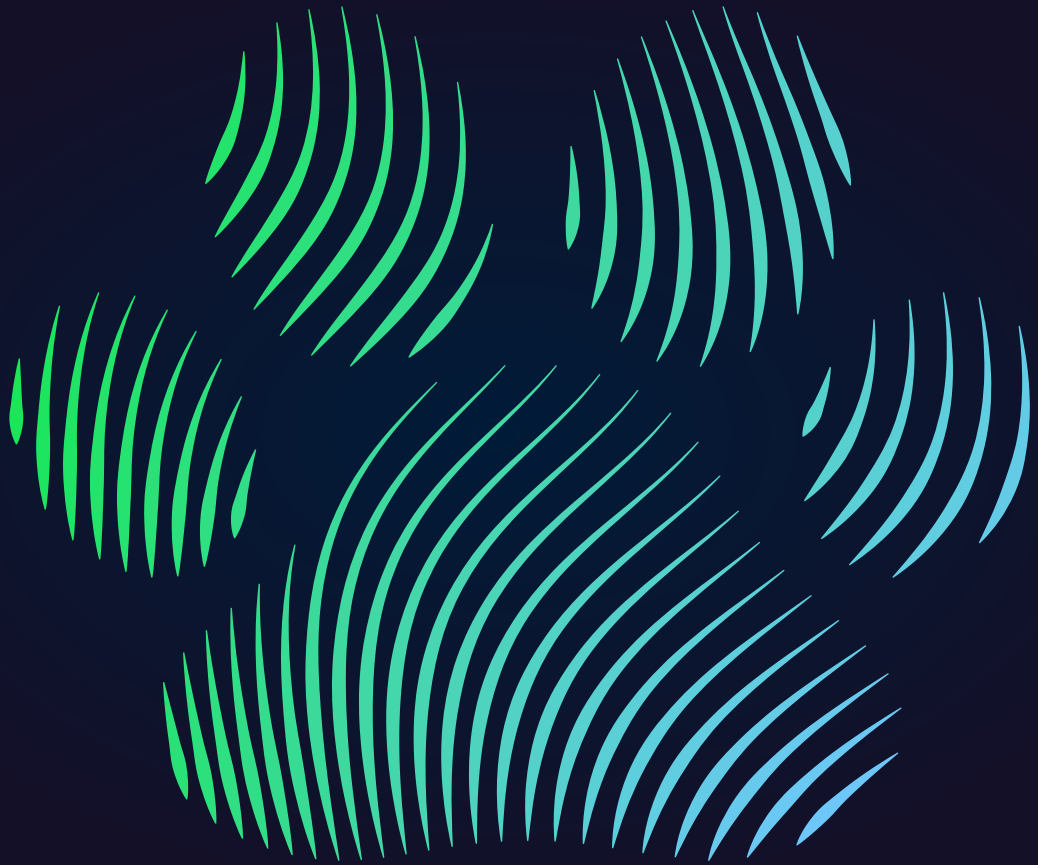


Android Studio Koala Essentials



Java Edition

Android Studio Koala Essentials

Java Edition

Android Studio Koala Essentials – Java Edition

ISBN: 978-1-951442-96-5

© 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



<https://www.payloadbooks.com>

Table of Contents

1. Introduction	1
1.1 Downloading the Code Samples	1
1.2 Feedback	1
1.3 Errata	2
2. Setting up an Android Studio Development Environment	3
2.1 System requirements	3
2.2 Downloading the Android Studio package	3
2.3 Installing Android Studio	4
2.3.1 Installation on Windows	4
2.3.2 Installation on macOS	4
2.3.3 Installation on Linux	5
2.4 Installing additional Android SDK packages	5
2.5 Installing the Android SDK Command-line Tools	8
2.5.1 Windows 8.1	9
2.5.2 Windows 10	10
2.5.3 Windows 11	10
2.5.4 Linux	10
2.5.5 macOS	10
2.6 Android Studio memory management	10
2.7 Updating Android Studio and the SDK	11
2.8 Summary	12
3. Creating an Example Android App in Android Studio	13
3.1 About the Project	13
3.2 Creating a New Android Project	13
3.3 Creating an Activity	14
3.4 Defining the Project and SDK Settings	14
3.5 Modifying the Example Application	15
3.6 Modifying the User Interface	16
3.7 Reviewing the Layout and Resource Files	22
3.8 Adding Interaction	25
3.9 Summary	26
4. Creating an Android Virtual Device (AVD) in Android Studio	27
4.1 About Android Virtual Devices	27
4.2 Starting the Emulator	29
4.3 Running the Application in the AVD	30
4.4 Running on Multiple Devices	31
4.5 Stopping a Running Application	32
4.6 Running the Emulator in a Separate Window	32
4.7 Removing the Device Frame	35
4.8 Summary	37
5. Using and Configuring the Android Studio AVD Emulator	39

Table of Contents

5.1 The Emulator Environment	39
5.2 Emulator Toolbar Options	39
5.3 Working in Zoom Mode	41
5.4 Resizing the Emulator Window.....	41
5.5 Extended Control Options.....	41
5.5.1 Location.....	42
5.5.2 Displays.....	42
5.5.3 Cellular	42
5.5.4 Battery.....	42
5.5.5 Camera.....	42
5.5.6 Phone	42
5.5.7 Directional Pad.....	42
5.5.8 Microphone.....	42
5.5.9 Fingerprint	42
5.5.10 Virtual Sensors	43
5.5.11 Snapshots.....	43
5.5.12 Record and Playback	43
5.5.13 Google Play	43
5.5.14 Settings	43
5.5.15 Help.....	43
5.6 Working with Snapshots.....	43
5.7 Configuring Fingerprint Emulation	44
5.8 The Emulator in Tool Window Mode.....	46
5.9 Common Android Settings.....	46
5.10 Creating a Resizable Emulator.....	47
5.11 Summary	48
6. A Tour of the Android Studio User Interface	49
6.1 The Welcome Screen	49
6.2 The Menu Bar	50
6.3 The Main Window	50
6.4 The Tool Windows	52
6.5 The Tool Window Menus.....	55
6.6 Android Studio Keyboard Shortcuts	56
6.7 Switcher and Recent Files Navigation	56
6.8 Changing the Android Studio Theme	57
6.9 Summary	58
7. Testing Android Studio Apps on a Physical Android Device.....	59
7.1 An Overview of the Android Debug Bridge (ADB).....	59
7.2 Enabling USB Debugging ADB on Android Devices.....	59
7.2.1 macOS ADB Configuration.....	60
7.2.2 Windows ADB Configuration.....	61
7.2.3 Linux adb Configuration.....	62
7.3 Resolving USB Connection Issues	62
7.4 Enabling Wireless Debugging on Android Devices	63
7.5 Testing the adb Connection.....	65
7.6 Device Mirroring.....	65
7.7 Summary	65

8. The Basics of the Android Studio Code Editor	67
8.1 The Android Studio Editor.....	67
8.2 Splitting the Editor Window.....	70
8.3 Code Completion.....	70
8.4 Statement Completion.....	72
8.5 Parameter Information.....	72
8.6 Parameter Name Hints.....	72
8.7 Code Generation.....	73
8.8 Code Folding.....	74
8.9 Quick Documentation Lookup.....	75
8.10 Code Reformatting.....	75
8.11 Finding Sample Code.....	76
8.12 Live Templates.....	76
8.13 Summary.....	77
9. Introducing Gemini in Android Studio	79
9.1 Introducing Gemini AI.....	79
9.2 Enabling Gemini in Android Studio.....	79
9.3 Gemini configuration.....	81
9.4 Asking Gemini questions.....	82
9.5 Question contexts.....	83
9.6 Inline code completion.....	83
9.7 Summary.....	84
10. An Overview of the Android Architecture	85
10.1 The Android Software Stack.....	85
10.2 The Linux Kernel.....	86
10.3 Hardware Abstraction Layer.....	86
10.4 Android Runtime – ART.....	86
10.5 Android Libraries.....	86
10.5.1 C/C++ Libraries.....	87
10.6 Application Framework.....	88
10.7 Applications.....	88
10.8 Summary.....	88
11. The Anatomy of an Android App	89
11.1 Android Activities.....	89
11.2 Android Fragments.....	89
11.3 Android Intents.....	90
11.4 Broadcast Intents.....	90
11.5 Broadcast Receivers.....	90
11.6 Android Services.....	90
11.7 Content Providers.....	91
11.8 The Application Manifest.....	91
11.9 Application Resources.....	91
11.10 Application Context.....	91
11.11 Summary.....	91
12. An Overview of Android View Binding	93

Table of Contents

12.1 Find View by Id	93
12.2 View Binding	93
12.3 Converting the AndroidSample project.....	94
12.4 Enabling View Binding.....	94
12.5 Using View Binding.....	94
12.6 Choosing an Option	96
12.7 View Binding in the Book Examples	96
12.8 Migrating a Project to View Binding.....	96
12.9 Summary	97
13. Understanding Android Application and Activity Lifecycles.....	99
13.1 Android Applications and Resource Management.....	99
13.2 Android Process States	99
13.2.1 Foreground Process	100
13.2.2 Visible Process	100
13.2.3 Service Process	100
13.2.4 Background Process.....	100
13.2.5 Empty Process	101
13.3 Inter-Process Dependencies	101
13.4 The Activity Lifecycle.....	101
13.5 The Activity Stack.....	101
13.6 Activity States	102
13.7 Configuration Changes	102
13.8 Handling State Change.....	103
13.9 Summary	103
14. Handling Android Activity State Changes.....	105
14.1 New vs. Old Lifecycle Techniques.....	105
14.2 The Activity and Fragment Classes.....	105
14.3 Dynamic State vs. Persistent State.....	107
14.4 The Android Lifecycle Methods	108
14.5 Lifetimes	109
14.6 Foldable Devices and Multi-Resume	110
14.7 Disabling Configuration Change Restarts	110
14.8 Lifecycle Method Limitations.....	111
14.9 Summary	111
15. Android Activity State Changes by Example.....	113
15.1 Creating the State Change Example Project	113
15.2 Designing the User Interface	114
15.3 Overriding the Activity Lifecycle Methods	115
15.4 Filtering the Logcat Panel.....	117
15.5 Running the Application.....	119
15.6 Experimenting with the Activity.....	119
15.7 Summary	120
16. Saving and Restoring the State of an Android Activity	121
16.1 Saving Dynamic State	121
16.2 Default Saving of User Interface State	121
16.3 The Bundle Class	122

16.4 Saving the State	123
16.5 Restoring the State	124
16.6 Testing the Application.....	124
16.7 Summary	124
17. Understanding Android Views, View Groups and Layouts	127
17.1 Designing for Different Android Devices.....	127
17.2 Views and View Groups	127
17.3 Android Layout Managers	127
17.4 The View Hierarchy	129
17.5 Creating User Interfaces.....	130
17.6 Summary	130
18. A Guide to the Android Studio Layout Editor Tool	131
18.1 Basic vs. Empty Views Activity Templates.....	131
18.2 The Android Studio Layout Editor	135
18.3 Design Mode.....	135
18.4 The Palette	136
18.5 Design Mode and Layout Views.....	137
18.6 Night Mode	138
18.7 Code Mode.....	138
18.8 Split Mode	139
18.9 Setting Attributes.....	139
18.10 Transforms	141
18.11 Tools Visibility Toggles.....	142
18.12 Converting Views.....	143
18.13 Displaying Sample Data	144
18.14 Creating a Custom Device Definition	145
18.15 Changing the Current Device.....	145
18.16 Layout Validation	146
18.17 Summary.....	147
19. A Guide to the Android ConstraintLayout.....	149
19.1 How ConstraintLayout Works.....	149
19.1.1 Constraints.....	149
19.1.2 Margins.....	150
19.1.3 Opposing Constraints.....	150
19.1.4 Constraint Bias	151
19.1.5 Chains.....	152
19.1.6 Chain Styles.....	152
19.2 Baseline Alignment.....	153
19.3 Configuring Widget Dimensions.....	153
19.4 Guideline Helper	154
19.5 Group Helper.....	154
19.6 Barrier Helper.....	154
19.7 Flow Helper.....	156
19.8 Ratios	157
19.9 ConstraintLayout Advantages	157
19.10 ConstraintLayout Availability.....	158

Table of Contents

19.11 Summary	158
20. A Guide to Using ConstraintLayout in Android Studio	159
20.1 Design and Layout Views.....	159
20.2 Autoconnect Mode	161
20.3 Inference Mode.....	161
20.4 Manipulating Constraints Manually.....	161
20.5 Adding Constraints in the Inspector	163
20.6 Viewing Constraints in the Attributes Window.....	163
20.7 Deleting Constraints.....	164
20.8 Adjusting Constraint Bias	165
20.9 Understanding ConstraintLayout Margins.....	165
20.10 The Importance of Opposing Constraints and Bias	167
20.11 Configuring Widget Dimensions.....	169
20.12 Design Time Tools Positioning.....	170
20.13 Adding Guidelines	171
20.14 Adding Barriers	173
20.15 Adding a Group.....	174
20.16 Working with the Flow Helper	175
20.17 Widget Group Alignment and Distribution.....	175
20.18 Converting other Layouts to ConstraintLayout.....	177
20.19 Summary	177
21. Working with ConstraintLayout Chains and Ratios in Android Studio	179
21.1 Creating a Chain.....	179
21.2 Changing the Chain Style	181
21.3 Spread Inside Chain Style.....	182
21.4 Packed Chain Style.....	182
21.5 Packed Chain Style with Bias.....	182
21.6 Weighted Chain.....	182
21.7 Working with Ratios	183
21.8 Summary	185
22. An Android Studio Layout Editor ConstraintLayout Tutorial	187
22.1 An Android Studio Layout Editor Tool Example	187
22.2 Preparing the Layout Editor Environment	187
22.3 Adding the Widgets to the User Interface.....	188
22.4 Adding the Constraints	191
22.5 Testing the Layout.....	193
22.6 Using the Layout Inspector	193
22.7 Summary	194
23. Manual XML Layout Design in Android Studio	195
23.1 Manually Creating an XML Layout	195
23.2 Manual XML vs. Visual Layout Design.....	198
23.3 Summary	198
24. Managing Constraints using Constraint Sets.....	199
24.1 Java Code vs. XML Layout Files	199
24.2 Creating Views.....	199

24.3 View Attributes.....	200
24.4 Constraint Sets.....	200
24.4.1 Establishing Connections.....	200
24.4.2 Applying Constraints to a Layout.....	200
24.4.3 Parent Constraint Connections.....	200
24.4.4 Sizing Constraints.....	201
24.4.5 Constraint Bias.....	201
24.4.6 Alignment Constraints.....	201
24.4.7 Copying and Applying Constraint Sets.....	201
24.4.8 ConstraintLayout Chains.....	201
24.4.9 Guidelines.....	202
24.4.10 Removing Constraints.....	202
24.4.11 Scaling.....	202
24.4.12 Rotation.....	203
24.5 Summary.....	203
25. An Android ConstraintSet Tutorial.....	205
25.1 Creating the Example Project in Android Studio.....	205
25.2 Adding Views to an Activity.....	205
25.3 Setting View Attributes.....	207
25.4 Creating View IDs.....	207
25.5 Configuring the Constraint Set.....	208
25.6 Adding the EditText View.....	209
25.7 Converting Density Independent Pixels (dp) to Pixels (px).....	211
25.8 Summary.....	212
26. A Guide to Using Apply Changes in Android Studio.....	213
26.1 Introducing Apply Changes.....	213
26.2 Understanding Apply Changes Options.....	213
26.3 Using Apply Changes.....	214
26.4 Configuring Apply Changes Fallback Settings.....	215
26.5 An Apply Changes Tutorial.....	215
26.6 Using Apply Code Changes.....	215
26.7 Using Apply Changes and Restart Activity.....	216
26.8 Using Run App.....	216
26.9 Summary.....	216
27. A Guide to Gradle Version Catalogs.....	217
27.1 Library and Plugin Dependencies.....	217
27.2 Project Gradle Build File.....	217
27.3 Module Gradle Build Files.....	217
27.4 Version Catalog File.....	218
27.5 Adding Dependencies.....	219
27.6 Library Updates.....	220
27.7 Summary.....	220
28. An Overview and Example of Android Event Handling.....	221
28.1 Understanding Android Events.....	221
28.2 Using the android:onClick Resource.....	221
28.3 Event Listeners and Callback Methods.....	222

Table of Contents

28.4 An Event Handling Example	222
28.5 Designing the User Interface	223
28.6 The Event Listener and Callback Method	223
28.7 Consuming Events	225
28.8 Summary	226
29. Android Touch and Multi-touch Event Handling	227
29.1 Intercepting Touch Events	227
29.2 The MotionEvent Object	227
29.3 Understanding Touch Actions	228
29.4 Handling Multiple Touches	228
29.5 An Example Multi-Touch Application	228
29.6 Designing the Activity User Interface	229
29.7 Implementing the Touch Event Listener	229
29.8 Running the Example Application	232
29.9 Summary	233
30. Detecting Common Gestures Using the Android Gesture Detector Class	235
30.1 Implementing Common Gesture Detection	235
30.2 Creating an Example Gesture Detection Project	236
30.3 Implementing the Listener Class	236
30.4 Creating the GestureDetector Instance	238
30.5 Implementing the onTouchEvent() Method	239
30.6 Testing the Application	239
30.7 Summary	239
31. Implementing Custom Gesture and Pinch Recognition on Android	241
31.1 The Android Gesture Builder Application	241
31.2 The GestureOverlayView Class	241
31.3 Detecting Gestures	241
31.4 Identifying Specific Gestures	241
31.5 Installing and Running the Gesture Builder Application	241
31.6 Creating a Gestures File	242
31.7 Creating the Example Project	242
31.8 Extracting the Gestures File from the SD Card	242
31.9 Adding the Gestures File to the Project	243
31.10 Designing the User Interface	243
31.11 Loading the Gestures File	244
31.12 Registering the Event Listener	245
31.13 Implementing the onGesturePerformed Method	245
31.14 Testing the Application	246
31.15 Configuring the GestureOverlayView	246
31.16 Intercepting Gestures	247
31.17 Detecting Pinch Gestures	247
31.18 A Pinch Gesture Example Project	247
31.19 Summary	249
32. An Introduction to Android Fragments	251
32.1 What is a Fragment?	251
32.2 Creating a Fragment	251

32.3 Adding a Fragment to an Activity using the Layout XML File	252
32.4 Adding and Managing Fragments in Code	254
32.5 Handling Fragment Events	255
32.6 Implementing Fragment Communication.....	256
32.7 Summary	257
33. Using Fragments in Android Studio - An Example.....	259
33.1 About the Example Fragment Application	259
33.2 Creating the Example Project.....	259
33.3 Creating the First Fragment Layout.....	259
33.4 Migrating a Fragment to View Binding	261
33.5 Adding the Second Fragment.....	262
33.6 Adding the Fragments to the Activity.....	263
33.7 Making the Toolbar Fragment Talk to the Activity	264
33.8 Making the Activity Talk to the Text Fragment	267
33.9 Testing the Application.....	268
33.10 Summary.....	269
34. Modern Android App Architecture with Jetpack.....	271
34.1 What is Android Jetpack?	271
34.2 The “Old” Architecture.....	271
34.3 Modern Android Architecture.....	271
34.4 The ViewModel Component	272
34.5 The LiveData Component.....	272
34.6 ViewModel Saved State.....	273
34.7 LiveData and Data Binding.....	273
34.8 Android Lifecycles	274
34.9 Repository Modules.....	274
34.10 Summary.....	275
35. An Android ViewModel Tutorial.....	277
35.1 About the Project	277
35.2 Creating the ViewModel Example Project.....	277
35.3 Removing Unwanted Project Elements.....	277
35.4 Designing the Fragment Layout.....	278
35.5 Implementing the View Model.....	279
35.6 Associating the Fragment with the View Model.....	280
35.7 Modifying the Fragment	281
35.8 Accessing the ViewModel Data.....	282
35.9 Testing the Project.....	282
35.10 Summary.....	283
36. An Android Jetpack LiveData Tutorial.....	285
36.1 LiveData - A Recap	285
36.2 Adding LiveData to the ViewModel.....	285
36.3 Implementing the Observer.....	287
36.4 Summary	289
37. An Overview of Android Jetpack Data Binding.....	291
37.1 An Overview of Data Binding.....	291

Table of Contents

37.2 The Key Components of Data Binding	291
37.2.1 The Project Build Configuration	291
37.2.2 The Data Binding Layout File	292
37.2.3 The Layout File Data Element	293
37.2.4 The Binding Classes	294
37.2.5 Data Binding Variable Configuration	294
37.2.6 Binding Expressions (One-Way)	295
37.2.7 Binding Expressions (Two-Way)	296
37.2.8 Event and Listener Bindings	296
37.3 Summary	297
38. An Android Jetpack Data Binding Tutorial	299
38.1 Removing the Redundant Code	299
38.2 Enabling Data Binding	300
38.3 Adding the Layout Element	301
38.4 Adding the Data Element to Layout File	302
38.5 Working with the Binding Class	302
38.6 Assigning the ViewModel Instance to the Data Binding Variable	303
38.7 Adding Binding Expressions	304
38.8 Adding the Conversion Method	305
38.9 Adding a Listener Binding	305
38.10 Testing the App	305
38.11 Summary	306
39. An Android ViewModel Saved State Tutorial	307
39.1 Understanding ViewModel State Saving	307
39.2 Implementing ViewModel State Saving	307
39.3 Saving and Restoring State	309
39.4 Adding Saved State Support to the ViewModelDemo Project	309
39.5 Summary	310
40. Working with Android Lifecycle-Aware Components	311
40.1 Lifecycle Awareness	311
40.2 Lifecycle Owners	311
40.3 Lifecycle Observers	312
40.4 Lifecycle States and Events	313
40.5 Summary	314
41. An Android Jetpack Lifecycle Awareness Tutorial	315
41.1 Creating the Example Lifecycle Project	315
41.2 Creating a Lifecycle Observer	315
41.3 Adding the Observer	317
41.4 Testing the Observer	317
41.5 Creating a Lifecycle Owner	317
41.6 Testing the Custom Lifecycle Owner	319
41.7 Summary	319
42. An Overview of the Navigation Architecture Component	321
42.1 Understanding Navigation	321
42.2 Declaring a Navigation Host	322

42.3 The Navigation Graph	324
42.4 Accessing the Navigation Controller	325
42.5 Triggering a Navigation Action	325
42.6 Passing Arguments.....	326
42.7 Summary	326
43. An Android Jetpack Navigation Component Tutorial	327
43.1 Creating the NavigationDemo Project	327
43.2 Adding Navigation to the Build Configuration.....	327
43.3 Creating the Navigation Graph Resource File.....	328
43.4 Declaring a Navigation Host.....	329
43.5 Adding Navigation Destinations.....	330
43.6 Designing the Destination Fragment Layouts.....	332
43.7 Adding an Action to the Navigation Graph.....	334
43.8 Implement the OnFragmentInteractionListener	335
43.9 Adding View Binding Support to the Destination Fragments	336
43.10 Triggering the Action	337
43.11 Passing Data Using Safeargs	338
43.12 Summary.....	341
44. An Introduction to MotionLayout.....	343
44.1 An Overview of MotionLayout	343
44.2 MotionLayout	343
44.3 MotionScene.....	343
44.4 Configuring ConstraintSets	344
44.5 Custom Attributes.....	345
44.6 Triggering an Animation.....	347
44.7 Arc Motion.....	348
44.8 Keyframes.....	348
44.8.1 Attribute Keyframes.....	348
44.8.2 Position Keyframes	349
44.9 Time Linearity	352
44.10 KeyTrigger.....	352
44.11 Cycle and Time Cycle Keyframes	353
44.12 Starting an Animation from Code.....	353
44.13 Summary.....	354
45. An Android MotionLayout Editor Tutorial.....	355
45.1 Creating the MotionLayoutDemo Project	355
45.2 ConstraintLayout to MotionLayout Conversion	355
45.3 Configuring Start and End Constraints	357
45.4 Previewing the MotionLayout Animation	360
45.5 Adding an OnClick Gesture	360
45.6 Adding an Attribute Keyframe to the Transition.....	362
45.7 Adding a CustomAttribute to a Transition.....	364
45.8 Adding Position Keyframes	366
45.9 Summary	368
46. A MotionLayout KeyCycle Tutorial	369
46.1 An Overview of Cycle Keyframes	369

Table of Contents

46.2 Using the Cycle Editor	373
46.3 Creating the KeyCycleDemo Project	374
46.4 Configuring the Start and End Constraints	374
46.5 Creating the Cycles	376
46.6 Previewing the Animation	378
46.7 Adding the KeyFrameSet to the MotionScene	378
46.8 Summary	380
47. Working with the Floating Action Button and Snackbar	381
47.1 The Material Design	381
47.2 The Design Library	381
47.3 The Floating Action Button (FAB)	381
47.4 The Snackbar	382
47.5 Creating the Example Project	383
47.6 Reviewing the Project	383
47.7 Removing Navigation Features	384
47.8 Changing the Floating Action Button	384
47.9 Adding an Action to the Snackbar	386
47.10 Summary	386
48. Creating a Tabbed Interface using the TabLayout Component	387
48.1 An Introduction to the ViewPager2	387
48.2 An Overview of the TabLayout Component	387
48.3 Creating the TabLayoutDemo Project	388
48.4 Creating the First Fragment	389
48.5 Duplicating the Fragments	390
48.6 Adding the TabLayout and ViewPager2	391
48.7 Creating the Pager Adapter	392
48.8 Performing the Initialization Tasks	393
48.9 Testing the Application	395
48.10 Customizing the TabLayout	395
48.11 Summary	397
49. Working with the RecyclerView and CardView Widgets	399
49.1 An Overview of the RecyclerView	399
49.2 An Overview of the CardView	401
49.3 Summary	402
50. An Android RecyclerView and CardView Tutorial	403
50.1 Creating the CardDemo Project	403
50.2 Modifying the Basic Views Activity Project	403
50.3 Designing the CardView Layout	404
50.4 Adding the RecyclerView	405
50.5 Adding the Image Files	405
50.6 Creating the RecyclerView Adapter	406
50.7 Initializing the RecyclerView Component	408
50.8 Testing the Application	409
50.9 Responding to Card Selections	409
50.10 Summary	411

51. A Layout Editor Sample Data Tutorial	413
51.1 Adding Sample Data to a Project	413
51.2 Using Custom Sample Data	417
51.3 Summary	420
52. Working with the AppBar and Collapsing Toolbar Layouts	421
52.1 The Anatomy of an AppBar	421
52.2 The Example Project	422
52.3 Coordinating the RecyclerView and Toolbar	422
52.4 Introducing the Collapsing Toolbar Layout	424
52.5 Changing the Title and Scrim Color	427
52.6 Summary	428
53. An Overview of Android Services.....	429
53.1 Intent Service	429
53.2 Bound Service.....	429
53.3 The Anatomy of a Service	430
53.4 Controlling Destroyed Service Restart Options.....	430
53.5 Declaring a Service in the Manifest File.....	430
53.6 Starting a Service Running on System Startup.....	431
53.7 Summary	432
54. An Overview of Android Intents	433
54.1 An Overview of Intents	433
54.2 Explicit Intents.....	433
54.3 Returning Data from an Activity	434
54.4 Implicit Intents	435
54.5 Using Intent Filters.....	436
54.6 Automatic Link Verification	436
54.7 Manually Enabling Links	439
54.8 Checking Intent Availability	440
54.9 Summary	441
55. Android Explicit Intents – A Worked Example	443
55.1 Creating the Explicit Intent Example Application	443
55.2 Designing the User Interface Layout for MainActivity	443
55.3 Creating the Second Activity Class.....	444
55.4 Designing the User Interface Layout for SecondActivity	445
55.5 Reviewing the Application Manifest File	445
55.6 Creating the Intent	446
55.7 Extracting Intent Data	447
55.8 Launching SecondActivity as a Sub-Activity.....	448
55.9 Returning Data from a Sub-Activity.....	449
55.10 Testing the Application.....	449
55.11 Summary	450
56. Android Implicit Intents – A Worked Example	451
56.1 Creating the Android Studio Implicit Intent Example Project	451
56.2 Designing the User Interface	451
56.3 Creating the Implicit Intent	452

Table of Contents

56.4 Adding a Second Matching Activity	452
56.5 Adding the Web View to the UI	453
56.6 Obtaining the Intent URL	453
56.7 Modifying the MyWebView Project Manifest File	455
56.8 Installing the MyWebView Package on a Device	456
56.9 Testing the Application	457
56.10 Manually Enabling the Link	457
56.11 Automatic Link Verification	459
56.12 Summary	461
57. Android Broadcast Intents and Broadcast Receivers	463
57.1 An Overview of Broadcast Intents	463
57.2 An Overview of Broadcast Receivers	464
57.3 Obtaining Results from a Broadcast	465
57.4 Sticky Broadcast Intents	465
57.5 The Broadcast Intent Example	466
57.6 Creating the Example Application	466
57.7 Creating and Sending the Broadcast Intent	466
57.8 Creating the Broadcast Receiver	467
57.9 Registering the Broadcast Receiver	468
57.10 Testing the Broadcast Example	469
57.11 Listening for System Broadcasts	469
57.12 Summary	470
58. Android Local Bound Services – A Worked Example	471
58.1 Understanding Bound Services	471
58.2 Bound Service Interaction Options	471
58.3 A Local Bound Service Example	471
58.4 Adding a Bound Service to the Project	472
58.5 Implementing the Binder	472
58.6 Binding the Client to the Service	475
58.7 Completing the Example	476
58.8 Testing the Application	477
58.9 Summary	477
59. Android Remote Bound Services – A Worked Example	479
59.1 Client to Remote Service Communication	479
59.2 Creating the Example Application	479
59.3 Designing the User Interface	479
59.4 Implementing the Remote Bound Service	479
59.5 Configuring a Remote Service in the Manifest File	481
59.6 Launching and Binding to the Remote Service	482
59.7 Sending a Message to the Remote Service	483
59.8 Summary	484
60. An Overview of Java Threads, Handlers and Executors	485
60.1 The Application Main Thread	485
60.2 Thread Handlers	485
60.3 A Threading Example	485
60.4 Building the App	486

60.5 Creating a New Thread.....	487
60.6 Implementing a Thread Handler.....	488
60.7 Passing a Message to the Handler.....	490
60.8 Java Executor Concurrency.....	490
60.9 Working with Runnable Tasks.....	491
60.10 Shutting down an Executor Service.....	492
60.11 Working with Callable Tasks and Futures.....	492
60.12 Handling a Future Result.....	494
60.13 Scheduling Tasks.....	495
60.14 Summary.....	496
61. Making Runtime Permission Requests in Android.....	497
61.1 Understanding Normal and Dangerous Permissions.....	497
61.2 Creating the Permissions Example Project.....	499
61.3 Checking for a Permission.....	499
61.4 Requesting Permission at Runtime.....	501
61.5 Providing a Rationale for the Permission Request.....	502
61.6 Testing the Permissions App.....	504
61.7 Summary.....	504
62. An Android Notifications Tutorial.....	505
62.1 An Overview of Notifications.....	505
62.2 Creating the NotifyDemo Project.....	507
62.3 Designing the User Interface.....	507
62.4 Creating the Second Activity.....	507
62.5 Creating a Notification Channel.....	508
62.6 Requesting Notification Permission.....	509
62.7 Creating and Issuing a Notification.....	512
62.8 Launching an Activity from a Notification.....	514
62.9 Adding Actions to a Notification.....	516
62.10 Bundled Notifications.....	517
62.11 Summary.....	519
63. An Android Direct Reply Notification Tutorial.....	521
63.1 Creating the DirectReply Project.....	521
63.2 Designing the User Interface.....	521
63.3 Requesting Notification Permission.....	522
63.4 Creating the Notification Channel.....	523
63.5 Building the RemoteInput Object.....	524
63.6 Creating the PendingIntent.....	525
63.7 Creating the Reply Action.....	526
63.8 Receiving Direct Reply Input.....	528
63.9 Updating the Notification.....	529
63.10 Summary.....	531
64. Foldable Devices and Multi-Window Support.....	533
64.1 Foldables and Multi-Window Support.....	533
64.2 Using a Foldable Emulator.....	534
64.3 Entering Multi-Window Mode.....	535
64.4 Enabling and using Freeform Support.....	536

Table of Contents

64.5 Checking for Freeform Support	536
64.6 Enabling Multi- Window Support in an App	536
64.7 Specifying Multi- Window Attributes	537
64.8 Detecting Multi- Window Mode in an Activity	538
64.9 Receiving Multi- Window Notifications	538
64.10 Launching an Activity in Multi- Window Mode	539
64.11 Configuring Freeform Activity Size and Position.....	539
64.12 Summary	540
65. An Overview of Android SQLite Databases	541
65.1 Understanding Database Tables	541
65.2 Introducing Database Schema	541
65.3 Columns and Data Types	541
65.4 Database Rows	542
65.5 Introducing Primary Keys	542
65.6 What is SQLite?	542
65.7 Structured Query Language (SQL).....	542
65.8 Trying SQLite on an Android Virtual Device (AVD)	543
65.9 Android SQLite Classes.....	544
65.9.1 Cursor	545
65.9.2 SQLiteDatabase	545
65.9.3 SQLiteOpenHelper	545
65.9.4 ContentValues.....	546
65.10 The Android Room Persistence Library.....	546
65.11 Summary	546
66. An Android SQLite Database Tutorial	547
66.1 About the Database Example.....	547
66.2 Creating the SQLDemo Project.....	547
66.3 Designing the User interface	547
66.4 Creating the Data Model.....	548
66.5 Implementing the Data Handler.....	549
66.6 The Add Handler Method.....	551
66.7 The Query Handler Method	551
66.8 The Delete Handler Method	552
66.9 Implementing the Activity Event Methods.....	552
66.10 Testing the Application.....	554
66.11 Summary	554
67. Understanding Android Content Providers.....	555
67.1 What is a Content Provider?.....	555
67.2 The Content Provider	555
67.2.1 onCreate()	555
67.2.2 query()	555
67.2.3 insert()	555
67.2.4 update()	556
67.2.5 delete()	556
67.2.6 getType()	556
67.3 The Content URI	556

67.4 The Content Resolver	556
67.5 The <provider> Manifest Element	557
67.6 Summary	557
68. An Android Content Provider Tutorial	559
68.1 Copying the SQLDemo Project.....	559
68.2 Adding the Content Provider Package.....	559
68.3 Creating the Content Provider Class.....	560
68.4 Constructing the Authority and Content URI.....	561
68.5 Implementing URI Matching in the Content Provider.....	562
68.6 Implementing the Content Provider onCreate() Method	563
68.7 Implementing the Content Provider insert() Method	563
68.8 Implementing the Content Provider query() Method	564
68.9 Implementing the Content Provider update() Method	565
68.10 Implementing the Content Provider delete() Method.....	567
68.11 Declaring the Content Provider in the Manifest File.....	568
68.12 Modifying the Database Handler.....	569
68.13 Summary.....	571
69. An Android Content Provider Client Tutorial.....	573
69.1 Creating the SQLDemoClient Project.....	573
69.2 Designing the User interface	573
69.3 Accessing the Content Provider	573
69.4 Adding the Query Permission.....	574
69.5 Testing the Project.....	575
69.6 Summary	575
70. The Android Room Persistence Library	577
70.1 Revisiting Modern App Architecture	577
70.2 Key Elements of Room Database Persistence.....	577
70.2.1 Repository	578
70.2.2 Room Database	578
70.2.3 Data Access Object (DAO)	578
70.2.4 Entities.....	578
70.2.5 SQLite Database	578
70.3 Understanding Entities.....	579
70.4 Data Access Objects.....	582
70.5 The Room Database	583
70.6 The Repository.....	584
70.7 In-Memory Databases.....	585
70.8 Database Inspector.....	585
70.9 Summary	585
71. An Android TableLayout and TableRow Tutorial	587
71.1 The TableLayout and TableRow Layout Views.....	587
71.2 Creating the Room Database Project	588
71.3 Converting to a LinearLayout.....	588
71.4 Adding the TableLayout to the User Interface.....	589
71.5 Configuring the TableRows	590
71.6 Adding the Button Bar to the Layout	591

Table of Contents

71.7 Adding the RecyclerView.....	592
71.8 Adjusting the Layout Margins	593
71.9 Summary	593
72. An Android Room Database and Repository Tutorial.....	595
72.1 About the RoomDemo Project.....	595
72.2 Modifying the Build Configuration.....	595
72.3 Building the Entity	596
72.4 Creating the Data Access Object.....	598
72.5 Adding the Room Database.....	599
72.6 Adding the Repository	599
72.7 Adding the ViewModel	602
72.8 Creating the Product Item Layout	603
72.9 Adding the RecyclerView Adapter.....	604
72.10 Preparing the Main Activity	605
72.11 Adding the Button Listeners.....	606
72.12 Adding LiveData Observers	607
72.13 Initializing the RecyclerView.....	608
72.14 Testing the RoomDemo App.....	608
72.15 Using the Database Inspector.....	608
72.16 Summary.....	609
73. Creating, Testing, and Uploading an Android App Bundle.....	611
73.1 The Release Preparation Process.....	611
73.2 Android App Bundles.....	611
73.3 Register for a Google Play Developer Console Account.....	612
73.4 Configuring the App in the Console	613
73.5 Enabling Google Play App Signing.....	614
73.6 Creating a Keystore File	614
73.7 Creating the Android App Bundle.....	615
73.8 Generating Test APK Files	617
73.9 Uploading the App Bundle to the Google Play Developer Console.....	618
73.10 Exploring the App Bundle	619
73.11 Managing Testers	620
73.12 Rolling the App Out for Testing.....	620
73.13 Uploading New App Bundle Revisions.....	621
73.14 Analyzing the App Bundle File	622
73.15 Summary.....	623
74. An Overview of Android In-App Billing	625
74.1 Preparing a Project for In-App Purchasing	625
74.2 Creating In-App Products and Subscriptions	625
74.3 Billing Client Initialization.....	626
74.4 Connecting to the Google Play Billing Library.....	627
74.5 Querying Available Products.....	628
74.6 Starting the Purchase Process.....	628
74.7 Completing the Purchase	629
74.8 Querying Previous Purchases.....	630
74.9 Summary	631

75. An Android In-App Purchasing Tutorial	633
75.1 About the In-App Purchasing Example Project.....	633
75.2 Creating the InAppPurchase Project.....	633
75.3 Adding Libraries to the Project.....	633
75.4 Designing the User Interface.....	634
75.5 Adding the App to the Google Play Store.....	635
75.6 Creating an In-App Product.....	635
75.7 Enabling License Testers.....	636
75.8 Initializing the Billing Client.....	637
75.9 Querying the Product.....	638
75.10 Launching the Purchase Flow.....	640
75.11 Handling Purchase Updates.....	640
75.12 Consuming the Product.....	641
75.13 Restoring a Previous Purchase.....	642
75.14 Testing the App.....	643
75.15 Troubleshooting.....	644
75.16 Summary.....	644
76. Creating and Managing Overflow Menus on Android.....	645
76.1 The Overflow Menu.....	645
76.2 Creating an Overflow Menu.....	645
76.3 Displaying an Overflow Menu.....	646
76.4 Responding to Menu Item Selections.....	646
76.5 Creating Checkable Item Groups.....	647
76.6 Menus and the Android Studio Menu Editor.....	648
76.7 Creating the Example Project.....	649
76.8 Designing the Menu.....	649
76.9 Modifying the <code>onOptionsItemSelected()</code> Method.....	651
76.10 Testing the Application.....	652
76.11 Summary.....	653
77. An Android Studio Primary/Detail Flow Tutorial.....	655
77.1 The Primary/Detail Flow.....	655
77.2 Creating a Primary/Detail Flow Activity.....	656
77.3 Adding the Primary/Detail Flow Activity.....	656
77.4 Modifying the Primary/Detail Flow Template.....	657
77.5 Changing the Content Model.....	657
77.6 Changing the Detail Pane.....	659
77.7 Modifying the <code>ItemDetailFragment</code> Class.....	660
77.8 Modifying the <code>ItemListFragment</code> Class.....	661
77.9 Adding Manifest Permissions.....	662
77.10 Running the Application.....	662
77.11 Summary.....	662
78. An Android Biometric Authentication Tutorial.....	663
78.1 An Overview of Biometric Authentication.....	663
78.2 Creating the Biometric Authentication Project.....	663
78.3 Configuring Device Fingerprint Authentication.....	664
78.4 Adding the Biometric Permission to the Manifest File.....	664

Table of Contents

78.5 Designing the User Interface	665
78.6 Adding a Toast Convenience Method	665
78.7 Checking the Security Settings.....	666
78.8 Configuring the Authentication Callbacks	667
78.9 Adding the CancellationSignal.....	668
78.10 Starting the Biometric Prompt	668
78.11 Testing the Project.....	669
78.12 Summary	670
79. Accessing Cloud Storage using the Android Storage Access Framework.....	671
79.1 The Storage Access Framework.....	671
79.2 Working with the Storage Access Framework.....	672
79.3 Filtering Picker File Listings	672
79.4 Handling Intent Results.....	673
79.5 Reading the Content of a File	673
79.6 Writing Content to a File	674
79.7 Deleting a File.....	675
79.8 Gaining Persistent Access to a File.....	675
79.9 Summary	675
80. An Android Storage Access Framework Example	677
80.1 About the Storage Access Framework Example.....	677
80.2 Creating the Storage Access Framework Example.....	677
80.3 Designing the User Interface	677
80.4 Adding the Activity Launchers.....	678
80.5 Creating a New Storage File.....	679
80.6 Saving to a Storage File.....	680
80.7 Opening and Reading a Storage File	681
80.8 Testing the Storage Access Application	682
80.9 Summary	684
81. Video Playback on Android using the VideoView and MediaController Classes.....	685
81.1 Introducing the Android VideoView Class	685
81.2 Introducing the Android MediaController Class	686
81.3 Creating the Video Playback Example	686
81.4 Designing the VideoPlayer Layout	686
81.5 Downloading the Video File.....	687
81.6 Configuring the VideoView.....	687
81.7 Adding the MediaController to the Video View.....	689
81.8 Setting up the onPreparedListener	689
81.9 Summary	690
82. Android Picture-in-Picture Mode.....	691
82.1 Picture-in-Picture Features	691
82.2 Enabling Picture-in-Picture Mode.....	692
82.3 Configuring Picture-in-Picture Parameters	692
82.4 Entering Picture-in-Picture Mode	693
82.5 Detecting Picture-in-Picture Mode Changes	693
82.6 Adding Picture-in-Picture Actions.....	694
82.7 Summary	694

83. An Android Picture-in-Picture Tutorial.....	697
83.1 Adding Picture-in-Picture Support to the Manifest.....	697
83.2 Adding a Picture-in-Picture Button	697
83.3 Entering Picture-in-Picture Mode	698
83.4 Detecting Picture-in-Picture Mode Changes	699
83.5 Adding a Broadcast Receiver	699
83.6 Adding the PiP Action.....	700
83.7 Testing the Picture-in-Picture Action	703
83.8 Summary	704
84. Android Audio Recording and Playback using MediaPlayer and MediaRecorder	705
84.1 Playing Audio	705
84.2 Recording Audio and Video using the MediaRecorder Class.....	706
84.3 About the Example Project	707
84.4 Creating the AudioApp Project.....	707
84.5 Designing the User Interface	707
84.6 Checking for Microphone Availability	708
84.7 Initializing the Activity.....	709
84.8 Implementing the recordAudio() Method.....	710
84.9 Implementing the stopAudio() Method.....	710
84.10 Implementing the playAudio() method.....	711
84.11 Configuring and Requesting Permissions	711
84.12 Testing the Application.....	713
84.13 Summary	714
85. Working with the Google Maps Android API in Android Studio	715
85.1 The Elements of the Google Maps Android API	715
85.2 Creating the Google Maps Project.....	716
85.3 Creating a Google Cloud Billing Account	716
85.4 Creating a New Google Cloud Project.....	717
85.5 Enabling the Google Maps SDK.....	718
85.6 Generating a Google Maps API Key.....	719
85.7 Adding the API Key to the Android Studio Project	720
85.8 Testing the Application.....	720
85.9 Understanding Geocoding and Reverse Geocoding	720
85.10 Adding a Map to an Application	722
85.11 Requesting Current Location Permission.....	722
85.12 Displaying the User's Current Location	724
85.13 Changing the Map Type.....	725
85.14 Displaying Map Controls to the User	726
85.15 Handling Map Gesture Interaction.....	727
85.15.1 Map Zooming Gestures.....	727
85.15.2 Map Scrolling/Panning Gestures	727
85.15.3 Map Tilt Gestures.....	727
85.15.4 Map Rotation Gestures.....	727
85.16 Creating Map Markers.....	728
85.17 Controlling the Map Camera	729
85.18 Summary	730

86. Printing with the Android Printing Framework	731
86.1 The Android Printing Architecture	731
86.2 The Print Service Plugins	731
86.3 Google Cloud Print	732
86.4 Printing to Google Drive.....	732
86.5 Save as PDF	733
86.6 Printing from Android Devices	733
86.7 Options for Building Print Support into Android Apps.....	734
86.7.1 Image Printing.....	734
86.7.2 Creating and Printing HTML Content	735
86.7.3 Printing a Web Page.....	736
86.7.4 Printing a Custom Document	737
86.8 Summary	737
87. An Android HTML and Web Content Printing Example	739
87.1 Creating the HTML Printing Example Application	739
87.2 Printing Dynamic HTML Content	739
87.3 Creating the Web Page Printing Example.....	742
87.4 Removing the Floating Action Button	742
87.5 Removing Navigation Features.....	742
87.6 Designing the User Interface Layout	744
87.7 Accessing the WebView from the Main Activity	744
87.8 Loading the Web Page into the WebView.....	745
87.9 Adding the Print Menu Option.....	746
87.10 Summary.....	748
88. A Guide to Android Custom Document Printing.....	749
88.1 An Overview of Android Custom Document Printing	749
88.1.1 Custom Print Adapters.....	749
88.2 Preparing the Custom Document Printing Project.....	750
88.3 Designing the UI	750
88.4 Creating the Custom Print Adapter.....	751
88.5 Implementing the onLayout() Callback Method	752
88.6 Implementing the onWrite() Callback Method	755
88.7 Checking a Page is in Range	757
88.8 Drawing the Content on the Page Canvas	758
88.9 Starting the Print Job	760
88.10 Testing the Application.....	761
88.11 Summary.....	761
89. An Introduction to Android App Links.....	763
89.1 An Overview of Android App Links	763
89.2 App Link Intent Filters	763
89.3 Handling App Link Intents	764
89.4 Associating the App with a Website.....	764
89.5 Summary	765
90. An Android Studio App Links Tutorial	767
90.1 About the Example App	767

90.2 The Database Schema	767
90.3 Loading and Running the Project	767
90.4 Adding the URL Mapping	769
90.5 Adding the Intent Filter	772
90.6 Adding Intent Handling Code	772
90.7 Testing the App	775
90.8 Creating the Digital Asset Links File	775
90.9 Testing the App Link	776
90.10 Summary	776
91. Working with Material Design 3 Theming	777
91.1 Material Design 2 vs. Material Design 3	777
91.2 Understanding Material Design Theming	777
91.3 Material Design 3 Theming	777
91.4 Building a custom theme	779
91.5 Summary	781
92. A Material Design 3 Theming and Dynamic Color Tutorial	783
92.1 Creating the ThemeDemo Project	783
92.2 Designing the User Interface	783
92.3 Building a new theme	785
92.4 Adding the Theme to the Project	786
92.5 Enabling Dynamic Color Support	787
92.6 Summary	789
93. An Overview of Gradle in Android Studio	791
93.1 An Overview of Gradle	791
93.2 Gradle and Android Studio	791
93.2.1 Sensible Defaults	791
93.2.2 Dependencies	791
93.2.3 Build Variants	792
93.2.4 Manifest Entries	792
93.2.5 APK Signing	792
93.2.6 ProGuard Support	792
93.3 The Property and Settings Gradle Build File	792
93.4 The Top-level Gradle Build File	793
93.5 Module Level Gradle Build Files	794
93.6 Configuring Signing Settings in the Build File	796
93.7 Running Gradle Tasks from the Command Line	797
93.8 Summary	798
Index	799

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 Koala Feature Drop 2024.1.2 using the Android API 35 SDK (Vanilla Ice Cream), 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 Koala Feature Drop” 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 Koala Feature Drop 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. When prompted to select the components to install, ensure that the *Android Studio* and *Android Virtual Device* options are both 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 complete the installation process.

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:

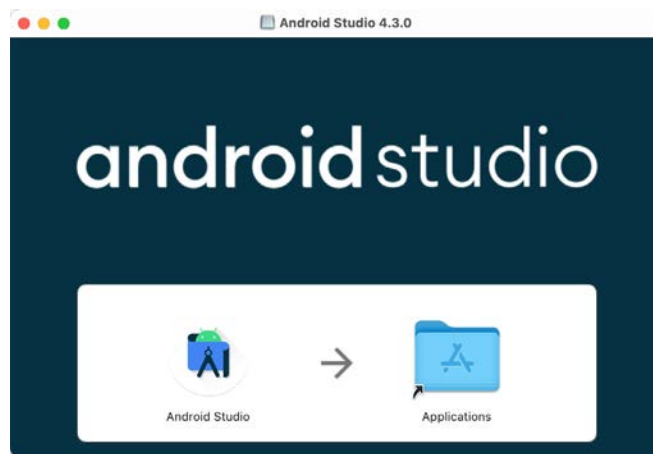


Figure 2-1

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 Installing additional Android SDK packages

When you launch Android Studio, the Welcome to Android Studio screen will appear as shown below:

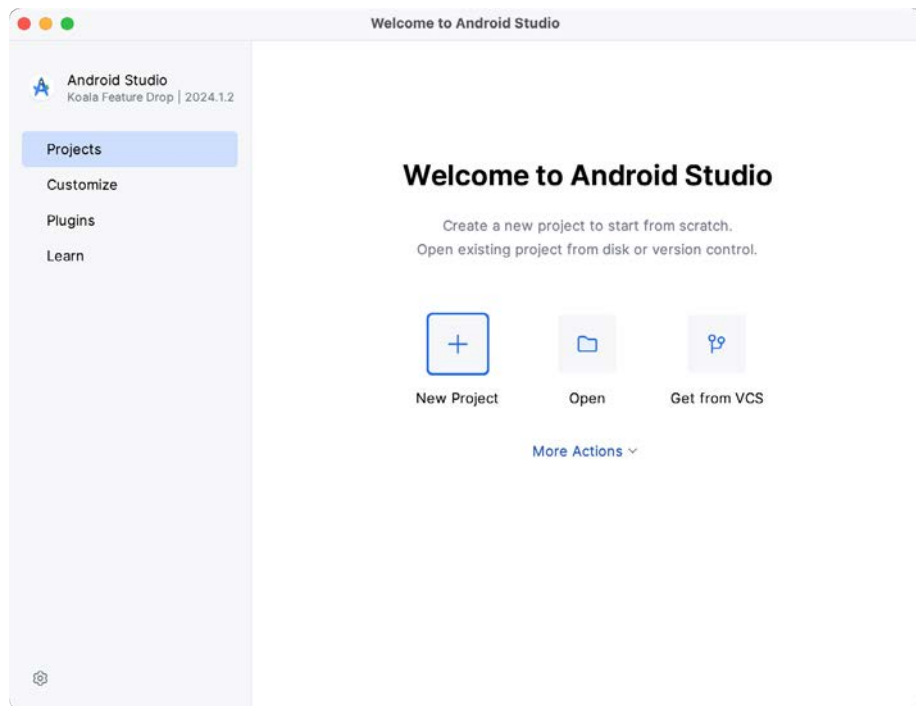


Figure 2-2

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 2-3:

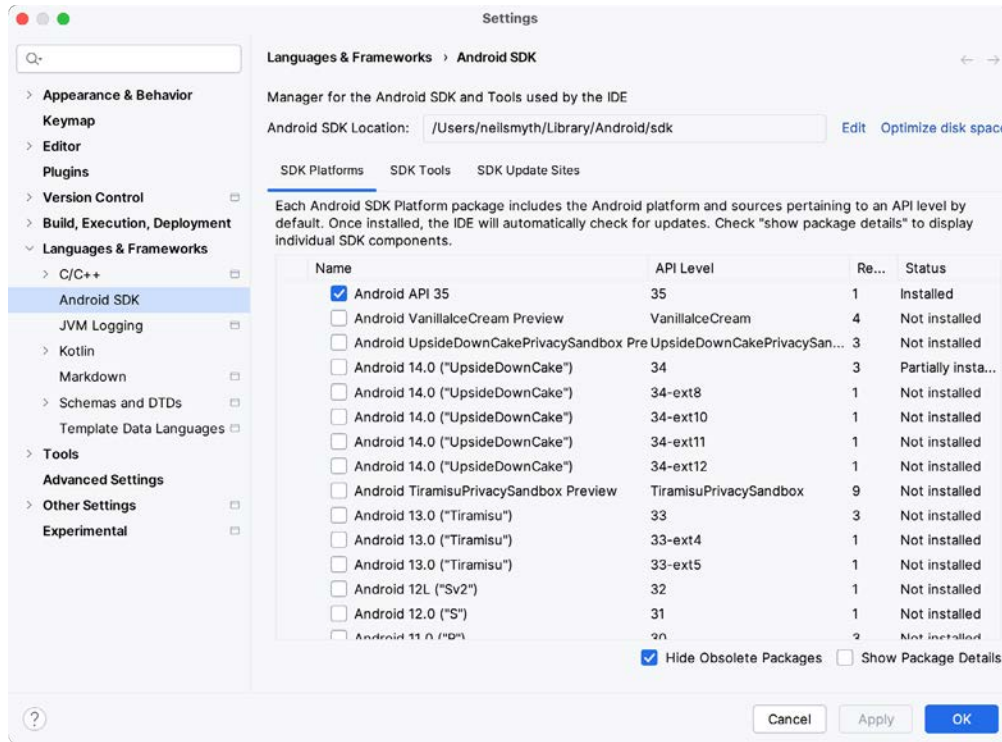


Figure 2-3

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 Koala Feature Drop, this is Android Vanilla Ice Cream (API Level 35). 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 Vanilla Ice Cream (API Level 35) 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 2-4:

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 2-4

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 2-5:



Figure 2-5

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 31-35

*Note that the Intel x86 Emulator Accelerator (HAXM installer) requires an Intel processor with VT-x support enabled. It cannot be installed on Apple silicon-based Macs or AMD-based PCs.

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:

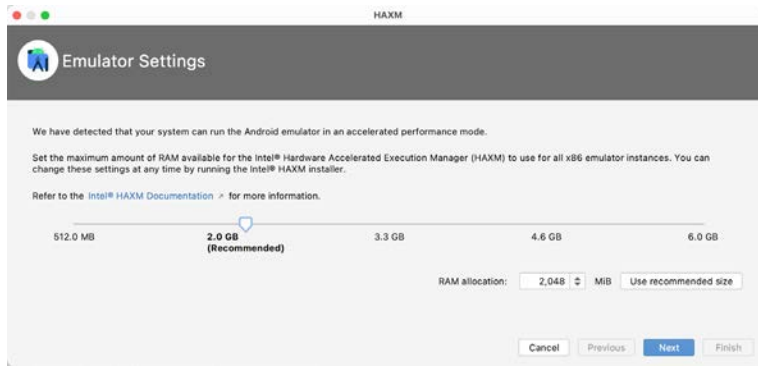


Figure 2-6

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.

2.5 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 2-7:

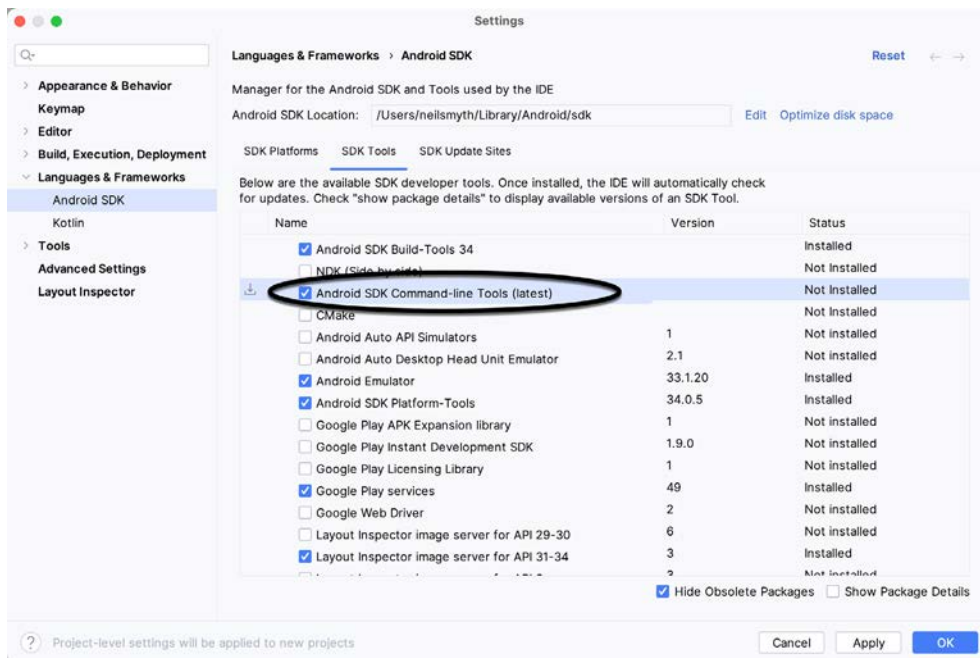


Figure 2-7

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.

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 2-8:



Figure 2-8

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

2.5.1 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:

Setting up an Android Studio Development Environment

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

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.5.5 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
```

2.6 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:

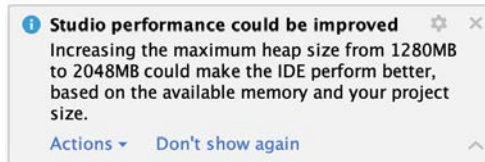


Figure 2-9

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 2-10 below:

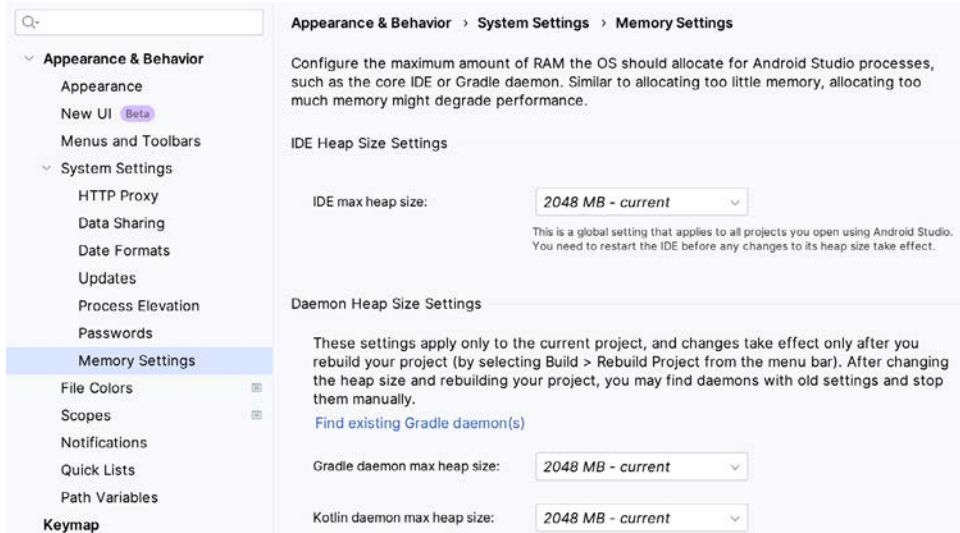


Figure 2-10

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.

2.7 Updating Android Studio and the SDK

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

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

2.8 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:

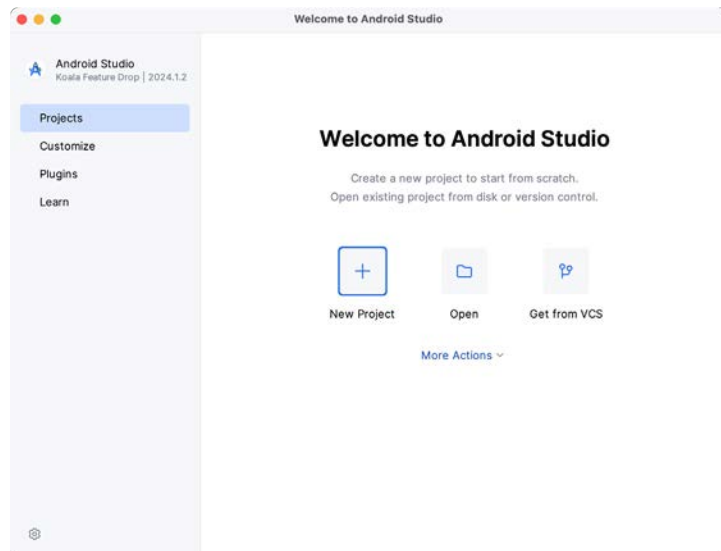


Figure 3-1

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.

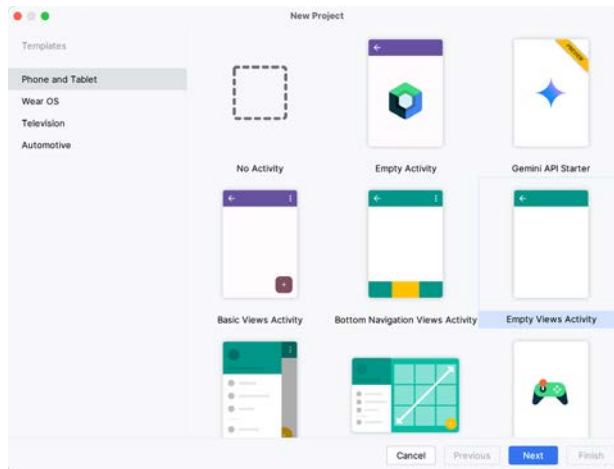


Figure 3-2

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:

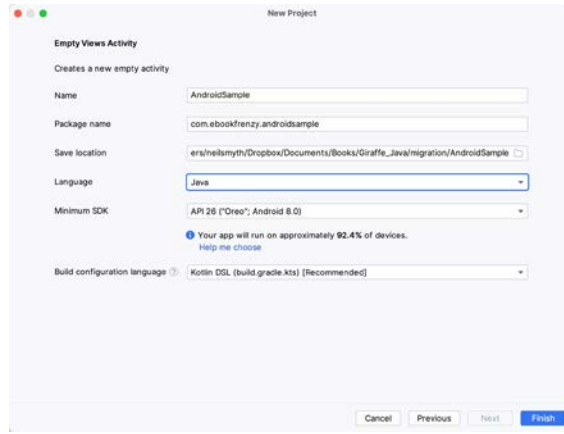


Figure 3-3

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

3.5 Modifying the Example Application

Once the project has been created, the main window will appear containing our *AndroidSample* project, as illustrated in Figure 3-4 below:

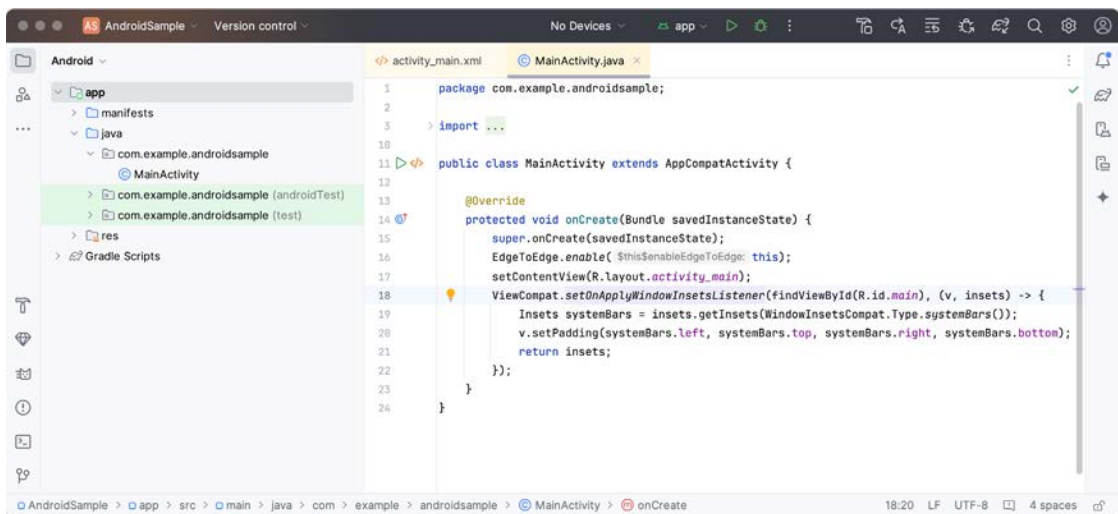


Figure 3-4

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 3-5. If the panel is not currently in Android mode, use the menu to switch mode:

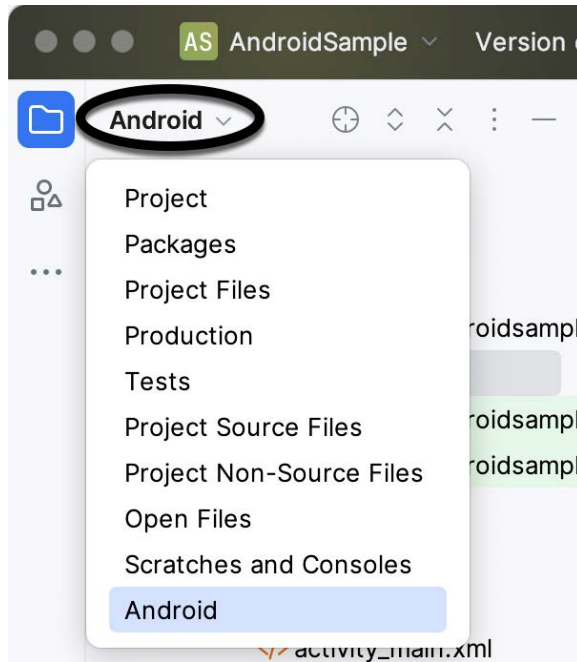


Figure 3-5

3.6 Modifying the User Interface

The user interface design for our activity is stored in a file named *activity_main.xml* which, in turn, is located under *app -> res -> layout* in the Project 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:

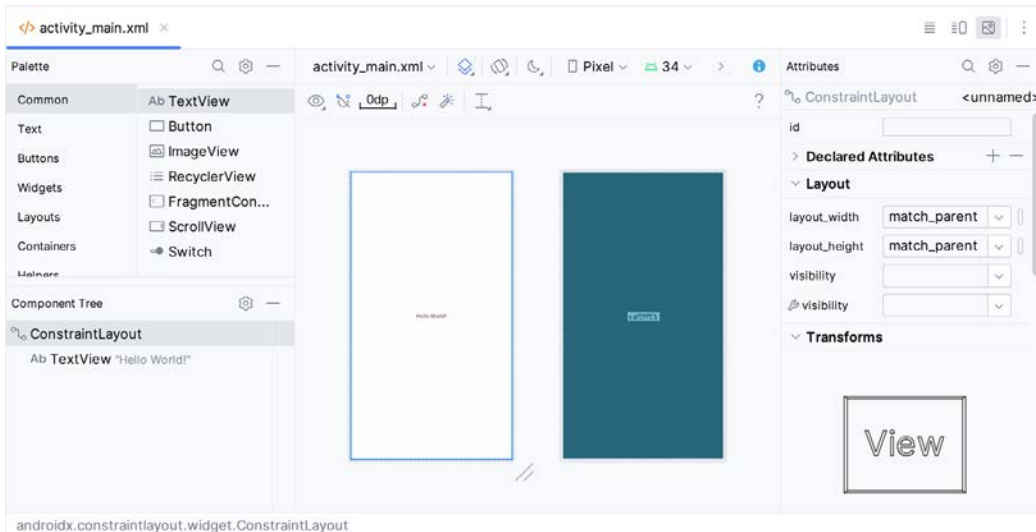




Figure 3-6

In the toolbar across the top of the Layout Editor window is a menu (currently set to *Pixel* in the above figure) which is reflected in the visual representation of the device within the Layout Editor panel. A range of other

device options are available by clicking on this menu.

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

As we can see in the device screen, the content layout already includes a label that displays a “Hello World!” message. Running down the left-hand side of the panel is a palette containing different categories of user interface components that may be used to construct a user interface, such as buttons, labels, and text fields. However, it should be noted that not all user interface components are visible to the user. One such category consists of *layouts*. Android supports a variety of layouts that provide different levels of control over how visual user interface components are positioned and managed on the screen. Though it is difficult to tell from looking at the visual representation of the user interface, the current design has been created using a `ConstraintLayout`. This can be confirmed by reviewing the information in the *Component Tree* panel, which, by default, is located in the lower left-hand corner of the Layout Editor panel and is shown in Figure 3-7:

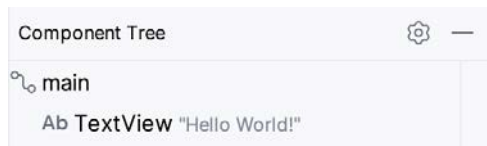


Figure 3-7

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 3-8). If necessary, re-enable Autoconnect mode by clicking on this button.

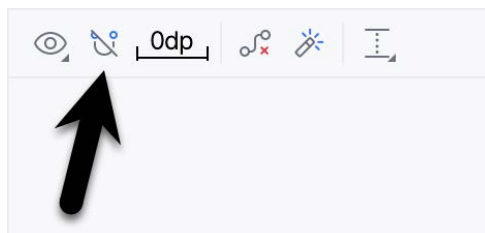


Figure 3-8

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

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

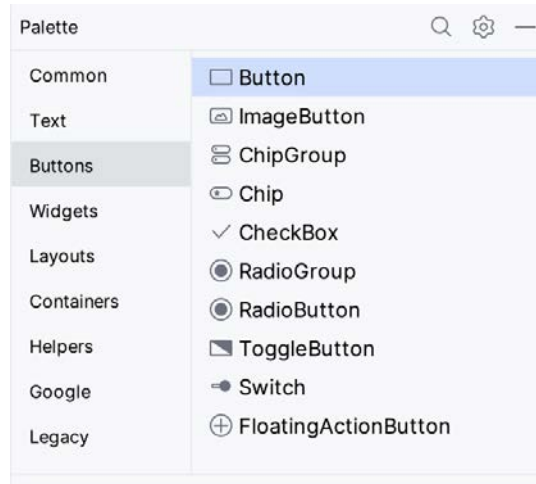


Figure 3-9

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

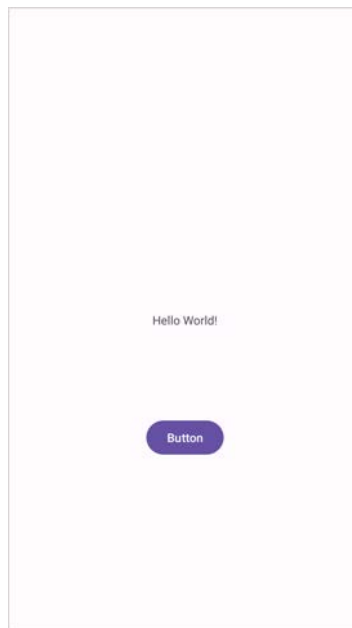


Figure 3-10

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



Figure 3-11

The second text property with a wrench next to it allows a text property to be set, which only appears within the Layout Editor tool but is not shown at runtime. This is useful for testing 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 3-12) to add any missing constraints to the layout:

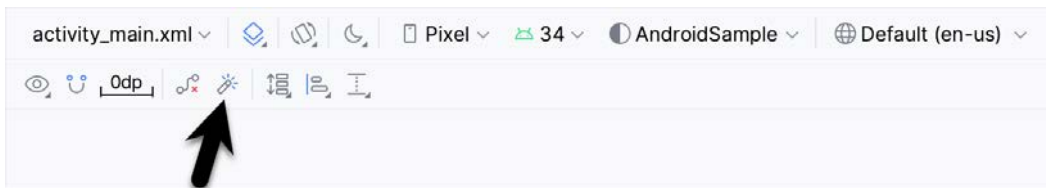


Figure 3-12

It is important to explain the warning button in the top right-hand corner of the Layout Editor tool, as indicated in Figure 3-13. This warning indicates potential problems with the layout. For details on any problems, click on the button:

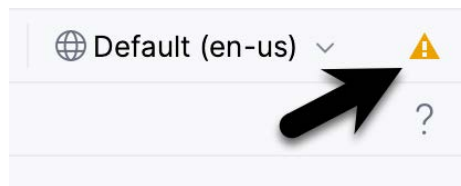


Figure 3-13

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

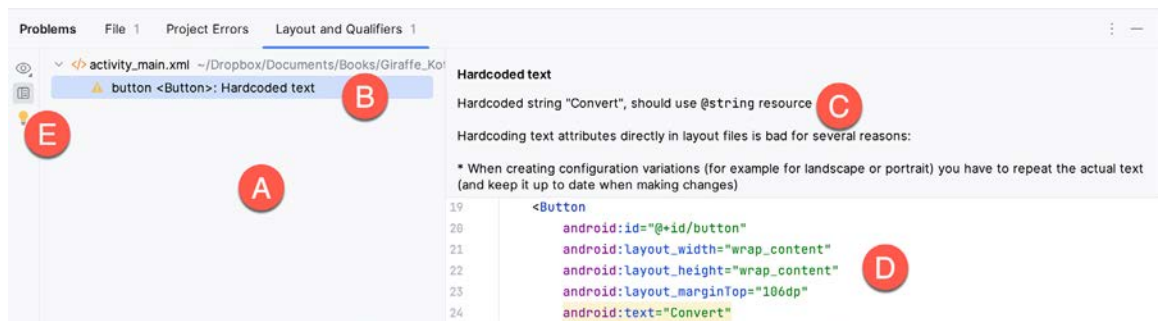


Figure 3-14

This tool window is divided into two panels. The left panel (marked A in the above figure) lists issues detected

Creating an Example Android App in Android Studio

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 3-15:

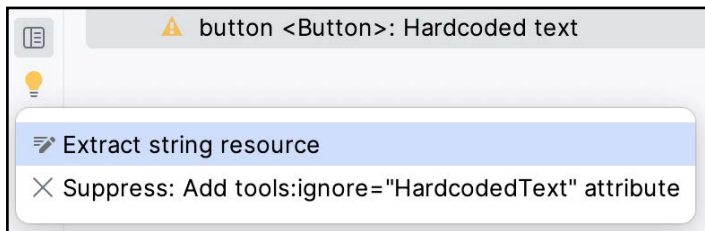


Figure 3-15

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

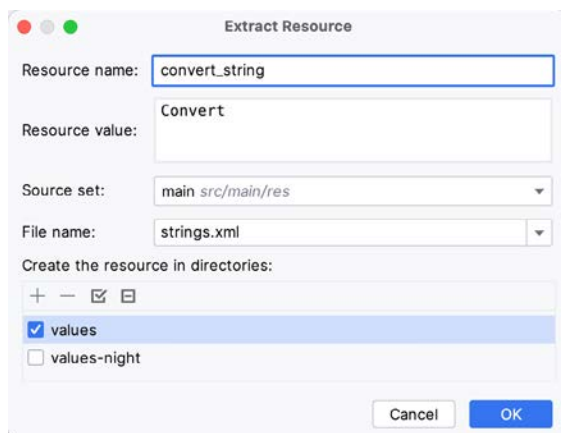


Figure 3-16

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

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

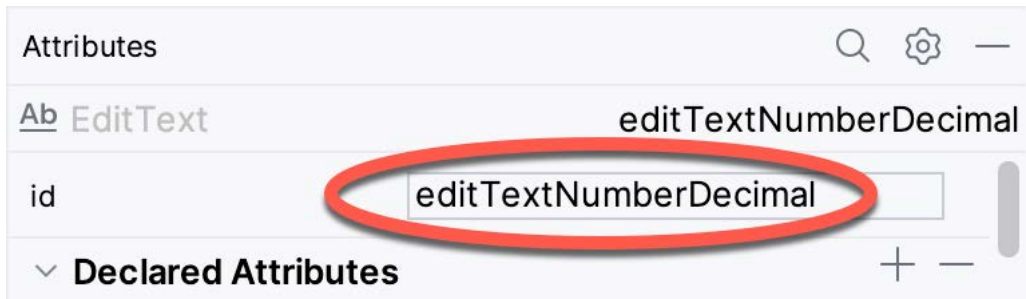


Figure 3-17

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

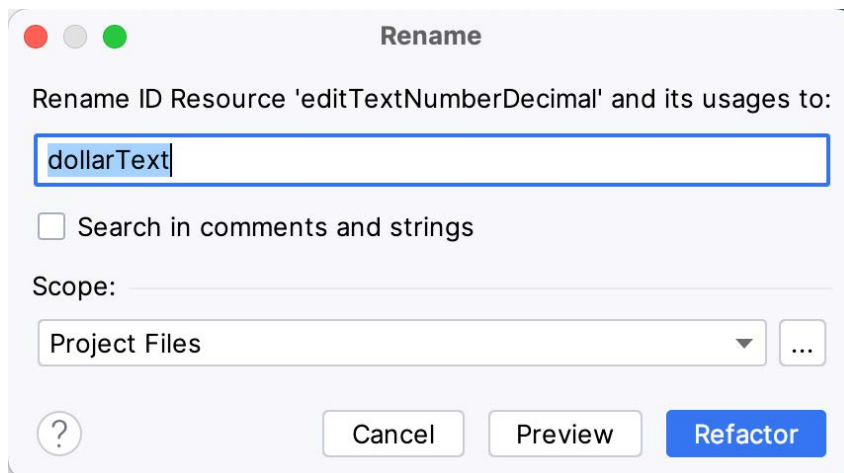


Figure 3-18

Repeat the steps to set the id of the `TextView` widget to `textView`, 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 3-19:

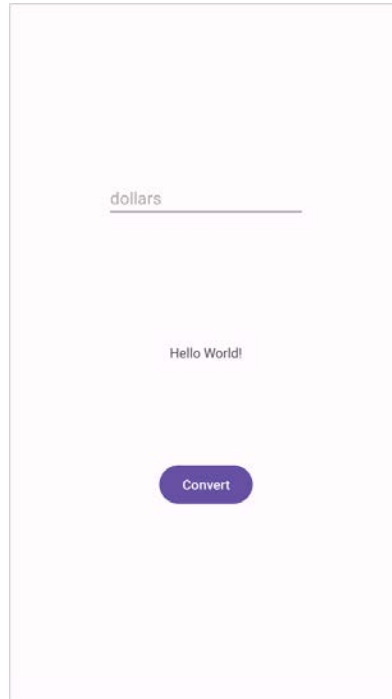


Figure 3-19

3.7 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 3-20 below:

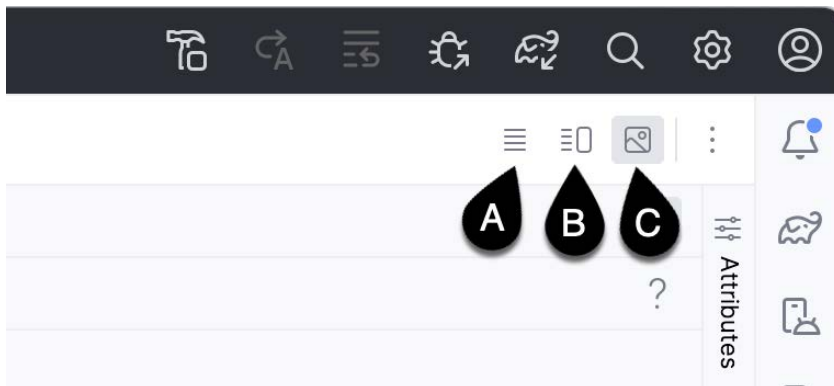


Figure 3-20

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 3-21:

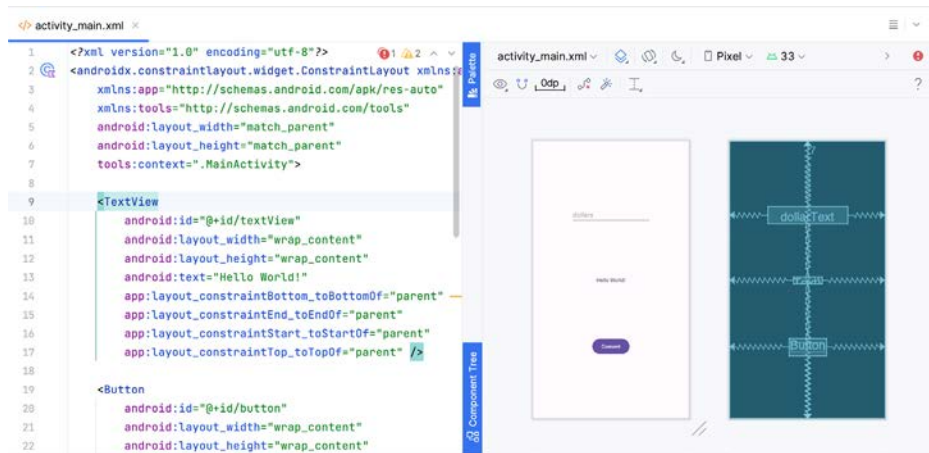


Figure 3-21

The button to the left of the View Modes button (marked B in Figure 3-20 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:

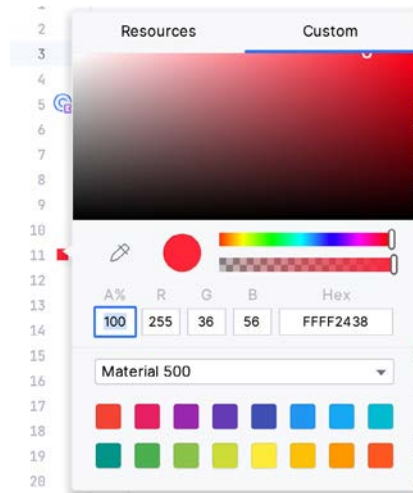


Figure 3-22

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

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

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

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:

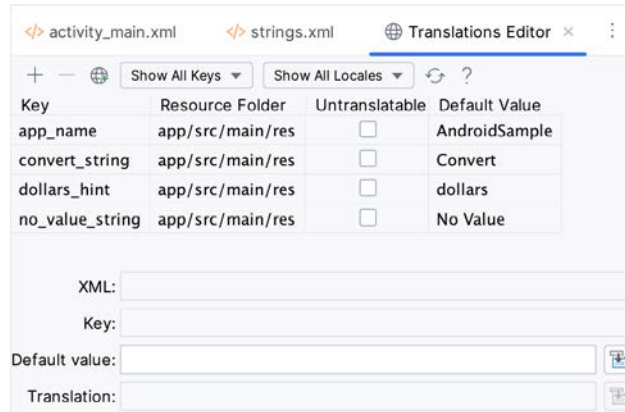


Figure 3-23

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

3.8 Adding Interaction

The final step in this example project is to make the app interactive so that when the user enters a dollar value into the EditText field and clicks the convert button, the converted euro value appears on the TextView. This involves the implementation of some event handling on the Button widget. Specifically, the Button needs to be configured so that a method in the app code is called when an *onClick* event is triggered. Event handling can be implemented in 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:

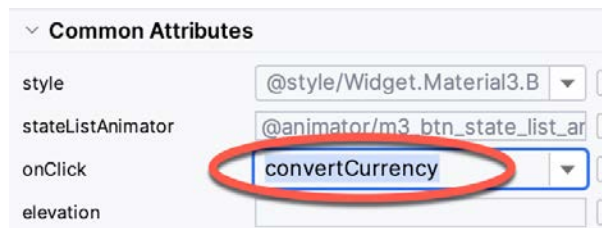


Figure 3-24

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.example.androidsample;

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);
    .
    .
}

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*”.

3.9 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.

10. An Overview of the Android Architecture

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

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

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

10.1 The Android Software Stack

Android is structured as a software stack comprising applications, an operating system, a runtime environment, middleware, services, and libraries. This architecture can best be represented visually, as Figure 10-1 outlines. Each layer of the stack, and the corresponding elements within each layer, are tightly integrated and carefully tuned to provide the optimal application development and execution environment for mobile devices. The remainder of this chapter will work through the different layers of the Android stack, starting at the bottom with the Linux Kernel.

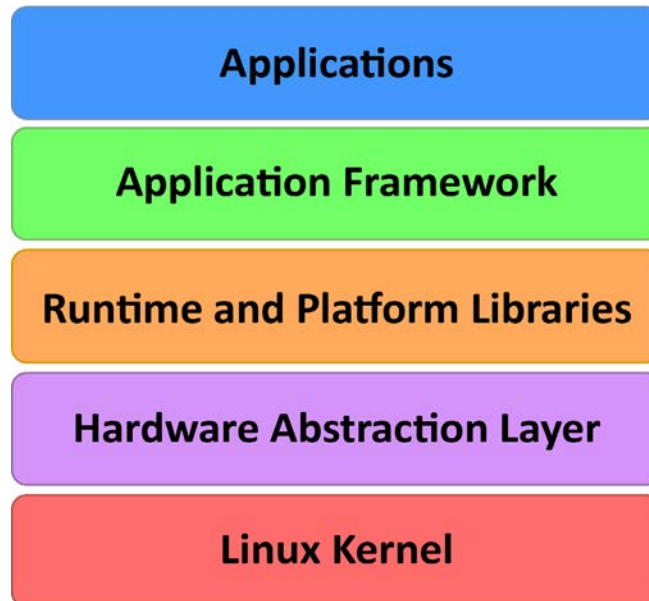


Figure 10-1

10.2 The Linux Kernel

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

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

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

10.3 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) comprises a set of library modules that interface with device components such as the camera, microphone, and accelerometer. When the Android stack needs to access a hardware component, it uses the HAL library modules. Each Android device manufacturer has an abstraction layer for its specific hardware configuration, allowing the standard Android libraries and frameworks to run on any device without being altered for specific hardware.

10.4 Android Runtime – ART

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

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

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

10.5 Android Libraries

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

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

- **android.app** – Provides access to the application model and is the cornerstone of all Android applications.
- **android.content** – Facilitates content access, publishing, and messaging between applications and application components.
- **android.database** – Used to access data published by content providers and includes SQLite database management classes.

- **android.graphics** – A low-level 2D graphics drawing API including colors, points, filters, rectangles, and canvases.
- **android.hardware** – Presents an API providing access to hardware such as the accelerometer and light sensor.
- **android.opengl** – A Java interface to the OpenGL ES 3D graphics rendering API.
- **android.os** – Provides applications with access to standard operating system services, including messages, system services, and inter-process communication.
- **android.media** – Provides classes to enable playback of audio and video.
- **android.net** – A set of APIs providing access to the network stack. Includes *android.net.wifi*, which provides access to the device's wireless stack.
- **android.print** – Includes a set of classes that enable content to be sent to configured printers from within Android applications.
- **android.provider** – A set of convenience classes that provide access to standard Android content provider databases such as those maintained by the calendar and contact applications.
- **android.text** – Used to render and manipulate text on a device display.
- **android.util** – A set of utility classes for performing tasks such as string and number conversion, XML handling and date and time manipulation.
- **android.view** – The fundamental building blocks of application user interfaces.
- **android.widget** – A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
- **android.webkit** – A set of classes intended to allow web-browsing capabilities to be built into applications.

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

10.5.1 C/C++ Libraries

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

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

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

10.6 Application Framework

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

The Android framework includes the following key services:

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

10.7 Applications

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

10.8 Summary

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

18. 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.

18.1 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 18-1 below:

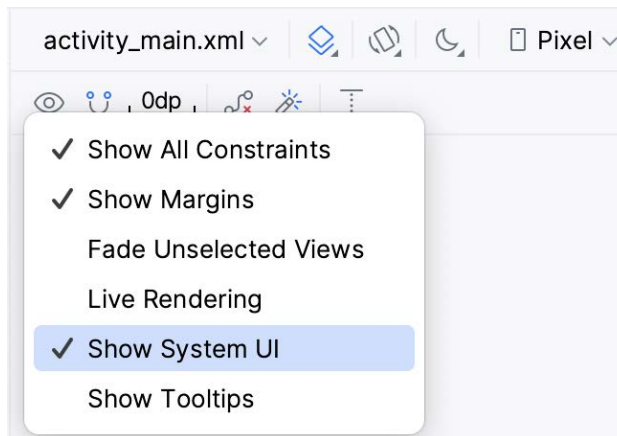


Figure 18-1

The Empty Views Activity template creates a single layout file consisting of a ConstraintLayout manager instance containing a TextView object, as shown in Figure 18-2:

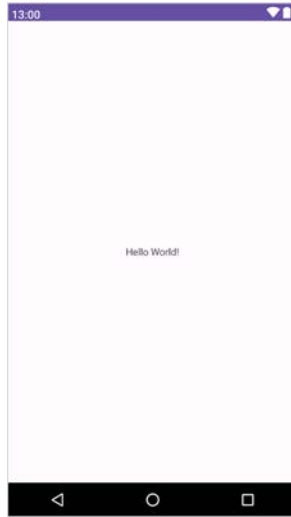


Figure 18-2

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 18-3), 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):

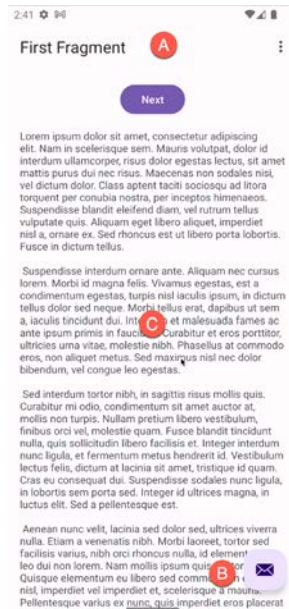


Figure 18-3

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 18-4). Pressing the Next button would navigate to the second screen (B), which, in turn, contains a button to return to the first screen:

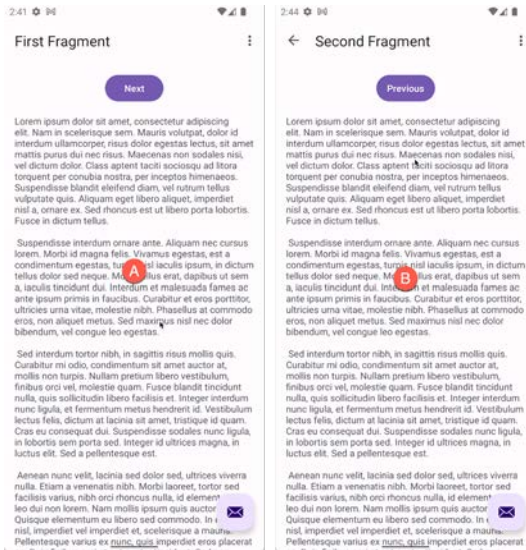


Figure 18-4

This app behavior uses two of 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);
    }
});
}

```

18.2 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.

18.3 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 18-5 highlights the key areas of the Android Studio Layout Editor tool in design mode:

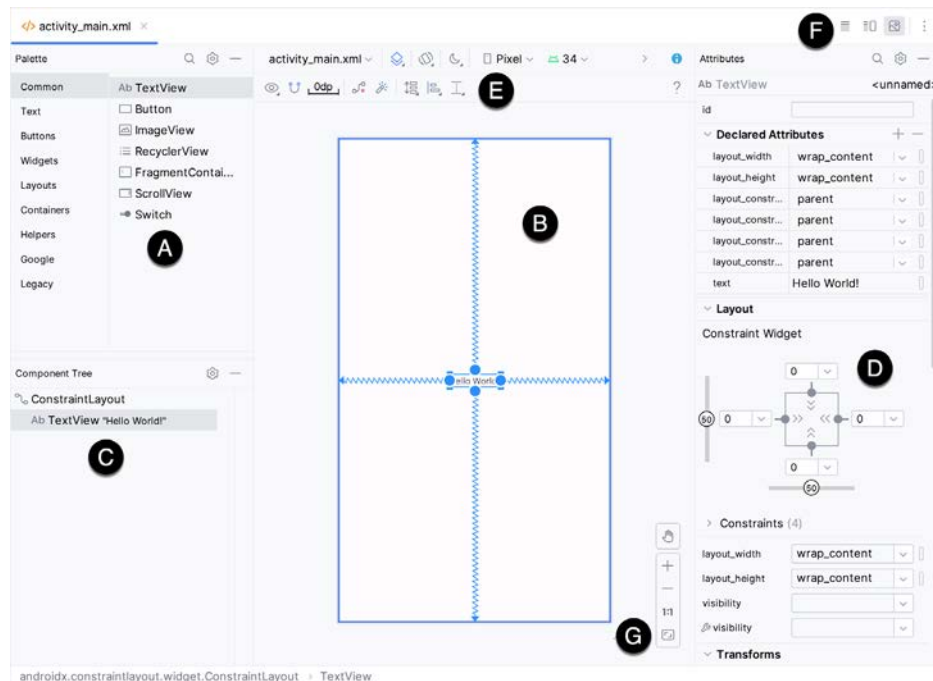


Figure 18-5

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.

18.4 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 18-6) 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:

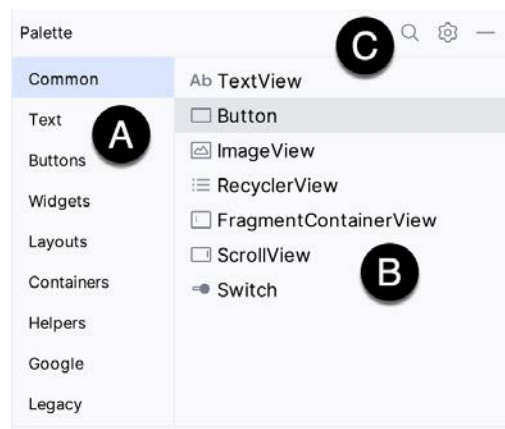


Figure 18-6

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 18-6 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.

18.5 Design Mode and Layout Views

The layout editor will appear in Design mode by default, as shown in Figure 18-5 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 18-7:

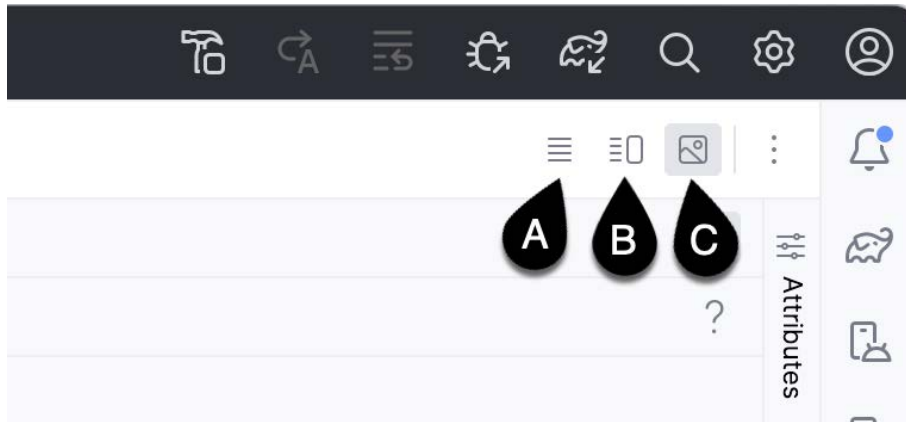


Figure 18-7

When the Layout Editor tool is in Design mode, the layout can be viewed in two ways. The view shown in Figure 18-5 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 18-8 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:

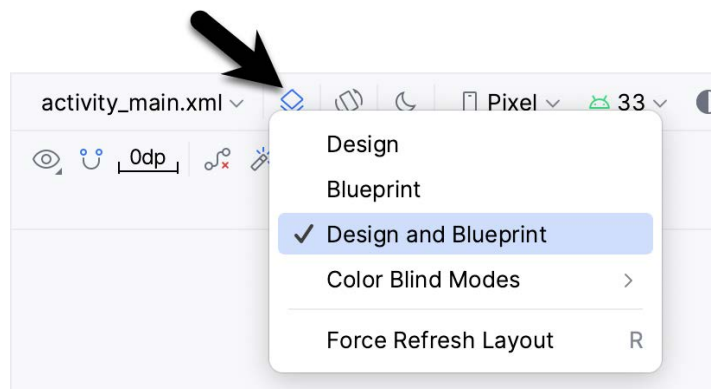


Figure 18-8

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 18-9:

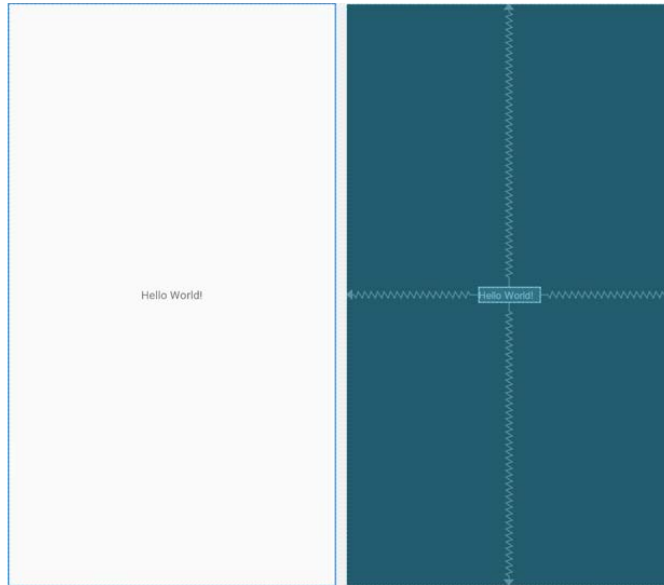


Figure 18-9

18.6 Night Mode

To view the layout in night mode during the design work, select the menu shown in Figure 18-10 below and change the setting to *Night*:

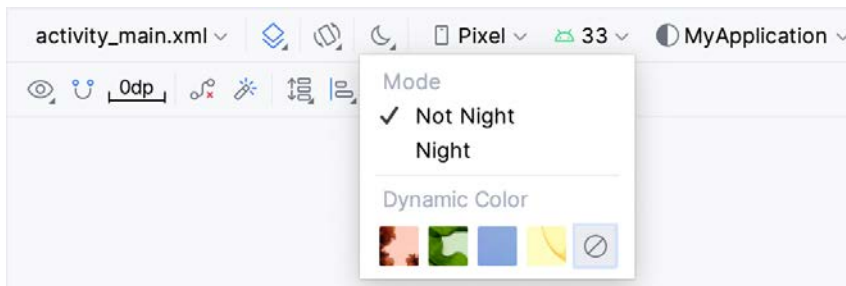


Figure 18-10

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*”.

18.7 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 18-7 above.

Figure 18-11 shows the Android Studio Layout Editor tool in Code mode, allowing changes to be made to the user interface declaration by modifying the XML:

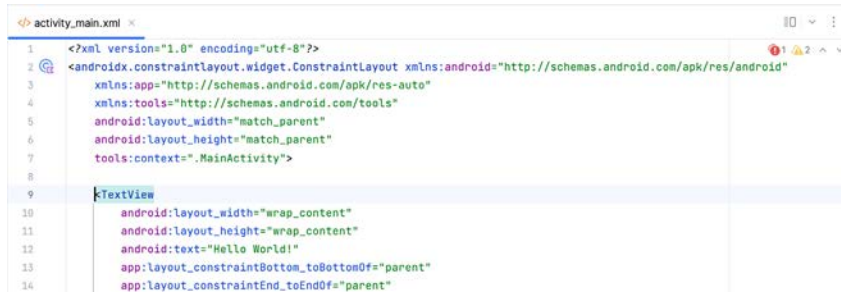


Figure 18-11

18.8 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 18-7 above.

Any changes to the XML are automatically reflected in the design canvas and vice versa. Figure 18-12 shows the editor in Split mode:

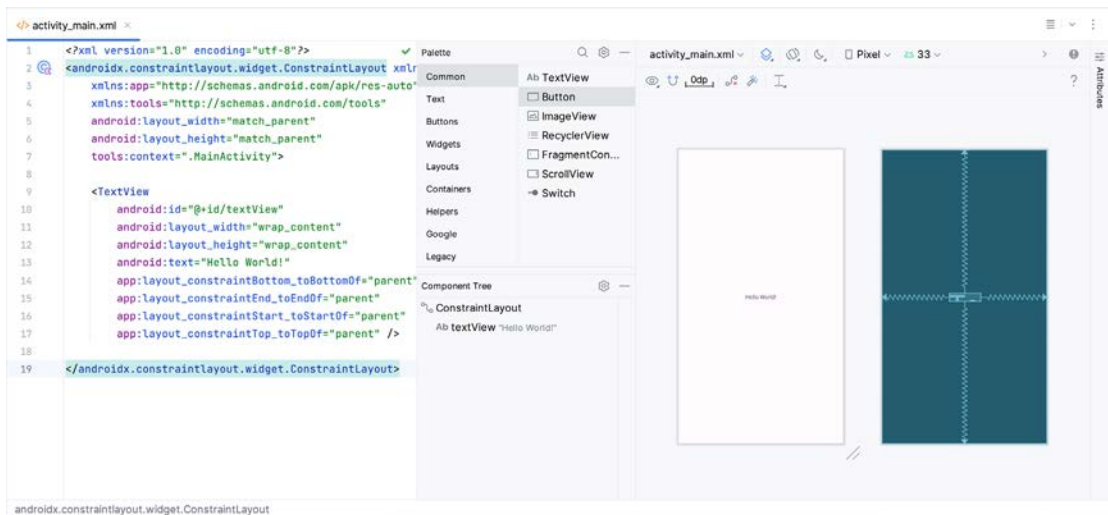


Figure 18-12

18.9 Setting Attributes

The Attributes panel provides access to all available settings for the currently selected component. Figure 18-13, for example, shows some of the attributes for the TextView widget:

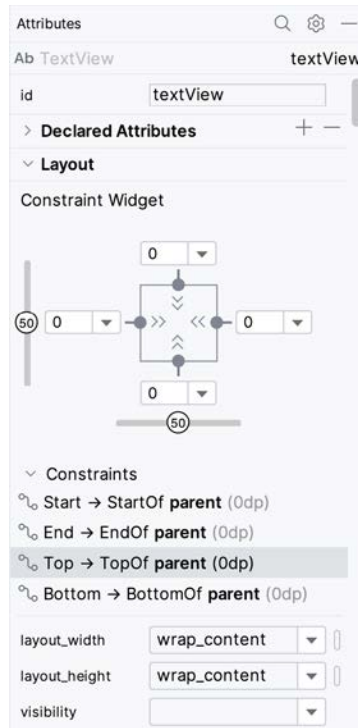


Figure 18-13

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 18-14 below:



Figure 18-14

Attributes for which a finite number of valid options are available will present a drop-down menu (Figure 18-15) from which a selection may be made.

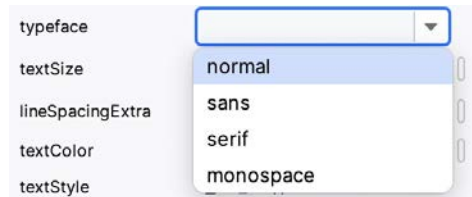


Figure 18-15

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.

18.10 Transforms

The transforms panel within the Attributes tool window (Figure 18-16) 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):

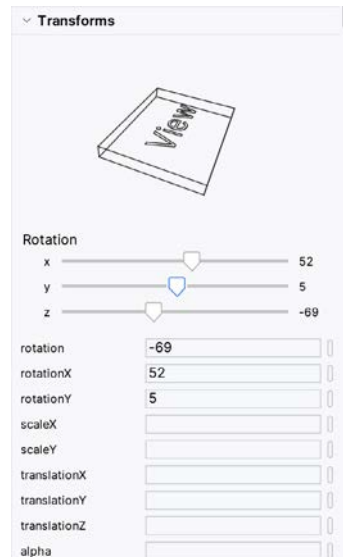


Figure 18-16

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.

18.11 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 18-17:

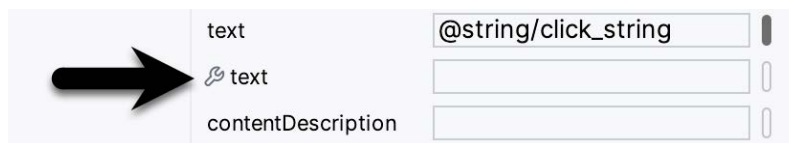


Figure 18-17

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 18-18, for example, shows the tools visibility toggle controls for a Button view named myButton:

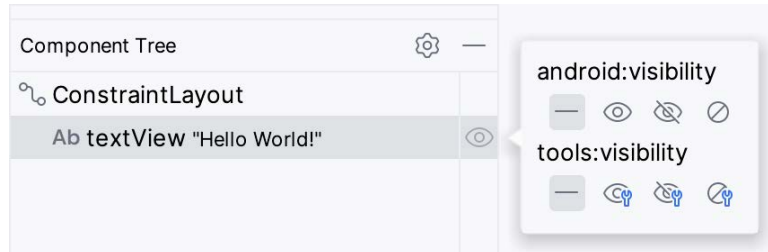


Figure 18-18

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 18-19 below:

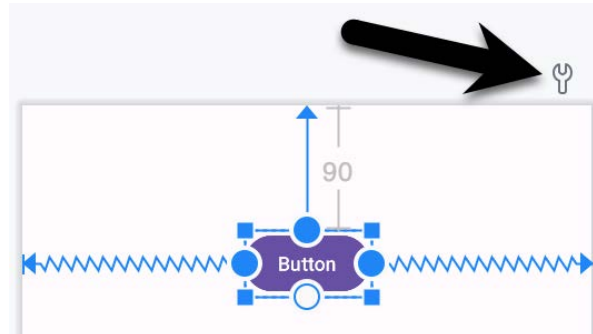


Figure 18-19

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”.

18.12 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 18-20):

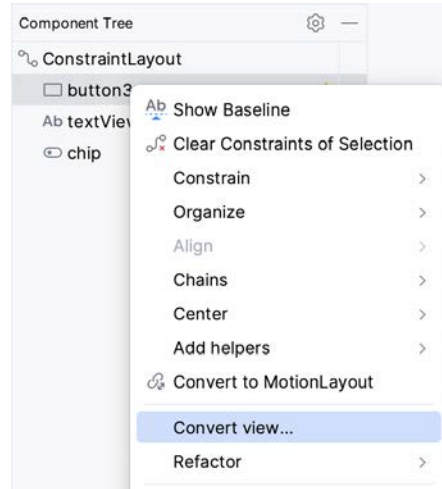


Figure 18-20

Once selected, a dialog containing a list of compatible view types to which the selected object is eligible for conversion will appear. Figure 18-21, for example, shows the types to which an existing `TextView` view may be converted:

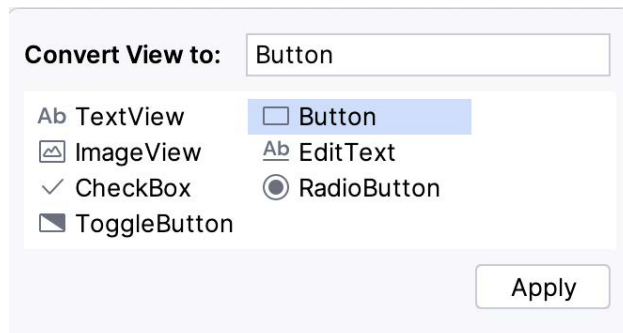


Figure 18-21

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

18.13 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 18-22, for example, shows the sample data options displayed when selecting sample data to appear in a `RecyclerView` list:

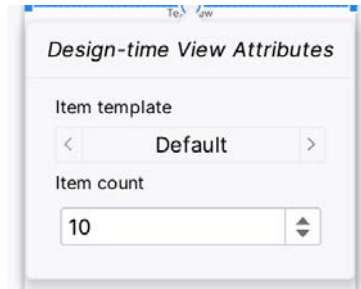


Figure 18-22

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 18-19 above.

18.14 Creating a Custom Device Definition

The device menu in the Layout Editor toolbar (Figure 18-23) 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*”.

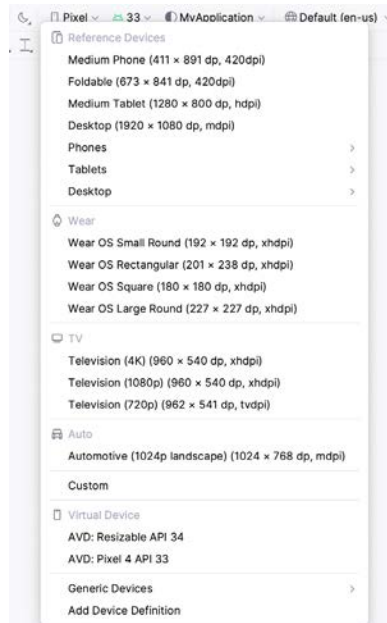


Figure 18-23

18.15 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 18-24), 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 18-24

18.16 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, select the *View -> Tool Windows -> Layout Validation* menu option. Once loaded, the panel will appear as shown in Figure 18-25, with the layout rendered on multiple device screen configurations:

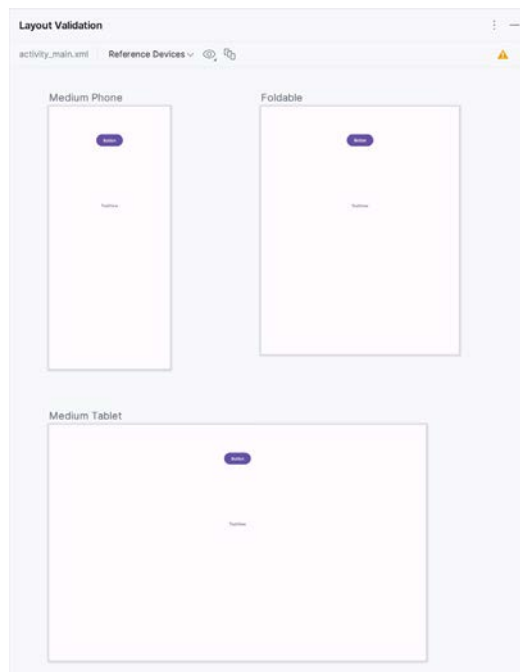


Figure 18-25

18.17 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.

34. 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.

34.1 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.

34.2 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.

34.3 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.

34.4 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.

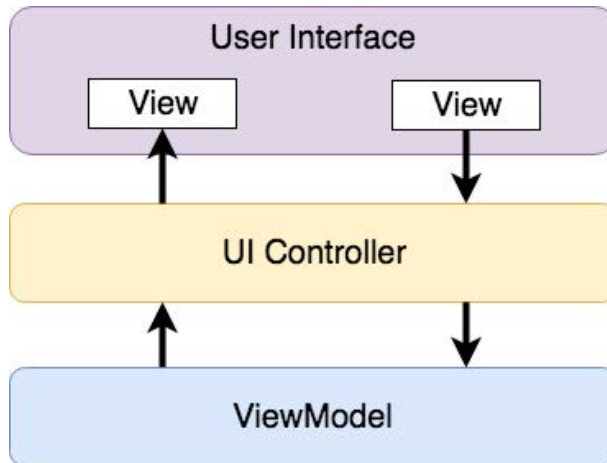


Figure 34-1

34.5 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.

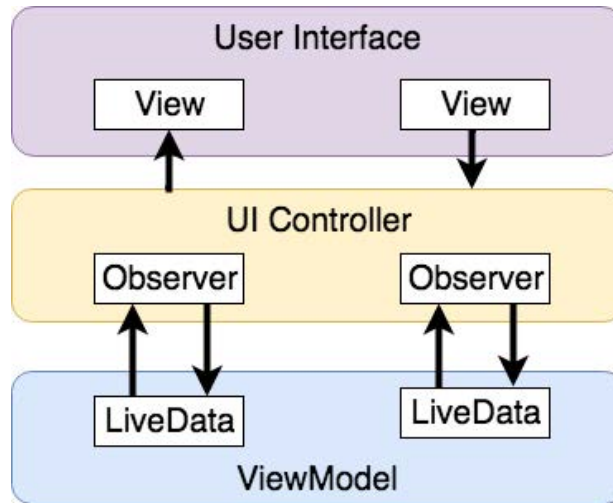


Figure 34-2

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.

34.6 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.

34.7 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.

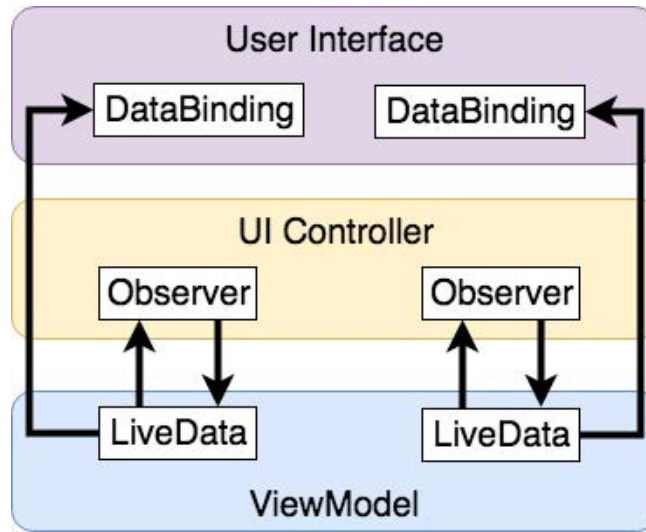


Figure 34-3

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

34.8 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.

34.9 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.

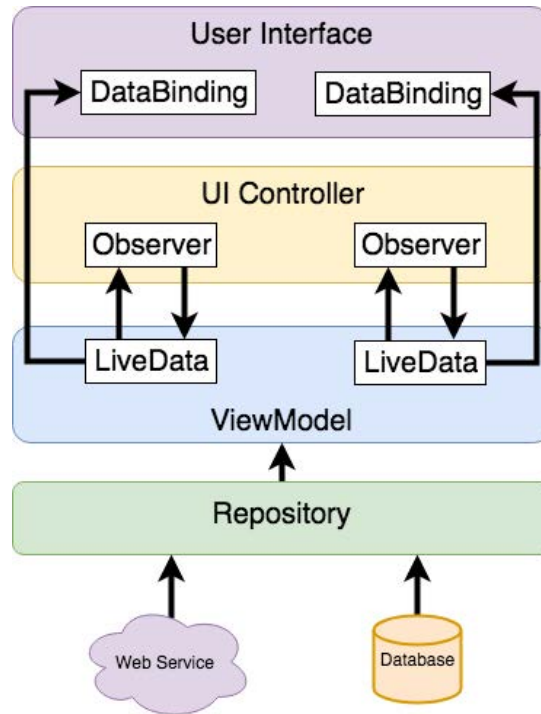


Figure 34-4

34.10 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 Android ViewModel Tutorial

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

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

35.1 About the Project

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

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

35.2 Creating the ViewModel Example Project

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

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

Enter *ViewModelDemo* into the Name field and specify *com.ebookfrenzy.viewmodeldemo* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.

35.3 Removing Unwanted Project Elements

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

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

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

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

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

35.4 Designing the Fragment Layout

The next step is to design the layout of the fragment. First, locate the *fragment_first.xml* file in the Project tool window and double-click on it to load it into the layout editor. Once the layout has loaded, select and delete the existing Button, TextView, and ConstraintLayout components. Next, right-click on the *NestedScrollView* instance in the Component Tree panel and select the *Convert NestedScrollView to ConstraintLayout* menu option as shown in Figure 35-1, and accept the default settings in the resulting dialog:

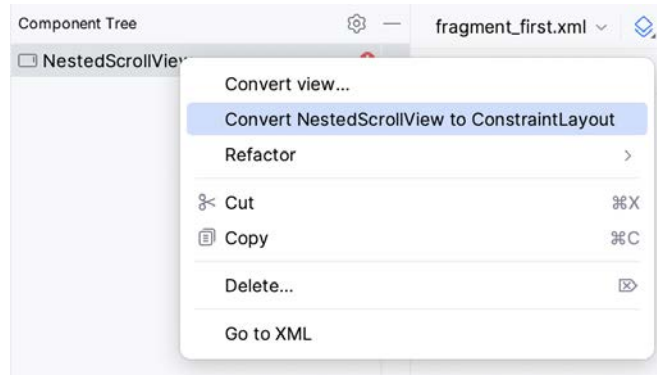


Figure 35-1

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

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

Drag a `Button` widget onto the layout to position it below the `TextView`, and change the text attribute to read “Convert”. With the button still selected, change the id property to `convertButton`. At this point, the layout should resemble that illustrated in Figure 35-2 (note that the three views have been constrained using a vertical chain):

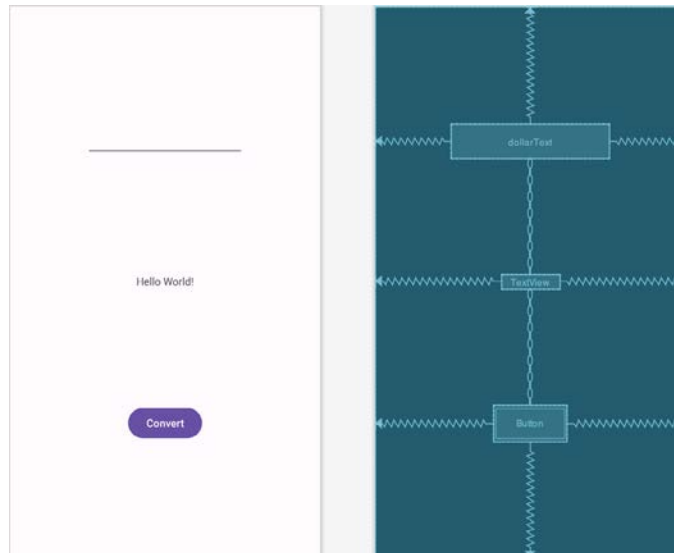


Figure 35-2

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

35.5 Implementing the View Model

With the user interface layout completed, the data model for the app needs to be created within the view model. Begin by locating the `com.ebookfrenzy.viewmodeldemo` entry in the Project tool window, right-clicking on it, and selecting the `New -> Java Class` menu option. Name the new class `MainViewModel` and press the keyboard enter

key. Edit the new class file so that it reads as follows:

```
package com.ebookfrenzy.viewmodeldemo.ui.main;

import androidx.lifecycle.ViewModel;

public class MainViewModel extends ViewModel {

    private static final Float rate = 0.74F;
    private String dollarText = "";
    private Float result = 0F;

    public void setAmount(String value) {
        this.dollarText = value;
        result = Float.parseFloat(dollarText)*rate;
    }

    public Float getResult()
    {
        return result;
    }
}
```

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

35.6 Associating the Fragment with the View Model

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

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

```
ViewModelProvider viewModelProvider = new ViewModelProvider(this);
```

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

```
ViewModel viewModel = viewModelProvider.get(MainViewModel.class);
```

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

```
.
.
import androidx.lifecycle.ViewModelProvider;
```

```

import androidx.annotation.Nullable;
.
.
public class FirstFragment extends Fragment {

    private MainViewModel viewModel;
.
.
    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        viewModel = new ViewModelProvider(this).get(MainViewModel.class);
    }
.
.

```

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

35.7 Modifying the Fragment

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

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

```

.
.
public class MainFragment extends Fragment {

    private MainViewModel viewModel;
.
.
    @Override
    public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);

        binding.convertButton.setOnClickListener(v -> {

        });
    }
.
.
}

```

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

35.8 Accessing the ViewModel Data

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

Since `LiveData` has yet to be used in the project, it will also be necessary to get the latest result value from the `ViewModel` each time the `Fragment` is created.

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

```
.
.
import java.util.Locale;
.
.
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle
savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    binding.resultText.setText(String.format(Locale.ENGLISH, "%.2f",
        viewModel.getResult()));

    binding.convertButton.setOnClickListener(v -> {
        if (!binding.dollarText.getText().toString().equals("")) {
            viewModel.setAmount(String.format(Locale.ENGLISH, "%s",
                binding.dollarText.getText()));
            binding.resultText.setText(String.format(Locale.ENGLISH, "%.2f",
                viewModel.getResult()));
        } else {
            binding.resultText.setText("No Value");
        }
    });
}
```

35.9 Testing the Project

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

When the original `AndroidSample` app was run, rotating the device caused the value displayed on the `resultText` `TextView` widget to be lost. Repeat this test now with the `ViewModelDemo` app and note that the current euro value is retained after the rotation. This is because the `ViewModel` remained in memory as the `Fragment` was destroyed and recreated, and code was added to the `onViewCreated()` method to update the `TextView` with the result data value from the `ViewModel` each time the `Fragment` re-started.

While this is an improvement on the original `AndroidSample` app, much more can be done to simplify the project by using `LiveData` and data binding, both of which are the topics of the next chapters.

35.10 Summary

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

48. Creating a Tabbed Interface using the TabLayout Component

The previous chapter outlined the concept of material design in Android. It introduced two of the components provided by the design support library in the form of the floating action button and the Snackbar. This chapter will demonstrate how to use another of the design library components, the TabLayout, which can be combined with the ViewPager class to create a tab-based interface within an Android activity.

48.1 An Introduction to the ViewPager2

Although not part of the design support library, ViewPager2 is a useful companion class when used with the TabLayout component to implement a tabbed user interface. The primary role of ViewPager2 is to allow the user to flip through different pages of information where a layout fragment most typically represents each page. The fragments associated with ViewPager2 are managed by an instance of the FragmentStateAdapter class.

At a minimum, the pager adapter assigned to ViewPager2 must implement two methods. The first, named *getItemCount()*, must return the total number of page fragments to be displayed to the user. The second method, *createFragment()*, is passed a page number and must return the corresponding fragment object ready to be presented to the user.

48.2 An Overview of the TabLayout Component

As previously discussed, TabLayout is one of the components introduced in material design and is included in the design support library. The purpose of the TabLayout is to present the user with a row of tabs that can be selected to display different pages to the user. The tabs can be fixed or scrollable, whereby the user can swipe left or right to view more tabs than will currently fit on the display. The information displayed on a tab can be text-based, an image, or a combination of text and images. Figure 48-1, for example, shows the tab bar for an app consisting of four tabs displaying images:



Figure 48-1

Figure 48-2, on the other hand, shows a TabLayout configuration consisting of four tabs displaying text in a scrollable configuration:



Figure 48-2

The remainder of this chapter will work through creating an example project that demonstrates the TabLayout component together with a ViewPager2 and four fragments.

48.3 Creating the TabLayoutDemo Project

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

Enter *TabLayoutDemo* into the Name field and specify *com.ebookfrenzy.tablayoutdemo* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.

Once the project has been created, load the *content_main.xml* file into the Layout Editor tool, select the NavHostFragment object, and then delete it. Since we will not be using the navigation features of the Basic Views Activity template, edit the *MainActivity.java* file and modify the *onCreate()* method to remove the navigation code:

```
@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();
        }
    });
}
```

Finally, delete the *onSupportNavigateUp()* method:

```
@Override
public boolean onSupportNavigateUp() {
    NavController navController = Navigation.findNavController(this, R.id.nav_
host_fragment_content_main);
    return NavigationUI.navigateUp(navController, appBarConfiguration)
        || super.onSupportNavigateUp();
}
```

48.4 Creating the First Fragment

Each tab on the TabLayout will display a different fragment when selected. Create the first of these fragments by right-clicking on the `app -> java -> com.ebookfrenzy.tablayoutdemo` entry in the Project tool window and selecting the `New -> Fragment -> Fragment (Blank)` option. In the resulting dialog, enter `Tab1Fragment` into the `Fragment Name:` field and `fragment_tab1` into the `Fragment Layout Name:` field. Click on the `Finish` button to create the new fragment:

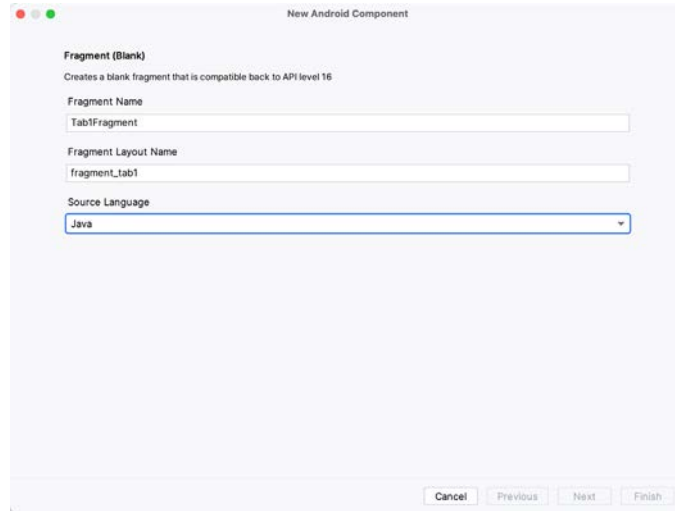


Figure 48-3

Edit the `Tab1Fragment.java` file, and if Android Studio has not added one automatically, add an `OnFragmentInteractionListener` interface declaration as follows:

```
.
.
import android.net.Uri;
.
.
    public interface OnFragmentInteractionListener {
        // TODO: Update argument type and name
        void onFragmentInteraction(Uri uri);
    }
.
.
```

Load the newly created `fragment_tab1.xml` file (located under `app -> res -> layout`) into the Layout Editor tool, right-click on the `FrameLayout` entry in the Component Tree panel, and select the `Convert FrameLayout to ConstraintLayout` menu option. In the resulting dialog, verify that all conversion options are selected before clicking on `OK`. Change the ID of the layout to `constraintLayout`.

Once the layout has been converted to a `ConstraintLayout`, delete the `TextView` from the layout. From the Palette, locate the `TextView` widget and drag and drop it so it is positioned in the center of the layout. Edit the object's text property to read "Tab 1 Fragment", extract the string to a resource named `tab_1_fragment`, and click the `Infer Constraints` toolbar button. At this point, the layout should match that of Figure 48-4:

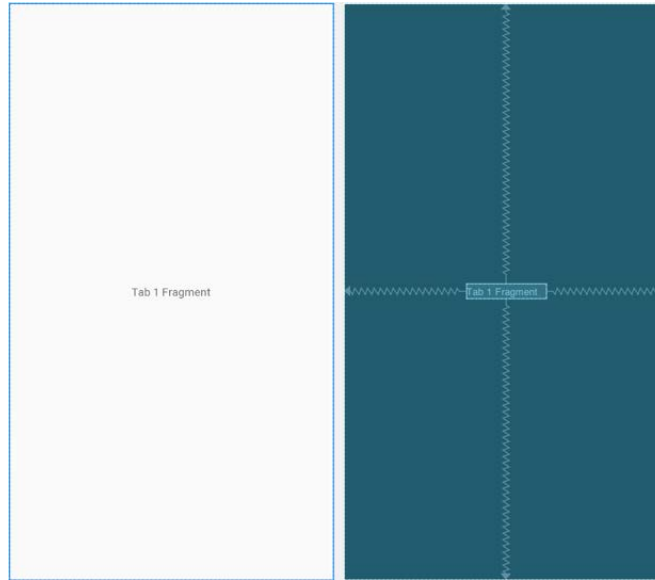


Figure 48-4

48.5 Duplicating the Fragments

So far, the project contains one of the four required fragments. It would be quicker to duplicate the first fragment instead of creating the remaining three fragments using the previous steps. Each fragment consists of a layout XML file and a Java class file, each needing to be duplicated.

Right-click on the *fragment_tab1.xml* file in the Project tool window and select the Copy option from the resulting menu. Right-click on the *layout* entry, this time selecting the Paste option. Name the new layout file *fragment_tab2.xml* in the resulting dialog before clicking the OK button. Edit the new *fragment_tab2.xml* file and change the text on the Text View to “Tab 2 Fragment”, following the usual steps to extract the string to a resource named *tab_2_fragment*.

To duplicate the *Tab1Fragment* class file, right-click on the class listed under *app -> java -> com.ebookfrenzy.tablayoutdemo* and select Copy. Right-click on the *com.ebookfrenzy.tablayoutdemo* entry and select Paste. In the Copy Class dialog, enter *Tab2Fragment* into the *New name:* field and click OK.

Edit the new *Tab2Fragment.java* file and modify the *onCreateView()* method to inflate the *fragment_tab2* layout file:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                        Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.fragment_tab2, container, false);
}
```

Perform the above duplication steps twice to create the fragment layout and class files for the remaining two fragments. On completion of these steps, the project structure should match that of Figure 48-5:

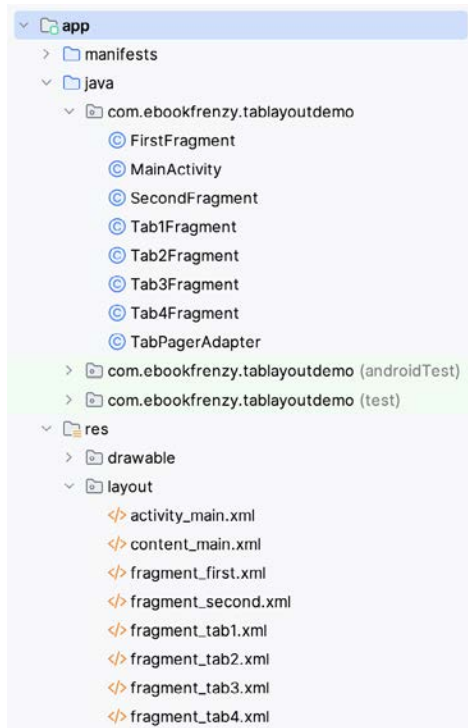


Figure 48-5

48.6 Adding the TabLayout and ViewPager2

With the fragment creation process now complete, the next step is to add the TabLayout and ViewPager2 to the main activity layout file. Edit the *activity_main.xml* file and add these elements as outlined in the following XML listing. Note that the TabLayout component is embedded into the AppBarLayout element while the ViewPager2 is placed after the AppBarLayout:

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

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/Theme.TabLayoutDemo.AppBarOverlay">

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
```

Creating a Tabbed Interface using the TabLayout Component

```
    android:background="?attr/colorPrimary"
    app:popupTheme="@style/Theme.TabLayoutDemo.PopupOverlay" />
```

```
    <com.google.android.material.tabs.TabLayout
        android:id="@+id/tabLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:tabMode="fixed"
        app:tabGravity="fill"/>
```

```
</com.google.android.material.appbar.AppBarLayout>
```

```
    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/view_pager"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior" />
```

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

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

```
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

48.7 Creating the Pager Adapter

This example will use the ViewPager2 approach to handling the fragments assigned to the TabLayout tabs, with ViewPager2 added to the layout resource file, a new class which subclasses FragmentStateAdapter needs to be added to the project to manage the fragments that will be displayed when the user selects the tab items.

Add a new class to the project by right-clicking on the *com.ebookfrenzy.tablayoutdemo* entry in the Project tool window and selecting the *New -> Java Class* menu option. In the new class dialog, enter *TabPagerAdapter* into the *Name:* field, select the *Class* item in the list, and press the keyboard Return key.

Edit the *TabPagerAdapter.java* file so that it reads as follows:

```
package com.ebookfrenzy.tablayoutdemo;

import androidx.annotation.NonNull;
import androidx.fragment.app.*;
import androidx.viewpager2.adapter.FragmentStateAdapter;
```

```

public class TabPagerAdapter extends FragmentStateAdapter {

    int tabCount;

    public TabPagerAdapter(@NonNull FragmentActivity fragmentActivity, int
numberOfTabs) {
        super(fragmentActivity);
        this.tabCount = numberOfTabs;
    }

    @NonNull
    @Override
    public Fragment createFragment(int position) {
        switch (position) {
            case 0:
                return new Tab1Fragment();
            case 1:
                return new Tab2Fragment();
            case 2:
                return new Tab3Fragment();
            case 3:
                return new Tab4Fragment();
            default:
                return null;
        }
    }

    @Override
    public int getItemCount() {
        return tabCount;
    }
}

```

The class is declared as extending the `FragmentStateAdapter` class, and a constructor is implemented, allowing the number of pages required to be passed to the class when an instance is created. The `createFragment()` method will be called when a specific page is required. A switch statement is used to identify the page number being requested and to return a corresponding fragment instance. Finally, the `getItemCount()` method returns the count value passed through when the object instance was created.

48.8 Performing the Initialization Tasks

The remaining tasks involve initializing the `TabLayout`, `ViewPager2`, and `TabPagerAdapter` instances and declaring the main activity class as implementing fragment interaction listeners for each of the four tab fragments. Edit the `MainActivity.java` file so that it reads as follows:

```

package com.ebookfrenzy.tablayoutdemo;

.

.

import android.net.Uri;

```

Creating a Tabbed Interface using the TabLayout Component

```
import com.google.android.material.tabs.TabLayoutMediator;
.
.
public class MainActivity extends AppCompatActivity implements
    Tab1Fragment.OnFragmentInteractionListener,
    Tab2Fragment.OnFragmentInteractionListener,
    Tab3Fragment.OnFragmentInteractionListener,
    Tab4Fragment.OnFragmentInteractionListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        .
        .

        configureTabLayout();
    }

    protected void configureTabLayout() {

        for (int i = 0; i < 4; i++) {
            binding.tabLayout.addTab(binding.tabLayout.newTab());
        }

        final TabPagerAdapter adapter = new TabPagerAdapter
            (this, binding.tabLayout.getTabCount());
        binding.viewPager.setAdapter(adapter);

        new TabLayoutMediator(binding.tabLayout, binding.viewPager,
            (tab, position) -> tab.setText("Tab " + (position + 1) +
                " Item")).attach();
    }

    @Override
    public void onFragmentInteraction(Uri uri) {
    }
    .
    .
}
```

The code begins by creating four tabs and adding them to the TabLayout instance as follows:

```
for (int i = 0; i < 4; i++) {
    binding.tabLayout.addTab(binding.tabLayout.newTab());
}
```

Next, an instance of the TabPagerAdapter class is created. Note that the code to create the TabPagerAdapter instance passes through the number of tabs assigned to the TabLayout component. The TabPagerAdapter instance is then assigned as the adapter for the ViewPager2 instance:


```
final TabPagerAdapter adapter = new TabPagerAdapter
    (this, binding.tabLayout.getTabCount());
binding.viewPager.setAdapter(adapter);
```

Finally, an instance of the TabLayoutMediator class is used to connect the TabLayout with the ViewPager2 object:

```
new TabLayoutMediator(binding.tabLayout, binding.viewPager,
    (tab, position) -> tab.setText("Tab " + (position + 1) + " Item")).
attach();
```

This class ensures that the TabLayout tabs remain synchronized with the currently selected fragment. This process involves ensuring that the correct text is displayed on each tab. In this case, the text is configured to read “Tab <*n*> Item” where <*n*> is replaced by the number of the currently selected tab.

48.9 Testing the Application

Compile and run the app on a device or emulator and make sure that selecting a tab causes the corresponding fragment to appear in the content area of the screen:



Figure 48-6

48.10 Customizing the TabLayout

The TabLayout in this example project is configured using *fixed* mode. This mode works well for a limited number of tabs with short titles. A greater number of tabs or longer titles can quickly become a problem when using fixed mode, as illustrated by Figure 48-7:

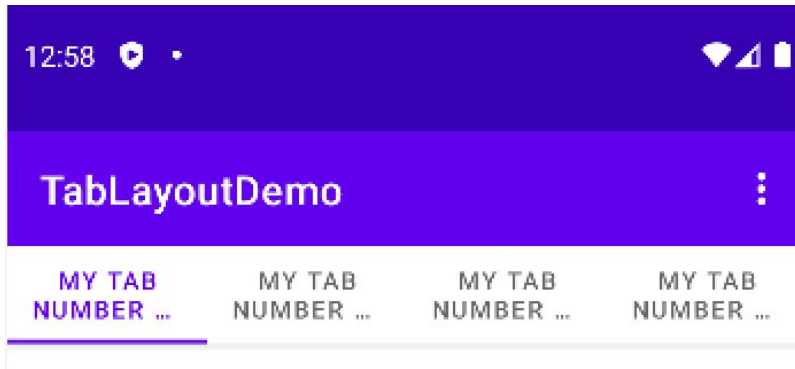


Figure 48-7

To fit the tabs into the available display width, the TabLayout has used multiple lines of text. Even so, the second line is truncated, making it impossible to see the full title. The best solution to this problem is to switch the TabLayout to scrollable mode. In this mode, the titles appear in full-length, single-line format allowing the user to swipe to scroll horizontally through the available items, as demonstrated in Figure 48-8:

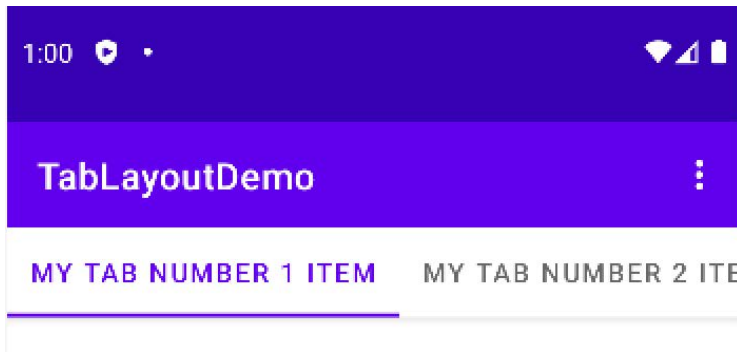


Figure 48-8

To switch a TabLayout to scrollable mode, change the *app:tabMode* property in the *activity_main.xml* layout resource file from “fixed” to “scrollable”:

```
<android.support.design.widget.TabLayout
    android:id="@+id/tabLayout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:tabMode="scrollable"
    app:tabGravity="fill"/>
</android.support.design.widget.AppBarLayout>
```

When in fixed mode, the TabLayout may be configured to control how the tab items are displayed to take up the available space on the screen. This is controlled via the *app:tabGravity* property, the results of which are more noticeable on wider displays such as tablets in landscape orientation. When set to “fill”, for example, the items will be distributed evenly across the width of the TabLayout, as shown in Figure 48-9:



Figure 48-9

Changing the property value to “center” will cause the items to be positioned relative to the center of the tab bar:

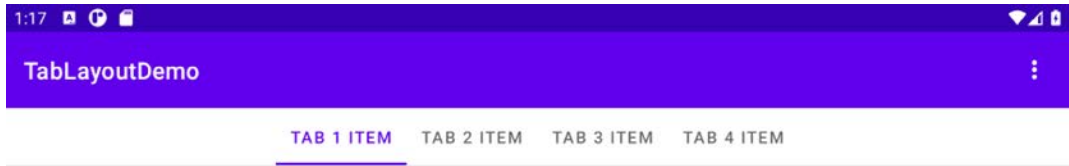


Figure 48-10

48.11 Summary

TabLayout is one of the components introduced in the Android material design implementation. The purpose of the TabLayout component is to present a series of tab items that display different content to the user when selected. The tab items can display text, images, or a combination. When combined with the ViewPager2 class and fragments, tab layouts can be created relatively easily, with each tab item selection displaying a different fragment.

70. 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.

70.1 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 70-1 outlines the recommended architecture for a typical Android app:

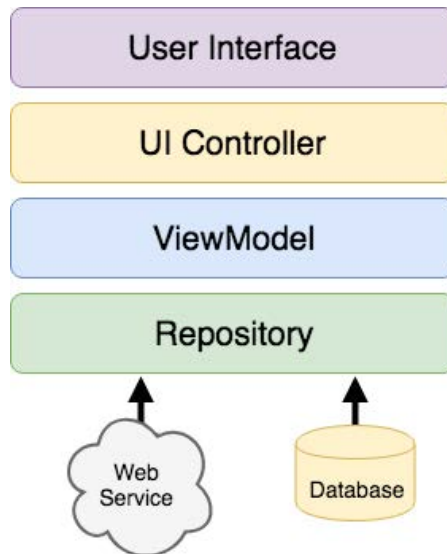


Figure 70-1

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.

70.2 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:

70.2.1 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.

70.2.2 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.

70.2.3 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.

70.2.4 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.

70.2.5 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 70-2 illustrates how these different elements interact to provide Room-based database storage within an Android app:

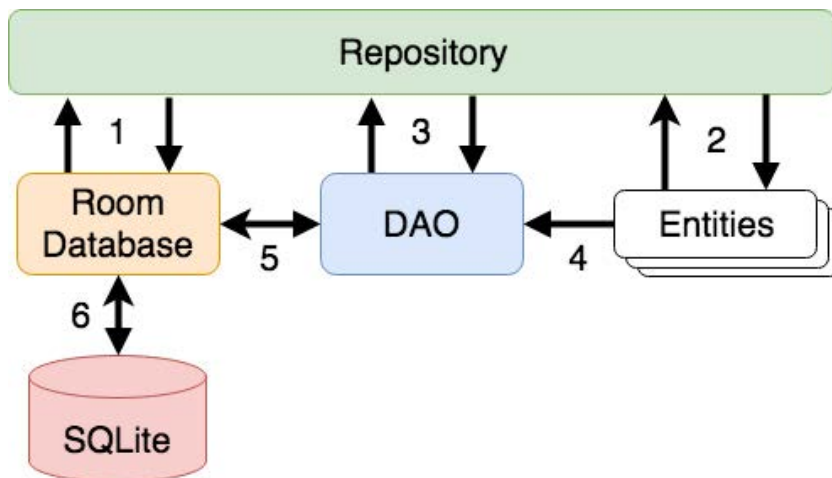


Figure 70-2

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.

70.3 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) {
```

The Android Room Persistence Library

```
        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.

70.4 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);
```

70.5 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) {
```

The Android Room Persistence Library

```
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.

70.6 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”).

70.7 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();
```

70.8 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 70-3:

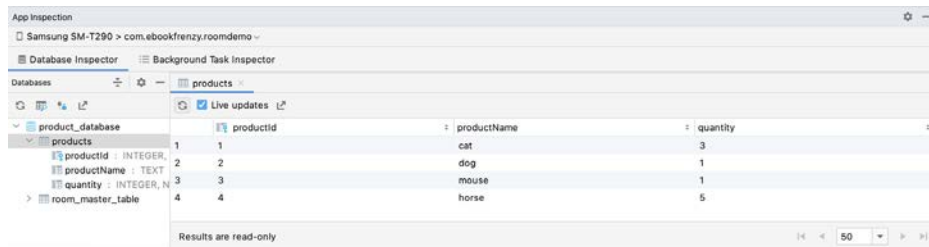


Figure 70-3

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

70.9 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.

91. Working with Material Design 3 Theming

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

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

91.1 Material Design 2 vs. Material Design 3

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

91.2 Understanding Material Design Theming

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

91.3 Material Design 3 Theming

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

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

    <application
    .
    .
```

Working with Material Design 3 Theming

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

As previously discussed, all of the files associated with the project theme are contained within the *colors.xml* and *themes.xml* files located in the *res -> values* folder, as shown in Figure 91-1:

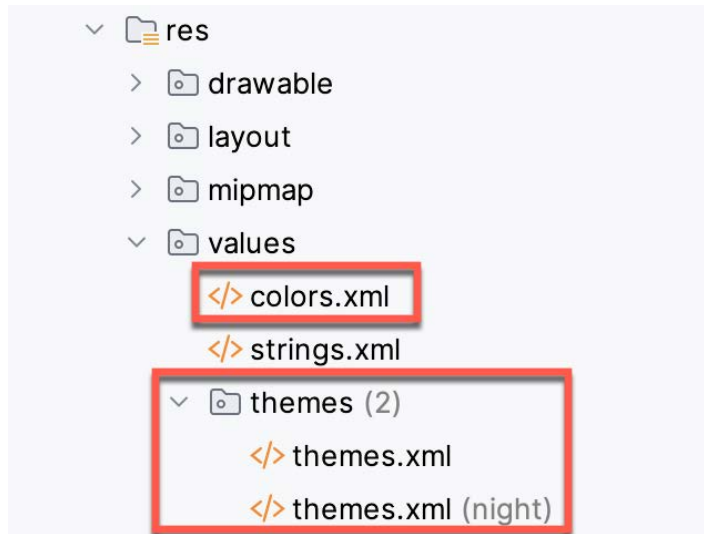


Figure 91-1

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

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

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

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

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

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

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

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

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

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

91.4 Building a custom theme

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

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

Figure 91-2 shows a recent version of the Theme Builder. However, Google redesigns the builder every few weeks, so it is possible that the user interface you see will not match the current design:

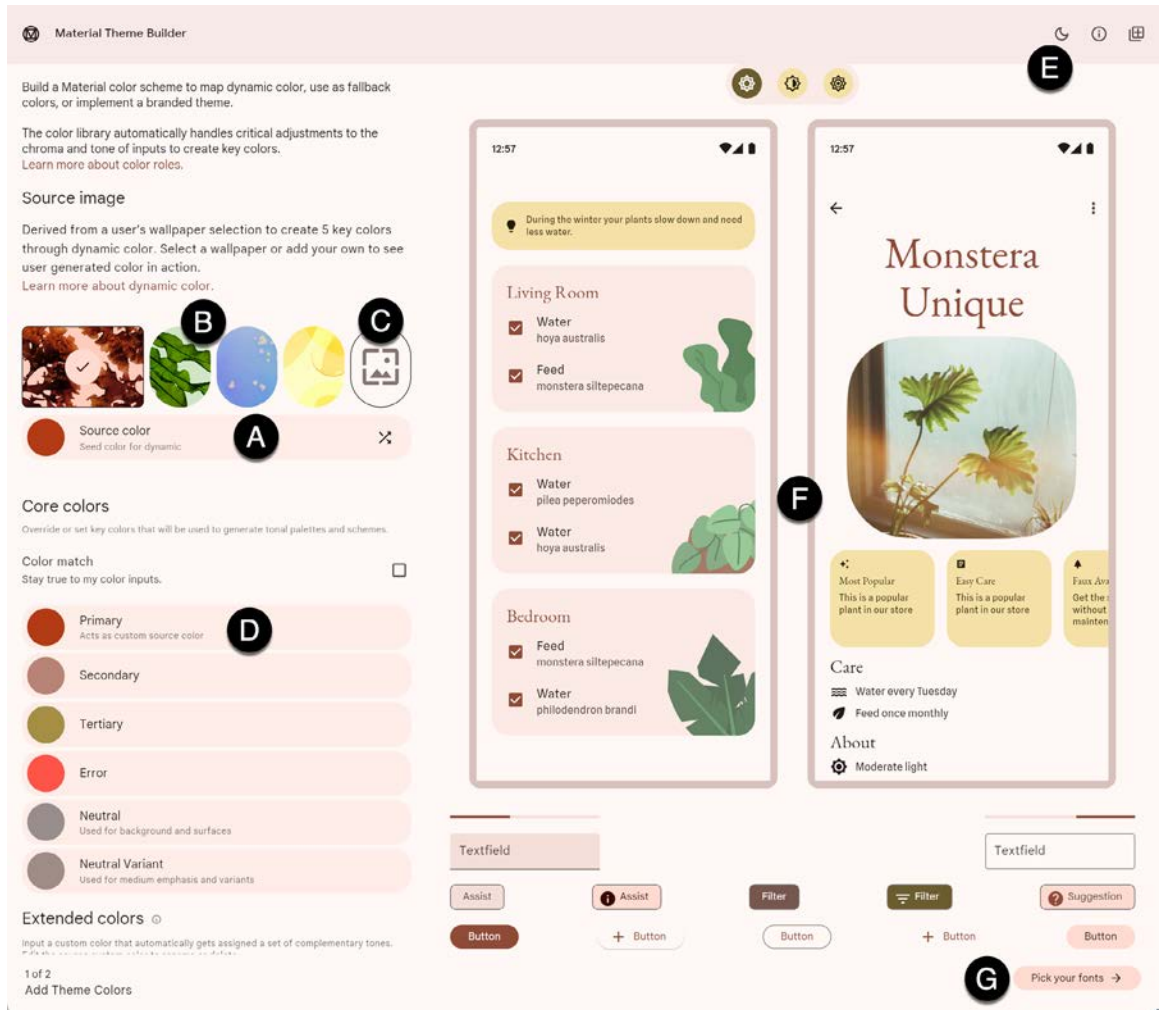


Figure 91-2

A custom Material color theme can be generated by providing a source color of your own choosing (marked A) in the figure above, selecting a standard wallpaper (B), uploading a source image (C), or making individual core color selections (D). The button marked E previews the color scheme in light and dark modes.

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

Once the theme colors have been defined, clicking on the Pick your fonts button (G) button will display the screen shown in Figure 91-3, where you can select the fonts to be used by your app:

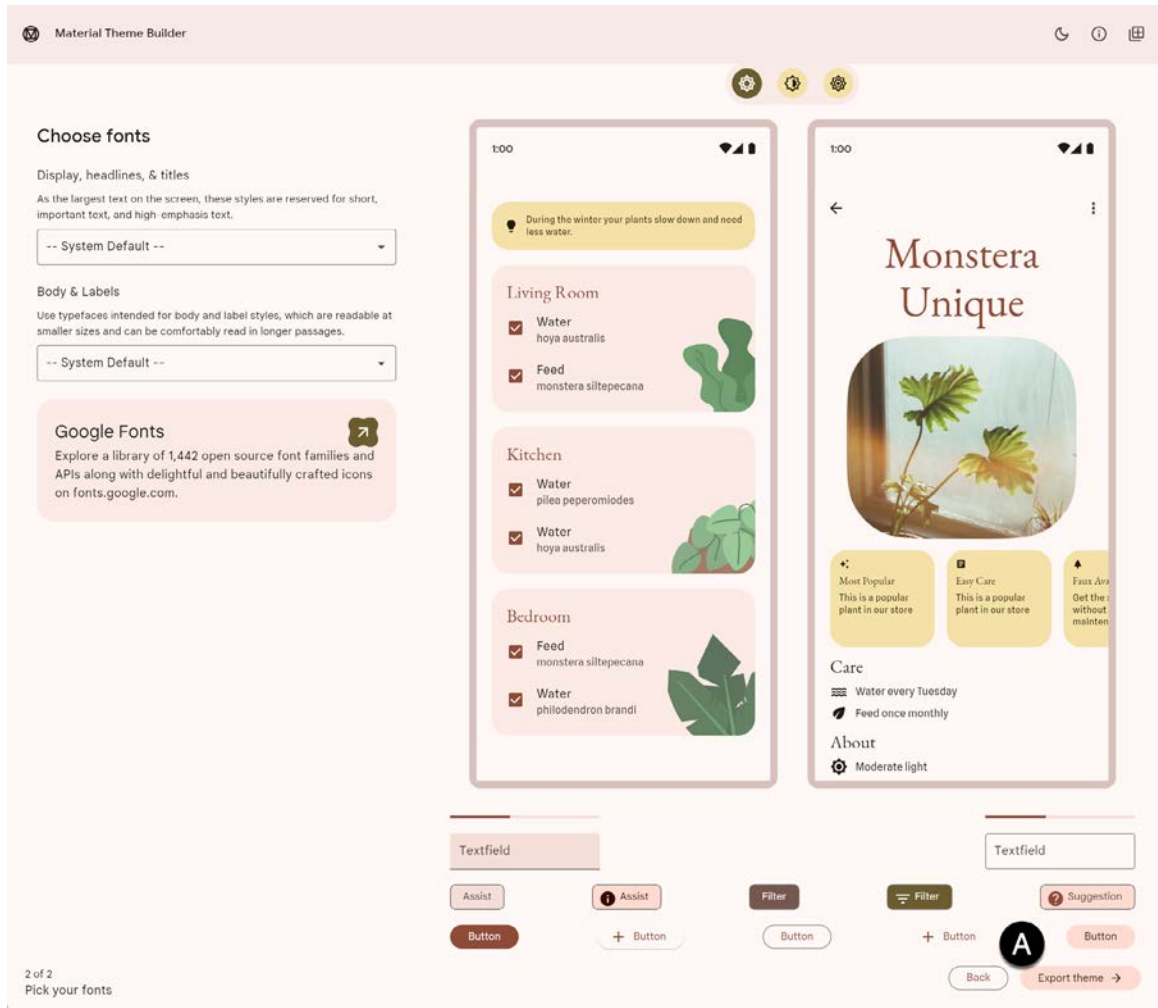


Figure 91-3

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

91.5 Summary

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

Index

Symbols

<application> 430
 <fragment> 253
 <fragment> element 253
 <menu> 645
 <provider> 557
 <receiver> 464
 <service> 430, 474, 481
 .well-known folder 437, 460, 764

A

AbsoluteLayout 128
 ACCESS_COARSE_LOCATION permission 498
 ACCESS_FINE_LOCATION permission 498
 acknowledgePurchase() method 629
 ACTION_CREATE_DOCUMENT 679
 ACTION_CREATE_INTENT 680
 ACTION_DOWN 228
 ACTION_MOVE 228
 ACTION_OPEN_DOCUMENT intent 672
 ACTION_POINTER_DOWN 228
 ACTION_POINTER_UP 228
 ACTION_UP 228
 ACTION_VIEW 455
 Active / Running state 102
 Activity 89, 105
 adding views in Java code 205
 class 105
 creation 14
 Entire Lifetime 109
 Foreground Lifetime 109
 lifecycle methods 108
 lifecycles 99
 returning data from 434
 state change example 113

 state changes 105
 states 102
 Visible Lifetime 109
 Activity Lifecycle 101
 Activity Manager 88
 ActivityResultLauncher 435
 Activity Stack 101
 Actual screen pixels 196
 adb
 command-line tool 59
 connection testing 65
 device pairing 63
 enabling on Android devices 59
 Linux configuration 62
 list devices 59
 macOS configuration 60
 overview 59
 restart server 60
 testing connection 65
 WiFi debugging 63
 Windows configuration 61
 Wireless debugging 63
 Wireless pairing 63
 addCategory() method 463
 addMarker() method 728
 addView() method 199
 ADD_VOICEMAIL permission 498
 android
 checkableBehavior 647
 exported 431
 gestureColor 246
 layout_behavior property 423
 onClick 255
 orderInCategory 646
 process 431, 481
 uncertainGestureColor 246
 Android
 Activity 89
 architecture 85

Index

- events 221
- intents 90
- onClick Resource 221
- runtime 86
- SDK Packages 5
- android.app 86
- Android Architecture Components 271
- android.content 86
- android.content.Intent 433
- android.database 86
- Android Debug Bridge. *See* ADB
- Android Development
 - System Requirements 3
- Android Devices
 - designing for different 127
- android.graphics 87
- android.hardware 87
- android.intent.action 469
- android.intent.action.BOOT_COMPLETED 431
- android.intent.action.MAIN 455
- android.intent.category.LAUNCHER 455
- Android Libraries 86
- android.media 87
- Android Monitor tool window 32
- Android Native Development Kit 87
- android.net 87
- android.opengl 87
- android.os 87
- android.permission.RECORD_AUDIO 707
- android.print 87
- Android Project
 - create new 13
- android.provider 87
- Android SDK Location
 - identifying 9
- Android SDK Manager 7, 9
- Android SDK Packages
 - version requirements 7
- Android SDK Tools
 - command-line access 8
 - Linux 10
 - macOS 10
 - Windows 7 9
 - Windows 8 9
- Android Software Stack 85
- Android Storage Access Framework 672
- Android Studio
 - changing theme 57
 - downloading 3
 - Editor Window 52
 - installation 4
 - Linux installation 5
 - macOS installation 4
 - Navigation Bar 51
 - Project tool window 52
 - Status Bar 52
 - Toolbar 51
 - Tool window bars 52
 - tool windows 52
 - updating 11
 - Welcome Screen 49
 - Windows installation 4
- android.text 87
- android.util 87
- android.view 87
- android.view.View 130
- android.view.ViewGroup 127, 130
- Android Virtual Device. *See* AVD
 - overview 27
- Android Virtual Device Manager 27
- android.webkit 87
- android.widget 87
- AndroidX libraries 792
- API Key 719
- APK analyzer 622
- APK file 615
- APK File
 - analyzing 622
- APK Signing 792
- APK Wizard dialog 614
- app
 - showAsAction 646
- App Architecture
 - modern 271

- AppBar
 - anatomy of 421
- appbar_scrolling_view_behavior 423
- App Bundles 611
 - creating 615
 - overview 611
 - revisions 621
 - uploading 618
- AppCompatActivity class 106
- App Inspector 53
- Application
 - stopping 32
- Application Context 91
- Application Framework 88
- Application Manifest 91
- Application Resources 91
- App Link
 - Adding Intent Filter 772
 - Digital Asset Links file 764, 437
 - Intent Filter Handling 772
 - Intent Filters 763
 - Intent Handling 764
 - Testing 776
 - URL Mapping 769
- App Links 763
 - auto verification 436
 - autoVerify 437
 - overview 763
- Apply Changes 213
 - Apply Changes and Restart Activity 213
 - Apply Code Changes 213
 - fallback settings 215
 - options 213
 - Run App 213
 - tutorial 215
- applyToActivitiesIfAvailable() method 788
- Architecture Components 271
- ART 86
- assetlinks.json , 764, 437
- Attribute Keyframes 348
- Audio
 - supported formats 705

- Audio Playback 705
- Audio Recording 705
- Auto Blocker 60
- Autoconnect Mode 161
- Automatic Link Verification 436, 459
- autoVerify 437, 772
- AVD
 - Change posture 48
 - cold boot 44
 - command-line creation 27
 - creation 27
 - device frame 35
 - Display mode 47
 - launch in tool window 35
 - overview 27
 - quickboot 44
 - Resizable 47
 - running an application 30
 - Snapshots 43
 - standalone 32
 - starting 29
 - Startup size and orientation 30

B

- Background Process 100
- Barriers 154
 - adding 173
 - constrained views 154
- Baseline Alignment 153
- beginTransaction() method 254
- BillingClient 630
 - acknowledgePurchase() method 629
 - consumeAsync() method 629
 - getPurchaseState() method 629
 - initialization 626, 637
 - launchBillingFlow() method 628
 - queryProductDetailsAsync() method 628
 - queryPurchasesAsync() method 630
- BillingResult 644
 - getDebugMessage() 644
- Binding Expressions 295
 - one-way 295

Index

- two-way 296
- BIND_JOB_SERVICE permission 431
- bindService() method 429, 471, 476
- Biometric Authentication 663
 - callbacks 667
 - overview 663
 - tutorial 663
- Biometric Prompt 668
- BitmapFactory 673
- black activity 14
- Blank template 131
- Blueprint view 159
- BODY_SENSORS permission 498
- Bound Service 429, 471
 - adding to a project 472
 - Implementing the Binder 472
 - Interaction options 471
- BoundService class 473
- Broadcast Intent 463
 - example 466
 - overview 90, 463
 - sending 466
 - Sticky 465
- Broadcast Receiver 463
 - adding to manifest file 468
 - creation 467
 - overview 90, 464
- BroadcastReceiver class 464
- BroadcastReceiver superclass 467
- BufferedReader object 682
- Build Variants , 54
 - tool window 54
- Bundle class 122
- Bundled Notifications 517
- C**
- Calendar permissions 498
- CALL_PHONE permission 498
- CAMERA permission 498
- Camera permissions 498
- CameraUpdateFactory class
 - methods 729
- CancellationSignal 668
- Canvas class 758
- CardView
 - layout file 401
 - responding to selection of 409
- CardView class 401
- CATEGORY_OPENABLE 672
- C/C++ Libraries 87
- Chain bias 182
- chain head 152
- chains 152
- Chains
 - creation of 179
- Chain style
 - changing 181
- chain styles 152
- CheckBox 127
- checkSelfPermission() method 502
- Circle class 715
- Code completion 70
- Code Editor
 - basics 67
 - Code completion 70
 - Code Generation 73
 - Code Reformatting 75
 - Document Tabs 68
 - Editing area 68
 - Gutter Area 68
 - Live Templates 76
 - Splitting 70
 - Statement Completion 72
 - Status Bar 69
- Code Generation 73
- Code Reformatting 75
- code samples
 - download 1
- cold boot 44
- CollapsingToolbarLayout
 - example 424
 - introduction 424
 - parallax mode 424
 - pin mode 424

- setting scrim color 427
- setting title 427
- with image 424
- Color class 759
- COLOR_MODE_COLOR 734, 754
- COLOR_MODE_MONOCHROME 734, 754
- Common Gestures 235
 - detection 235
- Component tree 17
- Constraint Bias 151
 - adjusting 165
- ConstraintLayout
 - advantages of 157
 - Availability 158
 - Barriers 154
 - Baseline Alignment 153
 - chain bias 182
 - chain head 152
 - chains 152
 - chain styles 152
 - Constraint Bias 151
 - Constraints 149
 - conversion to 177
 - convert to MotionLayout 355
 - deleting constraints 164
 - guidelines 171
 - Guidelines 154
 - manual constraint manipulation 161
 - Margins 150, 165
 - Opposing Constraints 150, 167
 - overview of 149
 - Packed chain 153, 182
 - ratios 157, 183
 - Spread chain 152
 - Spread inside 182
 - Spread inside chain 152
 - tutorial 187
 - using in Android Studio 159
 - Weighted chain 152, 182
 - Widget Dimensions 153, 169
 - Widget Group Alignment 175
- ConstraintLayout chains
 - creation of 179
 - in layout editor 179
- ConstraintLayout Chain style
 - changing 181
- Constraints
 - deleting 164
- ConstraintSet
 - addToHorizontalChain() method 202
 - addToVerticalChain() method 202
 - alignment constraints 201
 - apply to layout 200
 - applyTo() method 200
 - centerHorizontally() method 201
 - centerVertically() method 201
 - chains 201
 - clear() method 202
 - clone() method 201
 - connect() method 200
 - connect to parent 200
 - constraint bias 201
 - copying constraints 201
 - create 200
 - create connection 200
 - createHorizontalChain() method 201
 - createVerticalChain() method 201
 - guidelines 202
 - removeFromHorizontalChain() method 202
 - removeFromVerticalChain() method 202
 - removing constraints 202
 - rotation 203
 - scaling 202
 - setGuidelineBegin() method 202
 - setGuidelineEnd() method 202
 - setGuidelinePercent() method 202
 - setHorizontalBias() method 201
 - setRotationX() method 203
 - setRotationY() method 203
 - setScaleX() method 202
 - setScaleY() method 202
 - setTransformPivot() method 203
 - setTransformPivotX() method 203
 - setTransformPivotY() method 203

Index

- setVerticalBias() method 201
 - sizing constraints 201
 - tutorial 205
 - view IDs 207
 - ConstraintSet class 199, 200
 - Constraint Sets 200
 - ConstraintSets
 - configuring 344
 - consumeAsync() method 629
 - ConsumeParams 641
 - ConsumeResponseListener 629
 - Contacts permissions 498
 - container view 127
 - Content Provider 88, 555, 573
 - <provider> 557
 - accessing 573
 - Authority 561
 - client tutorial 573
 - ContentProvider class 555
 - Content Resolver 556
 - ContentResolver 569
 - content URI 556
 - Content URI 561, 573
 - ContentValues 563
 - delete() 556, 567
 - getType() 556
 - insert() 555, 563
 - onCreate() 555, 563
 - overview 91
 - query() 555, 564
 - tutorial 559
 - update() 556, 565
 - UriMatcher 562
 - UriMatcher class 556
 - ContentProvider class 555
 - Content Resolver 556
 - getContentResolver() 556
 - ContentResolver 569
 - getContentResolver() 556
 - content URI 556
 - Content URI 556, 561
 - ContentValues 563
 - Context class 91
 - CoordinatorLayout 128, 423
 - createPrintDocumentAdapter() method 749
 - Custom Attribute 345
 - Custom Document Printing 737, 749
 - Custom Gesture
 - recognition 241
 - Custom Print Adapter
 - implementation 751
 - Custom Print Adapters 749
 - Custom Theme
 - building 779
 - Cycle Editor 373
 - Cycle Keyframe 353
 - Cycle Keyframes
 - overview 369
- ## D
- dangerous permissions
 - list of 498
 - Data Access Object (DAO) 578
 - Data Access Objects (DAO) 582
 - Database Inspector 585, 608
 - live updates 609
 - SQL query 609
 - Database Rows 542
 - Database Schema 541
 - Database Tables 541
 - Data binding
 - binding expressions 295
 - Data Binding 273
 - binding classes 294
 - enabling 300
 - event and listener binding 296
 - key components 291
 - overview 291
 - tutorial 299
 - with LiveData 273
 - DDMS 32
 - Debugging
 - enabling on device 59
 - debug.keystore file 437, 459

- DefaultLifecycleObserver 312, 315
- deltaRelative 350
- Density-independent pixels 195
- Density Independent Pixels
 - converting to pixels 211
- Device Definition
 - custom 145
- Device File Explorer 54
- device frame 35
- Device Mirroring 65
 - enabling 65
- device pairing 63
- Digital Asset Links file 764, 437, 437
- Direct Reply Input 528
- document provider 671
- dp 195
- Dynamic Colors
 - applyToActivitiesIfAvailable() method 788
 - enabling in Android 787
- Dynamic State 107
 - saving 121

E

- Empty Process 101
- Empty template 131
- Emulator
 - battery 42
 - cellular configuration 42
 - configuring fingerprints 44
 - directional pad 42
 - extended control options 41
 - Extended controls 41
 - fingerprint 42
 - location configuration 42
 - phone settings 42
 - Resizable 47
 - resize 41
 - rotate 40
 - Screen Record 43
 - Snapshots 43
 - starting 29
 - take screenshot 40

- toolbar 39
- toolbar options 39
- tool window mode 46
- Virtual Sensors 43
- zoom 40
- enablePendingPurchases() method 629
- enabling ADB support 59
- Event Handling 221
 - example 222
- Event Listener 223
- Event Listeners 222
- Events
 - consuming 225
- execSQL() 550
- explicit
 - intent 90
- explicit intent 433
- Explicit Intent 433
- Extended Control
 - options 41

F

- Files
 - switching between 68
- findPointerIndex() method 228
- findViewById() 93
- Fingerprint
 - emulation 44
- Fingerprint authentication
 - device configuration 664
 - permission 664
 - steps to implement 663
- Fingerprint Authentication
 - overview 663
 - tutorial 663
- FLAG_INCLUDE_STOPPED_PACKAGES 463
- flexible space area 421
- floating action button 14, 132
 - changing appearance of 384
 - margins 382
 - removing 133
 - sizes 382

Index

- Foldable Devices 110
 - multi-resume 110
- Foldable Emulator 534
- Foldables 533
- Foreground Process 100
- Forward-geocoding 721
- Fragment
 - creation 251
 - event handling 255
 - XML file 252
- FragmentManager class 107
- Fragment Communication 256
- Fragments 251
 - adding in code 254
 - duplicating 390
 - example 259
 - overview 251
- FragmentManager class 393
- FrameLayout 128
- G**
- Gemini 79
 - asking questions 82
 - configuration 81
 - enabling 79
 - in Android Studio 79
 - inline code completion 83
 - overview 79
 - playground 82
 - question context 83
 - tool window 80
- Geocoder object 722
- Geocoding 720
- Gesture Builder Application 241
 - building and running 241
- Gesture Detector class 235
- GestureDetectorCompat
 - instance creation 238
- GestureDetectorCompat class 235
- GestureDetector.OnDoubleTapListener 235, 236
- GestureDetector.OnGestureListener 236
- GestureLibrary 241
- GestureOverlayView 241
 - configuring color 246
 - configuring multiple strokes 246
- GestureOverlayView class 241
- GesturePerformedListener 241
- Gestures
 - interception of 247
- Gestures File
 - creation 242
 - extract from SD card 242
 - loading into application 244
- GET_ACCOUNTS permission 498
- getAction() method 469
- getContentResolver() 556
- getDebugMessage() 644
- getFromLocation() method 722
- getId() method 200
- getIntent() method 434
- getItemId() method 647
- getPointerCount() method 228
- getPointerId() method 228
- getPurchaseState() method 629
- getService() method 475
- getWritableDatabase() 551
- GNU/Linux 86
- Google Cloud
 - billing account 716
 - new project 717
- Google Cloud Print 732
- Google Drive 672
 - printing to 732
- GoogleMap 715
 - map types 725
- GoogleMap.MAP_TYPE_HYBRID 725
- GoogleMap.MAP_TYPE_NONE 725
- GoogleMap.MAP_TYPE_NORMAL 725
- GoogleMap.MAP_TYPE_SATELLITE 725
- GoogleMap.MAP_TYPE_TERRAIN 725
- Google Maps Android API 715
 - Controlling the Map Camera 729
 - displaying controls 726
 - Map Markers 728

- overview 715
- Google Maps SDK 715
 - API Key 719
 - Credentials 719
 - enabling 718
 - Maps SDK for Android 719
- Google Play App Signing 614
- Google Play Console 635
 - Creating an in-app product 635
 - License Testers 636
- Google Play Developer Console 612
- Gradle
 - APK signing settings 796
 - Build Variants 792
 - command line tasks 797
 - dependencies 791
 - Manifest Entries 792
 - overview 791
 - sensible defaults 791
- Gradle Build File
 - top level 793
- Gradle Build Files
 - module level 794
- gradle.properties file 792
- GridLayout 128
- GridLayoutManager 399

H

- HAL 86
- Handler class 480
- Hardware Abstraction Layer 86
- HP Print Services Plugin 731
- HTML printing 735
- HTML Printing
 - example 739

I

- IBinder 429, 473
- IBinder object 471, 481
- Image Printing 734
- implicit
 - intent 90

- implicit intent 433
- Implicit Intent 435
- Implicit Intents
 - example 451
- importance hierarchy 99
- in 195
- INAPP 630
- In-App Products 625
- In-App Purchasing 633
 - acknowledgePurchase() method 629
 - BillingClient 626
 - BillingResult 644
 - consumeAsync() method 629
 - ConsumeParams 641
 - ConsumeResponseListener 629
 - Consuming purchases 641
 - enablePendingPurchases() method 629
 - getPurchaseState() method 629
 - launchBillingFlow() method 628
 - Libraries 633
 - newBuilder() method 626
 - onBillingServiceDisconnected() callback 638
 - onBillingServiceDisconnected() method 627
 - onBillingSetupFinished() listener 638
 - onProductDetailsResponse() callback 638
 - Overview 625
 - ProductDetail 628
 - ProductDetails 639
 - products 625
 - ProductType 630
 - ProductType.INAPP 630
 - ProductType.SUBS 630
 - Purchase Flow 640
 - PurchaseResponseListener 630
 - PurchasesUpdatedListener 629
 - PurchaseUpdatedListener 640
 - purchase updates 640
 - queryProductDetailsAsync() 638
 - queryProductDetailsAsync() method 628
 - queryPurchasesAsync() 642
 - queryPurchasesAsync() method 630
 - runOnUiThread() 639

Index

- subscriptions 625
- tutorial 633
- In-Memory Database 585
- Intent 90
 - explicit 90
 - implicit 90
- Intent Availability
 - checking for 440
- Intent.CATEGORY_OPENABLE 680
- Intent Filters 436
 - App Link 763
- Intents 433
 - ActivityResultLauncher 435
 - overview 433
 - registerForActivityResult() 448
- Intent Service 429
- Intent URL 453

J

- Java Native Interface 87
- Jetpack 271
 - overview 271
- JobIntentService 429
 - BIND_JOB_SERVICE permission 431
 - onHandleWork() method 429

K

- KeyAttribute 348
- Keyboard Shortcuts 56
- KeyCycle 369
 - Cycle Editor 373
 - tutorial 369
- Keyframe 362
- Keyframes 348
- KeyFrameSet 378
- KeyPosition 349
 - deltaRelative 350
 - parentRelative 349
 - pathRelative 350
- Keystore File
 - creation 614
- KeyTimeCycle 369

- keytool 437
- KeyTrigger 352
- Killed state 102

L

- Large Language Model 79
- launchBillingFlow() method 628
- layout_collapseMode
 - parallax 426
 - pin 426
- layout_constraintDimensionRatio 184
- layout_constraintHorizontal_bias 182
- layout_constraintVertical_bias 182
- layout editor
 - ConstraintLayout chains 179
- Layout Editor 16, 187
 - Autoconnect Mode 161
 - code mode 138
 - Component Tree 136
 - design mode 135
 - device screen 136
 - example project 187
 - Inference Mode 161
 - palette 136
 - properties panel 136
 - Sample Data 144
 - Setting Properties 139
 - toolbar 136
 - user interface design 187
 - view conversion 143
- Layout Editor Tool
 - changing orientation 17
 - overview 135
- Layout Inspector 55
- Layout Managers 127
- LayoutResultCallback object 754
- Layouts 127
- layout_scrollFlags
 - enterAlwaysCollapsed mode 423
 - enterAlways mode 423
 - exitUntilCollapsed mode 423
 - scroll mode 423

- Layout Validation 146
 - libc 87
 - libs.versions.toml file 218
 - License Testers 636
 - Lifecycle
 - awareness 311
 - components 274
 - owners 311
 - states and events 313
 - tutorial 315
 - Lifecycle-Aware Components 311
 - Lifecycle Methods 108
 - Lifecycle Observer 315
 - creating a 315
 - Lifecycle Owner
 - creating a 317
 - Lifecycles
 - modern 274
 - LinearLayout 128
 - LinearLayoutManager 399
 - LinearLayoutManager layout 408
 - Linux Kernel 86
 - list devices 59
 - LiveData 272, 285
 - adding to ViewModel 285
 - observer 287
 - tutorial 285
 - Live Templates 76
 - LLM 79
 - Local Bound Service 471
 - example 471
 - Location Manager 88
 - Location permission 498
 - Logcat
 - tool window 54
 - LogCat
 - enabling 117
- M**
- MANAGE_EXTERNAL_STORAGE 499
 - adb enabling 499
 - testing 499
 - Manifest File
 - permissions 455
 - Maps 715
 - MapView 715
 - adding to a layout 722
 - Marker class 715
 - Master/Detail Flow
 - creation 656
 - two pane mode 655
 - match_parent properties 195
 - Material design 381
 - Material Design 2 777
 - Material Design 2 Theming 777
 - Material Design 3 777
 - Material Theme Builder 779
 - Material You 777
 - MediaController
 - adding to VideoView instance 689
 - MediaController class 686
 - methods 686
 - MediaPlayer class 705
 - methods 705
 - MediaRecorder class 705
 - methods 706
 - recording audio 706
 - Memory Indicator 69
 - Menu Editor 648
 - Menu Item Selections 646
 - Menus 645
 - menu editor 648
 - Messenger object 481
 - Microphone
 - checking for availability 708
 - Microphone permissions 498
 - mm 195
 - MotionEvent 227, 228, 249
 - getActionMasked() 228
 - MotionLayout 343
 - arc motion 348
 - Attribute Keyframes 348
 - ConstraintSets 344
 - Custom Attribute 364

Index

- Custom Attributes 345
- Cycle Editor 373
- Editor 355
- KeyAttribute 348
- KeyCycle 369
- Keyframes 348
- KeyFrameSet 378
- KeyPosition 349
- KeyTimeCycle 369
- KeyTrigger 352
- OnClick 347, 360
- OnSwipe 347
- overview 343
- Position Keyframes 349
- previewing animation 360
- Trigger Keyframe 352
- Tutorial 355
- MotionScene
 - ConstraintSets 344
 - Custom Attributes 345
 - file 344
 - overview 343
 - transition 344
- moveCamera() method 729
- multiple devices
 - testing app on 31
- Multiple Touches
 - handling 228
- multi-resume 110
- Multi-Touch
 - example 228
- Multi-touch Event Handling 227
- Multi-Window
 - attributes 537
- Multi-Window Mode
 - detecting 538
 - entering 535
 - launching activity into 539
- Multi-Window Notifications 538
- multi-window support 110
- Multi-Window Support
 - enabling 536

- My Location Layer 715

N

- Navigation 321
 - adding destinations 330
 - overview 321
 - pass data with safeargs 338
 - passing arguments 326
 - stack 321
 - tutorial 327
- Navigation Action
 - triggering 325
- Navigation Architecture Component 321
- Navigation Component
 - tutorial 327
- Navigation Controller
 - accessing 325
- Navigation Graph 324, 328
 - adding actions 334
 - creating a 328
- Navigation Host 322
 - declaring 329
- newBuilder() method 626
- normal permissions 497
- Notification
 - adding actions 516
 - Direct Reply Input 528
 - issuing a basic 512
 - launch activity from a 514
 - PendingIntent 524
 - Reply Action 526
 - updating direct reply 529
- Notifications
 - bundled 517
 - overview 505
- Notifications Manager 88
- O**
- Observer
 - implementing a LiveData 287
- onAttach() method 256
- onBillingServiceDisconnected() callback 638

- onBillingServiceDisconnected() method 627
 - onBillingSetupFinished() listener 638
 - onBind() method 430, 471, 479
 - onBindViewHolder() method 407
 - OnClick 347
 - onClickListener 222, 223, 226
 - onClick() method 221
 - onCreateContextMenuListener 222
 - onCreate() method 100, 108, 430
 - onCreateOptionsMenu() method 646
 - onCreateView() method 109
 - onDestroy() method 108, 430
 - onDoubleTap() method 235
 - onDown() method 235
 - onFling() method 235
 - onFocusChangeListener 222
 - OnFragmentInteractionListener
 - implementation 335
 - onGesturePerformed() method 241
 - onHandleWork() method 430
 - onKeyListener 222
 - onLayoutFailed() method 754
 - onLayoutFinished() method 755
 - onLongClickListener 222
 - onLongClick() method 225
 - onLongPress() method 235
 - onMapReady() method 724
 - onOptionsItemSelected() method 646
 - onOptionsItemSelected() method 651
 - onPageFinished() callback 740
 - onPause() method 108
 - onProductDetailsResponse() callback 638
 - onReceive() method 100, 464, 465, 467
 - onRequestPermissionsResult() method 501, 712, 510, 522
 - onRestart() method 108
 - onRestoreInstanceState() method 109
 - onResume() method 100, 108
 - onSaveInstanceState() method 109
 - onScaleBegin() method 247
 - onScaleEnd() method 247
 - onScale() method 247
 - onScroll() method 235
 - OnSeekBarChangeListener 266
 - onServiceConnected() method 471, 475, 482
 - onServiceDisconnected() method 471, 475, 482
 - onShowPress() method 235
 - onSingleTapUp() method 235
 - onStartCommand() method 430
 - onStart() method 108
 - onStop() method 108
 - onTouchEvent() method 235, 247
 - onTouchListener 222
 - onTouch() method 227
 - onUpgrade() 550
 - onViewCreated() method 109
 - onViewStatusRestored() method 109
 - openFileDescriptor() method 672
 - OpenJDK 3
 - Overflow Menu 645
 - creation 645
 - displaying 646
 - overview 645
 - XML file 645
 - Overflow Menus
 - Checkable Item Groups 647
- ## P
- Package Explorer 15
 - Package Manager 88
 - PackageManager class 708
 - PackageManager.FEATURE_MICROPHONE 708
 - PackageManager.PERMISSION_DENIED 499
 - PackageManager.PERMISSION_GRANTED 499
 - Package Name 14
 - Packed chain 153, 182
 - PageRange 756, 757
 - Paint class 759
 - parentRelative 349
 - parent view 129
 - pathRelative 350
 - Paused state 102
 - PdfDocument 737
 - PdfDocument.Page 749, 756
 - PendingIntent class 524

Index

- Permission
 - checking for 499
- permissions
 - normal 497
- Persistent State 107
- Phone permissions 498
- picker 671
- Pinch Gesture
 - detection 247
 - example 247
- Pinch Gesture Recognition 241
- Position Keyframes 349
- POST_NOTIFICATIONS permission 498, 522
- PrintAttributes 754
- PrintDocumentAdapter 737, 749
- Printing
 - color 734
 - monochrome 734
- Printing framework
 - architecture 731
- Printing Framework 731
- Print Job
 - starting 760
- PrintManager service 741
- Problems
 - tool window 54, 55
- process
 - priority 99
 - state 99
- PROCESS_OUTGOING_CALLS permission 498
- Process States 99
- ProductDetail 628
- ProductDetails 639
- ProductType 630
- Profiler
 - tool window 55
- ProgressBar 127
- proguard-rules.pro file 796
- ProGuard Support 792
- Project Name 14
- Project tool window 15, 53
- pt 195

- PurchaseResponseListener 630
- PurchasesUpdatedListener 629
- PurchaseUpdatedListener 640
- putExtra() method 433, 463
- px 196

Q

- queryProductDetailsAsync() 638
- queryPurchasesAsync() 642
- quickboot snapshot 44
- Quick Documentation 75

R

- RadioButton 127
- ratios 183
- READ_CALENDAR permission 498
- READ_CALL_LOG permission 498
- READ_CONTACTS permission 498
- READ_EXTERNAL_STORAGE permission 499
- READ_PHONE_STATE permission 498
- READ_SMS permission 498
- RECEIVE_MMS permission 498
- RECEIVE_SMS permission 498
- RECEIVE_WAP_PUSH permission 498
- Recent Files Navigation 56
- RECORD_AUDIO permission 498
- Recording Audio
 - permission 707
- RecyclerView 399
 - adding to layout file 400
 - LayoutManager 399
 - initializing 408
 - LayoutManager 399
 - StaggeredLayoutManager 399
- RecyclerView Adapter
 - creation of 406
- RecyclerView.Adapter 400, 406
 - getItemCount() method 400
 - onBindViewHolder() method 400
 - onCreateViewHolder() method 400
- RecyclerView.ViewHolder
 - getAdapterPosition() method 410

- registerForActivityResult() method 434, 448
- registerReceiver() method 465
- RelativeLayout 128
- releasePersistableUriPermission() method 675
- Release Preparation 611
- Remote Bound Service 479
 - client communication 479
 - implementation 479
 - manifest file declaration 481
- RemoteInput.Builder() method 524
- RemoteInput Object 524
- Remote Service
 - launching and binding 482
 - sending a message 483
- Repository
 - tutorial 595
- Repository Modules 274
- Resizable Emulator 47
- Resource
 - string creation 20
- Resource File 22
- Resource Management 99
- Resource Manager 53, 88
- result receiver 465
- Reverse-geocoding 721
- Reverse Geocoding 720
- Room
 - Data Access Object (DAO) 578
 - entities 578, 579
 - In-Memory Database 585
 - Repository 578
- Room Database 578
 - tutorial 595
- Room Database Persistence 577
- Room Persistence Library 546, 577
- root element 127
- root view 129
- Run
 - tool window 53
- Running Devices
 - tool window 65
- runOnUiThread() 639

S

- safeargs 338
- Sample Data 144, 413
 - tutorial 413
- Saved State 273, 307
- SavedStateHandle 308, 309
 - contains() method 309
 - keys() method 309
 - remove() method 309
- Saved State module 307
- SavedStateViewModelFactory 308
- ScaleGestureDetector class 247
- Scale-independent 195
- SDK Packages 5
- Secure Sockets Layer (SSL) 87
- SeekBar 259
- sendBroadcast() method 463, 465
- sendOrderedBroadcast() method 463, 465
- SEND_SMS permission 498
- sendStickyBroadcast() method 463
- Sensor permissions 498
- Service
 - anatomy 430
 - launch at system start 431
 - manifest file entry 430
 - overview 90
 - run in separate process 431
- ServiceConnection class 482
- Service Process 100
- Service Restart Options 430
- setAudioEncoder() method 706
- setAudioSource() method 706
- setBackground-color() 200
- setCompassEnabled() method 726
- setContentView() method 199, 205
- setId() method 200
- setMyLocationButtonEnabled() method 727
- setOnClickListener() method 221, 223
- setOnDoubleTapListener() method 235, 238
- setOutputFile() method 706
- setOutputFormat() method 706
- setResult() method 435

Index

- setRotateGesturesEnabled() method 727
- setScrollGesturesEnabled() method 727
- setText() method 124
- setTiltGesturesEnabled() method 727
- settings.gradle file 792
- settings.gradle.kts file 792
- setTransition() 353
- setVideoSource() method 706
- setZoomControlsEnabled() method 726, 727
- SHA-256 certificate fingerprint 437
- shouldOverrideUrlLoading() method 740
- SimpleOnScaleGestureListener 247
- SimpleOnScaleGestureListener class 249
- SMS permissions 498
- Snackbar 381, 382, 383
- Snapshots
 - emulator 43
- sp 195
- Spread chain 152
- Spread inside 182
- Spread inside chain 152
- SQL 542
- SQL CREATE 550
- SQLite 541
 - AVD command-line use 543
 - Columns and Data Types 541
 - overview 542
 - Primary keys 542
 - tutorial 547
- SQLiteDatabase 550
- SQLiteOpenHelper 548, 549
- SQL SELECT 551, 552
- StaggeredGridLayoutManager 399
- startActivity() method 433
- startForeground() method 100
- START_NOT_STICKY 430
- START_REDELIVER_INTENT 430
- START_STICKY 430
- State
 - restoring 124
- State Change
 - handling 103

- Statement Completion 72
- Status Bar Widgets 69
 - Memory Indicator 69
- Sticky Broadcast Intents 465
- Stopped state 102
- Storage Access Framework 671
 - ACTION_CREATE_DOCUMENT 672
 - ACTION_OPEN_DOCUMENT 672
 - deleting a file 675
 - example 677
 - file creation 679
 - file filtering 672
 - file reading 673
 - file writing 674
 - intents 672
 - MIME Types 673
 - Persistent Access 675
 - picker 671
- Storage permissions 499
- StringBuilder object 682
- strings.xml file 24
- Structure
 - tool window 55
- Structured Query Language 542
- Structure tool window 55
- SUBS 630
- subscriptions 625
- SupportMapFragment class 715
- Switcher 56
- System Broadcasts 469
- system requirements 3

T

- TabLayout
 - adding to layout 391
 - app
 - tabGravity property 396
 - tabMode property 396
 - example 388
 - fixed mode 395
 - getItemCount() method 387
 - overview 387

- TableLayout 128, 587
 - TableRow 587
 - Telephony Manager 88
 - Templates
 - blank vs. empty 131
 - Terminal
 - tool window 54
 - Theme
 - building a custom 779
 - Theme Builder 779
 - Theming 777
 - tutorial 783
 - Time Cycle Keyframes 353
 - TODO
 - tool window 55
 - ToolBarListener 256
 - tools
 - layout 253
 - Tool window bars 52
 - Tool windows 52
 - Touch Actions 228
 - Touch Event Listener
 - implementation 229
 - Touch Events
 - intercepting 227
 - Touch handling 227
- U**
- UiSettings class 715
 - unbindService() method 429
 - unregisterReceiver() method 465
 - upload key 614
 - UriMatcher 556, 562
 - UriMatcher class 556
 - URL Mapping 769
 - USB connection issues
 - resolving 62
 - USE_BIOMETRIC 664
 - user interface state 107
 - USE_SIP permission 498
- V**
- Version catalog 217
 - dependencies 219
 - libraries 218
 - libs.versions.toml file 218
 - plugins 218
 - versions 218
 - Video Playback 685
 - VideoView class 685
 - methods 685
 - supported formats 685
 - view bindings
 - enabling 94
 - using 94
 - View class
 - setting properties 207
 - view conversion 143
 - ViewGroup 127
 - View Groups 127
 - View Hierarchy 129
 - ViewHolder class 400
 - sample implementation 407
 - ViewModel
 - adding LiveData 285
 - data access 282
 - overview 272
 - saved state 307
 - Saved State 273, 307
 - tutorial 277
 - ViewModelProvider 280
 - ViewModel Saved State 307
 - ViewPager
 - adding to layout 391
 - example 388
 - Views 127
 - Java creation 199
 - View System 88
 - Virtual Device Configuration dialog 28
 - Virtual Sensors 43
 - Visible Process 100
- W**
- WebViewClient 735, 740

Index

WebView view 453
Weighted chain 152, 182
Welcome screen 49
Widget Dimensions 153
Widget Group Alignment 175
Widgets palette 188
WiFi debugging 63
Wireless debugging 63
Wireless pairing 63
wrap_content properties 197
WRITE_CALENDAR permission 498
WRITE_CALL_LOG permission 498
WRITE_CONTACTS permission 498
WRITE_EXTERNAL_STORAGE permission 499

X

XML Layout File
 manual creation 195
 vs. Java Code 199