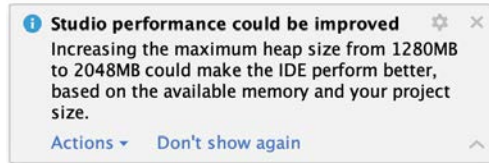


Figure 2-11

To view and modify the current memory configuration, select the *File -> Settings...* main menu option (*Android Studio -> Settings...* on macOS) and, in the resulting dialog, select *Appearance & Behavior* followed by the *Memory Settings* option listed under *System Settings* in the left-hand navigation panel, as illustrated in Figure 2-12 below:

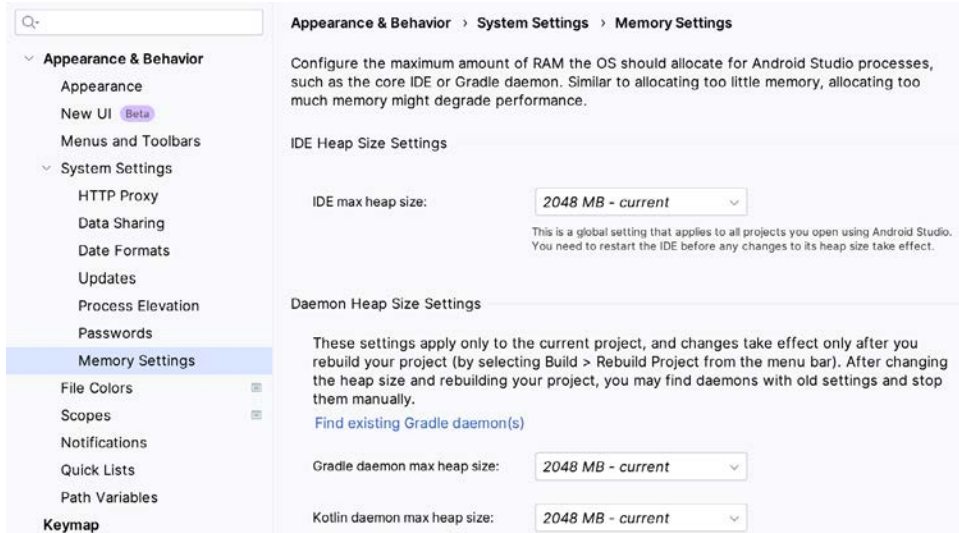


Figure 2-12

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

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

2.8 Updating Android Studio and the SDK

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

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

2.9 Summary

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

3. Creating an Example Android App in Android Studio

The preceding chapters of this book have 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:

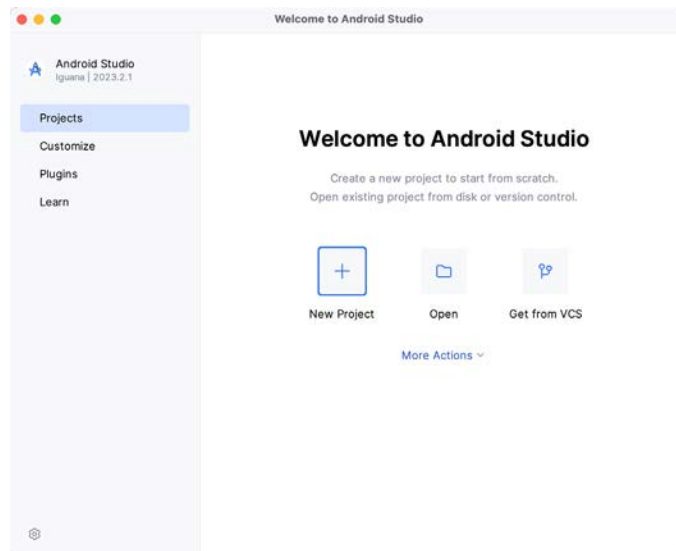


Figure 3-1

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.

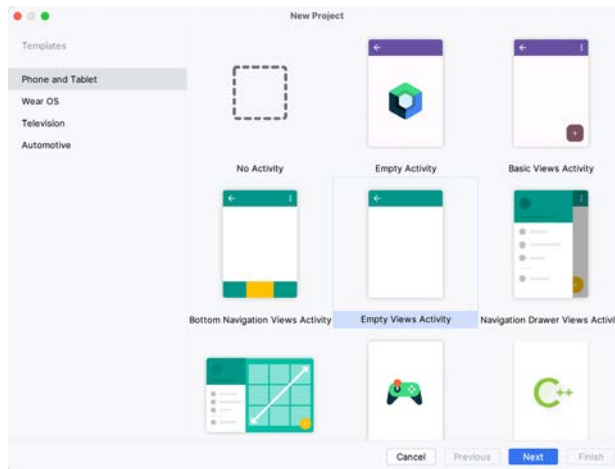


Figure 3-2

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:

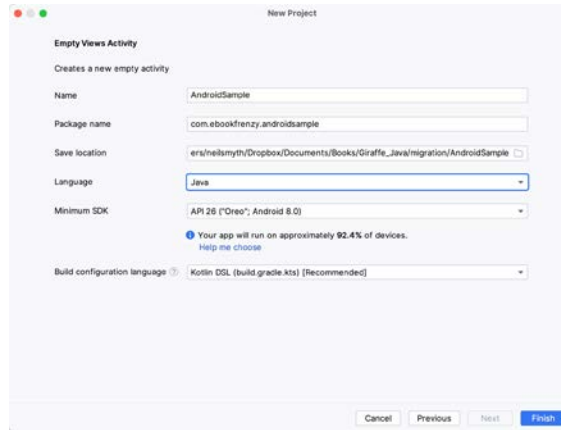


Figure 3-3

Finally, change the *Language* menu to *Java* and select *Kotlin DSL (build.gradle.kts)* as the build configuration 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 Iguana version. If your installation of Android Studio resembles b below, then you will need to enable the new UI before proceeding:

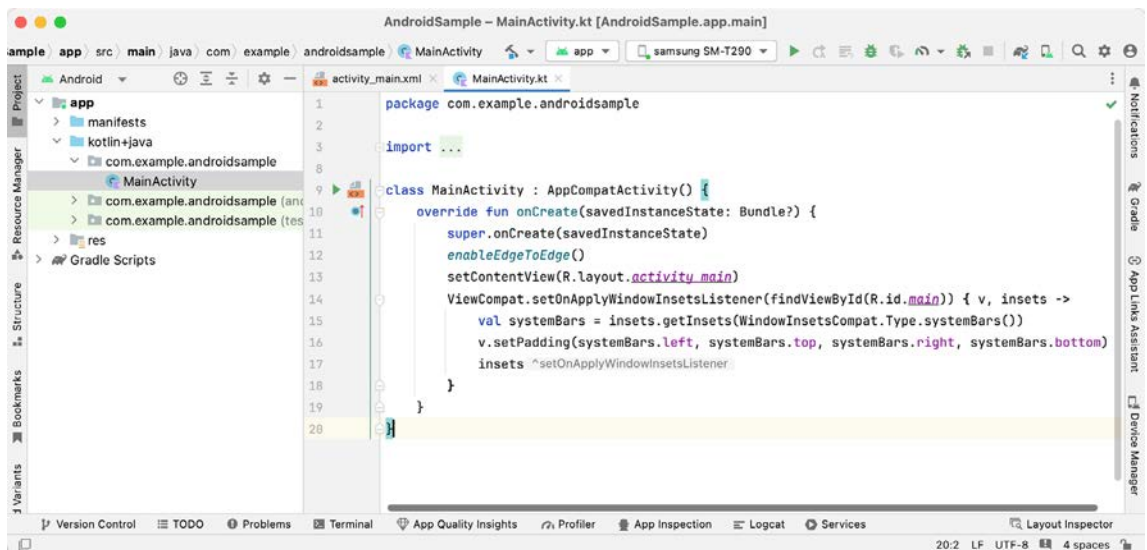


Figure 3-4

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:

Creating an Example Android App in Android Studio

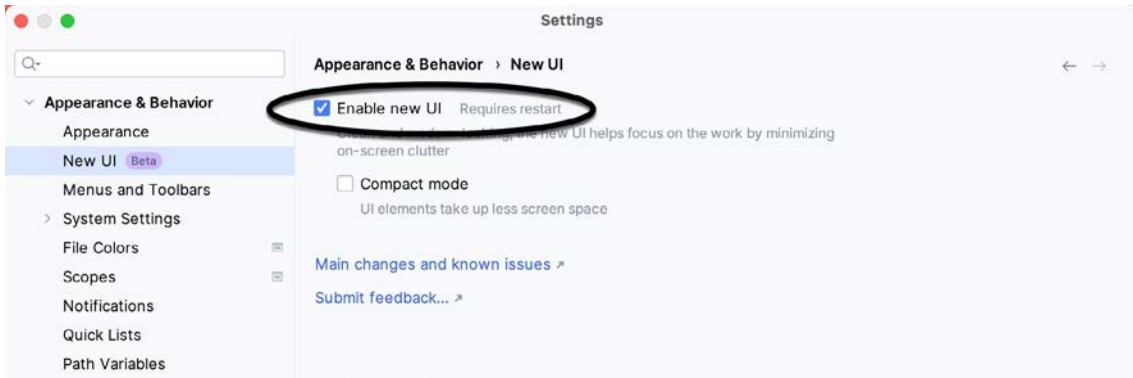


Figure 3-5

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 3-5 below:

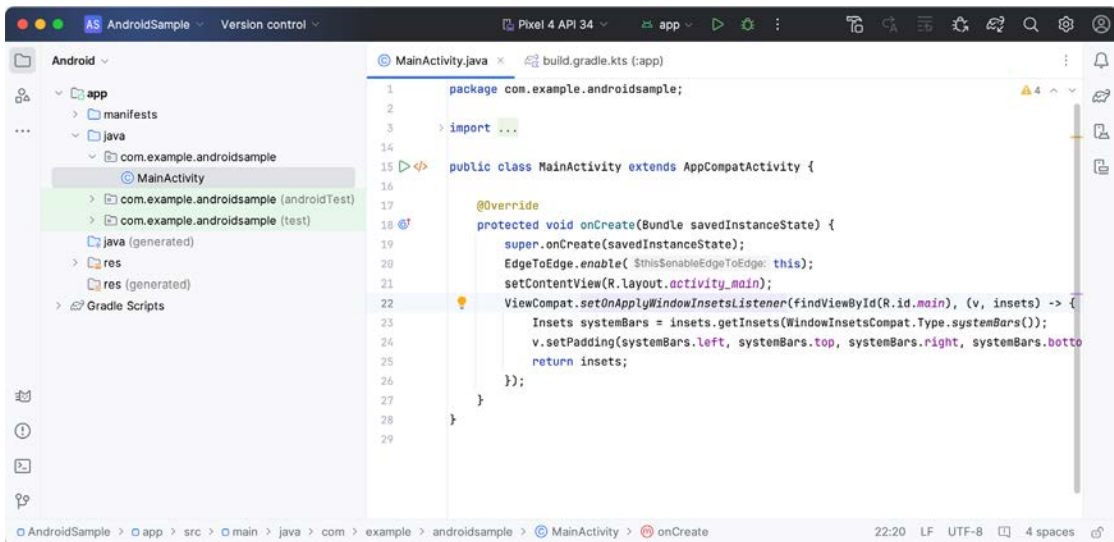


Figure 3-6

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 3-6. If the panel is not currently in Android mode, use the menu to switch mode:

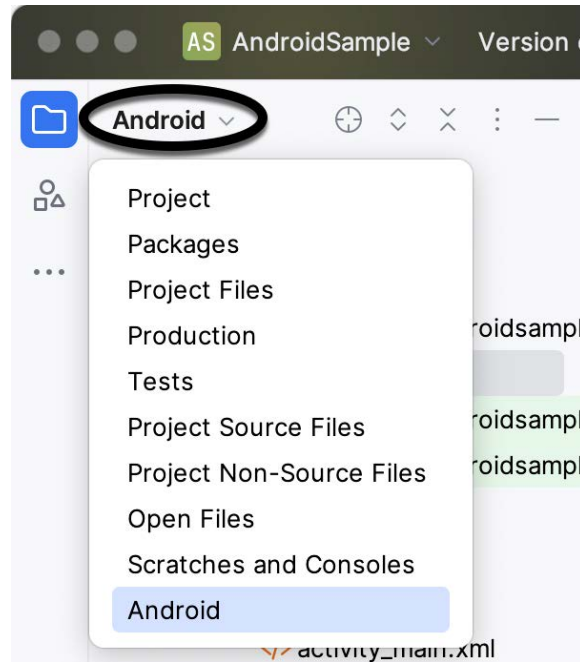


Figure 3-7

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:

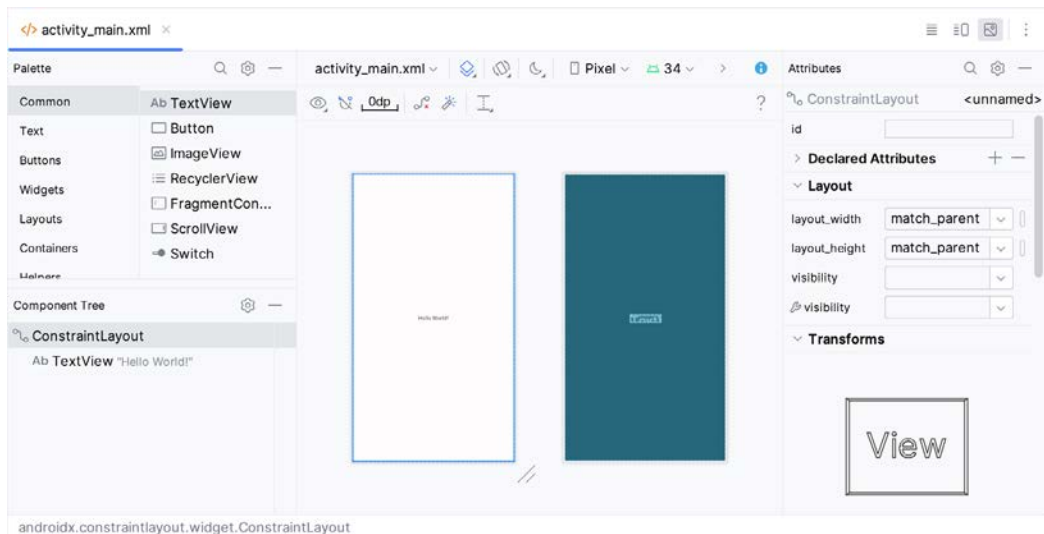




Figure 3-8

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 3-8:

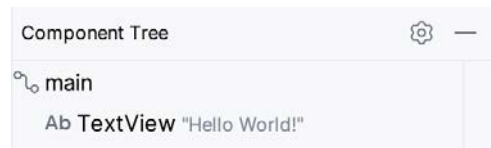


Figure 3-9

As we can see from the component tree hierarchy, the user interface layout consists of a *ConstraintLayout* parent called *main* 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 3-9). If necessary, re-enable Autoconnect mode by clicking on this button.

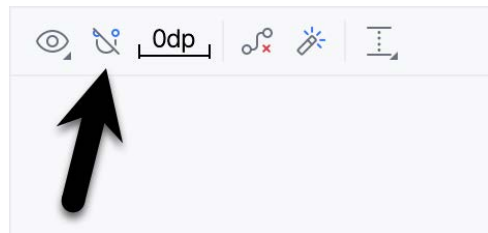


Figure 3-10

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 3-10, for example, the *Button* view is currently selected within the *Buttons* category:

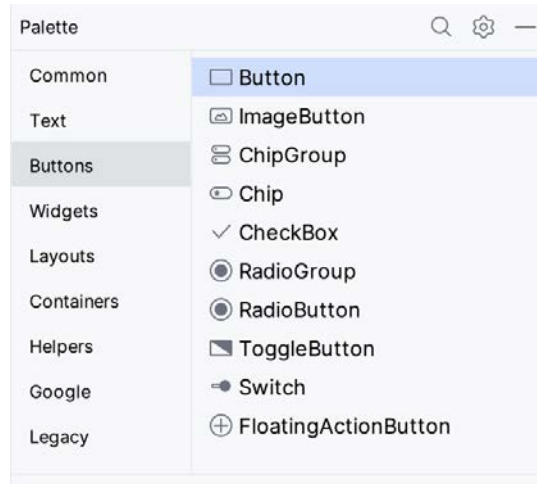


Figure 3-11

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:

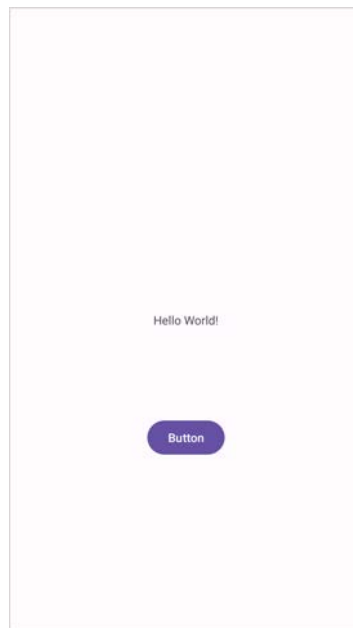


Figure 3-12

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 3-12:

Creating an Example Android App in Android Studio



Figure 3-13

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

Just in case the Autoconnect system failed to set all of the layout connections, click on the Infer Constraints button (Figure 3-13) to add any missing constraints to the layout:

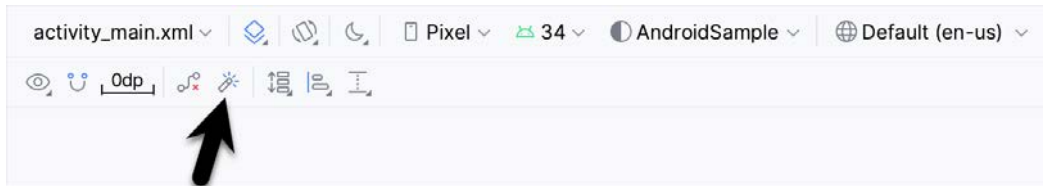


Figure 3-14

It is important to explain the warning button in the top right-hand corner of the Layout Editor tool, as indicated in Figure 3-14. 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 3-15) will appear, describing the nature of the problems:

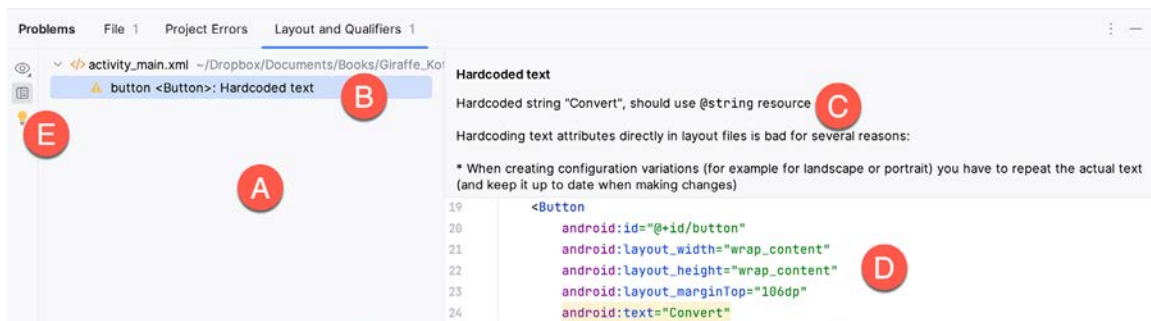


Figure 3-16

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 3-16:

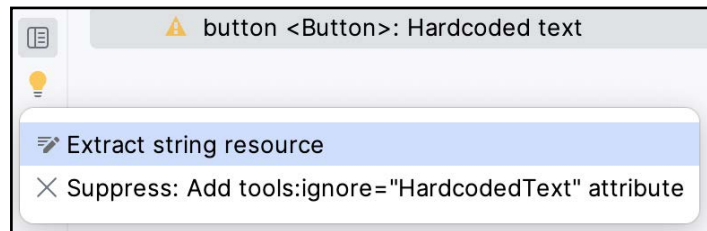


Figure 3-17

After selecting this option, the *Extract Resource* panel (Figure 3-17) 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:

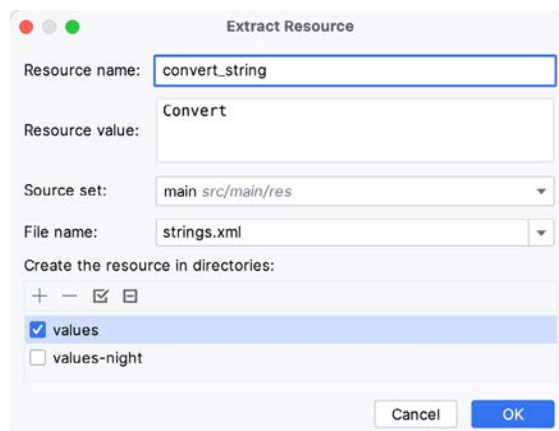


Figure 3-18

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 3-18:

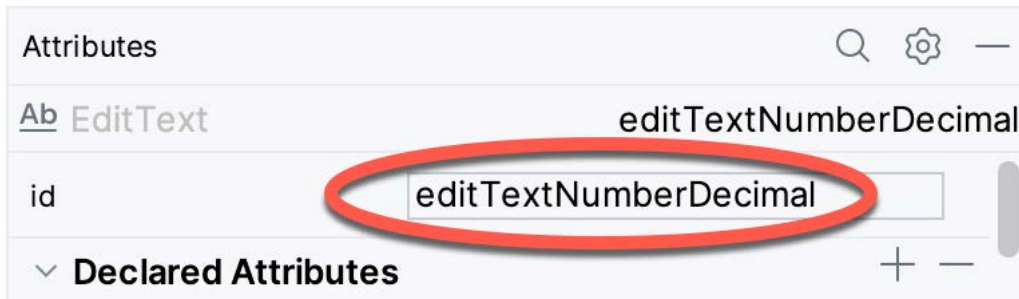


Figure 3-19

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:

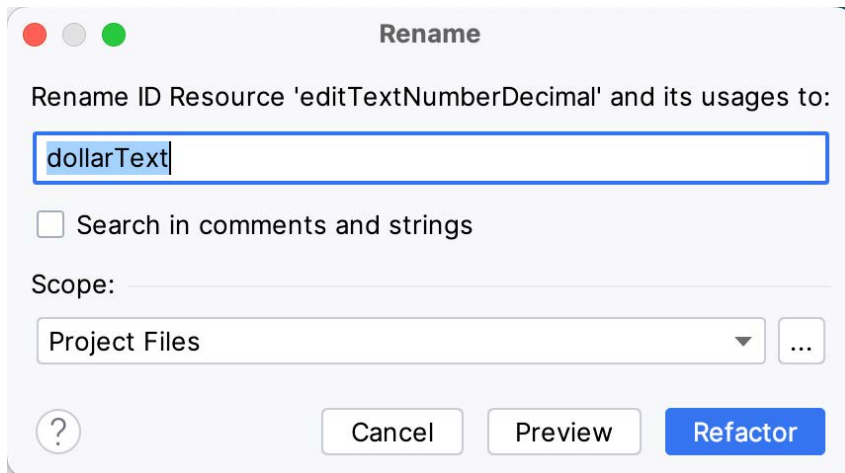


Figure 3-20

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 3-20:

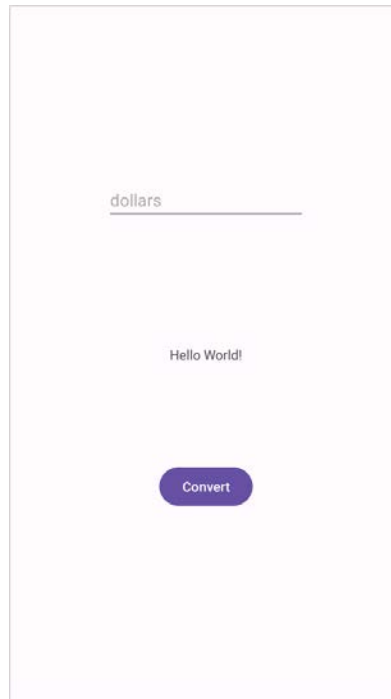


Figure 3-21

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 3-21 below:

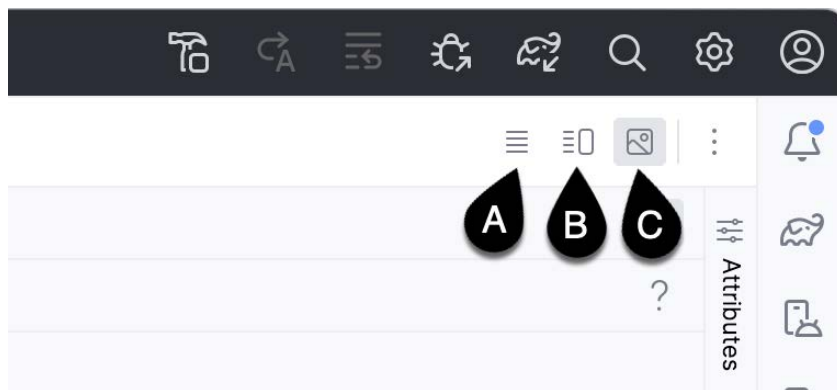


Figure 3-22

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 3-22:

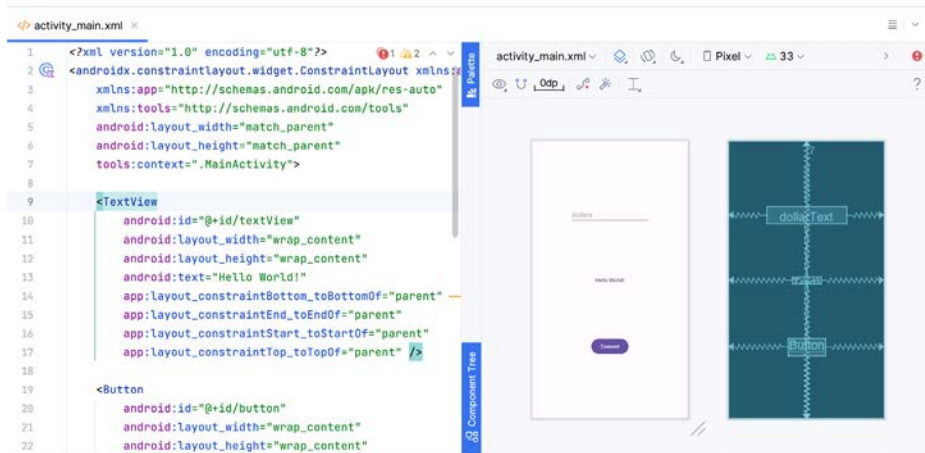


Figure 3-23

The button to the left of the View Modes button (marked B in Figure 3-21 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:id="@+id/main"
    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:

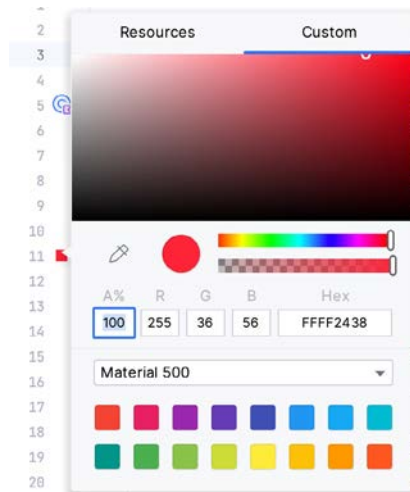


Figure 3-24

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:

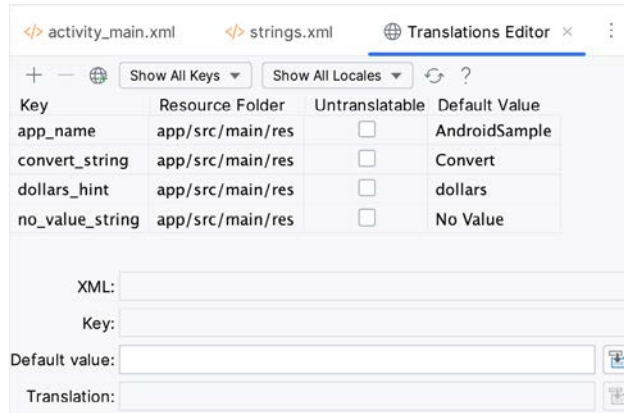


Figure 3-25

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:

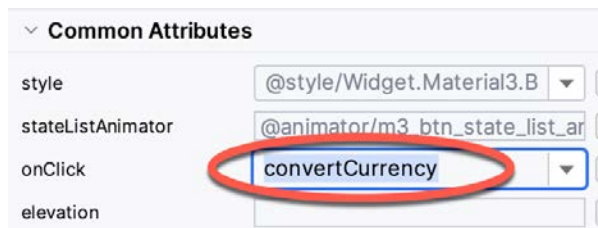


Figure 3-26

Next, double-click on the *MainActivity.java* file in the Project tool window (*app* -> *java* -> *<package name>* -> *MainActivity*) to load it into the code editor and add the code for the *convertCurrency* method to the class file so that it reads as follows, noting that it is also necessary to import some additional Android packages:

```
package com.ebookfrenzy.androidsample;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

.
```



```

import java.util.Locale;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void convertCurrency(View view) {

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

        if (!dollarText.getText().toString().isEmpty()) {

            float dollarValue = Float.parseFloat(dollarText.getText().toString());
            float euroValue = dollarValue * 0.85F;
            textView.setText(String.format(Locale.ENGLISH, "%.2f", euroValue));
        } else {
            textView.setText(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):

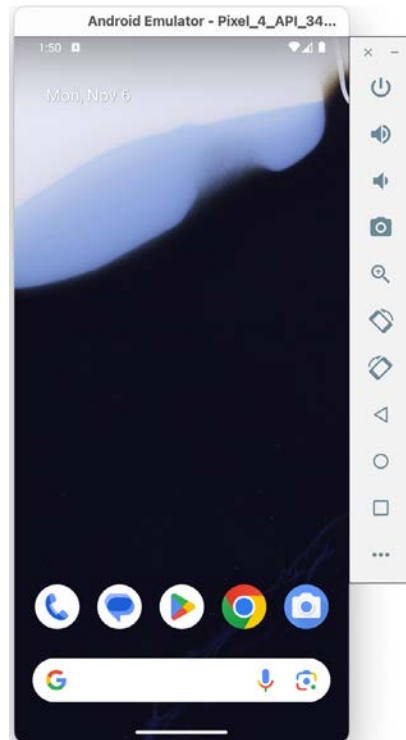


Figure 5-1

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.

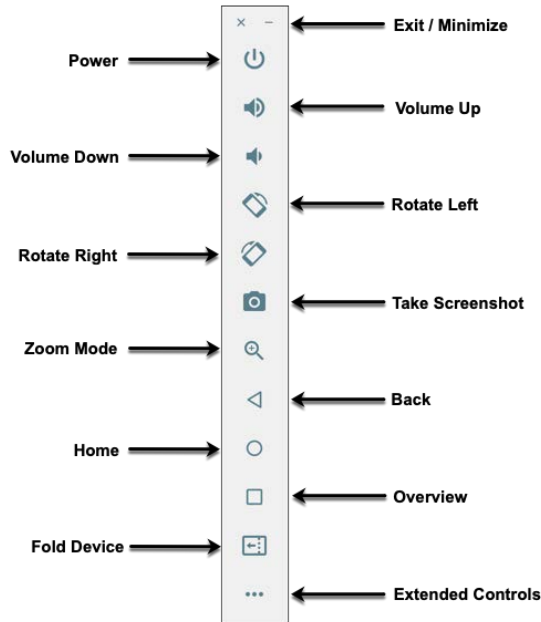


Figure 5-2

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:

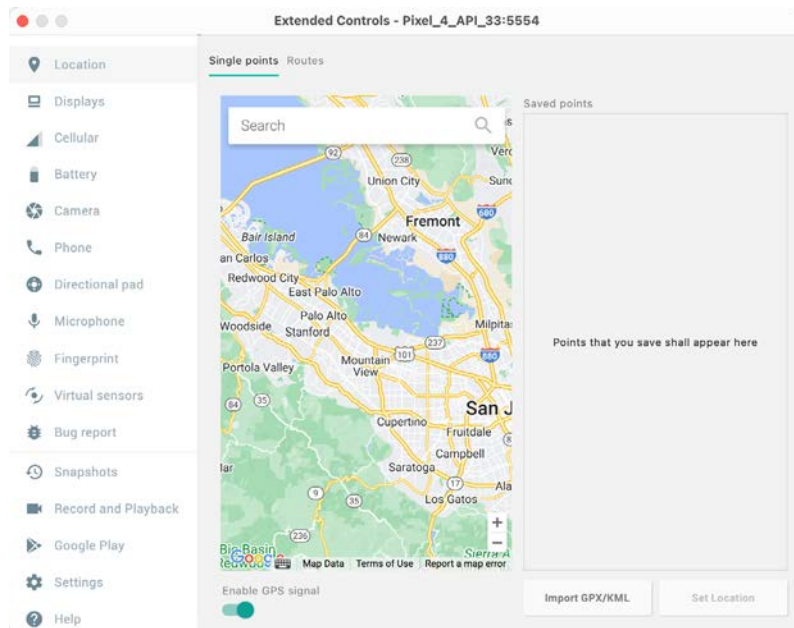


Figure 5-3

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:

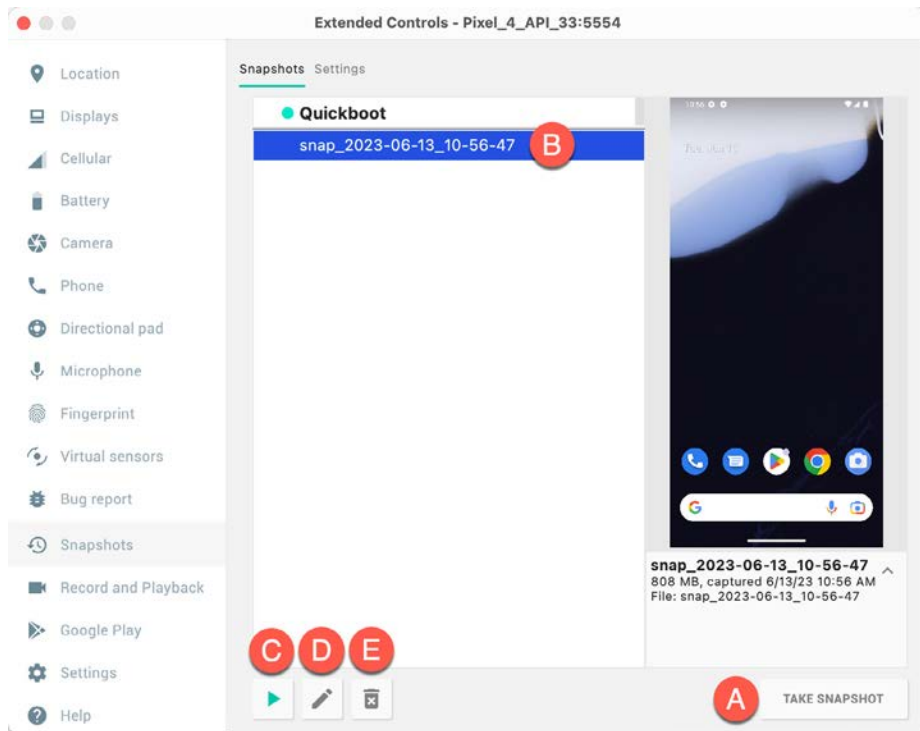


Figure 5-4

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:

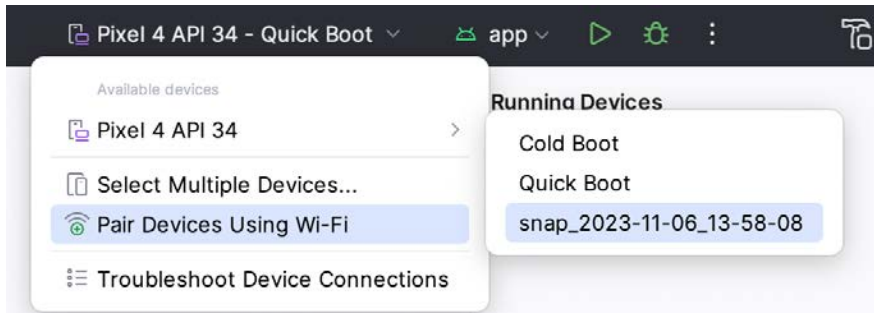


Figure 5-5

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:

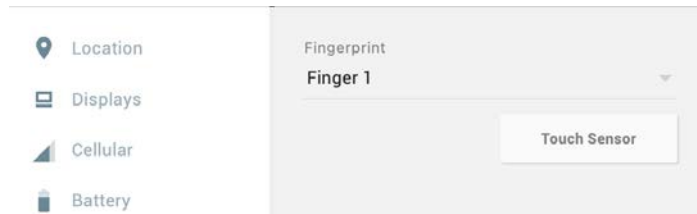


Figure 5-6

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

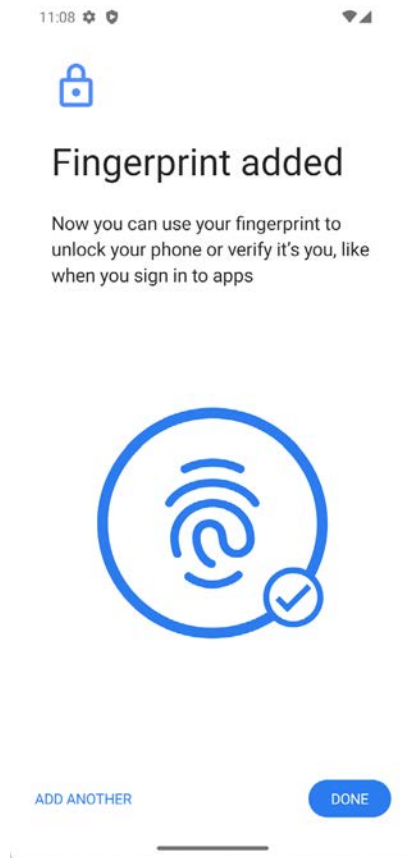


Figure 5-7

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:

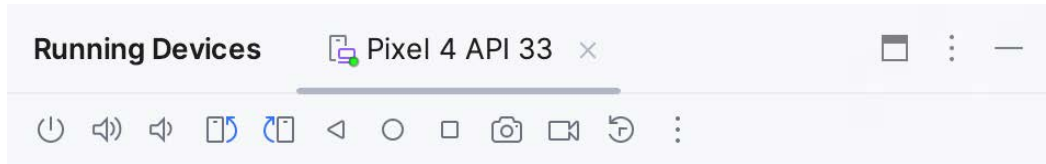


Figure 5-8

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:

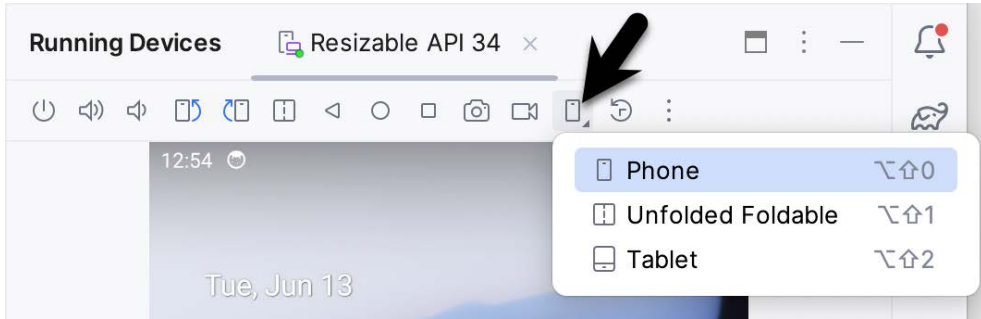


Figure 5-10

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

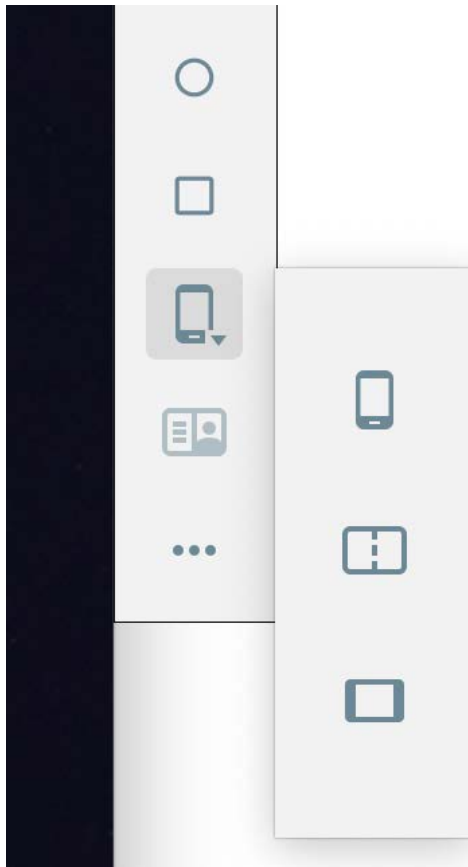


Figure 5-11

Once a foldable display mode has been selected, the Change posture menu may be used to test the app in open, closed, and half-open configurations:

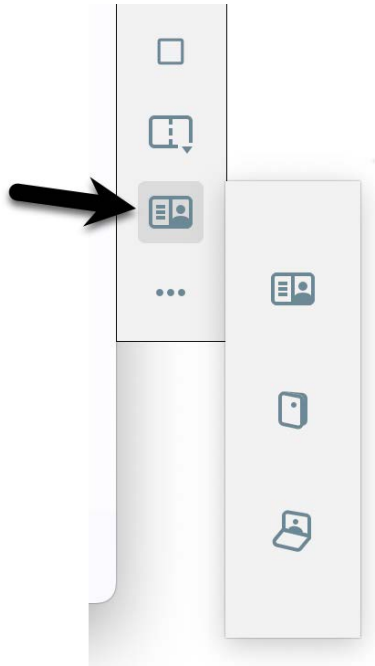


Figure 5-12

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.

7. Testing Android Studio Apps on a Physical Android Device

While much can be achieved by testing applications using an Android Virtual Device (AVD), there is no substitute for performing real-world application testing on a physical Android device, and some Android features are only available on physical Android devices.

Communication with both AVD instances and connected Android devices is handled by the *Android Debug Bridge (ADB)*. This chapter explains how to configure the adb environment to enable application testing on an Android device with macOS, Windows, and Linux-based systems.

7.1 An Overview of the Android Debug Bridge (ADB)

The primary purpose of the ADB is to facilitate interaction between a development system, in this case, Android Studio, and both AVD emulators and Android devices to run and debug applications. ADB allows you to connect to devices via WiFi or USB cable.

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

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

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

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

7.2 Enabling USB Debugging ADB on Android Devices

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

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

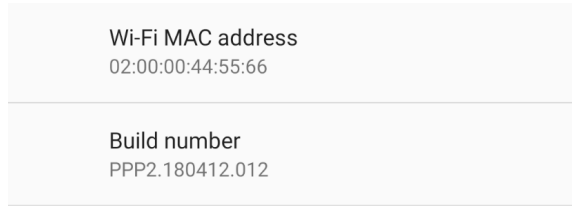


Figure 7-1

3. Return to the main Settings screen and note the appearance of a new option titled Developer options (on newer versions of Android, this option is listed on the System settings screen). Select this option, and on the resulting screen, locate the USB debugging option as illustrated in Figure 7-2:

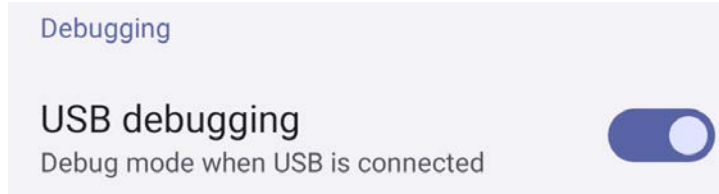


Figure 7-2

4. Enable the USB debugging option and tap the Allow button when confirmation is requested.

If you use a Samsung Galaxy device, you may need to turn off the Auto Blocker feature in the Settings app before enabling the debugging option.

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

7.2.1 macOS ADB Configuration

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

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

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

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

If the device is listed as *offline*, go to the Android device and check for the dialog shown in Figure 7-3 seeking permission to *Allow USB debugging*. Enable the checkbox next to the option that reads *Always allow from this computer* before clicking OK.



Figure 7-3

Repeating the `adb devices` command should now list the device as being available:

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

If the device is not listed, try logging out and back into the macOS desktop and rebooting the system if the problem persists.

7.2.2 Windows ADB Configuration

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

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

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

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

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

```
adb devices
```

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

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

If the device is listed as *offline* or *unauthorized*, go to the device display and check for the dialog shown in Figure 7-3 seeking permission to *Allow USB debugging*. Enable the checkbox next to the option that reads *Always allow from this computer* before clicking *OK*. Repeating the `adb devices` command should now list the device as being ready:

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

If the device is not listed, execute the following commands to restart the ADB server:

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

Testing Android Studio Apps on a Physical Android Device

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

```
android update adb
```

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

7.2.3 Linux adb Configuration

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

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

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

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

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

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

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

If the device is listed as *offline* or *unauthorized*, go to the Android device and check for the dialog shown in Figure 7-3 seeking permission to *Allow USB debugging*.

7.3 Resolving USB Connection Issues

If you are unable to successfully connect to the device using the above steps, display the run target menu (Figure 7-4) and select the *Troubleshoot Device Connections* option:

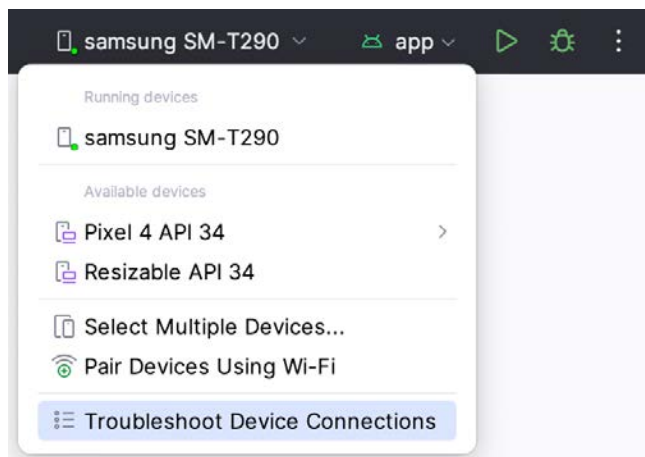


Figure 7-4

The connection assistant will scan for devices and report problems and possible solutions.

7.4 Enabling Wireless Debugging on Android Devices

Follow steps 1 through 3 from section 7.2 above, this time enabling the Wireless Debugging option as shown in Figure 7-5:

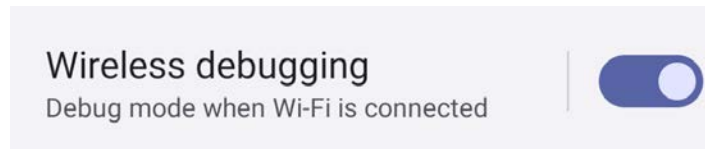


Figure 7-5

Next, tap the above Wireless debugging entry to display the screen shown in Figure 7-6:

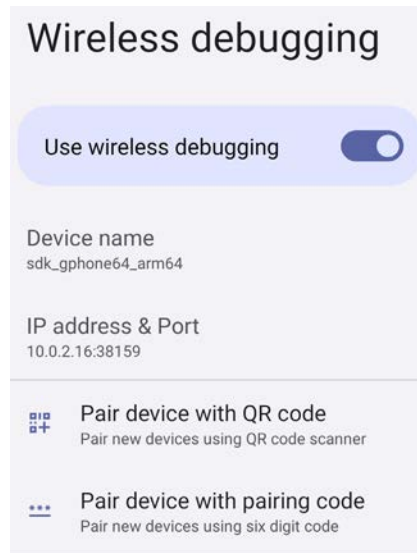


Figure 7-6

If your device has a camera, select *Pair device with QR code*, otherwise select the *Pair device with pairing code* option. Depending on your selection, the Settings app will either start a camera session or display a pairing code, as shown in Figure 7-7:

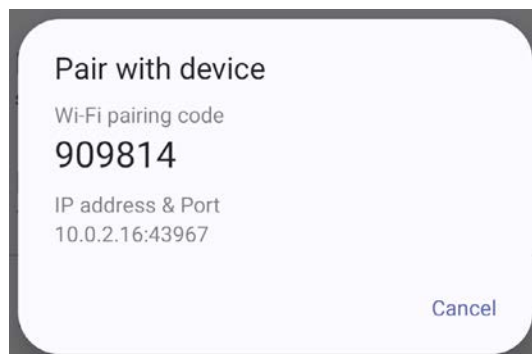


Figure 7-7

Testing Android Studio Apps on a Physical Android Device

With an option selected, return to Android Studio and select the *Pair Devices Using WiFi* option from the run target menu as illustrated in Figure 7-8:

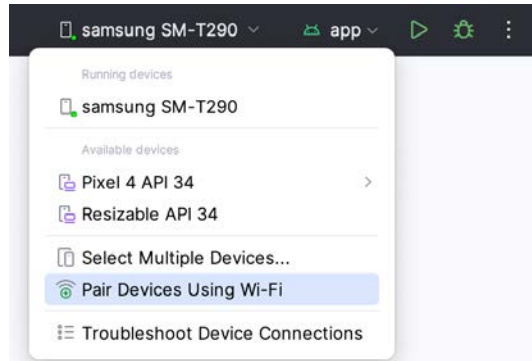


Figure 7-8

In the pairing dialog, select either *Pair using QR code* or *Pair using pairing code* depending on your previous selection in the Settings app on the device:



Figure 7-9

Either scan the QR code using the Android device or enter the pairing code displayed on the device screen into the Android Studio dialog (Figure 7-10) to complete the pairing process:



Figure 7-10

If the pairing process fails, try rebooting both the development system and the Android device and try again.

7.5 Testing the adb Connection

Assuming that the adb configuration has been successful on your chosen development platform, the next step is to try running the test application created in the chapter entitled “*Creating an Example Android App in Android Studio*” on the device. Launch Android Studio, open the AndroidSample project, and verify that the device appears in the device selection menu as highlighted in Figure 7-11:

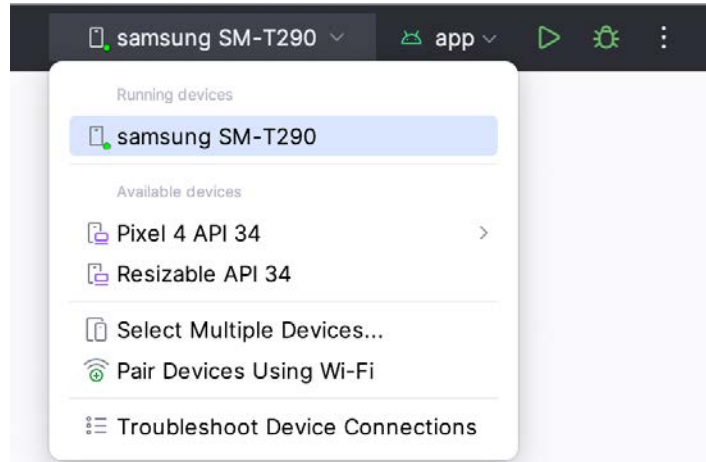


Figure 7-11

Select the device from the list and click the run button to install and run the app.

7.6 Device Mirroring

Device mirroring allows you to run an app on a physical device while viewing the display within Android Studio's Running Devices tool window. In other words, although your app is running on a physical device, it appears within Android Studio in the same way as an AVD instance.

With a device connected to Android Studio, display the *Running Devices* tool window and click the *Device Mirror settings* link to display the Settings dialog. Within the Settings dialog, enable the mirroring of physical Android devices and click OK. The next time the app is run, Android Studio will mirror the display of the physical device in the Running Devices tool window.

7.7 Summary

While the Android Virtual Device emulator provides an excellent testing environment, it is essential to remember that there is no real substitute for ensuring an application functions correctly on a physical Android device.

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

10. The Anatomy of an Android App

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

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

10.1 Android Activities

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

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

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

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

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

10.2 Android Fragments

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

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

screen.

10.3 Android Intents

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

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

10.4 Broadcast Intents

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

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

10.5 Broadcast Receivers

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

10.6 Android Services

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

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

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

10.7 Content Providers

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

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

10.8 The Application Manifest

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

10.9 Application Resources

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

10.10 Application Context

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

10.11 Summary

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

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

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

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

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

12. Understanding Android Application and Activity Lifecycles

In earlier chapters, we learned that Android applications run within processes and comprise multiple components in the form of activities, services, and broadcast receivers. This chapter aims to expand on this knowledge by looking at the lifecycle of applications and activities within the Android runtime system.

Regardless of the fanfare about how much memory and computing power resides in the mobile devices of today compared to the desktop systems of yesterday, it is important to keep in mind that these devices are still considered to be “resource constrained” by the standards of modern desktop and laptop-based systems, particularly in terms of memory. As such, a key responsibility of the Android system is to ensure that these limited resources are managed effectively and that the operating system and the applications running on it remain responsive to the user at all times. To achieve this, Android is given complete control over the lifecycle and state of the processes in which the applications run and the individual components that comprise those applications.

An important factor in developing Android applications, therefore, is to understand Android’s application and activity lifecycle management models of Android, and how an application can react to the state changes likely to be imposed upon it during its execution lifetime.

12.1 Android Applications and Resource Management

The operating system views each running Android application as a separate process. If the system identifies that resources on the device are reaching capacity, it will take steps to terminate processes to free up memory.

When determining which process to terminate to free up memory, the system considers both the *priority* and *state* of all currently running processes, combining these factors to create what is referred to by Google as an *importance hierarchy*. Processes are then terminated, starting with the lowest priority and working up the hierarchy until sufficient resources have been liberated for the system to function.

12.2 Android Process States

Processes host applications, and applications are made up of components. Within an Android system, the current state of a process is defined by the highest-ranking active component within the application it hosts. As outlined in Figure 12-1, a process can be in one of the following five states at any given time:

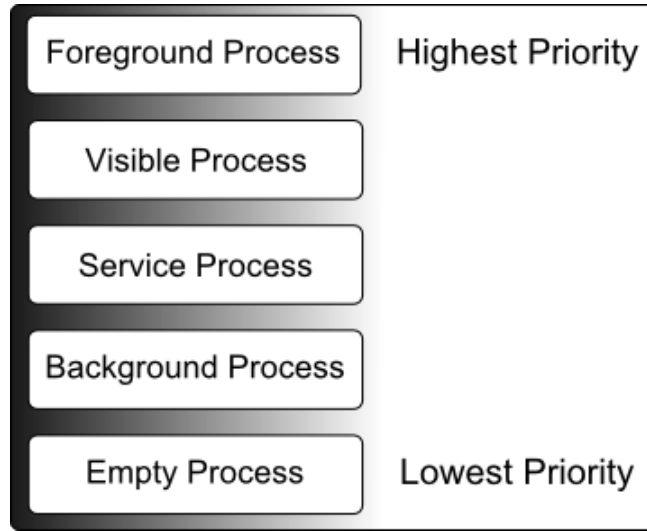


Figure 12-1

12.2.1 Foreground Process

These processes are assigned the highest level of priority. At any one time, there are unlikely to be more than one or two foreground processes active, which are usually the last to be terminated by the system. A process must meet one or more of the following criteria to qualify for foreground status:

- Hosts an activity with which the user is currently interacting.
- Hosts a Service connected to the activity with which the user is interacting.
- Hosts a Service that has indicated, via a call to *startForeground()*, that termination would disrupt the user experience.
- Hosts a Service executing either its *onCreate()*, *onResume()*, or *onStart()* callbacks.
- Hosts a Broadcast Receiver that is currently executing its *onReceive()* method.

12.2.2 Visible Process

A process containing an activity that is visible to the user but is not the activity with which the user is interacting is classified as a “visible process”. This is typically the case when an activity in the process is visible to the user, but another activity, such as a partial screen or dialog, is in the foreground. A process is also eligible for visible status if it hosts a Service that is, itself, bound to a visible or foreground activity.

12.2.3 Service Process

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

12.2.4 Background Process

A process that contains one or more activities that are not currently visible to the user and does not host a Service that qualifies for *Service Process* status. Processes that fall into this category are at high risk of termination if additional memory needs to be freed for higher-priority processes. Android maintains a dynamic list of background processes, terminating processes in chronological order such that processes that were the least recently in the foreground are killed first.

12.2.5 Empty Process

Empty processes no longer contain active applications and are held in memory, ready to serve as hosts for newly launched applications. This is analogous to keeping the doors open and the engine running on a bus in anticipation of passengers arriving. Such processes are considered the lowest priority and are the first to be killed to free up resources.

12.3 Inter-Process Dependencies

Determining the highest priority process is more complex than outlined in the preceding section because processes can often be interdependent. As such, when determining the priority of a process, the Android system will also consider whether the process is in some way serving another process of higher priority (for example, a service process acting as the content provider for a foreground process). As a basic rule, the Android documentation states that a process can never be ranked lower than another process that it is currently serving.

12.4 The Activity Lifecycle

As we have previously determined, the state of an Android process is primarily determined by the status of the activities and components that make up the application it hosts. It is important to understand, therefore, that these activities also transition through different states during the execution lifetime of an application. The current state of an activity is determined, in part, by its position in something called the Activity Stack.

12.5 The Activity Stack

The runtime system maintains an *Activity Stack* for each application running on an Android device. When an application is launched, the first of the application's activities to be started is placed onto the stack. When a second activity is started, it is placed on the top of the stack, and the previous activity is *pushed* down. The activity at the top of the stack is called the *active* (or *running*) activity. When the active activity exits, it is *popped* off the stack by the runtime and the activity located immediately beneath it in the stack becomes the current active activity. For example, the activity at the top of the stack might exit because the task for which it is responsible has been completed. Alternatively, the user may have selected a “Back” button on the screen to return to the previous activity, causing the current activity to be popped off the stack by the runtime system and destroyed. A visual representation of the Android Activity Stack is illustrated in Figure 12-2.

As shown in the diagram, new activities are pushed onto the top of the stack when they are started. The current active activity is located at the top of the stack until it is either pushed down the stack by a new activity or popped off the stack when it exits or the user navigates to the previous activity. If resources become constrained, the runtime will kill activities, starting with those at the bottom of the stack.

The Activity Stack is what is referred to in programming terminology as a Last-In-First-Out (LIFO) stack in that the last item to be pushed onto the stack is the first to be popped off.

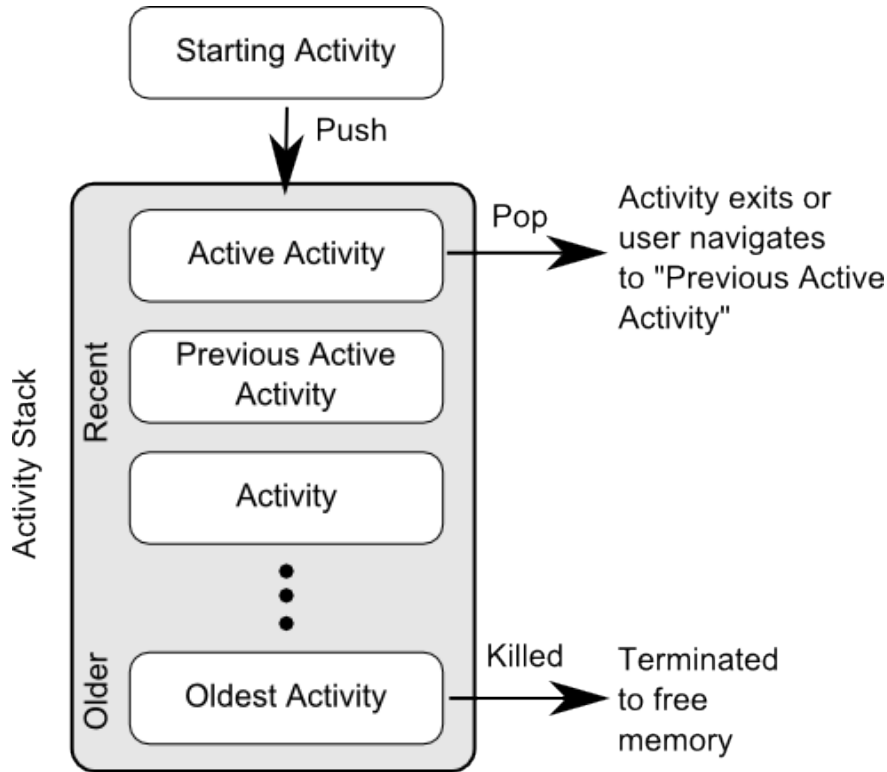


Figure 12-2

12.6 Activity States

An activity can be in one of several states during the course of its execution within an application:

- **Active / Running** – The activity is at the top of the Activity Stack, is the foreground task visible on the device screen, has focus, and is currently interacting with the user. This is the least likely activity to be terminated in the event of a resource shortage.
- **Paused** – The activity is visible to the user but does not currently have focus (typically because the current *active* activity partially obscures this activity). Paused activities are held in memory, remain attached to the window manager, retain all state information, and can quickly be restored to active status when moved to the top of the Activity Stack.
- **Stopped** – The activity is currently not visible to the user (in other words, it is obscured on the device display by other activities). As with paused activities, it retains all state and member information but is at higher risk of termination in low-memory situations.
- **Killed** – The runtime system has terminated the activity to free up memory and is no longer present on the Activity Stack. Such activities must be restarted if required by the application.

12.7 Configuration Changes

So far in this chapter, we have looked at two causes for the change in the state of an Android activity, namely the movement of an activity between the foreground and background and the termination of an activity by the runtime system to free up memory. In fact, there is a third scenario in which the state of an activity can dramatically change, which involves a change to the device configuration.

By default, any configuration change that impacts the appearance of an activity (such as rotating the orientation of the device between portrait and landscape or changing a system font setting) will cause the activity to be destroyed and recreated. The reasoning behind this is that such changes affect resources such as the layout of the user interface, and destroying and recreating impacted activities is the quickest way for an activity to respond to the configuration change. It is, however, possible to configure an activity so that the system does not restart it in response to specific configuration changes.

12.8 Handling State Change

It should be clear from this chapter that an application and, by definition, the components contained therein will transition through many states during its lifespan. Of particular importance is the fact that these state changes (up to and including complete termination) are imposed upon the application by the Android runtime subject to the user's actions and the availability of resources on the device.

In practice, however, these state changes are not imposed entirely without notice, and an application will, in most circumstances, be notified by the runtime system of the changes and given the opportunity to react accordingly. This will typically involve saving or restoring both internal data structures and user interface state, thereby allowing the user to switch seamlessly between applications and providing at least the appearance of multiple concurrently running applications.

Android provides two ways to handle the changes to the lifecycle states of the objects within an app. One approach involves responding to state change method calls from the operating system and is covered in detail in the next chapter entitled *“Handling Android Activity State Changes”*.

A new approach that Google recommends involves the lifecycle classes included with the Jetpack Android Architecture components, introduced in *“Modern Android App Architecture with Jetpack”* and explained in more detail in the chapter entitled *“Working with Android Lifecycle-Aware Components”*.

12.9 Summary

Mobile devices are typically considered to be resource constrained, particularly in terms of onboard memory capacity. Consequently, a prime responsibility of the Android operating system is to ensure that applications, and the operating system in general, remain responsive to the user.

Applications are hosted on Android within processes. Each application, in turn, comprises components in the form of activities and Services.

The Android runtime system has the power to terminate both processes and individual activities to free up memory. Process state is considered by the runtime system when deciding whether a process is a suitable candidate for termination. The state of a process largely depends upon the status of the activities hosted by that process.

The key message of this chapter is that an application moves through various states during its execution lifespan and has very little control over its destiny within the Android runtime environment. Those processes and activities not directly interacting with the user run a higher risk of termination by the runtime system. An essential element of Android application development, therefore, involves the ability of an application to respond to state change notifications from the operating system.

17. A Guide to the Android Studio Layout Editor Tool

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

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

17.1 Basic vs. Empty Views Activity Templates

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

When creating a new Android Studio project, several templates are available to be used as the starting point for the user interface of the main activity. The most basic templates are the Basic Views Activity and Empty Views Activity templates. Although these seem similar at first glance, there are considerable differences between the two options. To see these differences within the layout editor, use the View Options menu to enable Show System UI, as shown in Figure 17-1 below:

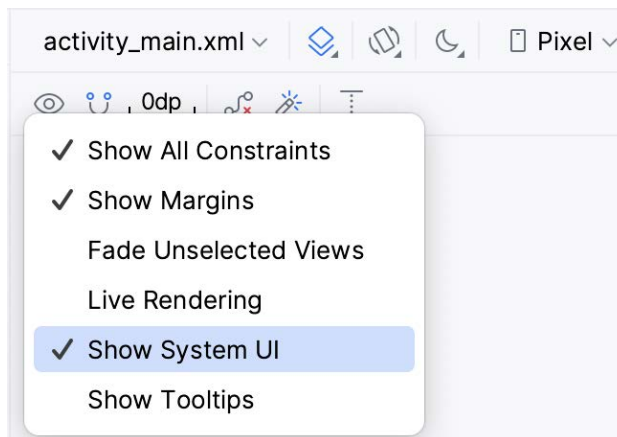


Figure 17-1

The Empty Views Activity template creates a single layout file consisting of a `ConstraintLayout` manager instance containing a `TextView` object, as shown in Figure 17-2:



Figure 17-2

The Basic Views Activity, on the other hand, consists of multiple layout files. The top-level layout file has a CoordinatorLayout as the root view, a configurable app bar (which contains a toolbar) that appears across the top of the device screen (marked A in Figure 17-3), and a floating action button (the email button marked B). In addition to these items, the *activity_main.xml* layout file contains a reference to a second file named *content_main.xml* containing the content layout (marked C):

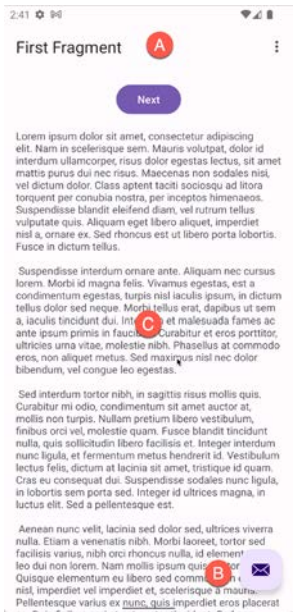


Figure 17-3

The Basic Views Activity contains layouts for two screens containing a button and a text view. This template aims to demonstrate how to implement navigation between multiple screens within an app. If an unmodified app using the Basic Views Activity template were to be run, the first of these two screens would appear (marked A in Figure 17-4). Pressing the Next button would navigate to the second screen (B), which, in turn, contains a button to return to the first screen:

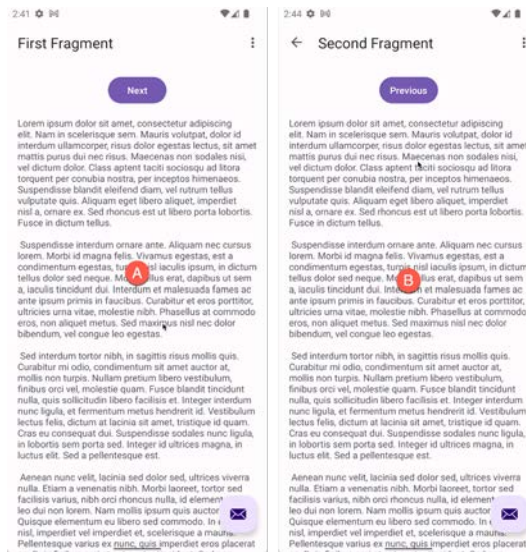


Figure 17-4

This app behavior uses two Android features referred to as *fragments* and *navigation*, which will be covered starting with the chapters entitled “An Introduction to Android Fragments” and “An Overview of the Navigation Architecture Component” respectively.

The `content_main.xml` file contains a special fragment, known as a Navigation Host Fragment which allows different content to be switched in and out of view depending on the settings configured in the `res -> layout -> nav_graph.xml` file. In the case of the Basic Views Activity template, the `nav_graph.xml` file is configured to switch between the user interface layouts defined in the `fragment_first.xml` and `fragment_second.xml` files based on the Next and Previous selections made by the user.

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

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

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

1. Double-click on the main `activity_main.xml` layout file in the Project tool window under `app -> res -> layout` to load it into the Layout Editor. With the layout loaded into the Layout Editor tool, select the floating action button and tap the keyboard `Delete` key to remove the object from the layout.
2. Locate and edit the Java code for the activity (located under `app -> java -> <package name> -> <activity class name>`) and remove the floating action button code from the `onCreate` method as follows:

A Guide to the Android Studio Layout Editor Tool

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

    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    setSupportActionBar(binding.toolbar);

    NavController navController = Navigation.findNavController(this, R.id.nav_
host_fragment_content_main);
    appBarConfiguration = new AppBarConfiguration.Builder(navController.
getGraph()).build();
    NavigationUI.setupActionBarWithNavController(this, navController,
appBarConfiguration);

    binding.fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_
LONG)
                .setAction("Action", null).show();
        }
    });
}
```

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

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

If you need to use the Basic Views Activity template but need neither the navigation features nor the second content fragment, follow these steps:

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

```
public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
}
```

```

binding.buttonFirst.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        NavHostFragment.findNavController(FirstFragment.this)
            .navigate(R.id.action_FirstFragment_to_SecondFragment);
    }
});
}

```

17.2 The Android Studio Layout Editor

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

Under the surface, the Layout Editor tool constructs an XML resource file containing the definition of the user interface that is being designed. As such, the Layout Editor tool operates in three distinct modes: Design, Code, and Split.

17.3 Design Mode

In design mode, the user interface can be visually manipulated by directly working with the view palette and the graphical representation of the layout. Figure 17-5 highlights the key areas of the Android Studio Layout Editor tool in design mode:

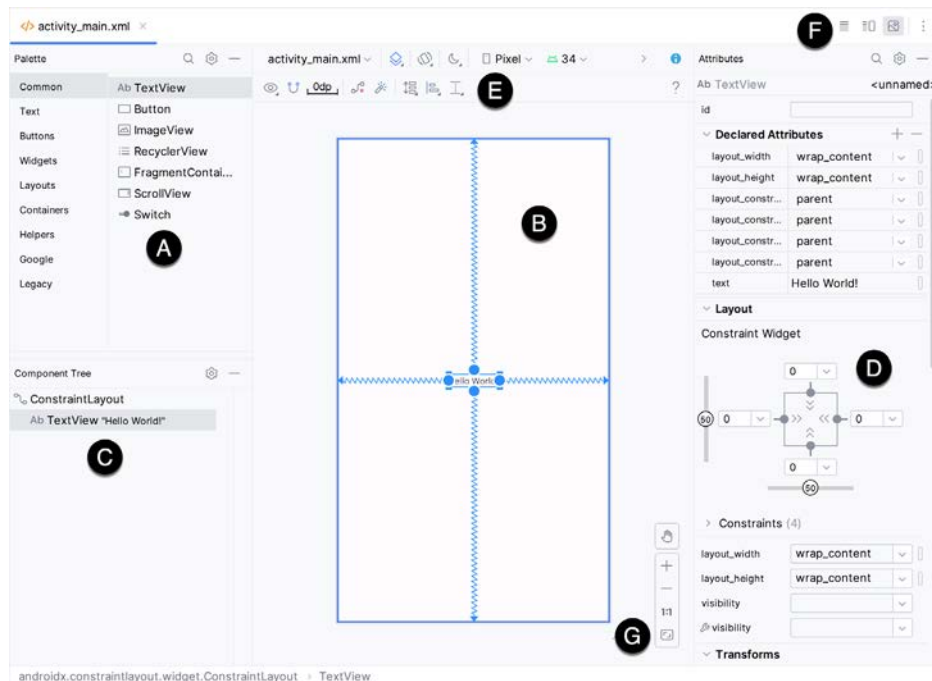


Figure 17-5

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

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

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

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

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

F – Mode Switching Controls – These three buttons provide a way to switch back and forth between the Layout Editor tool’s Design, Code, and Split modes.

G - Zoom and Pan Controls - This control panel allows you to zoom in and out of the design canvas, grab the canvas, and pan around to find obscured areas when zoomed in.

17.4 The Palette

The Layout Editor palette is organized into two panels designed to make it easy to locate and preview view components for addition to a layout design. The category panel (marked A in Figure 17-6) lists the different categories of view components supported by the Android SDK. When a category is selected from the list, the second panel (B) updates to display a list of the components that fall into that category:

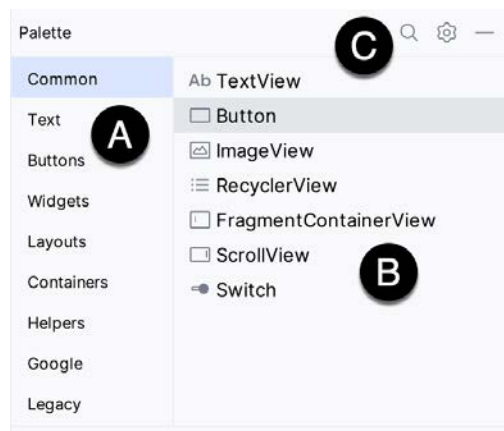


Figure 17-6

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

A search for a specific component within the selected category may be initiated by clicking the search button (marked C in Figure 17-6 above) in the palette toolbar and typing in the component name. As characters are typed, matching results will appear in the component list panel. If you are unsure of the component's category, select the All Results category before or during the search operation.

17.5 Design Mode and Layout Views

The layout editor will appear in Design mode by default, as shown in Figure 17-5 above. This mode provides a visual representation of the user interface. Design mode can be selected by clicking on the button marked C in Figure 17-7:

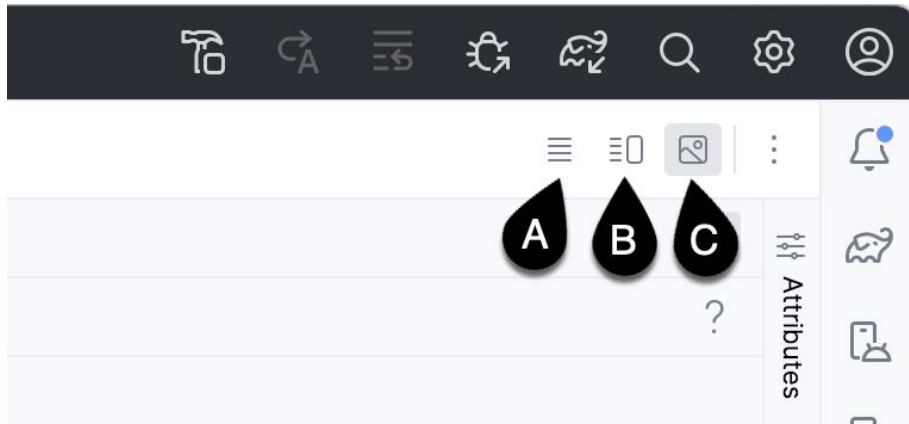


Figure 17-7

When the Layout Editor tool is in Design mode, the layout can be viewed in two ways. The view shown in Figure 17-5 above is the Design view and shows the layout and widgets as they will appear in the running app. A second mode, the Blueprint view, can be shown instead of or concurrently with the Design view. The toolbar menu in Figure 17-8 provides options to display the Design, Blueprint, or both views. Settings are also available to adjust for color blindness. A fifth option, *Force Refresh Layout*, causes the layout to rebuild and redraw. This can be useful when the layout enters an unexpected state or is not accurately reflecting the current design settings:

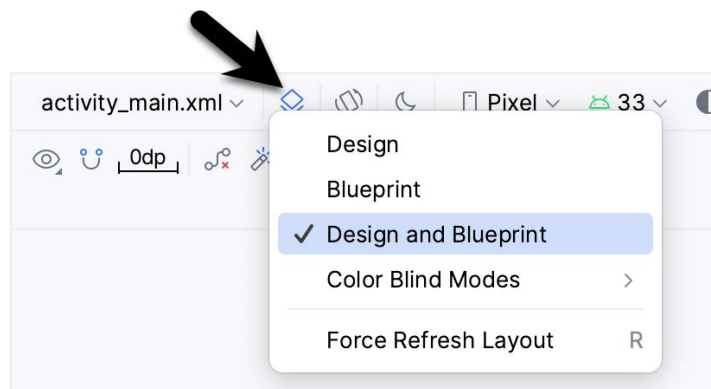


Figure 17-8

Whether to display the layout view, design view, or both is a matter of personal preference. A good approach is to begin with both displayed as shown in Figure 17-9:

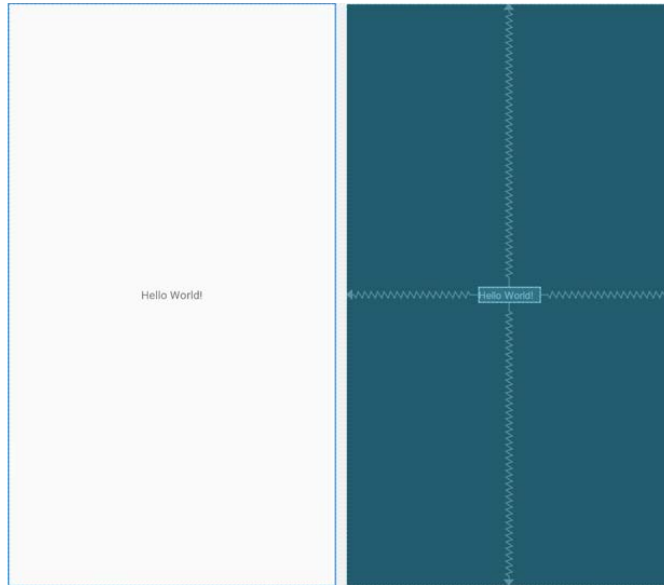


Figure 17-9

17.6 Night Mode

To view the layout in night mode during the design work, select the menu shown in Figure 17-10 below and change the setting to *Night*:

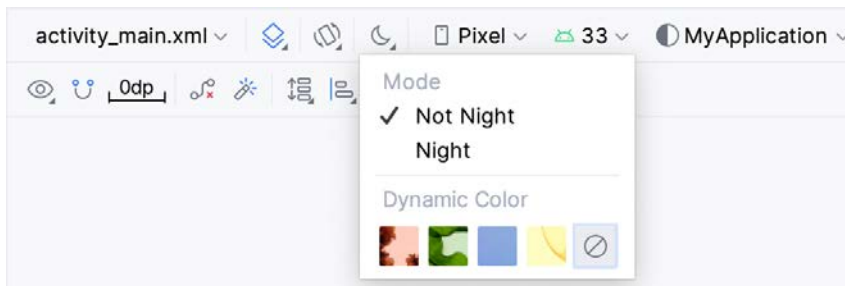


Figure 17-10

The mode menu also includes options for testing dynamic colors, a topic covered in the chapter “*A Material Design 3 Theming and Dynamic Color Tutorial*”.

17.7 Code Mode

It is important to remember when using the Android Studio Layout Editor tool that all it is doing is providing a user-friendly approach to creating XML layout resource files. The underlying XML can be viewed and directly edited during the design process by selecting the button marked A in Figure 17-7 above.

Figure 17-11 shows the Android Studio Layout Editor tool in Code mode, allowing changes to be made to the user interface declaration by modifying the XML:

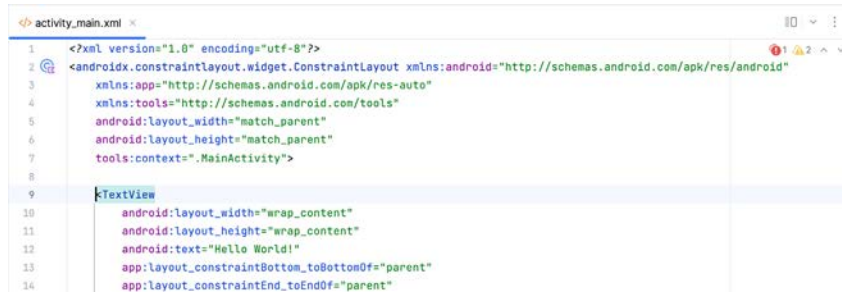


Figure 17-11

17.8 Split Mode

In Split mode, the editor shows the Design and Code views side-by-side, allowing the user interface to be modified visually using the design canvas and making changes directly to the XML declarations. Split mode is selected using the button marked B Figure 17-7 above.

Any changes to the XML are automatically reflected in the design canvas and vice versa. Figure 17-12 shows the editor in Split mode:

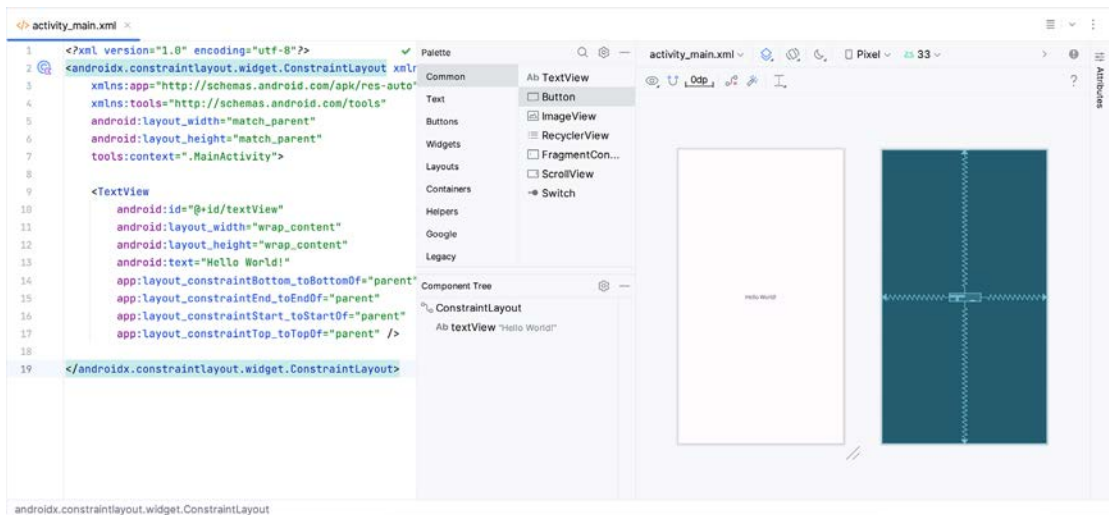


Figure 17-12

17.9 Setting Attributes

The Attributes panel provides access to all available settings for the currently selected component. Figure 17-13, for example, shows some of the attributes for the TextView widget:

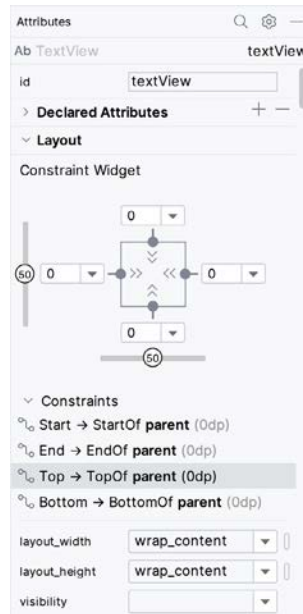


Figure 17-13

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

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

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

Some attributes contain a narrow button to the right of the value field. This indicates that the Resources dialog is available to assist in selecting a suitable property value. To display the dialog, click on the button. The appearance of this button changes to reflect whether or not the corresponding property value is stored in a resource file or hard-coded. If the value is stored in a resource file, the button to the right of the text property field will be filled in to indicate that the value is not hard-coded, as highlighted in Figure 17-14 below:



Figure 17-14

Attributes for which a finite number of valid options are available will present a drop-down menu (Figure 17-15) from which a selection may be made.

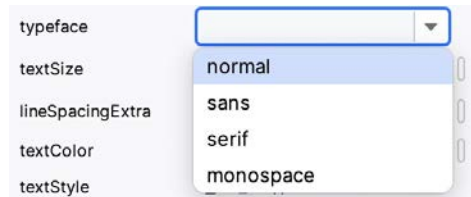


Figure 17-15

A dropper icon can be clicked to display the color selection palette. Similarly, when a flag icon appears, it can be clicked to display a list of options available for the attribute, while an image icon opens the resource manager panel allowing images and other resource types to be selected for the attribute.

17.10 Transforms

The transforms panel within the Attributes tool window (Figure 17-16) provides a set of controls and properties that control visual aspects of the currently selected object in terms of rotation, alpha (used to fade a view in and out), scale (size), and translation (offset from current position):

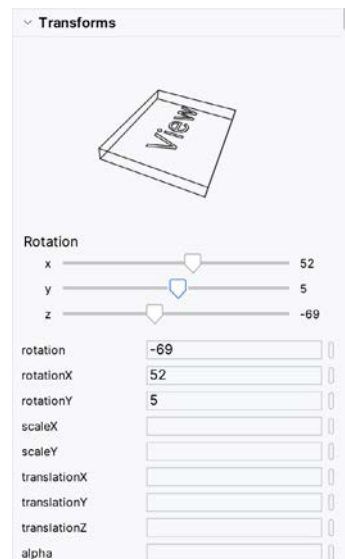


Figure 17-16

The panel contains a visual representation of the view, which updates as properties are changed. These changes are also reflected in the view within the layout canvas.

17.11 Tools Visibility Toggles

When reviewing the content of an Android Studio XML layout file in Code mode, you will notice that many attributes that define how a view appears and behaves begin with the *android:* prefix. This indicates that the attributes are set within the *android* namespace and will take effect when the app is run. The following excerpt from a layout file, for example, sets a variety of attributes on a Button view:

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Button"
```

```
.
```

In addition to the *android* namespace, Android Studio also provides a *tools* namespace. When attributes are set within this namespace, they only take effect within the layout editor preview. While designing a layout, you might find it helpful for an EditText view to display some text but require the view to be blank when the app runs. To achieve this, you would set the text property of the view using the *tools* namespace as follows:

```
<EditText
    android:id="@+id/editTextTextPersonName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="textPersonName"
    tools:text="Sample Text"
```

```
.
```

A tool attribute of this type is set in the Attributes tool window by entering the value into the property fields marked by the wrench icon, as shown in Figure 17-17:



Figure 17-17

Tools attributes are particularly useful for changing the visibility of a view during the design process. A layout may contain a view that is programmatically displayed and hidden when the app runs, depending on user actions. To simulate the hiding of the view, the following tools attribute could be added to the view XML declaration:

```
tools:visibility="invisible"
```

Although the view will no longer be visible when using the invisible setting, it is still present in the layout and occupies the same space it did when it was visible. To make the layout behave as though the view no longer exists, the visibility attribute should be set to *gone* as follows:

```
tools:visibility="gone"
```

In both examples above, the visibility settings only apply within the layout editor and will have no effect in the running app. To control visibility in both the layout editor and running app, the same attribute would be set using the *android* namespace:

```
android:visibility="gone"
```

While these visibility tools attributes are useful, having to manually edit the XML layout file is a cumbersome process. To make it easier to change these settings, Android Studio provides a set of toggles within the layout editor Component Tree panel. To access these controls, click in the margin to the right of the corresponding view in the panel. Figure 17-18, for example, shows the tools visibility toggle controls for a Button view named `myButton`:

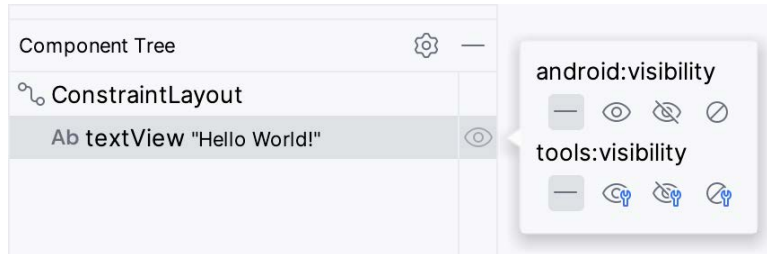


Figure 17-18

These toggles control the visibility of the corresponding view for both the `android` and `tools` namespaces and provide *not set*, *visible*, *invisible* and *gone* options. When conflicting attributes are set (for example, an `android` namespace toggle is set to *visible* while the `tools` value is set to *invisible*), the `tools` namespace takes precedence within the layout preview. When a toggle selection is made, Android Studio automatically adds the appropriate attribute to the XML view element in the layout file.

In addition to the visibility toggles in the Component Tree panel, the layout editor also includes the *tools visibility* and *position* toggle button shown highlighted in Figure 17-19 below:

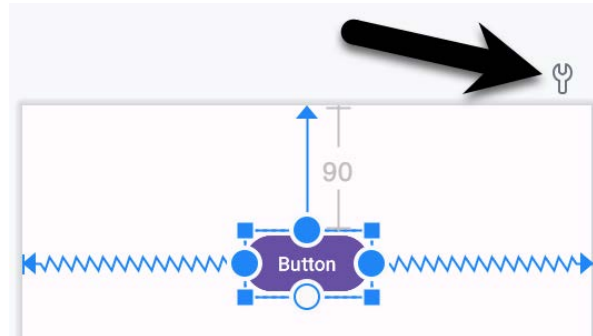


Figure 17-19

This button toggles the current `tools` visibility settings. If the Button view shown above currently has the `tools` visibility attribute set to *gone*, for example, toggling this button will make it visible. This makes it easy to quickly check the layout behavior as the view is added to and removed from the layout. This toggle is also useful for checking that the views in the layout are correctly constrained, a topic covered in the chapter entitled “A Guide to Using *ConstraintLayout* in Android Studio”.

17.12 Converting Views

Changing a view in a layout from one type to another (such as converting a `TextView` to an `EditText`) can be performed easily within the Android Studio layout editor by right-clicking on the view either within the screen layout or Component tree window and selecting the *Convert view...* menu option (Figure 17-20):

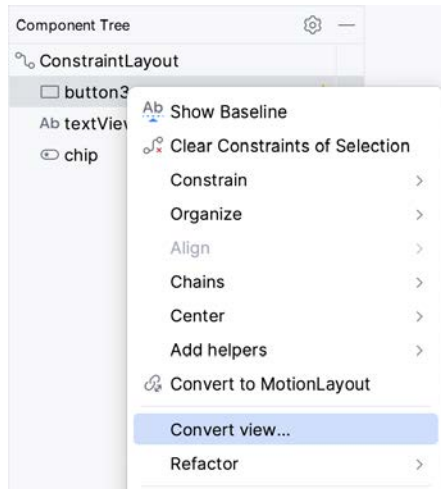


Figure 17-20

Once selected, a dialog containing a list of compatible view types to which the selected object is eligible for conversion will appear. Figure 17-21, for example, shows the types to which an existing TextView view may be converted:

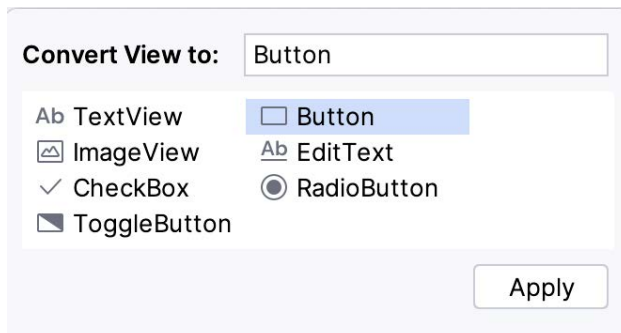


Figure 17-21

This technique is also helpful in converting layouts from one type to another (for example, converting a ConstraintLayout to a LinearLayout).

17.13 Displaying Sample Data

When designing layouts in Android Studio, situations will arise where the content to be displayed within the user interface will not be available until the app is completed and running. This can sometimes make it difficult to assess how the layout will appear at app runtime from within the layout editor. To address this issue, the layout editor allows sample data to be specified, which will populate views within the layout editor with sample images and data. This sample data only appears within the layout editor and is not displayed when the app runs. Sample data may be configured either by directly editing the XML for the layout or visually using the design-time helper by right-clicking on the widget in the design area and selecting the *Set Sample Data* menu option. The design-time helper panel will display a range of preconfigured options for sample data to be displayed on the selected view item, including combinations of text and images in various configurations. Figure 17-22, for example, shows the sample data options displayed when selecting sample data to appear in a RecyclerView list:

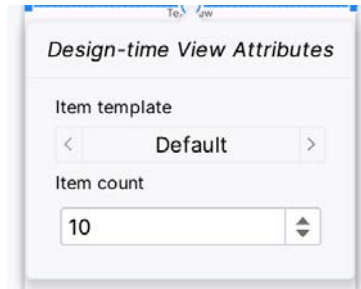


Figure 17-22

Alternatively, custom text and images may be provided for display during the layout design process. An example of using sample data within the layout editor is included in a later chapter entitled “A *Layout Editor Sample Data Tutorial*”. Since sample data is implemented as a *tools* attribute, the visibility of the data within the preview can be controlled using the toggle button highlighted in Figure 17-19 above.

17.14 Creating a Custom Device Definition

The device menu in the Layout Editor toolbar (Figure 17-23) provides a list of pre-configured device types, which, when selected, will appear as the device screen canvas. In addition to the pre-configured device types, any AVD instances previously configured within the Android Studio environment will also be listed within the menu. To add additional device configurations, display the device menu, select the *Add Device Definition* option and follow the steps outlined in the chapter entitled “*Creating an Android Virtual Device (AVD) in Android Studio*”.

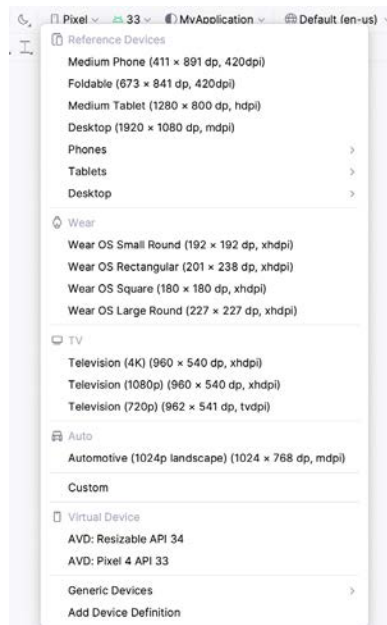


Figure 17-23

17.15 Changing the Current Device

As an alternative to the device selection menu, the current device format may be changed by selecting the *Custom* option from the device menu, clicking on the resize handle located next to the bottom right-hand corner of the device screen (Figure 17-24), and dragging to select an alternate device display format. As the screen

resizes, markers will appear indicating the various size options and orientations available for selection:



Figure 17-24

17.16 Layout Validation

The layout validation option allows the user interface layout to be previewed simultaneously on a range of Pixel-sized screens. To access the layout validation tool window, click on the tab on the right-hand edge of the Android Studio main window or use the Tool Window menu in the bottom left-hand corner of the window. Once loaded, the panel will appear as shown in Figure 17-25, with the layout rendered on multiple device screen configurations:

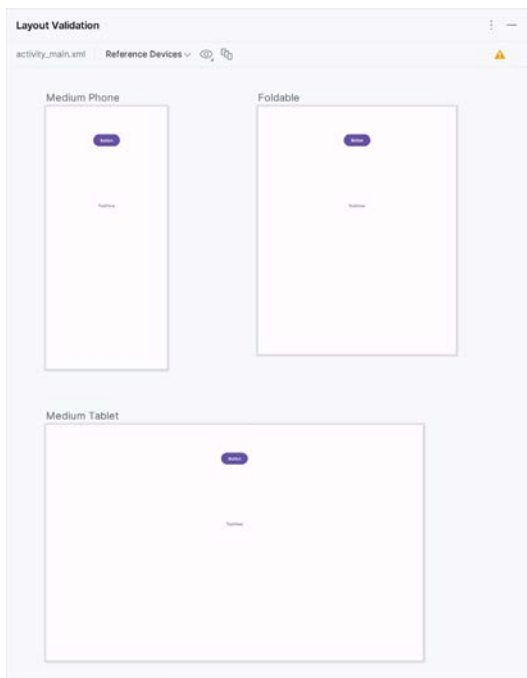


Figure 17-25

17.17 Summary

A key part of developing Android applications involves the creation of the user interface. This is performed within the Android Studio environment using the Layout Editor tool, which operates in three modes. In Design mode, view components are selected from a palette, positioned on a layout representing an Android device screen, and configured using a list of attributes. The underlying XML representing the user interface layout can be directly edited in Code mode. Split mode, on the other hand, allows the layout to be created and modified both visually and via direct XML editing. These modes combine to provide an extensive and intuitive user interface design environment.

The layout validation panel allows user interface layouts to be quickly previewed on various device screen sizes.

18. A Guide to the Android ConstraintLayout

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

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

18.1 How `ConstraintLayout` Works

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

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

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

18.1.1 Constraints

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

planes.

18.1.2 Margins

A margin is a form of constraint that specifies a fixed distance. Consider a Button object that needs to be positioned near the top right-hand corner of the device screen. This might be achieved by implementing margin constraints from the top and right-hand edges of the Button connected to the corresponding sides of the parent ConstraintLayout, as illustrated in Figure 18-1:

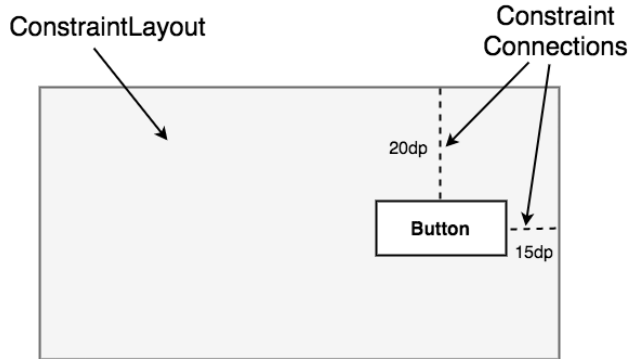


Figure 18-1

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

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

18.1.3 Opposing Constraints

Two constraints operating along the same axis on a single widget are considered *opposing constraints*. In other words, a widget with constraints on both its left and right-hand sides is considered to have horizontally opposing constraints. Figure 18-2, for example, illustrates the addition of both horizontally and vertically opposing constraints to the previous layout:

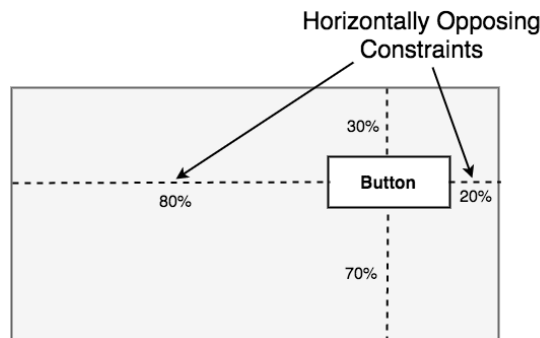


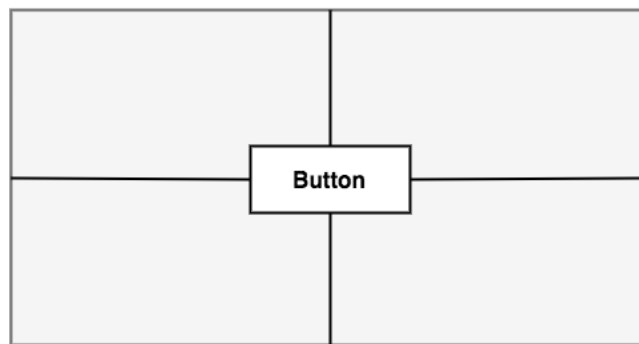
Figure 18-2

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

It is now important to understand that the layout outlined in Figure 18-2 has been implemented using not only opposing constraints, but also by applying *constraint bias*.

18.1.4 Constraint Bias

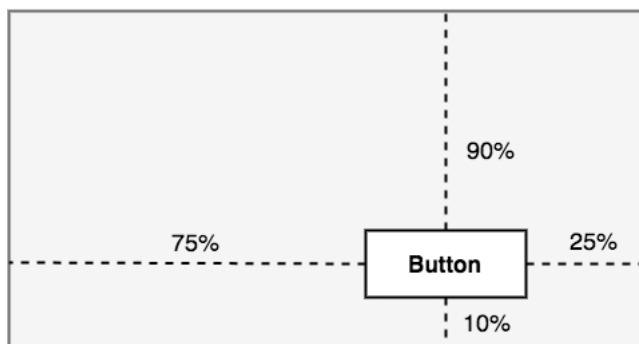
It has now been established that a widget in a ConstraintLayout can potentially be subject to opposing constraint connections. By default, opposing constraints are equal, resulting in the corresponding widget being centered along the axis of opposition. Figure 18-3, for example, shows a widget centered within the containing ConstraintLayout using opposing horizontal and vertical constraints:



Widget Centered by Opposing Constraints

Figure 18-3

To allow for the adjustment of widget position in the case of opposing constraints, the ConstraintLayout implements a feature known as *constraint bias*. Constraint bias allows the positioning of a widget along the axis of opposition to be biased by a specified percentage in favor of one constraint. Figure 18-4, for example, shows the previous constraint layout with a 75% horizontal bias and 10% vertical bias:



Widget Offset using Constraint Bias

Figure 18-4

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

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

18.1.5 Chains

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

Widgets are chained when connected by bi-directional constraints. Figure 18-5, for example, illustrates three widgets chained in this way:

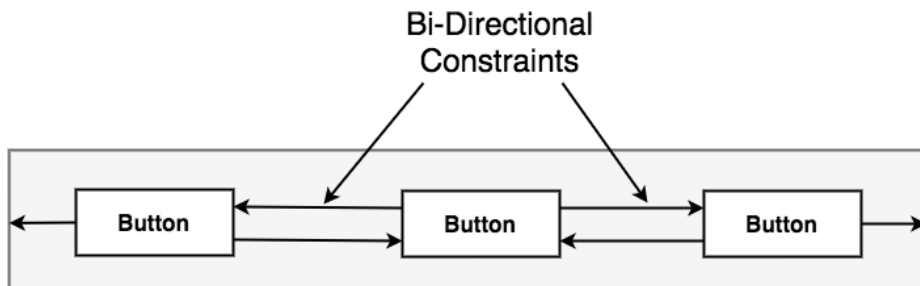


Figure 18-5

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

18.1.6 Chain Styles

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

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



Figure 18-6

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



Figure 18-7

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

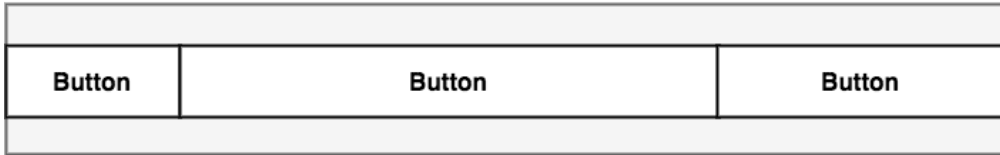


Figure 18-8

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

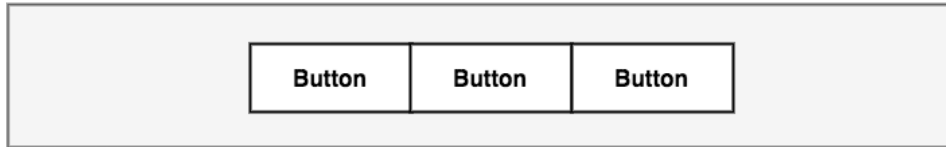


Figure 18-9

18.2 Baseline Alignment

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

For example, assume that the previous theoretical layout from Figure 18-1 requires a TextView widget to be positioned 40dp to the left of the Button. In this case, the TextView needs to be *baseline aligned* with the Button view. This means that the text within the Button needs to be vertically aligned with the text within the TextView. The additional constraints for this layout would need to be connected as illustrated in Figure 18-10:

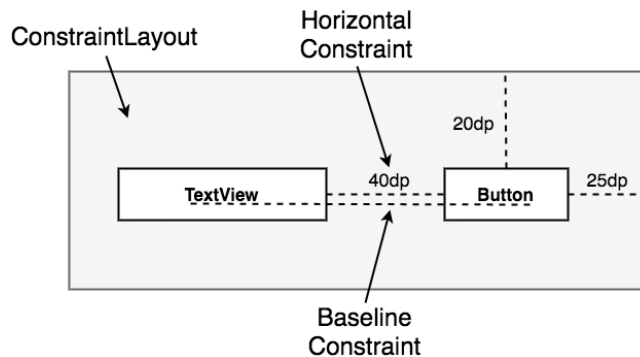


Figure 18-10

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

18.3 Configuring Widget Dimensions

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

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

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

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

18.4 Guideline Helper

Guidelines are special elements available within the `ConstraintLayout` that provide an additional target to which constraints may be connected. Multiple guidelines may be added to a `ConstraintLayout` instance which may, in turn, be configured in horizontal or vertical orientations. Once added, constraint connections may be established from widgets in the layout to the guidelines. This is particularly useful when multiple widgets must be aligned along an axis. In Figure 18-11, for example, three `Button` objects contained within a `ConstraintLayout` are constrained along a vertical guideline:

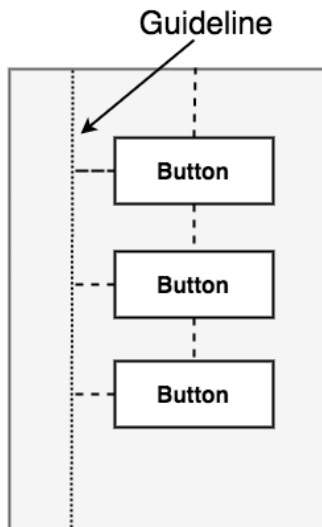


Figure 18-11

18.5 Group Helper

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

18.6 Barrier Helper

Rather like guidelines, barriers are virtual views that can be used to constrain views within a layout. As with guidelines, a barrier can be vertical or horizontal, and one or more views may be constrained to it (to avoid confusion, these will be referred to as *constrained views*). Unlike guidelines, where the guideline remains at a fixed position within the layout, however, the position of a barrier is defined by a set of so-called *reference views*. Barriers were introduced to address an issue that occurs with some frequency involving overlapping views. Consider, for example, the layout illustrated in Figure 18-12 below:

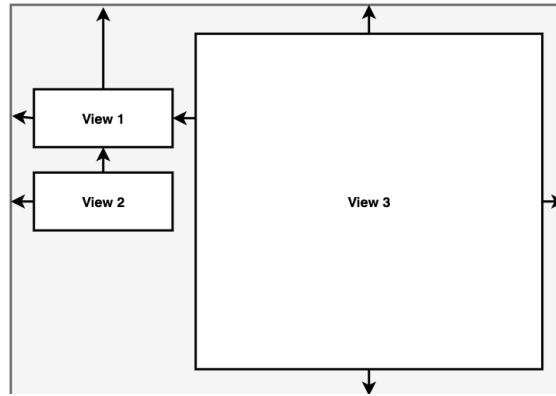


Figure 18-12

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

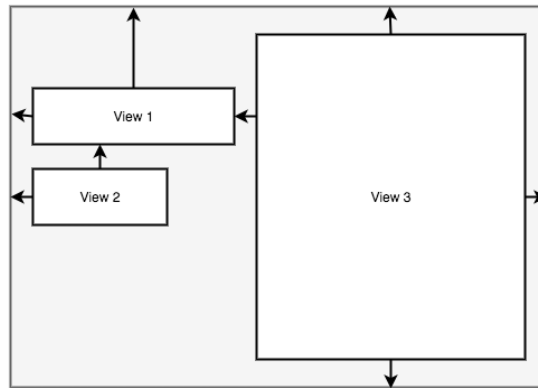


Figure 18-13

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

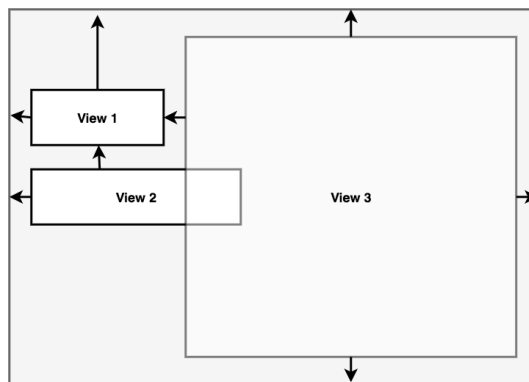


Figure 18-14

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

2, causing the views to overlap.

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

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

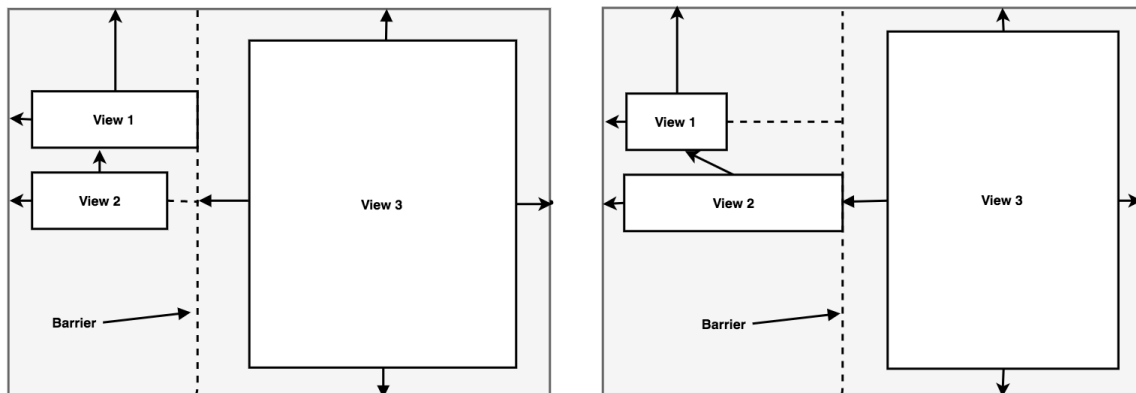


Figure 18-15

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

18.7 Flow Helper

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

Figure 18-16 represents the layout of five uniformly sized buttons positioned using a Flow helper instance in horizontal mode with no wrap settings:



Figure 18-16

Figure 18-17 shows the same buttons in a horizontal flow configuration with wrapping set to occur after every third widget:

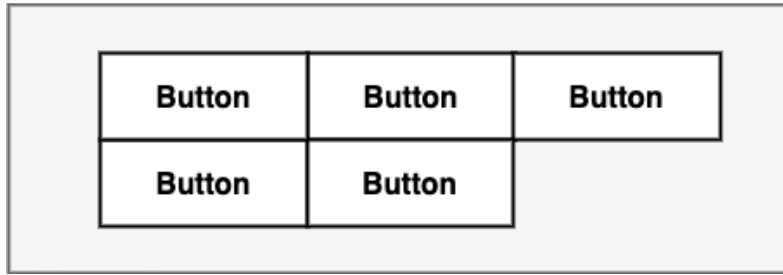


Figure 18-17

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

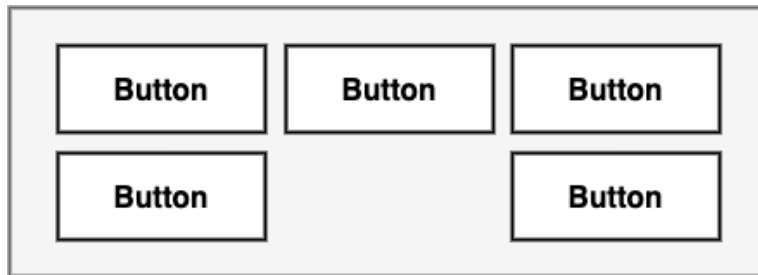


Figure 18-18

As a final demonstration of the flexibility of the Flow helper, Figure 18-19 shows five buttons of varying sizes configured in horizontal, packed chain mode with wrapping after each third widget. In addition, the grid content has been right-aligned by setting a horizontal-bias value of 1.0 (a value of 0.0 would cause left-alignment while 0.5 would center-align the grid content):

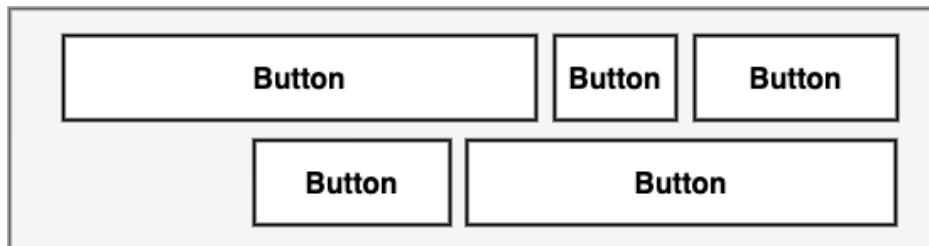


Figure 18-19

18.8 Ratios

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

18.9 ConstraintLayout Advantages

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

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

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

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

18.10 ConstraintLayout Availability

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

18.11 Summary

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

32. 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.

32.1 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.

32.2 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.

32.3 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.

32.4 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.

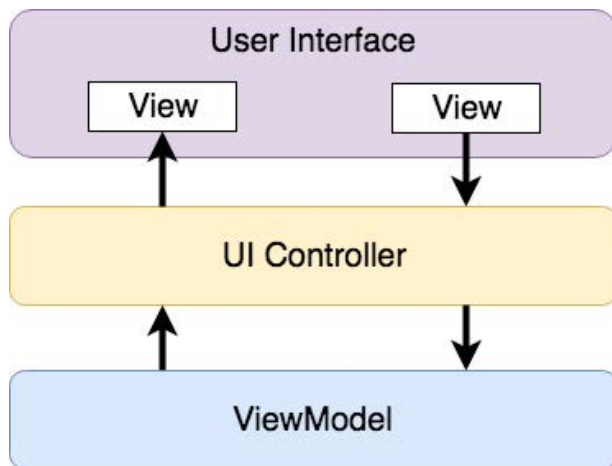


Figure 32-1

32.5 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.

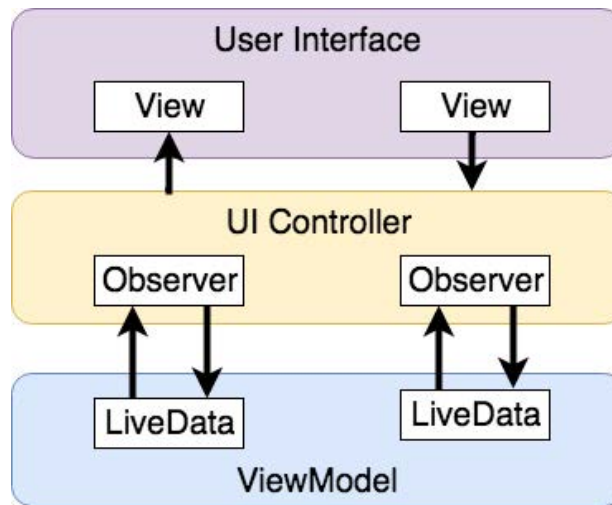


Figure 32-2

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.

32.6 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.

32.7 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.

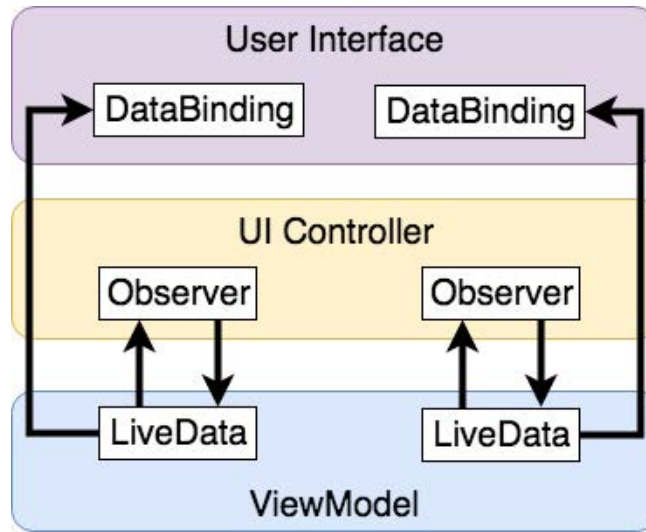


Figure 32-3

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

32.8 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.

32.9 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 Java 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.

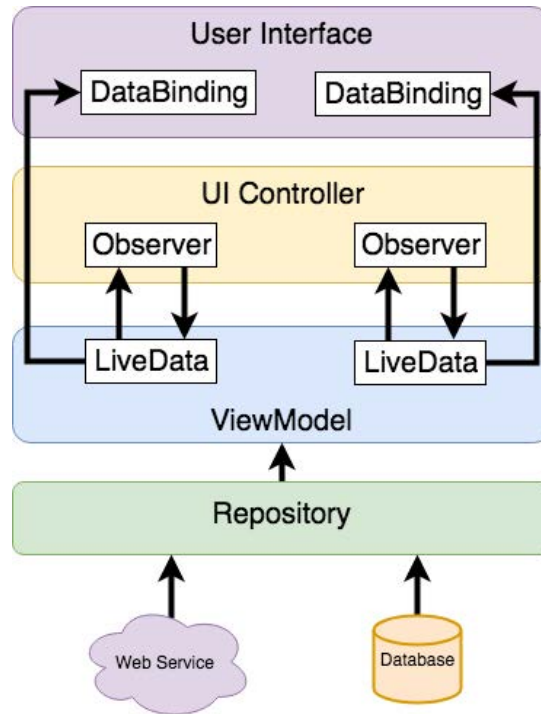


Figure 32-4

32.10 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.

35. An Overview of Android Jetpack Data Binding

In the chapter entitled “*Modern Android App Architecture with Jetpack*”, we introduced the concept of Android Data Binding. We explained how it is used to directly connect the views in a user interface layout to the methods and data located in other objects within an app without the need to write code. This chapter will provide more details on data binding, emphasizing how data binding is implemented within an Android Studio project. The tutorial in the next chapter (“*An Android Jetpack Data Binding Tutorial*”) will provide a practical example of data binding in action.

35.1 An Overview of Data Binding

The Android Jetpack Data Binding Library provides data binding support, primarily providing a simple way to connect the views in a user interface layout to the data stored within the app’s code (typically within `ViewModel` instances). Data binding also provides a convenient way to map user interface controls, such as `Button` widgets, to event and listener methods within other objects, such as `UI` controllers and `ViewModel` instances.

Data binding becomes particularly powerful when used in conjunction with the `LiveData` component. Consider, for example, an `EditText` view bound to a `LiveData` variable within a `ViewModel` using data binding. When connected in this way, any changes to the data value in the `ViewModel` will automatically appear within the `EditText` view, and when using two-way binding, any data typed into the `EditText` will automatically be used to update the `LiveData` value. Perhaps most impressive is that this can be achieved with no code beyond that necessary to initially set up the binding.

Connecting an interactive view, such as a `Button` widget, to a method within a `UI` controller traditionally required that the developer write code to implement a listener method to be called when the button is clicked. Data binding makes this as simple as referencing the method to be called within the `Button` element in the layout XML file.

35.2 The Key Components of Data Binding

An Android Studio project is not configured for data binding support by default. Several elements must be combined before an app can begin using data binding. These involve the project build configuration, the layout XML file, data binding classes, and the use of the data binding expression language. While this may appear overwhelming at first, when taken separately, these are quite simple steps that, once completed, are more than worthwhile in terms of saved coding effort. Each element will be covered in detail in the remainder of this chapter. Once these basics have been covered, the next chapter will work through a detailed tutorial demonstrating these steps.

35.2.1 The Project Build Configuration

Before a project can use data binding, it must be configured to use the Android Data Binding Library and to enable support for data binding classes and the binding syntax. Fortunately, this can be achieved with just a few lines added to the module level `build.gradle.kts` file (the one listed as `build.gradle.kts (Module: app)` under *Gradle Scripts* in the Project tool window). The following lists a partial build file with data binding enabled:

.

```
.
android {

    buildFeatures {
        dataBinding = true
    }
}
.
```

35.2.2 The Data Binding Layout File

As we have seen in previous chapters, the user interfaces for an app are typically contained within an XML layout file. Before the views contained within one of these layout files can take advantage of data binding, the layout file must be converted to a *data binding layout file*.

As outlined earlier in the book, XML layout files define the hierarchy of components in the layout, starting with a top-level or *root view*. Invariably, this root view takes the form of a layout container such as a `ConstraintLayout`, `FrameLayout`, or `LinearLayout` instance, as is the case in the `fragment_main.xml` file for the `ViewModelDemo` project:

```
<?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/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ui.main.MainFragment">
.
.
</androidx.constraintlayout.widget.ConstraintLayout>
```

To use data binding, the layout hierarchy must have a *layout* component as the root view, which, in turn, becomes the parent of the current root view.

In the case of the above example, this would require that the following changes be made to the existing layout file:

```
<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <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/main"
        android:layout_width="match_parent"
```

```

        android:layout_height="match_parent"
        tools:context=".ui.main.MainFragment">
    .
    .
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>

```

35.2.3 The Layout File Data Element

The data binding layout file needs some way to declare the classes within the project to which the views in the layout are to be bound (for example, a ViewModel or UI controller). Having declared these classes, the layout file will need a variable name to reference those instances within binding expressions.

This is achieved using the *data* element, an example of which is shown below:

```

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:android="http://schemas.android.com/apk/res/android">

    <data>
        <variable
            name="myViewModel"
            type="com.ebookfrenzy.myapp.ui.main.MainViewModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ui.main.MainFragment">
    .
    .
</layout>

```

The above data element declares a new variable named *myViewModel* of type *MainViewModel* (note that it is necessary to declare the full package name of the *MyViewModel* class when declaring the variable).

The data element can import other classes that may then be referenced within binding expressions elsewhere in the layout file. For example, if you have a class containing a method that needs to be called on a value before it is displayed to the user, the class could be imported as follows:

```

<data>
    <import type="com.ebookfrenzy.MyFormattingTools" />
    <variable
        name="viewModel"
        type="com.ebookfrenzy.myapp.ui.main.MainViewModel" />
</data>

```

35.2.4 The Binding Classes

For each class referenced in the *data* element within the binding layout file, Android Studio will automatically generate a corresponding *binding class*. This subclass of the Android `ViewDataBinding` class will be named based on the layout filename using word capitalization and the *Binding* suffix. Therefore, the binding class for a layout file named *fragment_main.xml* file will be named *FragmentMainBinding*. The binding class contains the bindings specified within the layout file and maps them to the variables and methods within the bound objects.

Although the binding class is generated automatically, code must be written to create an instance of the class based on the corresponding data binding layout file. Fortunately, this can be achieved by making use of the `DataBindingUtil` class.

The initialization code for an Activity or Fragment will typically set the content view or “inflate” the user interface layout file. This means that the code opens the layout file, parses the XML, and creates and configures all of the view objects in memory. In the case of an existing Activity class, the code to achieve this can be found in the *onCreate()* method and will read as follows:

```
setContentView(R.layout.activity_main);
```

In the case of a Fragment, this takes place in the *onCreateView()* method:

```
return inflater.inflate(R.layout.fragment_main, container, false);
```

All that is needed to create the binding class instances within an Activity class is to modify this initialization code as follows:

```
ActivityMainBinding binding;
```

```
binding = DataBindingUtil.setContentView(this, R.layout.activity_main, false);
```

In the case of a Fragment, the code would read as follows:

```
FragmentMainBinding binding;
```

```
binding = DataBindingUtil.inflate(  
    inflater, R.layout.fragment_main, container, false);
```

```
binding.setLifecycleOwner(this);  
View view = binding.getRoot();  
return view;
```

35.2.5 Data Binding Variable Configuration

As outlined above, the data binding layout file contains the *data* element, which contains *variable* elements consisting of variable names and the class types to which the bindings are to be established. For example:

```
<data>  
    <variable  
        name="viewModel"  
        type="com.ebookfrenzy.viewmodeldemo.ui.main.MainViewModel" />  
    <variable  
        name="uiController"  
        type="com.ebookfrenzy.viewmodeldemo_databinding.ui.main.MainFragment"  
    />  
</data>
```

In the above example, the first variable knows that it will be binding to an instance of a `ViewModel` class of type `MainViewModel` but has yet to be connected to an actual `MainViewModel` object instance. This requires the additional step of assigning the `MainViewModel` instance used within the app to the variable declared in the layout file. This is performed via a call to the `setVariable()` method of the data binding instance, a reference to which was obtained in the previous chapter:

```
MainViewModel mViewModel = new ViewModelProvider(this).get(MainViewModel.class);
binding.setVariable(viewModel, mViewModel);
```

The second variable in the above data element references a UI controller class in the form of a `Fragment` named `MainFragment`. In this situation, the code within a UI controller (be it an `Activity` or `Fragment`) would need to assign itself to the variable as follows:

```
binding.setVariable(uiController, this);
```

35.2.6 Binding Expressions (One-Way)

Binding expressions define how a particular view interacts with bound objects. For example, a binding expression on a `Button` might declare which method on an object is called in response to a click. Alternatively, a binding expression might define which data value stored in a `ViewModel` is to appear within a `TextView` and how it is to be presented and formatted.

Binding expressions use a declarative language that allows logic and access to other classes and methods to decide how bound data is used. Expressions can, for example, include mathematical expressions, method calls, string concatenations, access to array elements, and comparison operations. In addition, all standard Java language libraries are imported by default, so many things that can be achieved in Java can also be performed in a binding expression. As already discussed, the data element may also be used to import custom classes to add more capability to expressions.

A binding expression begins with an `@` symbol followed by the expression enclosed in curly braces (`{}`).

Consider, for example, a `ViewModel` instance containing a variable named *result*. Assume that this class has been assigned to a variable named *viewModel* within the data binding layout file and needs to be bound to a `TextView` object so that the view always displays the latest result value. If this value were stored as a `String` object, this would be declared within the layout file as follows:

```
<TextView
    android:id="@+id/resultText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{viewModel.result}"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

In the above XML, the *text* property is set to the value stored in the *result* `LiveData` property of the `viewModel` object.

Consider, however, that the result is stored within the model as a `Float` value instead of a `String`. That being the case, the above expression would cause a compilation error. Clearly, the `Float` value must be converted to a string before the `TextView` can display it. To resolve issues such as this, the binding expression can include the necessary steps to complete the conversion using the standard Java language classes:

```
android:text="@{String.valueOf(viewModel.result)}"
```

An Overview of Android Jetpack Data Binding

When running the app after making this change, it is important to be aware that the following warning may appear in the Android Studio console:

```
warning: myViewModel.result.getValue() is a boxed field but needs to be un-boxed
to execute String.valueOf(viewModel.result.getValue()).
```

Values in Java can take the form of primitive values such as the *boolean* type (referred to as being *unboxed*) or wrapped in a Java object such as the *Boolean* type and accessed via reference to that object (i.e., *boxed*). The unboxing process involves unwrapping the primitive value from the object.

To avoid this message, wrap the offending operation in a *safeUnbox()* call as follows:

```
android:text="@{String.valueOf(safeUnbox(myViewModel.result))}"
```

String concatenation may also be used. For example, to include the word “dollars” after the result string value, the following expression would be used:

```
android:text='@{String.valueOf(safeUnbox(myViewModel.result)) + " dollars"}'
```

Note that since the appended result string is wrapped in double quotes, the expression is now encapsulated with single quotes to avoid syntax errors.

The expression syntax also allows ternary statements to be declared. In the following expression, the view will display different text depending on whether or not the result value is greater than 10.

```
@{myViewModel.result > 10 ? "Out of range" : "In range"}
```

Expressions may also be constructed to access specific elements in a data array:

```
@{myViewModel.resultsArray[3]}
```

35.2.7 Binding Expressions (Two-Way)

The type of expression covered so far is called *one-way binding*. In other words, the layout is constantly updated as the corresponding value changes, but changes to the value from within the layout do not update the stored value.

A *two-way binding*, on the other hand, allows the data model to be updated in response to changes in the layout. An EditText view, for example, could be configured with a two-way binding so that when the user enters a different value, that value is used to update the corresponding data model value. When declaring a two-way expression, the syntax is similar to a one-way expression except that it begins with *@=*. For example:

```
android:text="@={myViewModel.result}"
```

35.2.8 Event and Listener Bindings

Binding expressions may also trigger method calls in response to events on a view. A Button view, for example, can be configured to call a method when clicked. In the chapter entitled “*Creating an Example Android App in Android Studio*”, for example, the *onClick* property of a button was configured to call a method within the app’s main activity named *convertCurrency()*. Within the XML file, this was represented as follows:

```
android:onClick="convertCurrency"
```

The *convertCurrency()* method was declared along the following lines:

```
public void convertCurrency(View view) {
    .
    .
}
```

Note that this type of method call is always passed a reference to the view on which the event occurred. The same effect can be achieved in data binding using the following expression (assuming the layout has been bound to a

class with a variable name of *uiController*):

```
android:onClick="@{uiController::convertCurrency}"
```

Another option, and one which provides the ability to pass parameters to the method, is referred to as a *listener binding*. The following expression uses this approach to call a method on the same *viewModel* instance with no parameters:

```
android:onClick='{() -> myViewModel.methodOne()}'
```

The following expression calls a method that expects three parameters:

```
android:onClick='{() -> myViewModel.methodTwo(viewModel.result, 10, "A String")}'
```

Binding expressions provide a rich and flexible language to bind user interface views to data and methods in other objects. This chapter has only covered the most common use cases. To learn more about binding expressions, review the Android documentation online at:

<https://developer.android.com/topic/libraries/data-binding/expressions>

35.3 Summary

Android data bindings provide a system for creating connections between the views in a user interface layout and the data and methods of other objects within the app architecture without writing code. Once some initial configuration steps have been performed, data binding involves using binding expressions within the view elements of the layout file. These binding expressions can be either one-way or two-way and may also be used to bind methods to be called in response to events such as button clicks within the user interface.

40. 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.

40.1 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 40-1:

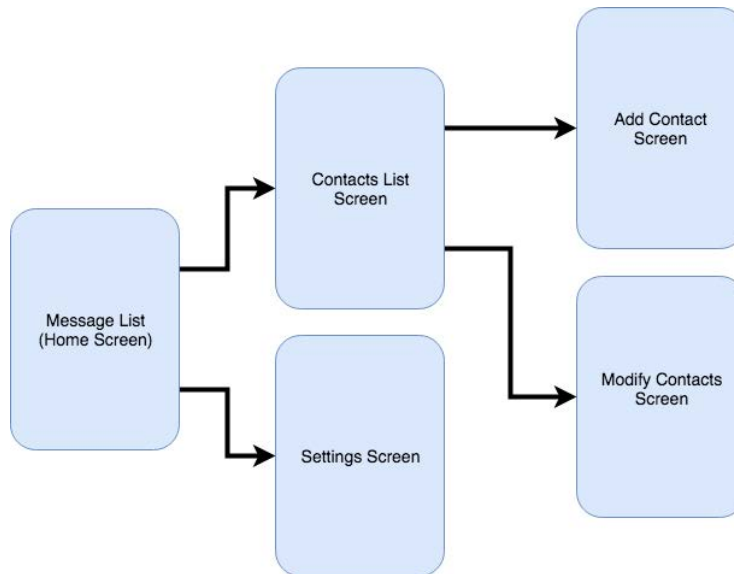


Figure 40-1

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 40-2, 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:

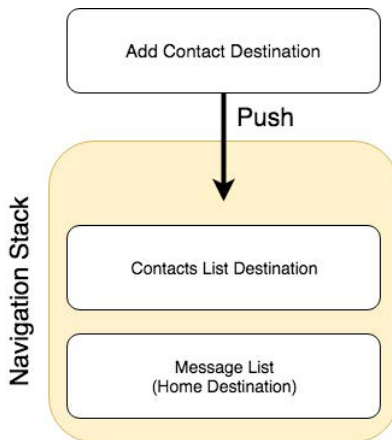


Figure 40-2

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 40-3, the user has navigated back from the Add Contact screen, popping it off the stack and making the Contacts List screen the current destination:

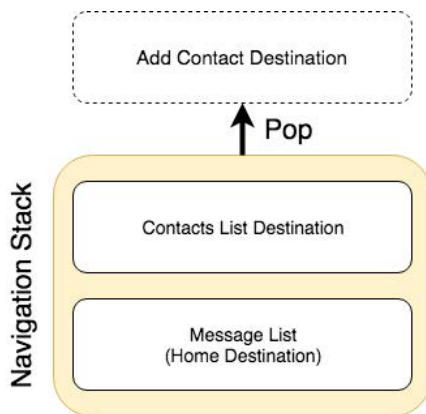


Figure 40-3

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.

40.2 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 40-4, for example, shows a typical activity screen and highlights the area represented by the navigation host fragment:

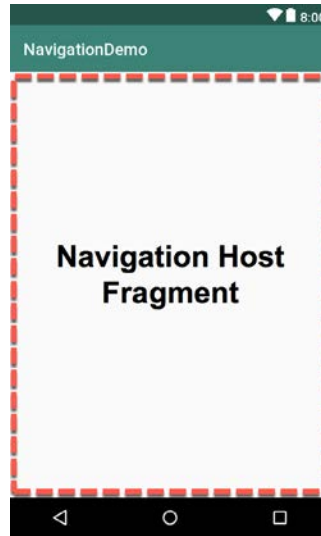


Figure 40-4

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.

40.3 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 40-5 shows the Android Studio navigation graph editor in Design mode:

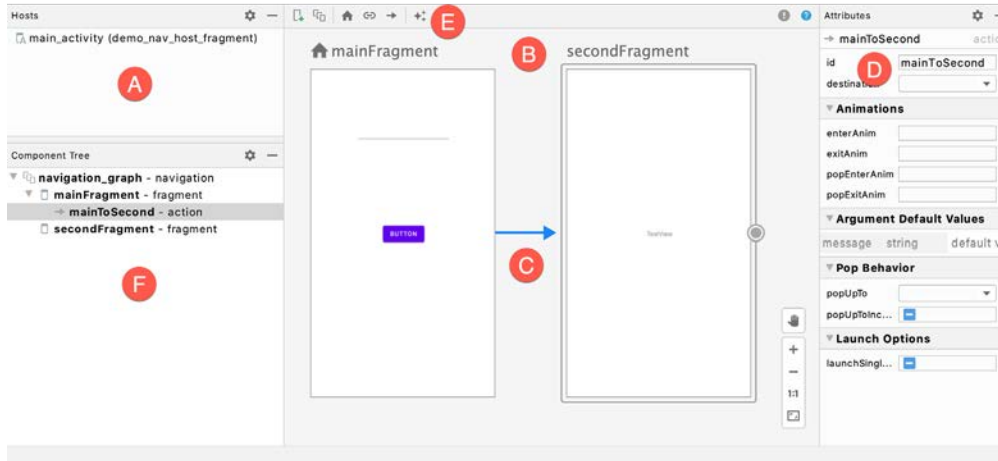


Figure 40-5

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 40-5 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.

40.4 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:

```

NavController controller =
    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:

```

NavController controller = Navigation.findNavController(binding.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:

```

NavController controller = NavHostFragment.findNavController(fragment);

```

40.5 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.

40.6 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 40-6, for example, an integer argument named *contactsCount* has been declared with a default value of 0:

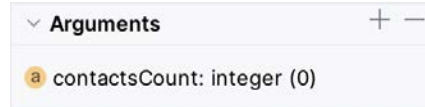


Figure 40-6

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*”).

40.7 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.

42. 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.

42.1 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.

42.2 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.

42.3 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.

42.4 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.

42.5 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>

.
.
```

42.6 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.

42.7 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 42-1 illustrates *startVertical* and *startHorizontal* arcs in comparison to the default straight line motion:

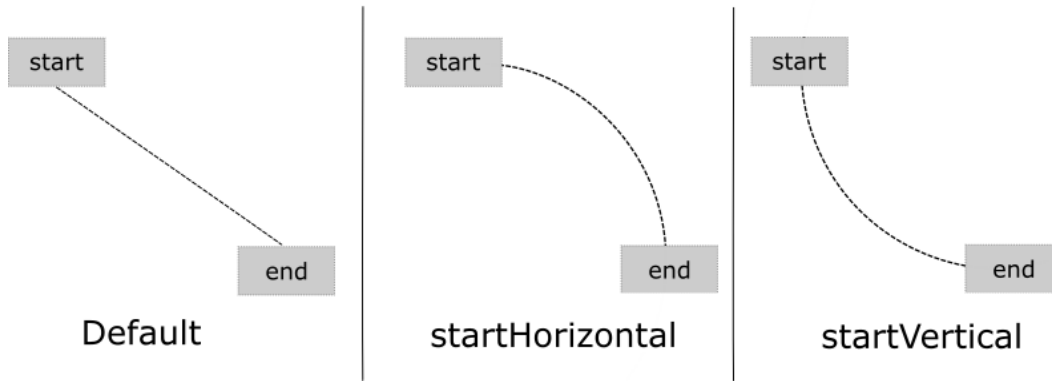


Figure 42-1

42.8 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:

42.8.1 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>
```

42.8.2 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):

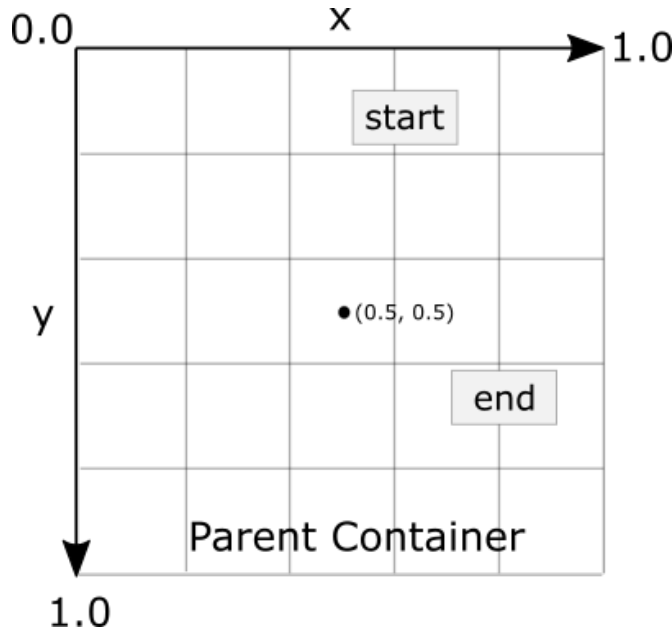


Figure 42-2

- **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):

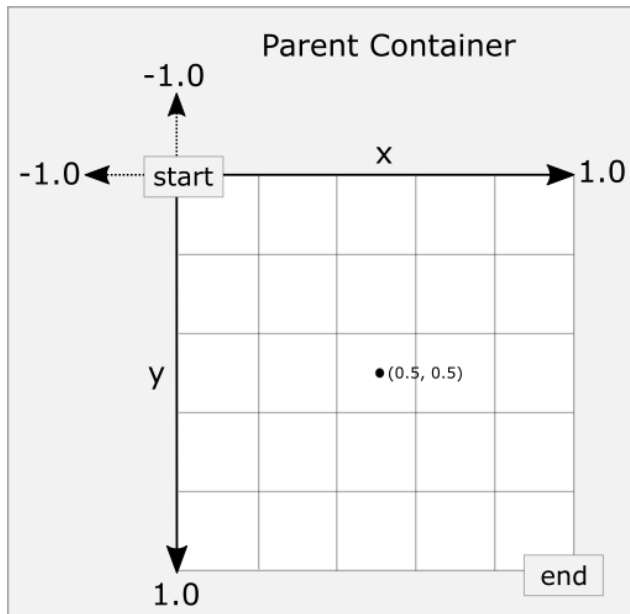


Figure 42-3

- **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:

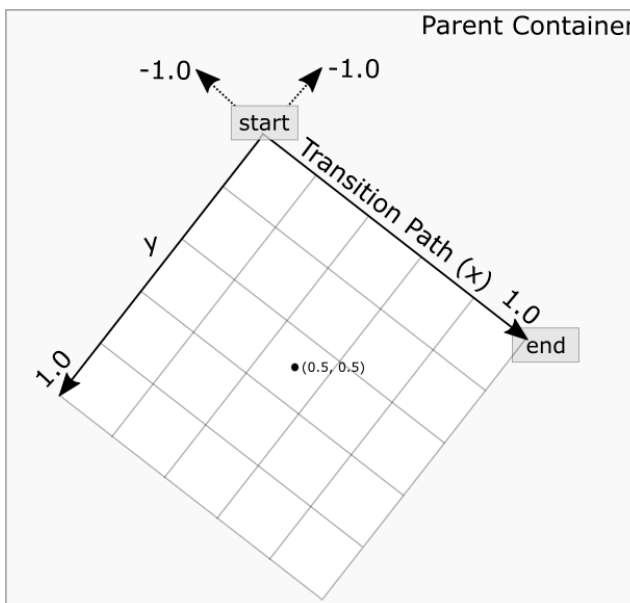


Figure 42-4

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 42-5:



Figure 42-5

Suppose, however, that the view is required to follow a path similar to that shown in Figure 42-6 below:

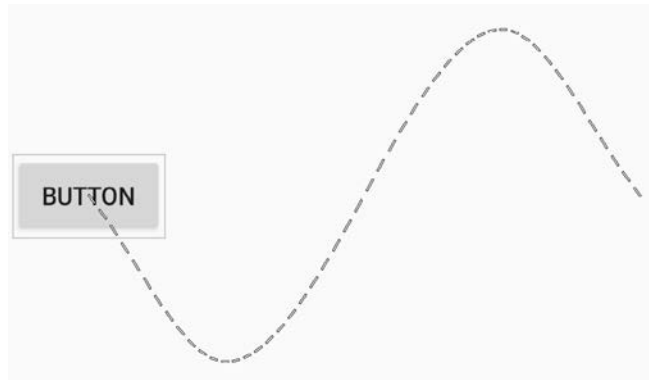


Figure 42-6

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 42-7 below:

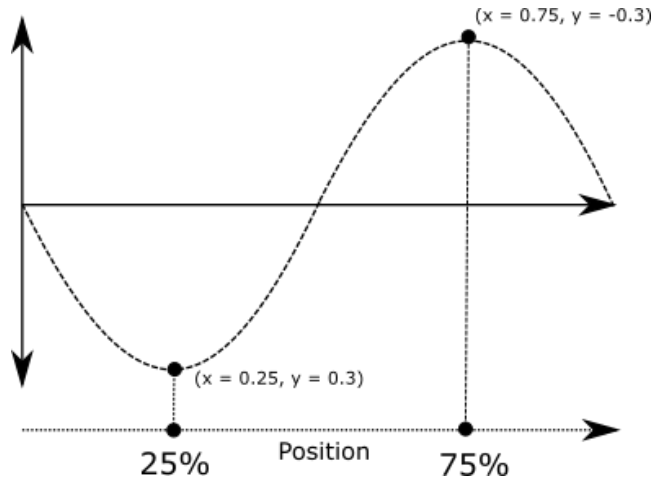


Figure 42-7

42.9 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">  

.  

.
```

42.10 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"/>
```

42.11 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.

42.12 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();
```

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

An Introduction to MotionLayout

as follows:

```
motionLayout.setTransitionListener(transitionListener);
```

```
MotionLayout.TransitionListener transitionListener =
    new MotionLayout.TransitionListener() {
        @Override
        public void onTransitionStarted(MotionLayout motionLayout,
                                         int startId, int endId) {
            // Called when the transition starts
        }

        @Override
        public void onTransitionChange(MotionLayout motionLayout, int startId,
                                       int endId, float progress) {
            // Called each time a property changes. Track progress value to find
            // current position
        }

        @Override
        public void onTransitionCompleted(MotionLayout motionLayout, int currentId) {
            // Called when the transition is complete
        }

        @Override
        public void onTransitionTrigger(MotionLayout motionLayout, int triggerId,
                                       boolean positive, float progress) {
            // Called when a trigger keyframe threshold is crossed
        }
    };
```

42.13 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.

49. A Layout Editor Sample Data Tutorial

The CardDemo project created in the previous chapter has provided a good example of how it can be difficult to assess from within the layout editor exactly how a user interface is going to appear until the completed app is tested. This is a problem that frequently occurs when the content to be displayed in a user interface is only generated or acquired once the user has the app installed and running.

For some time now, the Android Studio layout editor has provided the ability to specify simple attributes that are active only when the layout is being designed. A design-time only string resource could, for example, be assigned to a TextView within the layout editor that would not appear when the app runs. This capability has been extended significantly with the introduction of sample data support within the Android Studio layout editor and will be used in this chapter to improve the layout editor experience in the CardDemo project.

49.1 Adding Sample Data to a Project

During the design phase of the user interface layout, the RecyclerView instance (Figure 49-1) bears little resemblance to the running app tested at the end of the previous chapter:

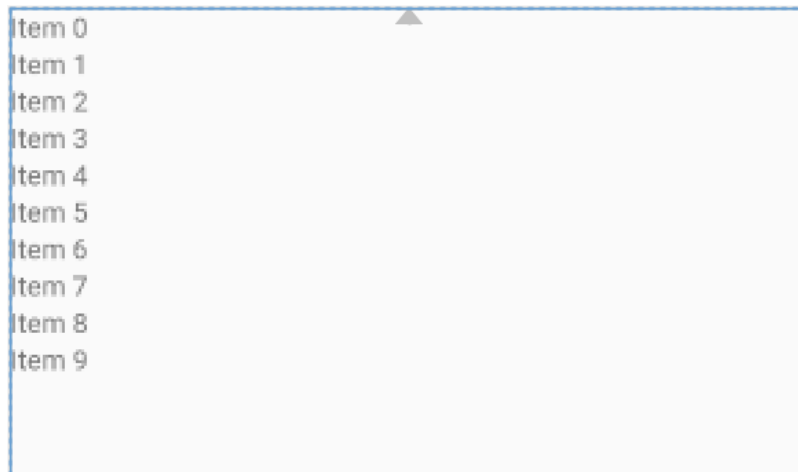


Figure 49-1

In the “*Modern Android App Architecture with Jetpack*” chapter earlier in the book the concept of sample data was introduced. To demonstrate sample data in use, the project will now be modified so that the fully populated cards appear within the RecyclerView from within the layout editor. Before doing that, however, it is worth noting that the layout editor has a collection of preconfigured sample data templates that can be used when designing user interfaces. To see some of these in action, load the *content_main.xml* layout file into the layout editor and select the RecyclerView instance. Right-click on the RecyclerView and select the *Set Sample Data* menu option to display the Design-time View Attributes panel:

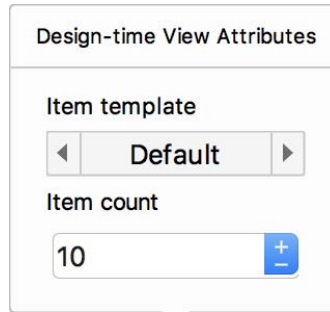


Figure 49-2

Change the template to the Email Client option and the item count to 12 and note that the RecyclerView changes to display data from the template:

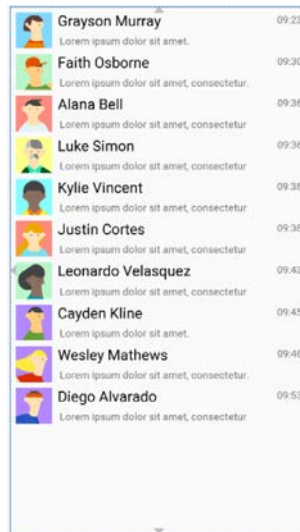


Figure 49-3

These templates can be useful for displaying sample data without any additional work and will often provide enough to complete the layout design. For this example, however, sample data is going to be used to display the cards within the RecyclerView as they are intended to appear in the running app. With the *content_main.xml* file still loaded in the layout editor, switch to Code mode and locate the RecyclerView element which should read as follows:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:itemCount="12"
    tools:listitem="@layout/recycler_view_item" />
```

Note the two special *tools* properties currently configured to display 12 items in the list, each using a layout contained within a file named *recycler_view_item.xml*. The layout editor provides a range of tools options that may be configured within a layout file. Though coverage of all of these settings is beyond the scope of this book, a full description of each can be found at the following URL:

https://developer.android.com/studio/write/tool-attributes#toolssample_resources

The *recycler_view_item.xml* file referenced above was generated automatically by the layout editor when the sample data template was selected and can be found in the project tool window.

Switch back to Design mode and, with the RecyclerView selected, use the Design-time View Attributes panel to switch the template back to the default settings. The *recycler_view_item.xml* file will be removed from the project along with the two *tools* property lines within the *content_main.xml* XML file (if Android Studio fails to remove the lines they may be deleted manually from within the Code view).

To switch to using the card layout for the sample data display, add a *listitem* property to reference the *card_layout.xml* file. With the RecyclerView selected in the layout, search for the *listitem* property in the Attributes tool window and enter *@layout/card_layout* into the property field as illustrated in Figure 49-4:

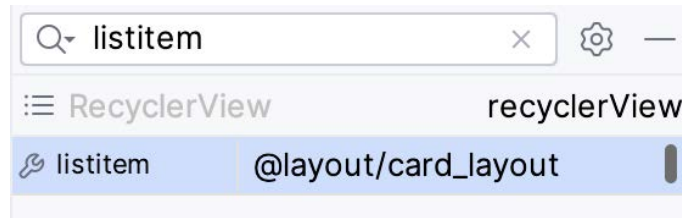


Figure 49-4

Note the card layout is now appearing for each list item, though without any images and using sample text data:



Figure 49-5

The next step is to display some images and different text on the views within the card layout. This can either

take the form of template sample data provided by the layout editor, or custom sample data added specifically for this project. Load the *card_layout.xml* file into the layout editor, select the *ImageView* and display the Design-time View Attributes panel as outlined earlier in the chapter. From the *srcCompat* menu, select the built-in backgrounds/scenic image set as illustrated in Figure 49-6 below:

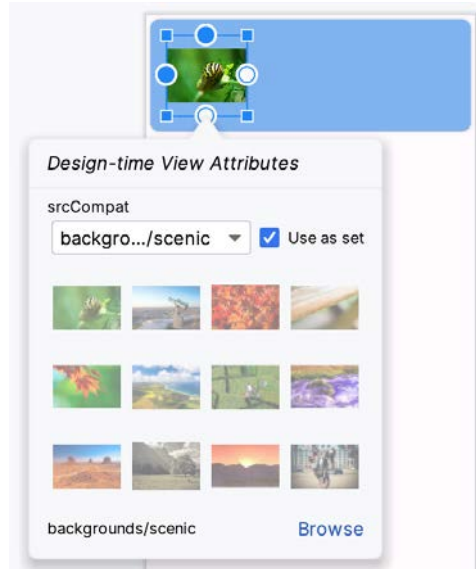


Figure 49-6

Next, right-click on the *itemTitle* *TextView* object, select the *Set Sample Data* menu option and, in the Design-time attributes panel, select the *cities* text option. Repeat this step for the *itemDetail* view, this time selecting the *full_names* option as shown in Figure 49-7:

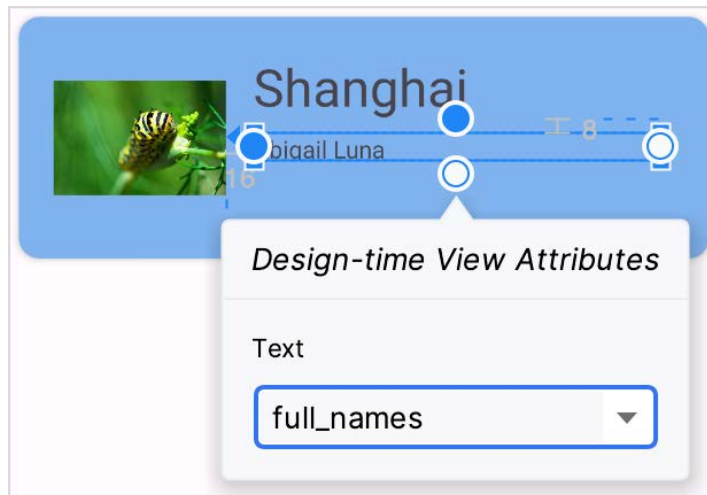


Figure 49-7

Open the *content_main.xml* file in Design mode and note that the *RecyclerView* is now using the built-in images and sample text data:



Figure 49-8

49.2 Using Custom Sample Data

The final step in this chapter is to demonstrate the use of custom sample data and images within the layout editor. This requires the creation of a sample data directory and the addition of some text and image files. Within the Project tool window, right-click on the *app* entry and select the *New -> Sample Data Directory* menu option, at which point a new directory named *sampledata* will appear within the Project tool window. If the new folder is not visible, switch the Project tool window from Android to Project mode and find the folder under *CardDemo* -> *app* as shown in Figure 49-9:

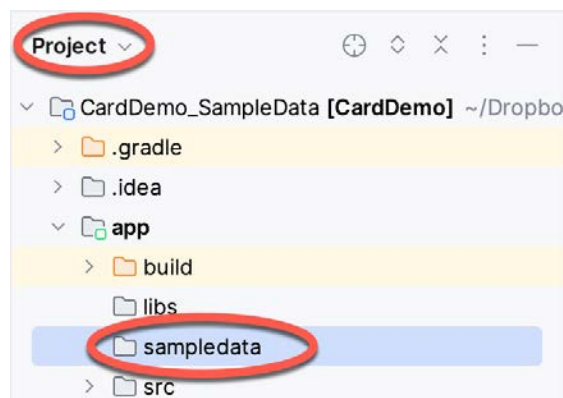


Figure 49-9

Right-click on the *sampledata* directory, create a directory named *images* and copy and paste the Android images into the new folder using the same steps outlined in the previous chapter. In the *card_layout.xml* file, display the Design-time View Attributes panel for the *ImageView* once again, this time clicking the Browse link and selecting the newly added Android images in the *Resources* dialog (if the *images* folder does not appear try rebuilding the project):

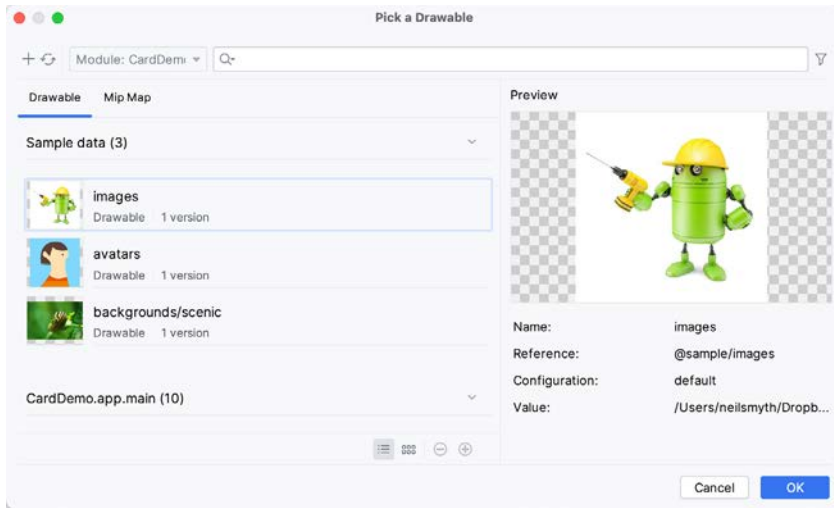


Figure 49-10

Right-click once again on the *sampleddata* directory, select the *New -> File* option, name the file *chapters*, and enter the following content:

```
Chapter One
Chapter Two
Chapter Three
Chapter Four
Chapter Five
Chapter Six
Chapter Seven
Chapter Eight
```

Next, create a second text file named *items* with the following content:

```
Item one details
Item two details
Item three details
Item four details
Item five details
Item six details
Item seven details
Item eight details
```

With the sample data text files created, all that remains is to reference them in the view elements of the *card_layout.xml* file as follows:

```
<TextView
    android:id="@+id/itemTitle"
    android:layout_width="236dp"
    android:layout_height="39dp"
    android:layout_marginStart="16dp"
    android:textSize="30sp"
    app:layout_constraintLeft_toRightOf="@+id/itemImage"
```



```
app:layout_constraintStart_toEndOf="@+id/itemImage"
app:layout_constraintTop_toTopOf="parent"
tools:text="@sample/chapters" />
```

```
<TextView
    android:id="@+id/itemDetail"
    android:layout_width="236dp"
    android:layout_height="16dp"
    android:layout_marginStart="16dp"
    android:layout_marginTop="8dp"
    app:layout_constraintLeft_toRightOf="@+id/itemImage"
    app:layout_constraintStart_toEndOf="@+id/itemImage"
    app:layout_constraintTop_toBottomOf="@+id/itemTitle"
    tools:text="@sample/items" />
```

Rebuild the app and return to the layout design in the *content_main.xml* file where the custom sample data and images should now be displayed within the RecyclerView list:

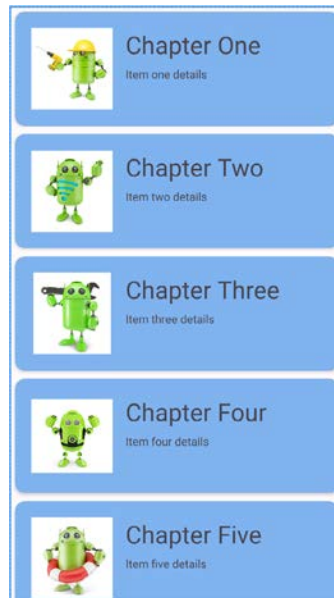


Figure 49-11

Instead of having two separate text files and a reference to the image set, another option is to declare the sample data within a JSON file. For example:

```
{
  "mydata": [
    {
      "chapter" : "Chapter One",
      "details": "Item one details",
      "image": "@sample/images"
    },
    {
```

A Layout Editor Sample Data Tutorial

```
        "chapter" : "Chapter Two",
        "details": "Item two details",
        "image": "@sample/images"
    },
    .
    .
}
```

Assuming the above was contained within a file named *chapterdata.json*, the sample data would then be referenced within the view XML elements as follows:

```
.
.
<ImageView
.
.
    tools:src="@sample/chapterdata.json/mydata/image" />
<TextView
.
.
    tools:text="@sample/chapterdata.json/mydata/chapter" />

<TextView
.
.
    tools:text="@sample/chapterdata.json/mydata/details" />
.
.
```

49.3 Summary

This chapter has demonstrated the use of sample data within the layout editor to provide a more realistic representation of how the user interface will appear at runtime. The steps covered in this tutorial included the use of both pre-existing sample data templates and the integration of custom sample data.

64. 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.

64.1 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.

64.2 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.

64.3 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.

64.4 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.

64.5 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.

64.6 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.

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>.

64.7 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.

64.8 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 -> Device 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:

```
adb 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
sqlite3 ./mydatabase.db
SQLite version 3.8.10.2 2015-05-20 18:17:19
```

An Overview of Android SQLite Databases

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.

64.9 Android SQLite Classes

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”, the Android SDK includes a set of classes that provide a programming layer on top of the SQLite database management system. The remainder of this chapter will provide a basic overview of each of the major classes within this category.

64.9.1 Cursor

A class provided specifically to access the results of a database query. For example, a SQL SELECT operation performed on a database will potentially return multiple matching rows from the database. A Cursor instance can be used to step through these results, which may then be accessed from within the application code using a variety of methods. Some key methods of this class are as follows:

- **close()** – Releases all resources used by the cursor and closes it.
- **getCount()** – Returns the number of rows contained within the result set.
- **moveToFirst()** – Moves to the first row within the result set.
- **moveToLast()** – Moves to the last row in the result set.
- **moveToNext()** – Moves to the next row in the result set.
- **move()** – Moves by a specified offset from the current position in the result set.
- **get<type>()** – Returns the value of the specified <type> contained at the specified column index of the row at the current cursor position (variations consist of *getString()*, *getInt()*, *getShort()*, *getFloat()*, and *getDouble()*).

64.9.2 SQLiteDatabase

This class provides the primary interface between the application code and underlying SQLite databases including the ability to create, delete, and perform SQL-based operations on databases. Some key methods of this class are as follows:

- **insert()** – Inserts a new row into a database table.
- **delete()** – Deletes rows from a database table.
- **query()** – Performs a specified database query and returns matching results via a Cursor object.
- **execSQL()** – Executes a single SQL statement that does not return result data.
- **rawQuery()** – Executes a SQL query statement and returns matching results in the form of a Cursor object.

64.9.3 SQLiteOpenHelper

A helper class designed to make it easier to create and update databases. This class must be subclassed within the code of the application seeking database access and the following callback methods implemented within that subclass:

- **onCreate()** – Called when the database is created for the first time. This method is passed the SQLiteDatabase object as an argument for the newly created database. This is the ideal location to initialize the database in terms of creating a table and inserting any initial data rows.
- **onUpgrade()** – Called in the event that the application code contains a more recent database version number reference. This is typically used when an application is updated on the device and requires that the database schema also be updated to handle storage of additional data.

In addition to the above mandatory callback methods, the *onOpen()* method, called when the database is opened, may also be implemented within the subclass.

The constructor for the subclass must also be implemented to call the super class, passing through the application context, the name of the database and the database version.

Notable methods of the `SQLiteOpenHelper` class include:

- **`getWritableDatabase()`** – Opens or creates a database for reading and writing. Returns a reference to the database in the form of a `SQLiteDatabase` object.
- **`getReadableDatabase()`** – Creates or opens a database for reading only. Returns a reference to the database in the form of a `SQLiteDatabase` object.
- **`close()`** – Closes the database.

64.9.4 ContentValues

`ContentValues` is a convenience class that allows key/value pairs to be declared consisting of table column identifiers and the values to be stored in each column. This class is of particular use when inserting or updating entries in a database table.

64.10 The Android Room Persistence Library

A limitation of the Android SDK SQLite classes is that they require moderate coding effort and don't 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 following chapters will provide an overview and tutorial on SQLite database management using SQLite and the Room persistence library.

64.11 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.

66. Understanding Android Content Providers

The previous chapter worked on creating an example application designed to store data using a SQLite database. When implemented this way, the data is private to the application and, as such, inaccessible to other applications running on the same device. While this may be the desired behavior for many application types, situations will inevitably arise whereby the data stored on behalf of an application could benefit other applications. A prime example is the data stored by the built-in Contacts application on an Android device. While the Contacts application is primarily responsible for managing the user's address book details, this data is also made accessible to any other applications needing access. This data sharing between Android applications is achieved through implementing *content providers*.

66.1 What is a Content Provider?

A content provider provides access to structured data between different Android applications. This data is exposed to applications either as tables of data (in much the same way as a SQLite database) or as a handle to a file. This essentially involves the implementation of a client/server arrangement whereby the application seeking access to the data is the client and the content provider is the server, performing actions and returning results on behalf of the client.

A successful content provider implementation involves several elements, each of which will be covered in detail in the remainder of this chapter.

66.2 The Content Provider

A content provider is created as a subclass of the *android.content.ContentProvider* class. Typically, the application responsible for managing the data to be shared will implement a content provider to facilitate sharing of that data with other applications.

Creating a content provider involves implementing methods to manage the data on behalf of other client applications. These methods are as follows:

66.2.1 onCreate()

This method is called when the content provider is first created and should be used to perform any initialization tasks required by the content provider.

66.2.2 query()

This method will be called when a client requests that data be retrieved from the content provider. This method identifies the data to be retrieved (single or multiple rows), performs the data extraction, and returns the results wrapped in a Cursor object.

66.2.3 insert()

This method is called when a new row needs to be inserted into the provider database. This method must identify the destination for the data, perform the insertion and return the full URI of the newly added row.

66.2.4 update()

The method called when existing rows need to be updated on behalf of the client. The method uses the arguments passed through to update the appropriate table rows and return the number of rows updated as a result of the operation.

66.2.5 delete()

Called when rows are to be deleted from a table. This method deletes the designated rows and returns a count of the number of rows deleted.

66.2.6 getType()

Returns the MIME type of the data stored by the content provider.

It is important when implementing these methods in a content provider to keep in mind that, with the exception of the *onCreate()* method, they can be called from many processes simultaneously and must, therefore, be thread safe.

Once a content provider has been implemented, the issue that then arises is how the provider is identified within the Android system. This is where the *content URI* comes into play.

66.3 The Content URI

An Android device will potentially contain several content providers. The system must, therefore, provide some way of identifying one provider from another. Similarly, a single content provider may provide access to multiple forms of content (typically in the form of database tables). Client applications, therefore, need a way to specify the underlying data for which access is required. This is achieved using content URIs.

The content URI is used to identify specific data within a specific content provider. The Authority section of the URI identifies the content provider and usually takes the form of the package name of the content provider. For example:

```
com.example.mydbapp.myprovider
```

A specific database table within the provider data structure may be referenced by appending the table name to the authority. For example, the following URI references a table named *products* within the content provider:

```
com.example.mydbapp.myprovider/products
```

Similarly, a specific row within the specified table may be referenced by appending the row ID to the URI. The following URI, for example, references the row in the products table in which the value stored in the ID column equals 3:

```
com.example.mydbapp.myprovider/products/3
```

When implementing the insert, query, update and delete methods in the content provider, it will be the responsibility of these methods to identify whether the incoming URI is targeting a specific row in a table, or references multiple rows, and act accordingly. This can potentially be a complex task given that a URI can extend to multiple levels. This process can, however, be eased significantly using the *UriMatcher* class, as will be outlined in the next chapter.

66.4 The Content Resolver

Access to a content provider is achieved via a *ContentResolver* object. An application can obtain a reference to its content resolver by calling the *getContentResolver()* method of the application context.

The content resolver object contains a set of methods that mirror those of the content provider (insert, query, delete etc.). The application simply makes calls to the methods, specifying the URI of the content on which the

operation is to be performed. The content resolver and content provider objects then communicate to perform the requested task on behalf of the application.

66.5 The <provider> Manifest Element

For a content provider to be visible within an Android system, it must be declared within the Android manifest file for the application in which it resides. This is achieved using the <provider> element, which must contain the following items:

- **android:authority** – The full authority URI of the content provider. For example `com.example.mydbapp.mydbapp.myprovider`.
- **android:name** – The name of the class that implements the content provider. In most cases, this will use the same value as the authority.

Similarly, the <provider> element may be used to define the permissions that must be held by client applications in order to qualify for access to the underlying data. If no permissions are declared, the default behavior is for permission to be allowed for all applications.

Permissions can be set to cover the entire content provider, or limited to specific tables and records.

66.6 Summary

The data belonging to an application is typically private to the application and inaccessible to other applications. Setting up a content provider is necessary when the data needs to be shared. This chapter has covered the basic elements that combine to enable data sharing between applications and outlined the concepts of the content provider, content URI, and content resolver.

In the next chapter, the SQLdemo project created previously will be extended to make the underlying customer data available via a content provider.

69. The Android Room Persistence Library

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

69.1 Revisiting Modern App Architecture

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

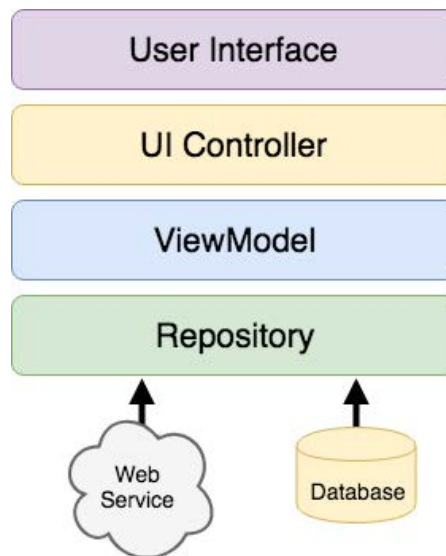


Figure 69-1

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

69.2 Key Elements of Room Database Persistence

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

69.2.1 Repository

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

69.2.2 Room Database

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

69.2.3 Data Access Object (DAO)

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

69.2.4 Entities

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

69.2.5 SQLite Database

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

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

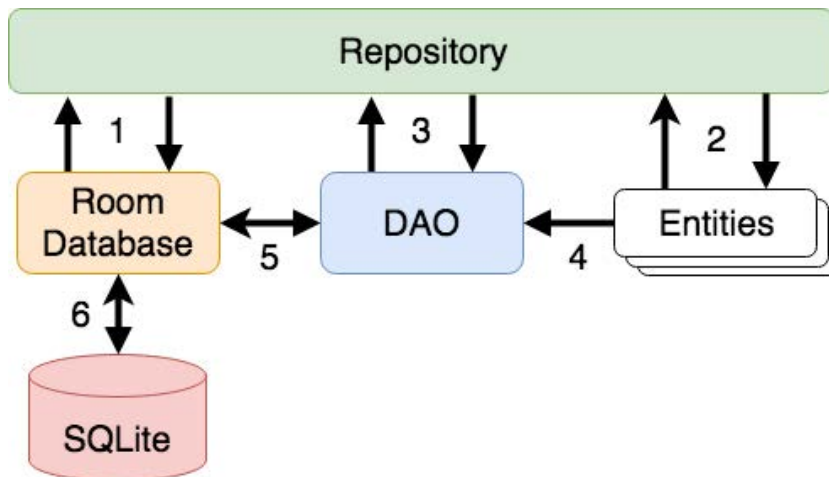


Figure 69-2

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

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

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

69.3 Understanding Entities

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

```
public class Customer {

    private int id;
    private String name;
    private String address;

    public Customer(String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public int getId() {
        return this.id;
    }

    public String getName() {
        return this.name;
    }

    public int getAddress() {
        return this.address;
    }

    public void setId(int id) {
```

```
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAddress(int quantity) {
        this.address = address;
    }
}
```

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

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

    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "customerId")
    private int id;

    @ColumnInfo(name = "customerName")
    private String name;

    private String address;

    public Customer(String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public int getId() {
        return this.id;
    }

    public String getName() {
        return this.name;
    }

    public String getAddress() {
        return this.address;
    }
}
```



```

    public void setId(@NonNull int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAddress(int quantity) {
        this.address = address;
    }
}

```

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

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

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

```

@PrimaryKey(autoGenerate = true)
@NonNull
@ColumnInfo(name = "customerId")
private int id;

```

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

```

@Ignore
private String myString;

```

Finally, the setter method for the id variable is modified to prevent attempts to assign a null value:

```

public void setId(@NonNull int id) {
    this.id = id;
}

```

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

```

@Entity(foreignKeys = {@ForeignKey(entity = Customer.class,
    parentColumns = "customerId",
    childColumns = "buyerId",
    onDelete = ForeignKey.CASCADE,
    onUpdate = ForeignKey.RESTRICT})
public class Purchase {

```

```
@PrimaryKey(autoGenerate = true)
@ColumnInfo(name = "purchaseId")
private int purchaseId;

@ColumnInfo(name = "buyerId")
private int buyerId;

}
```

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

69.4 Data Access Objects

A Data Access Object allows access to the data stored within a SQLite database. A DAO is declared as a standard Java interface with additional annotations that map specific SQL statements to methods that the repository may then call.

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

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

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

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

The `getAllCustomers()` method returns a List object containing a Customer entity object for each record retrieved from the database table. The DAO is also using LiveData so that the repository can observe changes to the database.

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

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

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

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

```
@Insert
void addCustomer(Customer customer);
```

This is referred to as a convenience annotation because the Room persistence library can infer that the Customer

entity passed to the *addCustomer()* method is to be inserted into the database without the need for the SQL insert statement to be provided. Multiple database records may also be inserted in a single transaction as follows:

```
@Insert
public void insertCustomers(Customer... customers);
```

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

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

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

```
@Delete
public void deleteCustomers(Customer... customers);
```

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

```
@Update
public void updateCustomers(Customer... customers);
```

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

```
@Delete
public int deleteCustomers(Customer... customers);
```

69.5 The Room Database

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

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

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

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

@Database(entities = {Customer.class}, version = 1)
public class CustomerRoomDatabase extends RoomDatabase {

    public abstract CustomerDao customerDao();

    private static CustomerRoomDatabase INSTANCE;

    static CustomerRoomDatabase getDatabase(final Context context) {
        if (INSTANCE == null) {
```

```
synchronized (CustomerRoomDatabase.class) {  
    if (INSTANCE == null) {  
        INSTANCE = Room.databaseBuilder(  
            context.getApplicationContext(),  
            CustomerRoomDatabase.class, "customer_database")  
                .build();  
    }  
}  
}  
return INSTANCE;  
}  
}
```

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

69.6 The Repository

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

```
public class CustomerRepository {  
  
    private CustomerDao customerDao;  
    private CustomerRoomDatabase db;  
  
    public CustomerRepository(Application application) {  
        db = CustomerRoomDatabase.getDatabase(application);  
        customerDao = db.customerDao();  
    }  
.  
.  
}
```

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

```
private LiveData<List<Customer>> allCustomers;  
allCustomers = customerDao.getAllCustomers();
```

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

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

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

Handlers and Executors”).

69.7 In-Memory Databases

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

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

```
// Create a file storage based database
INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                                CustomerRoomDatabase.class, "customer_database")
                                .build();

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

69.8 Database Inspector

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

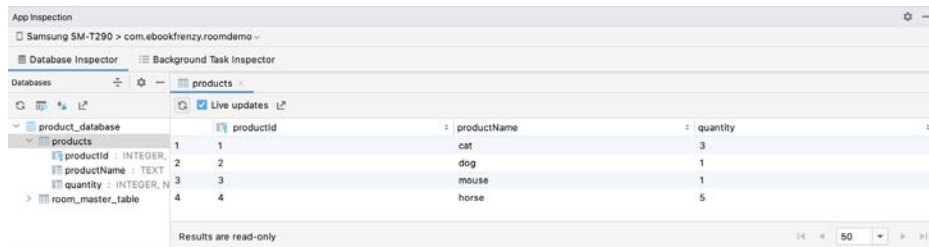


Figure 69-3

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

69.9 Summary

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

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

82. An Introduction to Android App Links

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

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

82.1 An Overview of Android App Links

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

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

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

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

82.2 App Link Intent Filters

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

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

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

        <data
            android:scheme="http"
```

An Introduction to Android App Links

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

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

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

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

82.3 Handling App Link Intents

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

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

```
Intent appLinkIntent = getIntent();
String appLinkAction = appLinkIntent.getAction();
Uri appLinkData = appLinkIntent.getData();
```

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

```
String userType = appLinkData.getLastPathSegment();
```

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

82.4 Associating the App with a Website

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

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

```
[{
```



```

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

```

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

82.5 Summary

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

Index

Symbols

<application> 432
 <fragment> 247
 <fragment> element 247
 <menu> 761
 <provider> 559
 <receiver> 466
 <service> 432, 476, 483
 .well-known folder 439, 462, 706

A

AbsoluteLayout 126
 ACCESS_COARSE_LOCATION permission 500
 ACCESS_FINE_LOCATION permission 500
 acknowledgePurchase() method 745
 ACTION_CREATE_DOCUMENT 622
 ACTION_CREATE_INTENT 622
 ACTION_DOWN 222
 ACTION_MOVE 222
 ACTION_OPEN_DOCUMENT intent 614
 ACTION_POINTER_DOWN 222
 ACTION_POINTER_UP 222
 ACTION_UP 222
 ACTION_VIEW 457
 Active / Running state 100
 Activity 87, 103
 adding views in Java code 203
 class 103
 creation 16
 Entire Lifetime 107
 Foreground Lifetime 107
 lifecycle methods 106
 lifecycles 97
 returning data from 436
 state change example 111

 state changes 103
 states 100
 Visible Lifetime 107
 Activity Lifecycle 99
 Activity Manager 86
 ActivityResultLauncher 437
 Activity Stack 99
 Actual screen pixels 194
 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 465
 addMarker() method 670
 addView() method 197
 ADD_VOICEMAIL permission 500
 android
 checkableBehavior 763
 exported 433
 gestureColor 240
 layout_behavior property 417
 onClick 249
 orderInCategory 762
 process 433, 483
 uncertainGestureColor 240
 Android
 Activity 87
 architecture 83

Index

- events 215
- intents 88
- onClick Resource 215
- runtime 84
- SDK Packages 6
- android.app 84
- Android Architecture Components 265
- android.content 84
- android.content.Intent 435
- android.database 84
- Android Debug Bridge. *See* ADB
- Android Development
 - System Requirements 3
- Android Devices
 - designing for different 125
- android.graphics 85
- android.hardware 85
- android.intent.action 471
- android.intent.action.BOOT_COMPLETED 433
- android.intent.action.MAIN 457
- android.intent.category.LAUNCHER 457
- Android Libraries 84
- android.media 85
- Android Monitor tool window 36
- Android Native Development Kit 85
- android.net 85
- android.opengl 85
- android.os 85
- android.permission.RECORD_AUDIO 649
- 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 614
- 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 128
- android.view.ViewGroup 125, 128
- Android Virtual Device. *See* AVD
 - overview 31
- Android Virtual Device Manager 31
- android.webkit 85
- android.widget 85
- AndroidX libraries 784
- API Key 661
- APK analyzer 738
- APK file 731
- APK File
 - analyzing 738
- APK Signing 784
- APK Wizard dialog 730
- app
 - showAsAction 762
- App Architecture

- modern 265
- AppBar
 - anatomy of 415
- appbar_scrolling_view_behavior 417
- App Bundles 727
 - creating 731
 - overview 727
 - revisions 737
 - uploading 734
- AppCompatActivity class 104
- App Inspector 57
- Application
 - stopping 36
- Application Context 89
- Application Framework 86
- Application Manifest 89
- Application Resources 89
- App Link
 - Adding Intent Filter 714
 - Digital Asset Links file 706, 439
 - Intent Filter Handling 714
 - Intent Filters 705
 - Intent Handling 706
 - Testing 718
 - URL Mapping 711
- App Links 705
 - auto verification 438
 - autoVerify 439
 - overview 705
- Apply Changes 211
 - Apply Changes and Restart Activity 211
 - Apply Code Changes 211
 - fallback settings 213
 - options 211
 - Run App 211
 - tutorial 213
- applyToActivitiesIfAvailable() method 779
- Architecture Components 265
- ART 84
- assetlinks.json , 706, 439
- Attribute Keyframes 342
- Audio

- supported formats 647
- Audio Playback 647
- Audio Recording 647
- Auto Blocker 64
- Autoconnect Mode 159
- Automatic Link Verification 438, 461
- autoVerify 439, 714
- AVD

- Change posture 51
- 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 98
- Barriers 152
 - adding 171
 - constrained views 152
- Baseline Alignment 151
- beginTransaction() method 248
- BillingClient 746
 - acknowledgePurchase() method 745
 - consumeAsync() method 745
 - getPurchaseState() method 745
 - initialization 742, 753
 - launchBillingFlow() method 744
 - queryProductDetailsAsync() method 744
 - queryPurchasesAsync() method 746
- BillingResult 760
 - getDebugMessage() 760
- Binding Expressions 289

Index

- one-way 289
- two-way 290
- BIND_JOB_SERVICE permission 433
- bindService() method 431, 473, 478
- Biometric Authentication 719
 - callbacks 723
 - overview 719
 - tutorial 719
- Biometric Prompt 724
- BitmapFactory 615
- black activity 16
- Blank template 129
- Blueprint view 157
- BODY_SENSORS permission 500
- Bound Service 431, 473
 - adding to a project 474
 - Implementing the Binder 474
 - Interaction options 473
- BoundService class 475
- Broadcast Intent 465
 - example 468
 - overview 88, 465
 - sending 468
 - Sticky 467
- Broadcast Receiver 465
 - adding to manifest file 470
 - creation 469
 - overview 88, 466
- BroadcastReceiver class 466
- BroadcastReceiver superclass 469
- BufferedReader object 625
- Build tool window 58
- Build Variants , 58
 - tool window 58
- Bundle class 120
- Bundled Notifications 519

C

- Calendar permissions 500
- CALL_PHONE permission 500
- CAMERA permission 500
- Camera permissions 500

- CameraUpdateFactory class
 - methods 671
- CancellationSignal 724
- Canvas class 700
- CardView
 - layout file 395
 - responding to selection of 403
- CardView class 395
- CATEGORY_OPENABLE 614
- C/C++ Libraries 85
- Chain bias 180
- chain head 150
- chains 150
- Chains
 - creation of 177
- Chain style
 - changing 179
- chain styles 150
- CheckBox 125
- checkSelfPermission() method 504
- Circle class 657
- Code completion 74
- Code Editor
 - basics 71
 - Code completion 74
 - Code Generation 77
 - 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 77
- Code Reformatting 79
- code samples
 - download 1
- cold boot 48
- CollapsingToolbarLayout
 - example 418
 - introduction 418

- parallax mode 418
- pin mode 418
- setting scrim color 421
- setting title 421
- with image 418
- Color class 701
- COLOR_MODE_COLOR 676, 696
- COLOR_MODE_MONOCHROME 676, 696
- Common Gestures 229
 - detection 229
- Component tree 20
- Constraint Bias 149
 - adjusting 163
- ConstraintLayout
 - advantages of 155
 - Availability 156
 - Barriers 152
 - Baseline Alignment 151
 - chain bias 180
 - chain head 150
 - chains 150
 - chain styles 150
 - Constraint Bias 149
 - Constraints 147
 - conversion to 175
 - convert to MotionLayout 349
 - deleting constraints 162
 - guidelines 169
 - Guidelines 152
 - manual constraint manipulation 159
 - Margins 148, 163
 - Opposing Constraints 148, 165
 - overview of 147
 - Packed chain 151, 180
 - ratios 155, 181
 - Spread chain 150
 - Spread inside 180
 - Spread inside chain 150
 - tutorial 185
 - using in Android Studio 157
 - Weighted chain 150, 180
 - Widget Dimensions 151, 167
 - Widget Group Alignment 173
- ConstraintLayout chains
 - creation of 177
 - in layout editor 177
- ConstraintLayout Chain style
 - changing 179
- Constraints
 - deleting 162
- ConstraintSet
 - addToHorizontalChain() method 200
 - addToVerticalChain() method 200
 - alignment constraints 199
 - apply to layout 198
 - applyTo() method 198
 - centerHorizontally() method 199
 - centerVertically() method 199
 - chains 199
 - clear() method 200
 - clone() method 199
 - connect() method 198
 - connect to parent 198
 - constraint bias 199
 - copying constraints 199
 - create 198
 - create connection 198
 - createHorizontalChain() method 199
 - createVerticalChain() method 199
 - guidelines 200
 - removeFromHorizontalChain() method 200
 - removeFromVerticalChain() method 200
 - removing constraints 200
 - rotation 201
 - scaling 200
 - setGuidelineBegin() method 200
 - setGuidelineEnd() method 200
 - setGuidelinePercent() method 200
 - setHorizontalBias() method 199
 - setRotationX() method 201
 - setRotationY() method 201
 - setScaleX() method 200
 - setScaleY() method 200
 - setTransformPivot() method 201

Index

- setTransformPivotX() method 201
- setTransformPivotY() method 201
- setVerticalBias() method 199
- sizing constraints 199
- tutorial 203
- view IDs 205
- ConstraintSet class 197, 198
- Constraint Sets 198
- ConstraintSets
 - configuring 338
- consumeAsync() method 745
- ConsumeParams 757
- ConsumeResponseListener 745
- Contacts permissions 500
- container view 125
- Content Provider 86, 557, 575
 - <provider> 559
 - accessing 575
 - Authority 563
 - client tutorial 575
 - ContentProvider class 557
 - Content Resolver 558
 - ContentResolver 571
 - content URI 558
 - Content URI 563, 575
 - ContentValues 565
 - delete() 558, 569
 - getType() 558
 - insert() 557, 565
 - onCreate() 557, 565
 - overview 89
 - query() 557, 566
 - tutorial 561
 - update() 558, 567
 - UriMatcher 564
 - UriMatcher class 558
- ContentProvider class 557
- Content Resolver 558
 - getContentResolver() 558
- ContentResolver 571
 - getContentResolver() 558
- content URI 558
- Content URI 558, 563
- ContentValues 565
- Context class 89
- CoordinatorLayout 126, 417
- createPrintDocumentAdapter() method 691
- Custom Attribute 339
- Custom Document Printing 679, 691
- Custom Gesture
 - recognition 235
- Custom Print Adapter
 - implementation 693
- Custom Print Adapters 691
- Custom Theme
 - building 773
- Cycle Editor 367
- Cycle Keyframe 347
- Cycle Keyframes
 - overview 363

D

- dangerous permissions
 - list of 500
- Dark Theme 36
 - enable on device 36
- Data Access Object (DAO) 580
- Data Access Objects (DAO) 584
- Database Inspector 587, 610
 - live updates 611
 - SQL query 611
- Database Rows 544
- Database Schema 543
- Database Tables 543
- Data binding
 - binding expressions 289
- Data Binding 267
 - binding classes 288
 - enabling 294
 - event and listener binding 290
 - key components 285
 - overview 285
 - tutorial 293
 - with LiveData 267

- DDMS 36
- Debugging
 - enabling on device 63
- debug.keystore file 439, 461
- DefaultLifecycleObserver 306, 309
- deltaRelative 344
- Density-independent pixels 193
- Density Independent Pixels
 - converting to pixels 208
- Device Definition
 - custom 143
- Device File Explorer 58
- device frame 40
- Device Mirroring 69
 - enabling 69
- device pairing 67
- Digital Asset Links file 706, 439, 439
- Direct Reply Input 530
- document provider 613
- dp 193
- Dynamic Colors
 - applyToActivitiesIfAvailable() method 779
 - enabling in Android 779
- Dynamic State 105
 - saving 119

E

- Empty Process 99
- Empty template 129
- 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 745
- enabling ADB support 63
- etings.gradle file 784
- Event Handling 215
 - example 216
- Event Listener 217
- Event Listeners 216
- Events
 - consuming 219
- execSQL() 552
- explicit
 - intent 88
- explicit intent 435
- Explicit Intent 435
- Extended Control
 - options 45

F

- Files
 - switching between 72
- findPointerIndex() method 222
- findViewById() 91
- Fingerprint
 - emulation 48
- Fingerprint authentication
 - device configuration 720
 - permission 720
 - steps to implement 719
- Fingerprint Authentication
 - overview 719
 - tutorial 719
- FLAG_INCLUDE_STOPPED_PACKAGES 465
- flexible space area 415

Index

floating action button 16, 130
 changing appearance of 378
 margins 376
 removing 131
 sizes 376

Foldable Devices 108
 multi-resume 108

Foldable Emulator 536

Foldables 535

Foreground Process 98

Forward-geocoding 663

Fragment
 creation 245
 event handling 249
 XML file 246

FragmentManager class 105

Fragment Communication 250

Fragments 245
 adding in code 248
 duplicating 384
 example 253
 overview 245

FragmentManager class 387

FrameLayout 126

G

Geocoder object 664

Geocoding 662

Gesture Builder Application 235
 building and running 235

Gesture Detector class 229

GestureDetectorCompat 232
 instance creation 232

GestureDetectorCompat class 229

GestureDetector.OnDoubleTapListener 229, 230

GestureDetector.OnGestureListener 230

GestureLibrary 235

GestureOverlayView 235
 configuring color 240
 configuring multiple strokes 240

GestureOverlayView class 235

GesturePerformedListener 235

Gestures
 interception of 241

Gestures File
 creation 236
 extract from SD card 236
 loading into application 238

GET_ACCOUNTS permission 500

getAction() method 471

getContentResolver() 558

getDebugMessage() 760

getFromLocation() method 664

getId() method 198

getIntent() method 436

getItemId() method 763

getPointerCount() method 222

getPointerId() method 222

getPurchaseState() method 745

getService() method 477

getWritableDatabase() 553

GNU/Linux 84

Google Cloud
 billing account 658
 new project 659

Google Cloud Print 674

Google Drive 614
 printing to 674

GoogleMap 657
 map types 667

GoogleMap.MAP_TYPE_HYBRID 667

GoogleMap.MAP_TYPE_NONE 667

GoogleMap.MAP_TYPE_NORMAL 667

GoogleMap.MAP_TYPE_SATELLITE 667

GoogleMap.MAP_TYPE_TERRAIN 667

Google Maps Android API 657
 Controlling the Map Camera 671
 displaying controls 668

Map Markers 670

overview 657

Google Maps SDK 657
 API Key 661
 Credentials 661
 enabling 660

- Maps SDK for Android 661
- Google Play App Signing 730
- Google Play Console 751
 - Creating an in-app product 751
 - License Testers 752
- Google Play Developer Console 728
- Gradle
 - APK signing settings 788
 - Build Variants 784
 - command line tasks 789
 - dependencies 783
 - Manifest Entries 784
 - overview 783
 - sensible defaults 783
- Gradle Build File
 - top level 785
- Gradle Build Files
 - module level 786
- gradle.properties file 784
- GridLayout 126
- GridLayoutManager 393

H

- HAL 84
- Handler class 482
- Hardware Abstraction Layer 84
- HP Print Services Plugin 673
- HTML printing 677
- HTML Printing
 - example 681

I

- IBinder 431, 475
- IBinder object 473, 483
- Image Printing 676
- implicit
 - intent 88
- implicit intent 435
- Implicit Intent 437
- Implicit Intents
 - example 453
- importance hierarchy 97

- in 193
- INAPP 746
- In-App Products 741
- In-App Purchasing 749
 - acknowledgePurchase() method 745
 - BillingClient 742
 - BillingResult 760
 - consumeAsync() method 745
 - ConsumeParams 757
 - ConsumeResponseListener 745
 - Consuming purchases 757
 - enablePendingPurchases() method 745
 - getPurchaseState() method 745
 - launchBillingFlow() method 744
- Libraries 749
 - newBuilder() method 742
- onBillingServiceDisconnected() callback 754
- onBillingServiceDisconnected() method 743
- onBillingSetupFinished() listener 754
- onProductDetailsResponse() callback 754
- Overview 741
- ProductDetail 744
- ProductDetails 755
- products 741
- ProductType 746
- ProductType.INAPP 746
- ProductType.SUBS 746
- Purchase Flow 756
- PurchaseResponseListener 746
- PurchasesUpdatedListener 745
- PurchaseUpdatedListener 756
- purchase updates 756
- queryProductDetailsAsync() 754
- queryProductDetailsAsync() method 744
- queryPurchasesAsync() 758
- queryPurchasesAsync() method 746
- runOnUiThread() 755
- subscriptions 741
- tutorial 749
- In-Memory Database 587
- Intent 88
 - explicit 88

Index

- implicit 88
- Intent Availability
 - checking for 442
- Intent.CATEGORY_OPENABLE 622
- Intent Filters 438
 - App Link 705
- Intents 435
 - ActivityResultLauncher 437
 - overview 435
 - registerForActivityResult() 450
- Intent Service 431
- Intent URL 455

J

- Java Native Interface 85
- Jetpack 265
 - overview 265
- JobIntentService 431
 - BIND_JOB_SERVICE permission 433
 - onHandleWork() method 431

K

- KeyAttribute 342
- Keyboard Shortcuts 59
- KeyCycle 363
 - Cycle Editor 367
 - tutorial 363
- Keyframe 356
- Keyframes 342
- KeyFrameSet 372
- KeyPosition 343
 - deltaRelative 344
 - parentRelative 343
 - pathRelative 344
- Keystore File
 - creation 730
- KeyTimeCycle 363
- keytool 439
- KeyTrigger 346
- Killed state 100

L

- launchBillingFlow() method 744
- layout_collapseMode
 - parallax 420
 - pin 420
- layout_constraintDimentionRatio 182
- layout_constraintHorizontal_bias 180
- layout_constraintVertical_bias 180
- layout editor
 - ConstraintLayout chains 177
- Layout Editor 19, 185
 - Autoconnect Mode 159
 - code mode 136
 - Component Tree 134
 - design mode 133
 - device screen 134
 - example project 185
 - Inference Mode 159
 - palette 134
 - properties panel 134
 - Sample Data 142
 - Setting Properties 137
 - toolbar 134
 - user interface design 185
 - view conversion 141
- Layout Editor Tool
 - changing orientation 20
 - overview 133
- Layout Inspector 58
- Layout Managers 125
- LayoutResultCallback object 696
- Layouts 125
- layout_scrollFlags
 - enterAlwaysCollapsed mode 417
 - enterAlways mode 417
 - exitUntilCollapsed mode 417
 - scroll mode 417
- Layout Validation 144
- libc 85
- License Testers 752
- Lifecycle
 - awareness 305
 - components 268

- owners 305
- states and events 307
- tutorial 309
- Lifecycle-Aware Components 305
- Lifecycle Methods 106
- Lifecycle Observer 309
 - creating a 309
- Lifecycle Owner
 - creating a 311
- Lifecycles
 - modern 268
- LinearLayout 126
- LinearLayoutManager 393
- LinearLayoutManager layout 402
- Linux Kernel 84
- list devices 63
- LiveData 266, 279
 - adding to ViewModel 279
 - observer 281
 - tutorial 279
- Live Templates 80
- Local Bound Service 473
 - example 473
- Location Manager 86
- Location permission 500
- Logcat
 - tool window 57
- LogCat
 - enabling 115

M

- MANAGE_EXTERNAL_STORAGE 501
 - adb enabling 501
 - testing 501
- Manifest File
 - permissions 457
- Maps 657
- MapView 657
 - adding to a layout 664
- Marker class 657
- Master/Detail Flow
 - creation 424

- two pane mode 423
- match_parent properties 193
- Material design 375
- Material Design 2 771
- Material Design 2 Theming 771
- Material Design 3 771
- Material Theme Builder 773
- Material You 771
- MediaController
 - adding to VideoView instance 631
- MediaController class 628
 - methods 628
- MediaPlayer class 647
 - methods 647
- MediaRecorder class 647
 - methods 648
 - recording audio 648
- Memory Indicator 73
- Menu Editor 764
- Menu Item Selections 762
- Menus 761
 - menu editor 764
- Messenger object 483
- Microphone
 - checking for availability 650
- Microphone permissions 500
- mm 193
- MotionEvent 221, 222, 243
 - getActionMasked() 222
- MotionLayout 337
 - arc motion 342
 - Attribute Keyframes 342
 - ConstraintSets 338
 - Custom Attribute 358
 - Custom Attributes 339
 - Cycle Editor 367
 - Editor 349
 - KeyAttribute 342
 - KeyCycle 363
 - Keyframes 342
 - KeyFrameSet 372
 - KeyPosition 343

Index

- KeyTimeCycle 363
- KeyTrigger 346
- OnClick 341, 354
- OnSwipe 341
- overview 337
- Position Keyframes 343
- previewing animation 354
- Trigger Keyframe 346
- Tutorial 349
- MotionScene
 - ConstraintSets 338
 - Custom Attributes 339
 - file 338
 - overview 337
 - transition 338
- moveCamera() method 671
- multiple devices
 - testing app on 35
- Multiple Touches
 - handling 222
- multi-resume 108
- Multi-Touch
 - example 222
- Multi-touch Event Handling 221
- Multi-Window
 - attributes 539
- Multi-Window Mode
 - detecting 540
 - entering 537
 - launching activity into 541
- Multi-Window Notifications 540
- multi-window support 108
- Multi-Window Support
 - enabling 538
- My Location Layer 657

N

- Navigation 315
 - adding destinations 324
 - overview 315
 - pass data with safeargs 332
 - passing arguments 320

- stack 315
- tutorial 321
- Navigation Action
 - triggering 319
- Navigation Architecture Component 315
- Navigation Component
 - tutorial 321
- Navigation Controller
 - accessing 319
- Navigation Graph 318, 322
 - adding actions 328
 - creating a 322
- Navigation Host 316
 - declaring 323
- newBuilder() method 742
- normal permissions 499
- Notification
 - adding actions 518
 - Direct Reply Input 530
 - issuing a basic 514
 - launch activity from a 516
 - PendingIntent 526
 - Reply Action 528
 - updating direct reply 531
- Notifications
 - bundled 519
 - overview 507
- Notifications Manager 86

O

- Observer
 - implementing a LiveData 281
- onAttach() method 250
- onBillingServiceDisconnected() callback 754
- onBillingServiceDisconnected() method 743
- onBillingSetupFinished() listener 754
- onBind() method 432, 473, 481
- onBindViewHolder() method 401
- OnClick 341
- onClickListener 216, 217, 220
- onClick() method 215
- onCreateContextMenuListener 216

- onCreate() method 98, 106, 432
- onCreateOptionsMenu() method 762
- onCreateView() method 107
- onDestroy() method 106, 432
- onDoubleTap() method 229
- onDown() method 229
- onFling() method 229
- onFocusChangeListener 216
- OnFragmentInteractionListener
 - implementation 329
- onGesturePerformed() method 235
- onHandleWork() method 432
- onKeyListener 216
- onLayoutFailed() method 696
- onLayoutFinished() method 697
- onLongClickListener 216
- onLongClick() method 219
- onLongPress() method 229
- onMapReady() method 666
- onOptionsItemSelected() method 762
- onOptionsItemsSelected() method 767
- onPageFinished() callback 682
- onPause() method 106
- onProductDetailsResponse() callback 754
- onReceive() method 98, 466, 467, 469
- onRequestPermissionsResult() method 503, 654, 512, 524
- onRestart() method 106
- onRestoreInstanceState() method 107
- onResume() method 98, 106
- onSaveInstanceState() method 107
- onScaleBegin() method 241
- onScaleEnd() method 241
- onScale() method 241
- onScroll() method 229
- OnSeekBarChangeListener 260
- onServiceConnected() method 473, 477, 484
- onServiceDisconnected() method 473, 477, 484
- onShowPress() method 229
- onSingleTapUp() method 229
- onStartCommand() method 432
- onStart() method 106
- onStop() method 106

- onTouchEvent() method 229, 241
- onTouchListener 216
- onTouch() method 221
- onUpgrade() 552
- onViewCreated() method 107
- onViewStatusRestored() method 107
- openFileDescriptor() method 614
- OpenJDK 3
- Overflow Menu 761
 - creation 761
 - displaying 762
 - overview 761
 - XML file 761
- Overflow Menus
 - Checkable Item Groups 763

P

- Package Explorer 18
- Package Manager 86
- PackageManager class 650
- PackageManager.FEATURE_MICROPHONE 650
- PackageManager.PERMISSION_DENIED 501
- PackageManager.PERMISSION_GRANTED 501
- Package Name 16
- Packed chain 151, 180
- PageRange 698, 699
- Paint class 701
- parentRelative 343
- parent view 127
- pathRelative 344
- Paused state 100
- PdfDocument 679
- PdfDocument.Page 691, 698
- PendingIntent class 526
- Permission
 - checking for 501
- permissions
 - normal 499
- Persistent State 105
- Phone permissions 500
- picker 613
- Pinch Gesture

Index

- detection 241
- example 241
- Pinch Gesture Recognition 235
- Position Keyframes 343
- POST_NOTIFICATIONS permission 500, 524
- PrintAttributes 696
- PrintDocumentAdapter 679, 691
- Printing
 - color 676
 - monochrome 676
- Printing framework
 - architecture 673
- Printing Framework 673
- Print Job
 - starting 702
- PrintManager service 683
- Problems
 - tool window 58
- process
 - priority 97
 - state 97
- PROCESS_OUTGOING_CALLS permission 500
- Process States 97
- ProductDetail 744
- ProductDetails 755
- ProductType 746
- Profiler
 - tool window 58
- ProgressBar 125
- proguard-rules.pro file 788
- ProGuard Support 784
- Project Name 16
- Project tool window 18, 57
- pt 193
- PurchaseResponseListener 746
- PurchasesUpdatedListener 745
- PurchaseUpdatedListener 756
- putExtra() method 435, 465
- px 194

Q

- queryProductDetailsAsync() 754

- queryProductDetailsAsync() method 744
- queryPurchaseHistoryAsync() method 746
- queryPurchasesAsync() 758
- queryPurchasesAsync() method 746
- quickboot snapshot 48
- Quick Documentation 79

R

- RadioButton 125
- ratios 181
- READ_CALENDAR permission 500
- READ_CALL_LOG permission 500
- READ_CONTACTS permission 500
- READ_EXTERNAL_STORAGE permission 501
- READ_PHONE_STATE permission 500
- READ_SMS permission 500
- RECEIVE_MMS permission 500
- RECEIVE_SMS permission 500
- RECEIVE_WAP_PUSH permission 500
- Recent Files Navigation 60
- RECORD_AUDIO permission 500
- Recording Audio
 - permission 649
- RecyclerView 393
 - adding to layout file 394
 - LayoutManager 393
 - initializing 402
 - LayoutManager 393
 - StaggeredLayoutManager 393
- RecyclerView Adapter
 - creation of 400
- RecyclerView.Adapter 394, 400
 - getItemCount() method 394
 - onBindViewHolder() method 394
 - onCreateViewHolder() method 394
- RecyclerView.ViewHolder
 - getAdapterPosition() method 404
- registerForActivityResult() method 436, 450
- registerReceiver() method 467
- RelativeLayout 126
- releasePersistableUriPermission() method 617
- Release Preparation 727

- Remote Bound Service 481
 - client communication 481
 - implementation 481
 - manifest file declaration 483
- RemoteInput.Builder() method 526
- RemoteInput Object 526
- Remote Service
 - launching and binding 484
 - sending a message 485
- Repository
 - tutorial 597
- Repository Modules 268
- Resizable Emulator 50
- Resource
 - string creation 23
- Resource File 25
- Resource Management 97
- Resource Manager 57, 86
- result receiver 467
- Reverse-geocoding 663
- Reverse Geocoding 662
- Room
 - Data Access Object (DAO) 580
 - entities 580, 581
 - In-Memory Database 587
 - Repository 580
- Room Database 580
 - tutorial 597
- Room Database Persistence 579
- Room Persistence Library 548, 579
- root element 125
- root view 127
- Run
 - tool window 57
- Running Devices
 - tool window 69
- runOnUiThread() 755

S

- safeargs 332
- Sample Data 142, 407
 - tutorial 407

- Saved State 267, 301
- SavedStateHandle 302, 303
 - contains() method 303
 - keys() method 303
 - remove() method 303
- Saved State module 301
- SavedStateViewModelFactory 302
- ScaleGestureDetector class 241
- Scale-independent 193
- SDK Packages 6
- Secure Sockets Layer (SSL) 85
- SeekBar 253
- sendBroadcast() method 465, 467
- sendOrderedBroadcast() method 465, 467
- SEND_SMS permission 500
- sendStickyBroadcast() method 465
- Sensor permissions 500
- Service
 - anatomy 432
 - launch at system start 433
 - manifest file entry 432
 - overview 88
 - run in separate process 433
- ServiceConnection class 484
- Service Process 98
- Service Restart Options 432
- setAudioEncoder() method 648
- setAudioSource() method 648
- setBackgroundColor() 198
- setCompassEnabled() method 668
- setContentView() method 197, 203
- setId() method 198
- setMyLocationButtonEnabled() method 669
- setOnClickListener() method 215, 217
- setOnDoubleTapListener() method 229, 232
- setOutputFile() method 648
- setOutputFormat() method 648
- setResult() method 437
- setRotateGesturesEnabled() method 669
- setScrollGesturesEnabled() method 669
- setText() method 122
- setTiltGesturesEnabled() method 669

Index

- settings.gradle.kts file 784
- setTransition() 347
- setVideoSource() method 648
- setZoomControlsEnabled() method 668, 669
- SHA-256 certificate fingerprint 439
- shouldOverrideUrlLoading() method 682
- SimpleOnScaleGestureListener 241
- SimpleOnScaleGestureListener class 243
- SMS permissions 500
- Snackbar 375, 376, 377
- Snapshots
 - emulator 47
- sp 193
- Spread chain 150
- Spread inside 180
- Spread inside chain 150
- SQL 544
- SQL CREATE 552
- SQLite 543
 - AVD command-line use 545
 - Columns and Data Types 543
 - overview 544
 - Primary keys 544
 - tutorial 549
- SQLiteDatabase 552
- SQLiteOpenHelper 550, 551
- SQL SELECT 553, 554
- StaggeredGridLayoutManager 393
- startActivity() method 435
- startForeground() method 98
- START_NOT_STICKY 432
- START_REDELIVER_INTENT 432
- START_STICKY 432
- State
 - restoring 122
- State Change
 - handling 101
- Statement Completion 76
- Status Bar Widgets 73
 - Memory Indicator 73
- Sticky Broadcast Intents 467
- Stopped state 100

- Storage Access Framework 613
 - ACTION_CREATE_DOCUMENT 614
 - ACTION_OPEN_DOCUMENT 614
 - deleting a file 617
 - example 619
 - file creation 622
 - file filtering 614
 - file reading 615
 - file writing 616
 - intents 614
 - MIME Types 615
 - Persistent Access 617
 - picker 613
- Storage permissions 501
- StringBuilder object 625
- strings.xml file 27
- Structure
 - tool window 58
- Structured Query Language 544
- Structure tool window 58
- SUBS 746
- subscriptions 741
- SupportMapFragment class 657
- Switcher 60
- System Broadcasts 471
- system requirements 3

T

- TabLayout
 - adding to layout 385
 - app
 - tabGravity property 390
 - tabMode property 390
 - example 382
 - fixed mode 389
 - getItemCount() method 381
 - overview 381
- TableLayout 126, 589
- TableRow 589
- Telephony Manager 86
- Templates
 - blank vs. empty 129

- Terminal
 - tool window 58
- Theme
 - building a custom 773
- Theming 771
 - tutorial 775
- Time Cycle Keyframes 347
- TODO
 - tool window 59
- ToolBarListener 250
- tools
 - layout 247
- Tool window bars 56
- Tool windows 56
- Touch Actions 222
- Touch Event Listener
 - implementation 223
- Touch Events
 - intercepting 221
- Touch handling 221
- U**
- UiSettings class 657
- unbindService() method 431
- unregisterReceiver() method 467
- upload key 730
- UriMatcher 558, 564
- UriMatcher class 558
- URL Mapping 711
- USB connection issues
 - resolving 66
- USE_BIOMETRIC 720
- user interface state 105
- USE_SIP permission 500
- V**
- Video Playback 627
- VideoView class 627
 - methods 627
 - supported formats 627
- view bindings
 - enabling 92
 - using 92
- View class
 - setting properties 205
- view conversion 141
- ViewGroup 125
- View Groups 125
- View Hierarchy 127
- ViewHolder class 394
 - sample implementation 401
- ViewModel
 - adding LiveData 279
 - data access 276
 - overview 266
 - saved state 301
 - Saved State 267, 301
 - tutorial 271
- ViewModelProvider 274
- ViewModel Saved State 301
- ViewPager
 - adding to layout 385
 - example 382
- Views 125
 - Java creation 197
- View System 86
- Virtual Device Configuration dialog 32
- Virtual Sensors 47
- Visible Process 98
- W**
- WebViewClient 677, 682
- WebView view 455
- Weighted chain 150, 180
- Welcome screen 53
- Widget Dimensions 151
- Widget Group Alignment 173
- Widgets palette 186
- WiFi debugging 67
- Wireless debugging 67
- Wireless pairing 67
- wrap_content properties 195
- WRITE_CALENDAR permission 500
- WRITE_CALL_LOG permission 500

Index

WRITE_CONTACTS permission 500

WRITE_EXTERNAL_STORAGE permission 501

X

XML Layout File

 manual creation 193

 vs. Java Code 197