

Android Studio Koala Essentials



Kotlin Edition

Android Studio Koala Essentials

Kotlin Edition

Android Studio Koala Essentials – Kotlin Edition

ISBN: 978-1-951442-94-1

© 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.....	72
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. An Overview of the Android Architecture.....	79
9.1 The Android Software Stack.....	79
9.2 The Linux Kernel.....	80
9.3 Hardware Abstraction Layer.....	80
9.4 Android Runtime – ART.....	80
9.5 Android Libraries.....	80
9.5.1 C/C++ Libraries.....	81
9.6 Application Framework.....	82
9.7 Applications.....	82
9.8 Summary.....	82
10. The Anatomy of an Android App.....	83
10.1 Android Activities.....	83
10.2 Android Fragments.....	83
10.3 Android Intents.....	84
10.4 Broadcast Intents.....	84
10.5 Broadcast Receivers.....	84
10.6 Android Services.....	84
10.7 Content Providers.....	85
10.8 The Application Manifest.....	85
10.9 Application Resources.....	85
10.10 Application Context.....	85
10.11 Summary.....	85
11. An Introduction to Kotlin.....	87
11.1 What is Kotlin?.....	87
11.2 Kotlin and Java.....	87
11.3 Converting from Java to Kotlin.....	87
11.4 Kotlin and Android Studio.....	88
11.5 Experimenting with Kotlin.....	88
11.6 Semi-colons in Kotlin.....	89
11.7 Summary.....	89
12. Kotlin Data Types, Variables, and Nullability.....	91
12.1 Kotlin Data Types.....	91
12.1.1 Integer Data Types.....	92
12.1.2 Floating-Point Data Types.....	92

Table of Contents

12.1.3 Boolean Data Type.....	92
12.1.4 Character Data Type.....	92
12.1.5 String Data Type.....	92
12.1.6 Escape Sequences	93
12.2 Mutable Variables.....	94
12.3 Immutable Variables	94
12.4 Declaring Mutable and Immutable Variables.....	94
12.5 Data Types are Objects	94
12.6 Type Annotations and Type Inference	95
12.7 Nullable Type.....	96
12.8 The Safe Call Operator	96
12.9 Not-Null Assertion.....	97
12.10 Nullable Types and the let Function.....	97
12.11 Late Initialization (lateinit)	98
12.12 The Elvis Operator	99
12.13 Type Casting and Type Checking	99
12.14 Summary.....	100
13. Kotlin Operators and Expressions	101
13.1 Expression Syntax in Kotlin.....	101
13.2 The Basic Assignment Operator.....	101
13.3 Kotlin Arithmetic Operators	101
13.4 Augmented Assignment Operators	102
13.5 Increment and Decrement Operators	102
13.6 Equality Operators	103
13.7 Boolean Logical Operators	103
13.8 Range Operator	104
13.9 Bitwise Operators.....	104
13.9.1 Bitwise Inversion.....	104
13.9.2 Bitwise AND.....	105
13.9.3 Bitwise OR.....	105
13.9.4 Bitwise XOR.....	105
13.9.5 Bitwise Left Shift.....	106
13.9.6 Bitwise Right Shift.....	106
13.10 Summary.....	107
14. Kotlin Control Flow	109
14.1 Looping Control flow	109
14.1.1 The Kotlin <i>for-in</i> Statement.....	109
14.1.2 The <i>while</i> Loop	110
14.1.3 The <i>do ... while</i> loop	111
14.1.4 Breaking from Loops.....	111
14.1.5 The <i>continue</i> Statement	112
14.1.6 Break and Continue Labels.....	112
14.2 Conditional Control Flow.....	113
14.2.1 Using the <i>if</i> Expressions	113
14.2.2 Using <i>if ... else ...</i> Expressions	114
14.2.3 Using <i>if ... else if ...</i> Expressions	114
14.2.4 Using the <i>when</i> Statement.....	114
14.3 Summary	115

15. An Overview of Kotlin Functions and Lambdas	117
15.1 What is a Function?	117
15.2 How to Declare a Kotlin Function	117
15.3 Calling a Kotlin Function.....	118
15.4 Single Expression Functions.....	118
15.5 Local Functions	118
15.6 Handling Return Values	119
15.7 Declaring Default Function Parameters.....	119
15.8 Variable Number of Function Parameters	119
15.9 Lambda Expressions	120
15.10 Higher-order Functions	121
15.11 Summary.....	122
16. The Basics of Object Oriented Programming in Kotlin	123
16.1 What is an Object?	123
16.2 What is a Class?	123
16.3 Declaring a Kotlin Class.....	123
16.4 Adding Properties to a Class.....	124
16.5 Defining Methods	124
16.6 Declaring and Initializing a Class Instance.....	124
16.7 Primary and Secondary Constructors.....	124
16.8 Initializer Blocks.....	127
16.9 Calling Methods and Accessing Properties	127
16.10 Custom Accessors	127
16.11 Nested and Inner Classes	128
16.12 Companion Objects.....	129
16.13 Summary.....	131
17. An Introduction to Kotlin Inheritance and Subclassing.....	133
17.1 Inheritance, Classes and Subclasses.....	133
17.2 Subclassing Syntax	133
17.3 A Kotlin Inheritance Example.....	134
17.4 Extending the Functionality of a Subclass	135
17.5 Overriding Inherited Methods.....	136
17.6 Adding a Custom Secondary Constructor.....	137
17.7 Using the SavingsAccount Class	137
17.8 Summary	137
18. An Overview of Android View Binding.....	139
18.1 Find View by Id	139
18.2 View Binding	139
18.3 Converting the AndroidSample project.....	140
18.4 Enabling View Binding.....	140
18.5 Using View Binding	140
18.6 Choosing an Option	142
18.7 View Binding in the Book Examples	142
18.8 Migrating a Project to View Binding.....	142
18.9 Summary	143
19. Introducing Gemini in Android Studio.....	145
19.1 Introducing Gemini AI	145

Table of Contents

19.2 Enabling Gemini in Android Studio	145
19.3 Gemini configuration	147
19.4 Asking Gemini questions	148
19.5 Question contexts.....	149
19.6 Inline code completion.....	149
19.7 Summary	150
20. Understanding Android Application and Activity Lifecycles	151
20.1 Android Applications and Resource Management.....	151
20.2 Android Process States	151
20.2.1 Foreground Process	152
20.2.2 Visible Process	152
20.2.3 Service Process	152
20.2.4 Background Process.....	152
20.2.5 Empty Process	153
20.3 Inter-Process Dependencies	153
20.4 The Activity Lifecycle.....	153
20.5 The Activity Stack.....	153
20.6 Activity States	154
20.7 Configuration Changes	154
20.8 Handling State Change.....	155
20.9 Summary	155
21. Handling Android Activity State Changes.....	157
21.1 New vs. Old Lifecycle Techniques.....	157
21.2 The Activity and Fragment Classes.....	157
21.3 Dynamic State vs. Persistent State.....	159
21.4 The Android Lifecycle Methods	159
21.5 Lifetimes	161
21.6 Foldable Devices and Multi-Resume	162
21.7 Disabling Configuration Change Restarts	162
21.8 Lifecycle Method Limitations.....	162
21.9 Summary	163
22. Android Activity State Changes by Example	165
22.1 Creating the State Change Example Project	165
22.2 Designing the User Interface	166
22.3 Overriding the Activity Lifecycle Methods	167
22.4 Filtering the Logcat Panel.....	169
22.5 Running the Application.....	170
22.6 Experimenting with the Activity.....	171
22.7 Summary	172
23. Saving and Restoring the State of an Android Activity	173
23.1 Saving Dynamic State	173
23.2 Default Saving of User Interface State	173
23.3 The Bundle Class	174
23.4 Saving the State.....	175
23.5 Restoring the State	176
23.6 Testing the Application.....	176
23.7 Summary	176

24. Understanding Android Views, View Groups and Layouts	177
24.1 Designing for Different Android Devices	177
24.2 Views and View Groups	177
24.3 Android Layout Managers	177
24.4 The View Hierarchy	179
24.5 Creating User Interfaces	180
24.6 Summary	180
25. A Guide to the Android Studio Layout Editor	181
25.1 Basic vs. Empty Views Activity Templates	181
25.2 The Android Studio Layout Editor	185
25.3 Design Mode.....	185
25.4 The Palette	186
25.5 Design Mode and Layout Views.....	187
25.6 Night Mode	188
25.7 Code Mode.....	188
25.8 Split Mode	189
25.9 Setting Attributes.....	189
25.10 Transforms	191
25.11 Tools Visibility Toggles.....	192
25.12 Converting Views.....	193
25.13 Displaying Sample Data	194
25.14 Creating a Custom Device Definition	195
25.15 Changing the Current Device.....	195
25.16 Layout Validation	196
25.17 Summary.....	197
26. A Guide to the Android ConstraintLayout.....	199
26.1 How ConstraintLayout Works.....	199
26.1.1 Constraints.....	199
26.1.2 Margins	200
26.1.3 Opposing Constraints.....	200
26.1.4 Constraint Bias	201
26.1.5 Chains	202
26.1.6 Chain Styles.....	202
26.2 Baseline Alignment.....	203
26.3 Configuring Widget Dimensions.....	203
26.4 Guideline Helper	204
26.5 Group Helper.....	204
26.6 Barrier Helper.....	204
26.7 Flow Helper.....	206
26.8 Ratios	207
26.9 ConstraintLayout Advantages	207
26.10 ConstraintLayout Availability.....	208
26.11 Summary.....	208
27. A Guide to Using ConstraintLayout in Android Studio	209
27.1 Design and Layout Views.....	209
27.2 Autoconnect Mode	211
27.3 Inference Mode.....	211

Table of Contents

27.4 Manipulating Constraints Manually.....	211
27.5 Adding Constraints in the Inspector	213
27.6 Viewing Constraints in the Attributes Window.....	213
27.7 Deleting Constraints.....	214
27.8 Adjusting Constraint Bias	215
27.9 Understanding ConstraintLayout Margins.....	215
27.10 The Importance of Opposing Constraints and Bias	217
27.11 Configuring Widget Dimensions.....	219
27.12 Design Time Tools Positioning	220
27.13 Adding Guidelines	221
27.14 Adding Barriers	223
27.15 Adding a Group.....	224
27.16 Working with the Flow Helper	225
27.17 Widget Group Alignment and Distribution.....	225
27.18 Converting other Layouts to ConstraintLayout	227
27.19 Summary	227
28. Working with ConstraintLayout Chains and Ratios in Android Studio	229
28.1 Creating a Chain.....	229
28.2 Changing the Chain Style	231
28.3 Spread Inside Chain Style.....	232
28.4 Packed Chain Style.....	232
28.5 Packed Chain Style with Bias.....	232
28.6 Weighted Chain.....	232
28.7 Working with Ratios	233
28.8 Summary	235
29. An Android Studio Layout Editor ConstraintLayout Tutorial	237
29.1 An Android Studio Layout Editor Tool Example	237
29.2 Preparing the Layout Editor Environment	237
29.3 Adding the Widgets to the User Interface.....	238
29.4 Adding the Constraints	241
29.5 Testing the Layout	243
29.6 Using the Layout Inspector	243
29.7 Summary	244
30. Manual XML Layout Design in Android Studio	245
30.1 Manually Creating an XML Layout	245
30.2 Manual XML vs. Visual Layout Design.....	248
30.3 Summary	248
31. Managing Constraints using Constraint Sets.....	249
31.1 Kotlin Code vs. XML Layout Files.....	249
31.2 Creating Views.....	249
31.3 View Attributes.....	250
31.4 Constraint Sets.....	250
31.4.1 Establishing Connections.....	250
31.4.2 Applying Constraints to a Layout	250
31.4.3 Parent Constraint Connections.....	250
31.4.4 Sizing Constraints	251
31.4.5 Constraint Bias	251

31.4.6 Alignment Constraints	251
31.4.7 Copying and Applying Constraint Sets	251
31.4.8 ConstraintLayout Chains	251
31.4.9 Guidelines	252
31.4.10 Removing Constraints	252
31.4.11 Scaling	252
31.4.12 Rotation	253
31.5 Summary	253
32. An Android ConstraintSet Tutorial.....	255
32.1 Creating the Example Project in Android Studio	255
32.2 Adding Views to an Activity	255
32.3 Setting View Attributes.....	256
32.4 Creating View IDs.....	257
32.5 Configuring the Constraint Set	258
32.6 Adding the EditText View	259
32.7 Converting Density Independent Pixels (dp) to Pixels (px).....	260
32.8 Summary	261
33. A Guide to Using Apply Changes in Android Studio	263
33.1 Introducing Apply Changes	263
33.2 Understanding Apply Changes Options	263
33.3 Using Apply Changes.....	264
33.4 Configuring Apply Changes Fallback Settings	265
33.5 An Apply Changes Tutorial.....	265
33.6 Using Apply Code Changes	265
33.7 Using Apply Changes and Restart Activity.....	266
33.8 Using Run App	266
33.9 Summary	266
34. A Guide to Gradle Version Catalogs	267
34.1 Library and Plugin Dependencies.....	267
34.2 Project Gradle Build File	267
34.3 Module Gradle Build Files	267
34.4 Version Catalog File.....	268
34.5 Adding Dependencies	269
34.6 Library Updates.....	270
34.7 Summary	270
35. An Overview and Example of Android Event Handling	271
35.1 Understanding Android Events.....	271
35.2 Using the android:onClick Resource	271
35.3 Event Listeners and Callback Methods	272
35.4 An Event Handling Example	272
35.5 Designing the User Interface	273
35.6 The Event Listener and Callback Method	273
35.7 Consuming Events	275
35.8 Summary	276
36. Android Touch and Multi-touch Event Handling	277
36.1 Intercepting Touch Events	277

Table of Contents

36.2 The MotionEvent Object	278
36.3 Understanding Touch Actions.....	278
36.4 Handling Multiple Touches	278
36.5 An Example Multi-Touch Application	279
36.6 Designing the Activity User Interface	279
36.7 Implementing the Touch Event Listener.....	279
36.8 Running the Example Application.....	282
36.9 Summary	282
37. Detecting Common Gestures Using the Android Gesture Detector Class	283
37.1 Implementing Common Gesture Detection.....	283
37.2 Creating an Example Gesture Detection Project	284
37.3 Implementing the Listener Class.....	284
37.4 Creating the GestureDetector Instance.....	286
37.5 Implementing the onTouchEvent() Method.....	286
37.6 Testing the Application.....	287
37.7 Summary	287
38. Implementing Custom Gesture and Pinch Recognition on Android	289
38.1 The Android Gesture Builder Application.....	289
38.2 The GestureOverlayView Class	289
38.3 Detecting Gestures.....	289
38.4 Identifying Specific Gestures	289
38.5 Installing and Running the Gesture Builder Application	289
38.6 Creating a Gestures File	290
38.7 Creating the Example Project.....	290
38.8 Extracting the Gestures File from the SD Card	290
38.9 Adding the Gestures File to the Project	291
38.10 Designing the User Interface	291
38.11 Loading the Gestures File	292
38.12 Registering the Event Listener.....	293
38.13 Implementing the onGesturePerformed Method.....	293
38.14 Testing the Application.....	294
38.15 Configuring the GestureOverlayView.....	294
38.16 Intercepting Gestures.....	294
38.17 Detecting Pinch Gestures.....	295
38.18 A Pinch Gesture Example Project.....	295
38.19 Summary	297
39. An Introduction to Android Fragments.....	299
39.1 What is a Fragment?	299
39.2 Creating a Fragment	299
39.3 Adding a Fragment to an Activity using the Layout XML File.....	300
39.4 Adding and Managing Fragments in Code	302
39.5 Handling Fragment Events	303
39.6 Implementing Fragment Communication.....	303
39.7 Summary	305
40. Using Fragments in Android Studio - An Example.....	307
40.1 About the Example Fragment Application	307
40.2 Creating the Example Project.....	307

40.3 Creating the First Fragment Layout.....	307
40.4 Migrating a Fragment to View Binding	309
40.5 Adding the Second Fragment	310
40.6 Adding the Fragments to the Activity	311
40.7 Making the Toolbar Fragment Talk to the Activity	312
40.8 Making the Activity Talk to the Text Fragment	315
40.9 Testing the Application.....	316
40.10 Summary	316
41. Modern Android App Architecture with Jetpack	317
41.1 What is Android Jetpack?	317
41.2 The “Old” Architecture.....	317
41.3 Modern Android Architecture	317
41.4 The ViewModel Component	318
41.5 The LiveData Component.....	318
41.6 ViewModel Saved State.....	319
41.7 LiveData and Data Binding.....	319
41.8 Android Lifecycles	320
41.9 Repository Modules.....	320
41.10 Summary	321
42. An Android ViewModel Tutorial.....	323
42.1 About the Project	323
42.2 Creating the ViewModel Example Project.....	323
42.3 Removing Unwanted Project Elements.....	323
42.4 Designing the Fragment Layout.....	324
42.5 Implementing the View Model.....	325
42.6 Associating the Fragment with the View Model.....	326
42.7 Modifying the Fragment	327
42.8 Accessing the ViewModel Data	327
42.9 Testing the Project.....	328
42.10 Summary	328
43. An Android Jetpack LiveData Tutorial.....	329
43.1 LiveData - A Recap	329
43.2 Adding LiveData to the ViewModel.....	329
43.3 Implementing the Observer.....	331
43.4 Summary	332
44. An Overview of Android Jetpack Data Binding	333
44.1 An Overview of Data Binding.....	333
44.2 The Key Components of Data Binding	333
44.2.1 The Project Build Configuration.....	333
44.2.2 The Data Binding Layout File.....	334
44.2.3 The Layout File Data Element	335
44.2.4 The Binding Classes	336
44.2.5 Data Binding Variable Configuration.....	336
44.2.6 Binding Expressions (One-Way).....	337
44.2.7 Binding Expressions (Two-Way).....	338
44.2.8 Event and Listener Bindings.....	338
44.3 Summary	339

45. An Android Jetpack Data Binding Tutorial.....	341
45.1 Removing the Redundant Code.....	341
45.2 Enabling Data Binding.....	342
45.3 Adding the Layout Element.....	343
45.4 Adding the Data Element to Layout File.....	344
45.5 Working with the Binding Class.....	344
45.6 Assigning the ViewModel Instance to the Data Binding Variable.....	345
45.7 Adding Binding Expressions.....	346
45.8 Adding the Conversion Method.....	346
45.9 Adding a Listener Binding.....	347
45.10 Testing the App.....	347
45.11 Summary.....	347
46. An Android ViewModel Saved State Tutorial.....	349
46.1 Understanding ViewModel State Saving.....	349
46.2 Implementing ViewModel State Saving.....	349
46.3 Saving and Restoring State.....	350
46.4 Adding Saved State Support to the ViewModelDemo Project.....	351
46.5 Summary.....	352
47. Working with Android Lifecycle-Aware Components.....	353
47.1 Lifecycle Awareness.....	353
47.2 Lifecycle Owners.....	353
47.3 Lifecycle Observers.....	354
47.4 Lifecycle States and Events.....	354
47.5 Summary.....	355
48. An Android Jetpack Lifecycle Awareness Tutorial.....	357
48.1 Creating the Example Lifecycle Project.....	357
48.2 Creating a Lifecycle Observer.....	357
48.3 Adding the Observer.....	358
48.4 Testing the Observer.....	359
48.5 Creating a Lifecycle Owner.....	359
48.6 Testing the Custom Lifecycle Owner.....	361
48.7 Summary.....	361
49. An Overview of the Navigation Architecture Component.....	363
49.1 Understanding Navigation.....	363
49.2 Declaring a Navigation Host.....	364
49.3 The Navigation Graph.....	366
49.4 Accessing the Navigation Controller.....	367
49.5 Triggering a Navigation Action.....	367
49.6 Passing Arguments.....	368
49.7 Summary.....	368
50. An Android Jetpack Navigation Component Tutorial.....	369
50.1 Creating the NavigationDemo Project.....	369
50.2 Adding Navigation to the Build Configuration.....	369
50.3 Creating the Navigation Graph Resource File.....	370
50.4 Declaring a Navigation Host.....	371
50.5 Adding Navigation Destinations.....	372

50.6 Designing the Destination Fragment Layouts.....	374
50.7 Adding an Action to the Navigation Graph.....	376
50.8 Implement the OnFragmentManagerInteractionListener	377
50.9 Adding View Binding Support to the Destination Fragments	378
50.10 Triggering the Action	379
50.11 Passing Data Using Safeargs	379
50.12 Summary	382
51. An Introduction to MotionLayout.....	383
51.1 An Overview of MotionLayout	383
51.2 MotionLayout	383
51.3 MotionScene	383
51.4 Configuring ConstraintSets	384
51.5 Custom Attributes	385
51.6 Triggering an Animation.....	387
51.7 Arc Motion.....	388
51.8 Keyframes.....	388
51.8.1 Attribute Keyframes.....	388
51.8.2 Position Keyframes	389
51.9 Time Linearity	392
51.10 KeyTrigger.....	392
51.11 Cycle and Time Cycle Keyframes	393
51.12 Starting an Animation from Code.....	393
51.13 Summary	394
52. An Android MotionLayout Editor Tutorial.....	395
52.1 Creating the MotionLayoutDemo Project	395
52.2 ConstraintLayout to MotionLayout Conversion	395
52.3 Configuring Start and End Constraints	397
52.4 Previewing the MotionLayout Animation	400
52.5 Adding an OnClick Gesture	400
52.6 Adding an Attribute Keyframe to the Transition.....	402
52.7 Adding a CustomAttribute to a Transition.....	404
52.8 Adding Position Keyframes	406
52.9 Summary	408
53. A MotionLayout KeyCycle Tutorial	409
53.1 An Overview of Cycle Keyframes	409
53.2 Using the Cycle Editor.....	413
53.3 Creating the KeyCycleDemo Project.....	414
53.4 Configuring the Start and End Constraints.....	414
53.5 Creating the Cycles	416
53.6 Previewing the Animation	418
53.7 Adding the KeyFrameSet to the MotionScene	418
53.8 Summary	420
54. Working with the Floating Action Button and Snackbar	421
54.1 The Material Design.....	421
54.2 The Design Library	421
54.3 The Floating Action Button (FAB)	421
54.4 The Snackbar.....	422

Table of Contents

54.5 Creating the Example Project	423
54.6 Reviewing the Project	423
54.7 Removing Navigation Features.....	424
54.8 Changing the Floating Action Button	424
54.9 Adding an Action to the Snackbar	426
54.10 Summary	426
55. Creating a Tabbed Interface using the TabLayout Component	427
55.1 An Introduction to the ViewPager2	427
55.2 An Overview of the TabLayout Component	427
55.3 Creating the TabLayoutDemo Project.....	428
55.4 Creating the First Fragment.....	428
55.5 Duplicating the Fragments.....	430
55.6 Adding the TabLayout and ViewPager2.....	431
55.7 Creating the Pager Adapter.....	432
55.8 Performing the Initialization Tasks.....	433
55.9 Testing the Application.....	435
55.10 Customizing the TabLayout.....	435
55.11 Summary	436
56. Working with the RecyclerView and CardView Widgets	437
56.1 An Overview of the RecyclerView	437
56.2 An Overview of the CardView	439
56.3 Summary	440
57. An Android RecyclerView and CardView Tutorial	441
57.1 Creating the CardDemo Project.....	441
57.2 Modifying the Basic Views Activity Project	441
57.3 Designing the CardView Layout	442
57.4 Adding the RecyclerView.....	443
57.5 Adding the Image Files.....	443
57.6 Creating the RecyclerView Adapter.....	443
57.7 Initializing the RecyclerView Component.....	445
57.8 Testing the Application.....	446
57.9 Responding to Card Selections.....	447
57.10 Summary	448
58. Working with the AppBar and Collapsing Toolbar Layouts	449
58.1 The Anatomy of an AppBar	449
58.2 The Example Project	450
58.3 Coordinating the RecyclerView and Toolbar	450
58.4 Introducing the Collapsing Toolbar Layout	452
58.5 Changing the Title and Scrim Color	455
58.6 Summary	456
59. An Overview of Android Intents	457
59.1 An Overview of Intents	457
59.2 Explicit Intents.....	457
59.3 Returning Data from an Activity	458
59.4 Implicit Intents	459
59.5 Using Intent Filters.....	460

59.6 Automatic Link Verification	460
59.7 Manually Enabling Links	463
59.8 Checking Intent Availability	464
59.9 Summary	465
60. Android Explicit Intents – A Worked Example	467
60.1 Creating the Explicit Intent Example Application	467
60.2 Designing the User Interface Layout for MainActivity	467
60.3 Creating the Second Activity Class	468
60.4 Designing the User Interface Layout for SecondActivity	469
60.5 Reviewing the Application Manifest File	469
60.6 Creating the Intent	470
60.7 Extracting Intent Data	471
60.8 Launching SecondActivity as a Sub-Activity	472
60.9 Returning Data from a Sub-Activity	473
60.10 Testing the Application	473
60.11 Summary	473
61. Android Implicit Intents – A Worked Example	475
61.1 Creating the Android Studio Implicit Intent Example Project	475
61.2 Designing the User Interface	475
61.3 Creating the Implicit Intent	476
61.4 Adding a Second Matching Activity	476
61.5 Adding the Web View to the UI	477
61.6 Obtaining the Intent URL	477
61.7 Modifying the MyWebView Project Manifest File	479
61.8 Installing the MyWebView Package on a Device	480
61.9 Testing the Application	481
61.10 Manually Enabling the Link	481
61.11 Automatic Link Verification	483
61.12 Summary	485
62. Android Broadcast Intents and Broadcast Receivers	487
62.1 An Overview of Broadcast Intents	487
62.2 An Overview of Broadcast Receivers	488
62.3 Obtaining Results from a Broadcast	489
62.4 Sticky Broadcast Intents	489
62.5 The Broadcast Intent Example	489
62.6 Creating the Example Application	490
62.7 Creating and Sending the Broadcast Intent	490
62.8 Creating the Broadcast Receiver	491
62.9 Registering the Broadcast Receiver	492
62.10 Testing the Broadcast Example	492
62.11 Listening for System Broadcasts	493
62.12 Summary	493
63. An Introduction to Kotlin Coroutines	495
63.1 What are Coroutines?	495
63.2 Threads vs. Coroutines	495
63.3 Coroutine Scope	496
63.4 Suspend Functions	496

Table of Contents

63.5 Coroutine Dispatchers.....	496
63.6 Coroutine Builders.....	497
63.7 Jobs.....	497
63.8 Coroutines – Suspending and Resuming.....	498
63.9 Returning Results from a Coroutine	499
63.10 Using withContext.....	499
63.11 Coroutine Channel Communication	501
63.12 Summary.....	502
64. An Android Kotlin Coroutines Tutorial.....	503
64.1 Creating the Coroutine Example Application.....	503
64.2 Designing the User Interface	503
64.3 Implementing the SeekBar.....	504
64.4 Adding the Suspend Function.....	505
64.5 Implementing the launchCoroutines Method.....	506
64.6 Testing the App.....	506
64.7 Summary	507
65. An Overview of Android Services.....	509
65.1 Intent Service.....	509
65.2 Bound Service.....	509
65.3 The Anatomy of a Service	510
65.4 Controlling Destroyed Service Restart Options.....	510
65.5 Declaring a Service in the Manifest File.....	510
65.6 Starting a Service Running on System Startup.....	511
65.7 Summary	512
66. Android Local Bound Services – A Worked Example.....	513
66.1 Understanding Bound Services.....	513
66.2 Bound Service Interaction Options.....	513
66.3 A Local Bound Service Example.....	513
66.4 Adding a Bound Service to the Project.....	514
66.5 Implementing the Binder.....	514
66.6 Binding the Client to the Service	516
66.7 Completing the Example.....	517
66.8 Testing the Application.....	518
66.9 Summary	519
67. Android Remote Bound Services – A Worked Example	521
67.1 Client to Remote Service Communication.....	521
67.2 Creating the Example Application.....	521
67.3 Designing the User Interface	521
67.4 Implementing the Remote Bound Service.....	521
67.5 Configuring a Remote Service in the Manifest File.....	523
67.6 Launching and Binding to the Remote Service.....	523
67.7 Sending a Message to the Remote Service	525
67.8 Summary	525
68. An Introduction to Kotlin Flow.....	527
68.1 Understanding Flows.....	527
68.2 Creating the Sample Project	527

68.3 Adding the Kotlin Lifecycle Library	528
68.4 Declaring a Flow.....	528
68.5 Emitting Flow Data.....	529
68.6 Collecting Flow Data	529
68.7 Adding a Flow Buffer.....	531
68.8 Transforming Data with Intermediaries	532
68.9 Terminal Flow Operators	534
68.10 Flow Flattening.....	534
68.11 Combining Multiple Flows	536
68.12 Hot and Cold Flows.....	537
68.13 StateFlow	537
68.14 SharedFlow.....	538
68.15 Summary.....	540
69. An Android SharedFlow Tutorial	541
69.1 About the Project	541
69.2 Creating the SharedFlowDemo Project.....	541
69.3 Adding the Lifecycle Libraries.....	541
69.4 Designing the User Interface Layout	542
69.5 Adding the List Row Layout	542
69.6 Adding the RecyclerView Adapter.....	543
69.7 Adding the ViewModel	543
69.8 Configuring the ViewModelProvider.....	544
69.9 Collecting the Flow Values.....	545
69.10 Testing the SharedFlowDemo App.....	546
69.11 Handling Flows in the Background	547
69.12 Summary.....	549
70. An Overview of Android SQLite Databases	551
70.1 Understanding Database Tables	551
70.2 Introducing Database Schema	551
70.3 Columns and Data Types	551
70.4 Database Rows	552
70.5 Introducing Primary Keys	552
70.6 What is SQLite?	552
70.7 Structured Query Language (SQL).....	552
70.8 Trying SQLite on an Android Virtual Device (AVD)	553
70.9 Android SQLite Classes.....	554
70.9.1 Cursor.....	555
70.9.2 SQLiteDatabase	555
70.9.3 SQLiteOpenHelper	555
70.9.4 ContentValues.....	556
70.10 The Android Room Persistence Library.....	556
70.11 Summary.....	556
71. An Android SQLite Database Tutorial	557
71.1 About the Database Example.....	557
71.2 Creating the SQLDemo Project.....	557
71.3 Designing the User interface	557
71.4 Creating the Data Model.....	558
71.5 Implementing the Data Handler	558

Table of Contents

71.6 The Add Handler Method	560
71.7 The Query Handler Method	561
71.8 The Delete Handler Method	561
71.9 Implementing the Activity Event Methods.....	562
71.10 Testing the Application.....	563
71.11 Summary	563
72. Understanding Android Content Providers.....	565
72.1 What is a Content Provider?.....	565
72.2 The Content Provider	565
72.2.1 onCreate()	565
72.2.2 query()	565
72.2.3 insert()	565
72.2.4 update()	566
72.2.5 delete()	566
72.2.6 getType()	566
72.3 The Content URI	566
72.4 The Content Resolver	566
72.5 The <provider> Manifest Element	567
72.6 Summary	567
73. An Android Content Provider Tutorial.....	569
73.1 Copying the SQLDemo Project.....	569
73.2 Adding the Content Provider Package.....	569
73.3 Creating the Content Provider Class.....	570
73.4 Constructing the Authority and Content URI.....	571
73.5 Implementing URI Matching in the Content Provider.....	572
73.6 Implementing the Content Provider onCreate() Method	573
73.7 Implementing the Content Provider insert() Method	573
73.8 Implementing the Content Provider query() Method	574
73.9 Implementing the Content Provider update() Method	575
73.10 Implementing the Content Provider delete() Method	576
73.11 Declaring the Content Provider in the Manifest File	577
73.12 Modifying the Database Handler.....	578
73.13 Summary	580
74. An Android Content Provider Client Tutorial.....	581
74.1 Creating the SQLDemoClient Project.....	581
74.2 Designing the User interface	581
74.3 Accessing the Content Provider	581
74.4 Adding the Query Permission.....	582
74.5 Testing the Project.....	583
74.6 Summary	583
75. The Android Room Persistence Library	585
75.1 Revisiting Modern App Architecture	585
75.2 Key Elements of Room Database Persistence.....	585
75.2.1 Repository	586
75.2.2 Room Database	586
75.2.3 Data Access Object (DAO)	586
75.2.4 Entities.....	586

75.2.5 SQLite Database	586
75.3 Understanding Entities.....	587
75.4 Data Access Objects.....	589
75.5 The Room Database.....	590
75.6 The Repository.....	591
75.7 In-Memory Databases.....	592
75.8 Database Inspector.....	592
75.9 Summary.....	593
76. An Android TableLayout and TableRow Tutorial.....	595
76.1 The TableLayout and TableRow Layout Views.....	595
76.2 Creating the Room Database Project.....	596
76.3 Converting to a LinearLayout.....	596
76.4 Adding the TableLayout to the User Interface.....	597
76.5 Configuring the TableRows.....	598
76.6 Adding the Button Bar to the Layout.....	599
76.7 Adding the RecyclerView.....	600
76.8 Adjusting the Layout Margins.....	601
76.9 Summary.....	601
77. An Android Room Database and Repository Tutorial.....	603
77.1 About the RoomDemo Project.....	603
77.2 Modifying the Build Configuration.....	603
77.3 Building the Entity.....	604
77.4 Creating the Data Access Object.....	606
77.5 Adding the Room Database.....	607
77.6 Adding the Repository.....	608
77.7 Adding the ViewModel.....	610
77.8 Creating the Product Item Layout.....	611
77.9 Adding the RecyclerView Adapter.....	612
77.10 Preparing the Main Activity.....	613
77.11 Adding the Button Listeners.....	614
77.12 Adding LiveData Observers.....	614
77.13 Initializing the RecyclerView.....	615
77.14 Testing the RoomDemo App.....	615
77.15 Using the Database Inspector.....	616
77.16 Summary.....	616
78. Video Playback on Android using the VideoView and MediaController Classes.....	617
78.1 Introducing the Android VideoView Class.....	617
78.2 Introducing the Android MediaController Class.....	618
78.3 Creating the Video Playback Example.....	618
78.4 Designing the VideoPlayer Layout.....	618
78.5 Downloading the Video File.....	619
78.6 Configuring the VideoView.....	619
78.7 Adding the MediaController to the Video View.....	621
78.8 Setting up the onPreparedListener.....	621
78.9 Summary.....	622
79. Android Picture-in-Picture Mode.....	623
79.1 Picture-in-Picture Features.....	623

Table of Contents

79.2 Enabling Picture-in-Picture Mode.....	624
79.3 Configuring Picture-in-Picture Parameters	624
79.4 Entering Picture-in-Picture Mode	625
79.5 Detecting Picture-in-Picture Mode Changes	625
79.6 Adding Picture-in-Picture Actions.....	625
79.7 Summary	626
80. An Android Picture-in-Picture Tutorial.....	627
80.1 Adding Picture-in-Picture Support to the Manifest.....	627
80.2 Adding a Picture-in-Picture Button	627
80.3 Entering Picture-in-Picture Mode	628
80.4 Detecting Picture-in-Picture Mode Changes	629
80.5 Adding a Broadcast Receiver	629
80.6 Adding the PiP Action.....	630
80.7 Testing the Picture-in-Picture Action	633
80.8 Summary	633
81. Making Runtime Permission Requests in Android.....	635
81.1 Understanding Normal and Dangerous Permissions.....	635
81.2 Creating the Permissions Example Project.....	637
81.3 Checking for a Permission	637
81.4 Requesting Permission at Runtime.....	639
81.5 Providing a Rationale for the Permission Request	640
81.6 Testing the Permissions App.....	641
81.7 Summary	642
82. Android Audio Recording and Playback using MediaPlayer and MediaRecorder	643
82.1 Playing Audio	643
82.2 Recording Audio and Video using the MediaRecorder Class.....	644
82.3 About the Example Project	645
82.4 Creating the AudioApp Project.....	645
82.5 Designing the User Interface	645
82.6 Checking for Microphone Availability	646
82.7 Initializing the Activity.....	647
82.8 Implementing the recordAudio() Method.....	648
82.9 Implementing the stopAudio() Method.....	648
82.10 Implementing the playAudio() method.....	649
82.11 Configuring and Requesting Permissions	649
82.12 Testing the Application.....	651
82.13 Summary	651
83. An Android Notifications Tutorial	653
83.1 An Overview of Notifications.....	653
83.2 Creating the NotifyDemo Project	655
83.3 Designing the User Interface	655
83.4 Creating the Second Activity	655
83.5 Creating a Notification Channel	656
83.6 Requesting Notification Permission	657
83.7 Creating and Issuing a Notification	660
83.8 Launching an Activity from a Notification.....	662
83.9 Adding Actions to a Notification	664

83.10 Bundled Notifications.....	664
83.11 Summary.....	666
84. An Android Direct Reply Notification Tutorial	667
84.1 Creating the DirectReply Project.....	667
84.2 Designing the User Interface.....	667
84.3 Requesting Notification Permission.....	668
84.4 Creating the Notification Channel.....	669
84.5 Building the RemoteInput Object.....	670
84.6 Creating the PendingIntent.....	671
84.7 Creating the Reply Action.....	672
84.8 Receiving Direct Reply Input.....	673
84.9 Updating the Notification.....	674
84.10 Summary.....	675
85. Working with the Google Maps Android API in Android Studio	677
85.1 The Elements of the Google Maps Android API.....	677
85.2 Creating the Google Maps Project.....	678
85.3 Creating a Google Cloud Billing Account.....	678
85.4 Creating a New Google Cloud Project.....	679
85.5 Enabling the Google Maps SDK.....	680
85.6 Generating a Google Maps API Key.....	681
85.7 Adding the API Key to the Android Studio Project.....	682
85.8 Testing the Application.....	682
85.9 Understanding Geocoding and Reverse Geocoding.....	682
85.10 Adding a Map to an Application.....	684
85.11 Requesting Current Location Permission.....	684
85.12 Displaying the User's Current Location.....	686
85.13 Changing the Map Type.....	687
85.14 Displaying Map Controls to the User.....	688
85.15 Handling Map Gesture Interaction.....	688
85.15.1 Map Zooming Gestures.....	688
85.15.2 Map Scrolling/Panning Gestures.....	689
85.15.3 Map Tilt Gestures.....	689
85.15.4 Map Rotation Gestures.....	689
85.16 Creating Map Markers.....	689
85.17 Controlling the Map Camera.....	690
85.18 Summary.....	691
86. Printing with the Android Printing Framework.....	693
86.1 The Android Printing Architecture.....	693
86.2 The Print Service Plugins.....	693
86.3 Google Cloud Print.....	694
86.4 Printing to Google Drive.....	694
86.5 Save as PDF.....	695
86.6 Printing from Android Devices.....	695
86.7 Options for Building Print Support into Android Apps.....	696
86.7.1 Image Printing.....	696
86.7.2 Creating and Printing HTML Content.....	697
86.7.3 Printing a Web Page.....	698
86.7.4 Printing a Custom Document.....	699

86.8 Summary	699
87. An Android HTML and Web Content Printing Example	701
87.1 Creating the HTML Printing Example Application	701
87.2 Printing Dynamic HTML Content	701
87.3 Creating the Web Page Printing Example	704
87.4 Removing the Floating Action Button	704
87.5 Removing Navigation Features.....	704
87.6 Designing the User Interface Layout	705
87.7 Accessing the WebView from the Main Activity	706
87.8 Loading the Web Page into the WebView.....	706
87.9 Adding the Print Menu Option.....	707
87.10 Summary.....	709
88. A Guide to Android Custom Document Printing	711
88.1 An Overview of Android Custom Document Printing	711
88.1.1 Custom Print Adapters.....	711
88.2 Preparing the Custom Document Printing Project.....	712
88.3 Designing the UI	712
88.4 Creating the Custom Print Adapter.....	713
88.5 Implementing the onLayout() Callback Method	714
88.6 Implementing the onWrite() Callback Method	717
88.7 Checking a Page is in Range	719
88.8 Drawing the Content on the Page Canvas	720
88.9 Starting the Print Job	722
88.10 Testing the Application.....	723
88.11 Summary.....	723
89. An Introduction to Android App Links.....	725
89.1 An Overview of Android App Links	725
89.2 App Link Intent Filters	725
89.3 Handling App Link Intents	726
89.4 Associating the App with a Website.....	726
89.5 Summary	727
90. An Android Studio App Links Tutorial	729
90.1 About the Example App	729
90.2 The Database Schema	729
90.3 Loading and Running the Project	729
90.4 Adding the URL Mapping.....	731
90.5 Adding the Intent Filter.....	734
90.6 Adding Intent Handling Code.....	734
90.7 Testing the App.....	737
90.8 Creating the Digital Asset Links File	737
90.9 Testing the App Link.....	738
90.10 Summary.....	738
91. An Android Biometric Authentication Tutorial.....	739
91.1 An Overview of Biometric Authentication.....	739
91.2 Creating the Biometric Authentication Project	739
91.3 Configuring Device Fingerprint Authentication	740

91.4 Adding the Biometric Permission to the Manifest File.....	740
91.5 Designing the User Interface	741
91.6 Adding a Toast Convenience Method	741
91.7 Checking the Security Settings.....	742
91.8 Configuring the Authentication Callbacks.....	743
91.9 Adding the CancellationSignal.....	744
91.10 Starting the Biometric Prompt	744
91.11 Testing the Project.....	745
91.12 Summary	746
92. Creating, Testing, and Uploading an Android App Bundle.....	747
92.1 The Release Preparation Process.....	747
92.2 Android App Bundles.....	747
92.3 Register for a Google Play Developer Console Account.....	748
92.4 Configuring the App in the Console	749
92.5 Enabling Google Play App Signing	750
92.6 Creating a Keystore File	750
92.7 Creating the Android App Bundle.....	751
92.8 Generating Test APK Files	753
92.9 Uploading the App Bundle to the Google Play Developer Console.....	754
92.10 Exploring the App Bundle	755
92.11 Managing Testers	756
92.12 Rolling the App Out for Testing.....	756
92.13 Uploading New App Bundle Revisions.....	757
92.14 Analyzing the App Bundle File	758
92.15 Summary	759
93. An Overview of Android In-App Billing	761
93.1 Preparing a Project for In-App Purchasing	761
93.2 Creating In-App Products and Subscriptions	761
93.3 Billing Client Initialization.....	762
93.4 Connecting to the Google Play Billing Library.....	763
93.5 Querying Available Products.....	763
93.6 Starting the Purchase Process.....	764
93.7 Completing the Purchase.....	765
93.8 Querying Previous Purchases.....	766
93.9 Summary	766
94. An Android In-App Purchasing Tutorial	767
94.1 About the In-App Purchasing Example Project.....	767
94.2 Creating the InAppPurchase Project.....	767
94.3 Adding Libraries to the Project	767
94.4 Designing the User Interface	768
94.5 Adding the App to the Google Play Store	769
94.6 Creating an In-App Product.....	769
94.7 Enabling License Testers	770
94.8 Initializing the Billing Client	771
94.9 Querying the Product.....	772
94.10 Launching the Purchase Flow	773
94.11 Handling Purchase Updates	774
94.12 Consuming the Product.....	775

Table of Contents

94.13 Restoring a Previous Purchase	776
94.14 Testing the App.....	777
94.15 Troubleshooting	778
94.16 Summary	778
95. Accessing Cloud Storage using the Android Storage Access Framework.....	779
95.1 The Storage Access Framework.....	779
95.2 Working with the Storage Access Framework.....	780
95.3 Filtering Picker File Listings.....	780
95.4 Handling Intent Results.....	781
95.5 Reading the Content of a File	781
95.6 Writing Content to a File	782
95.7 Deleting a File.....	783
95.8 Gaining Persistent Access to a File.....	783
95.9 Summary	783
96. An Android Storage Access Framework Example	785
96.1 About the Storage Access Framework Example.....	785
96.2 Creating the Storage Access Framework Example.....	785
96.3 Designing the User Interface	785
96.4 Adding the Activity Launchers.....	786
96.5 Creating a New Storage File.....	787
96.6 Saving to a Storage File.....	788
96.7 Opening and Reading a Storage File	789
96.8 Testing the Storage Access Application	790
96.9 Summary	792
97. An Android Studio Primary/Detail Flow Tutorial	793
97.1 The Primary/Detail Flow.....	793
97.2 Creating a Primary/Detail Flow Activity	794
97.3 Adding the Primary/Detail Flow Activity.....	794
97.4 Modifying the Primary/Detail Flow Template	795
97.5 Changing the Content Model.....	795
97.6 Changing the Detail Pane	797
97.7 Modifying the ItemDetailFragment Class	798
97.8 Modifying the ItemListFragment Class.....	799
97.9 Adding Manifest Permissions.....	799
97.10 Running the Application.....	800
97.11 Summary.....	800
98. Working with Material Design 3 Theming	801
98.1 Material Design 2 vs. Material Design 3	801
98.2 Understanding Material Design Theming	801
98.3 Material Design 3 Theming	801
98.4 Building a custom theme	803
98.5 Summary	805
99. A Material Design 3 Theming and Dynamic Color Tutorial.....	807
99.1 Creating the ThemeDemo Project	807
99.2 Designing the User Interface	807
99.3 Building a new theme	809

99.4 Adding the Theme to the Project	810
99.5 Enabling Dynamic Color Support	811
99.6 Summary	813
100. An Overview of Gradle in Android Studio	815
100.1 An Overview of Gradle	815
100.2 Gradle and Android Studio	815
100.2.1 Sensible Defaults	815
100.2.2 Dependencies	815
100.2.3 Build Variants	816
100.2.4 Manifest Entries	816
100.2.5 APK Signing	816
100.2.6 ProGuard Support.....	816
100.3 The Property and Settings Gradle Build File.....	816
100.4 The Top-level Gradle Build File	817
100.5 Module Level Gradle Build Files.....	818
100.6 Configuring Signing Settings in the Build File	820
100.7 Running Gradle Tasks from the Command Line	821
100.8 Summary	822
Index	823

2. Setting up an Android Studio Development Environment

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

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

2.1 System requirements

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

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

2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio 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:

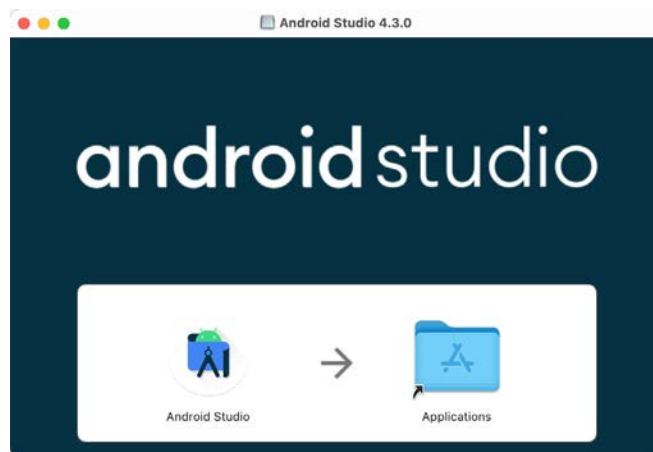


Figure 2-1

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:

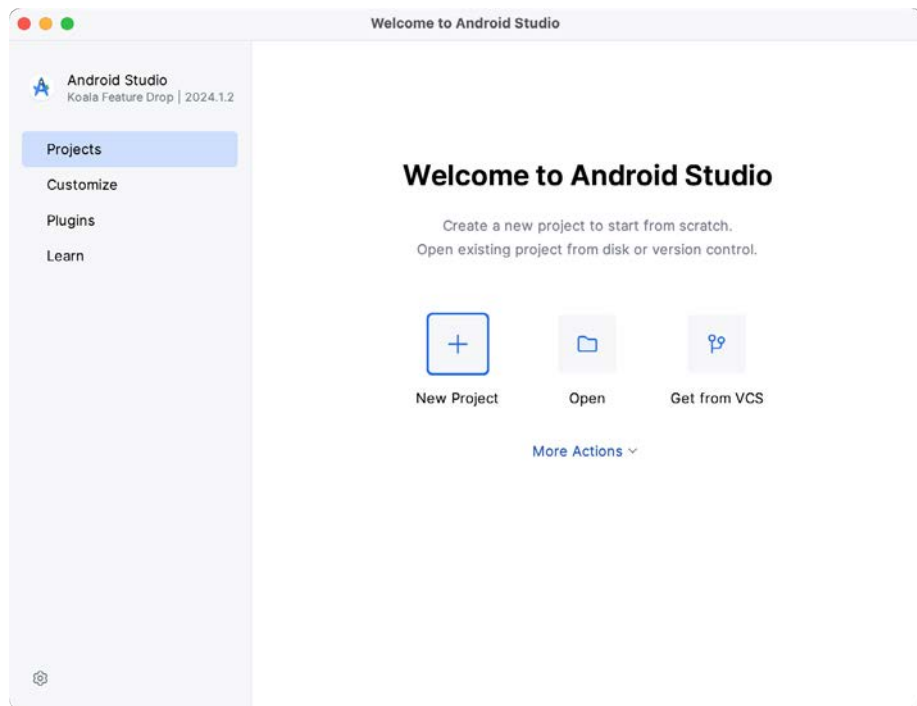


Figure 2-2

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:

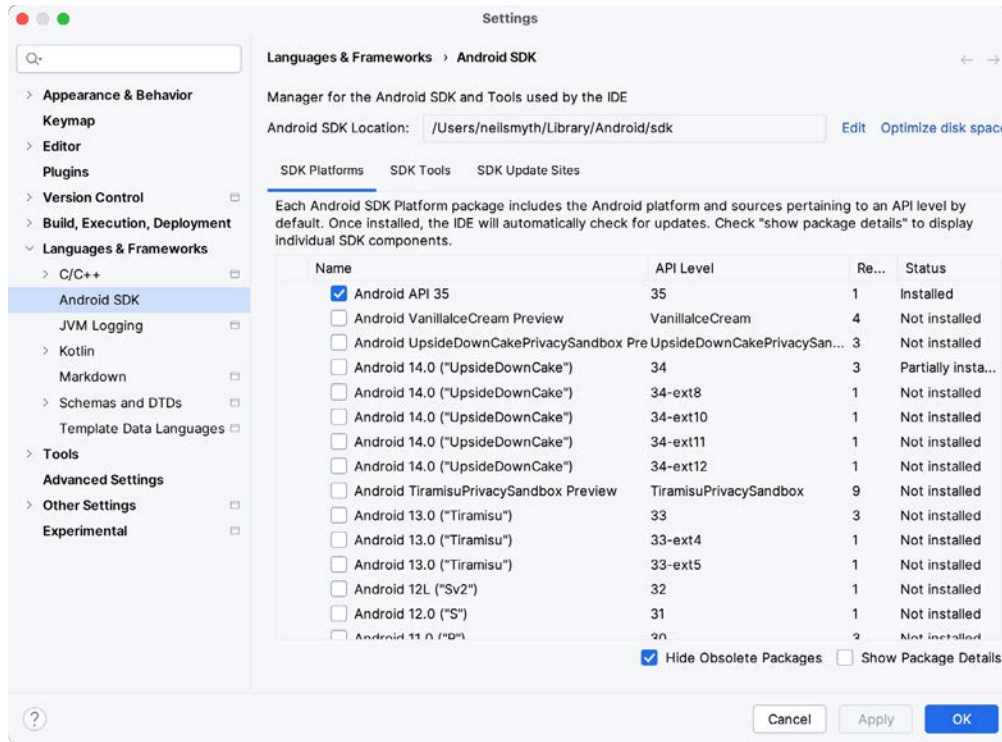


Figure 2-3

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:

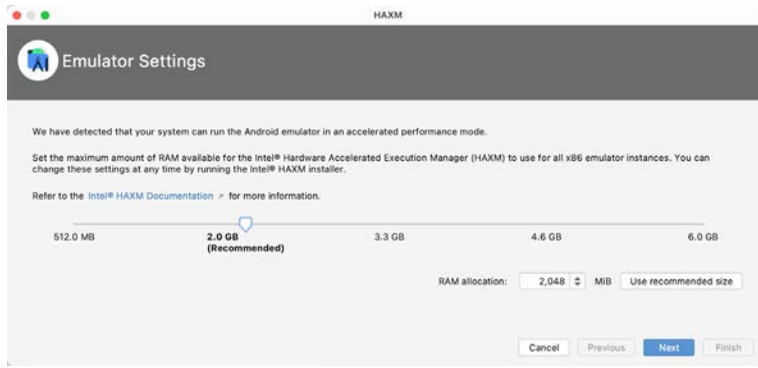


Figure 2-6

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:

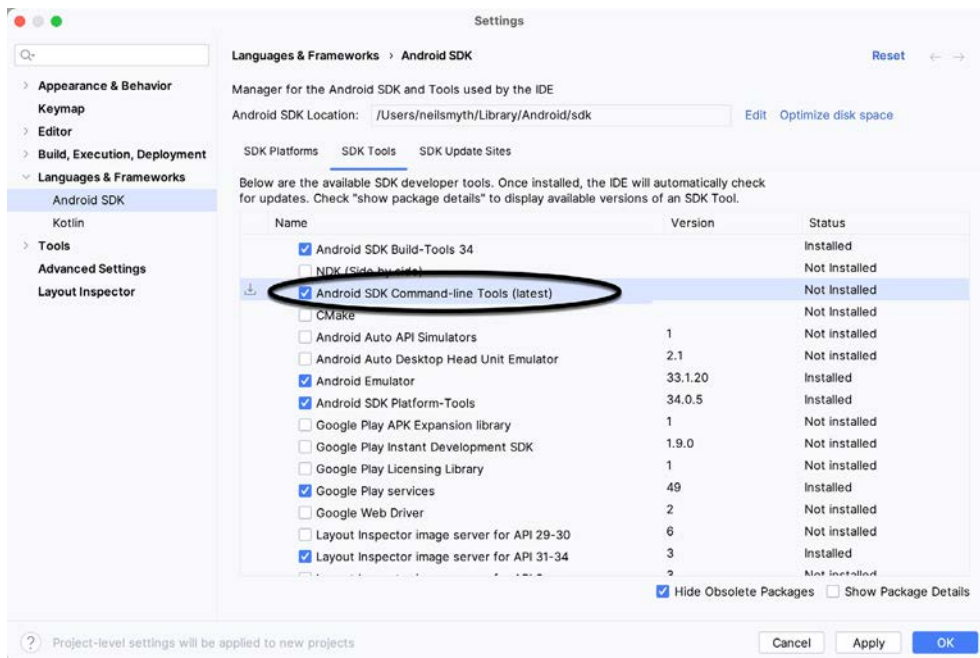


Figure 2-7

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:

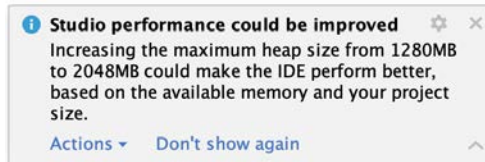


Figure 2-9

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:

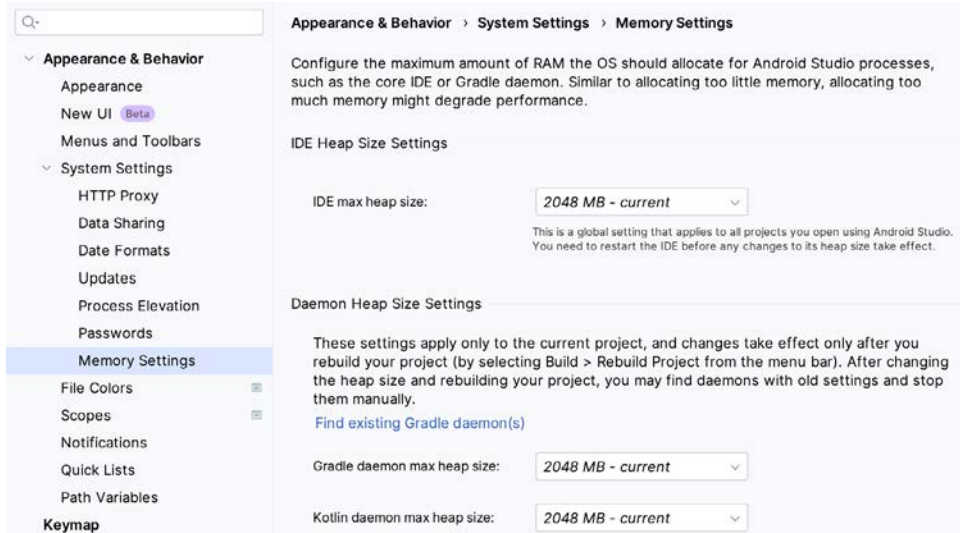


Figure 2-10

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:

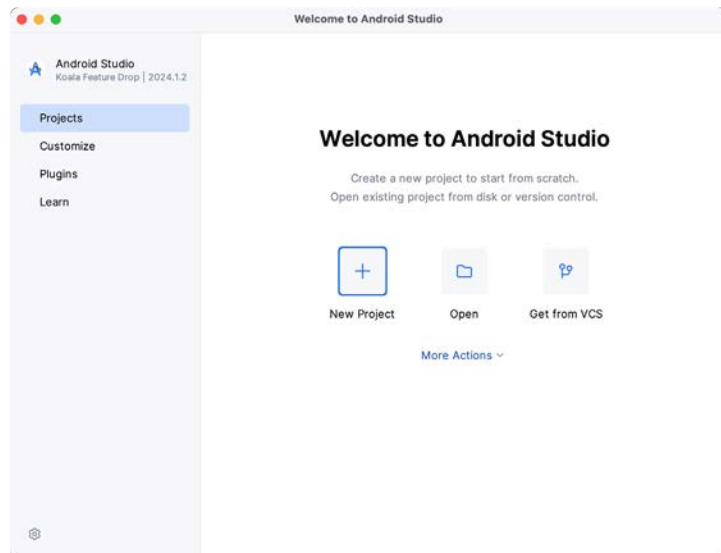


Figure 3-1

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.

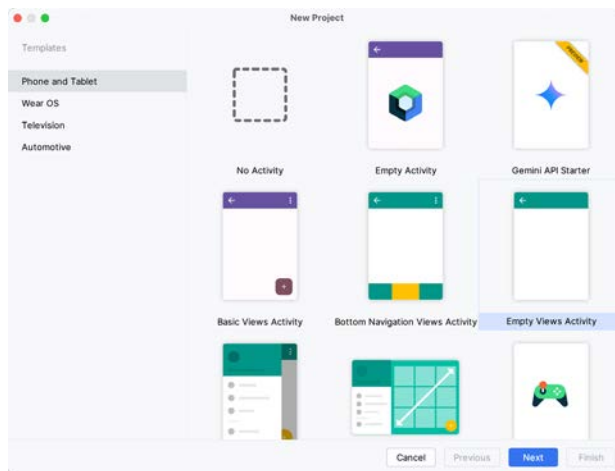


Figure 3-2

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:

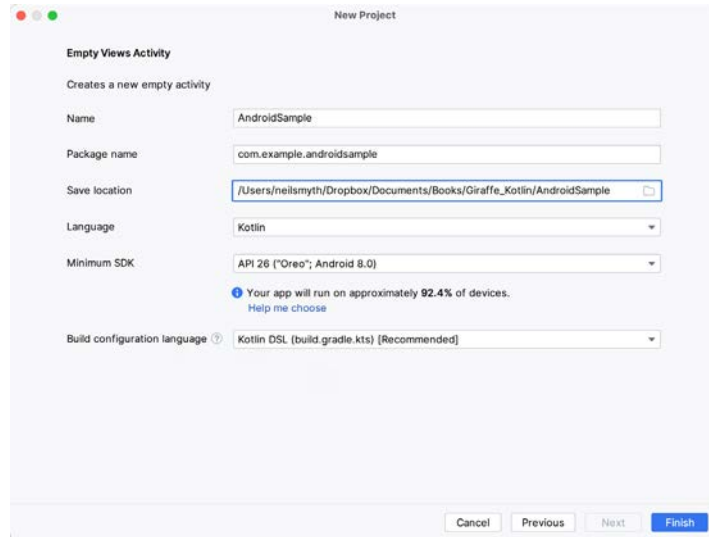


Figure 3-3

Finally, change the *Language* menu to *Kotlin* 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:

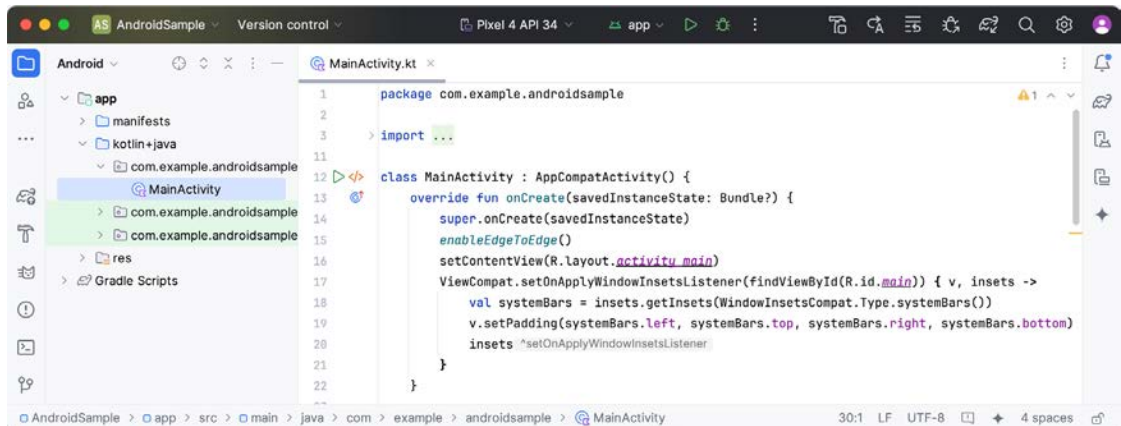


Figure 3-4

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:

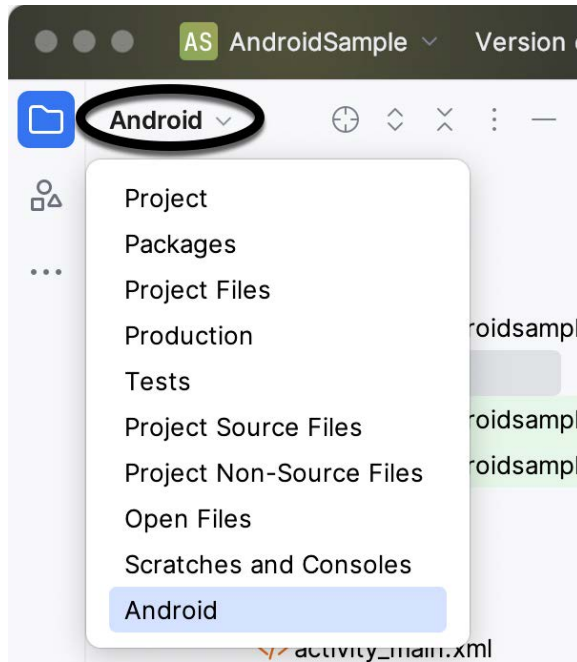


Figure 3-5

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:

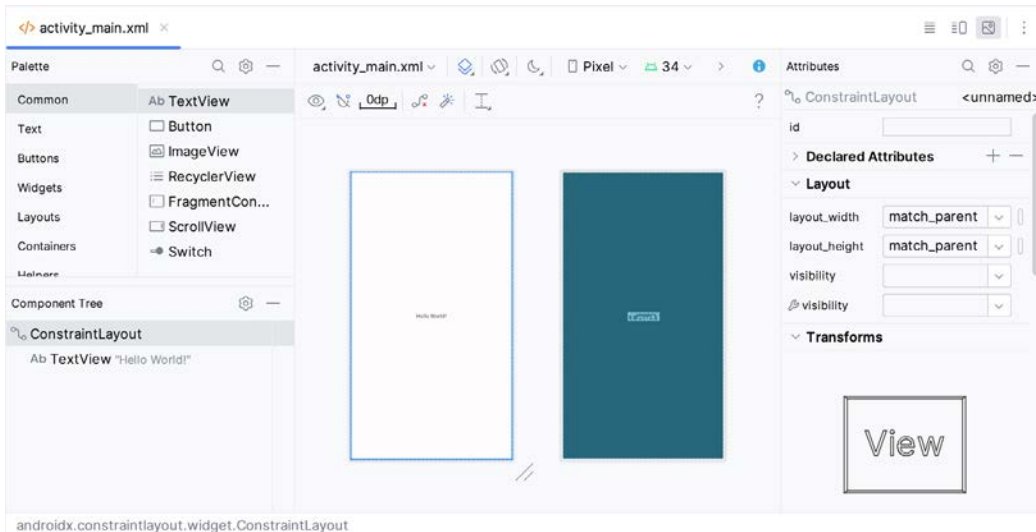




Figure 3-6

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:

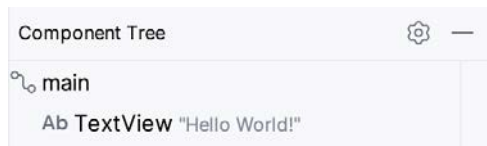


Figure 3-7

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.

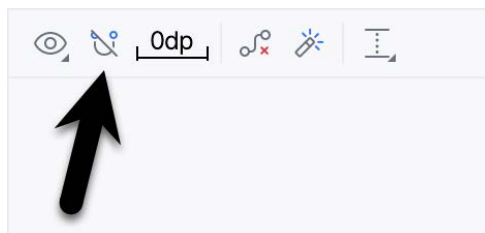


Figure 3-8

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:

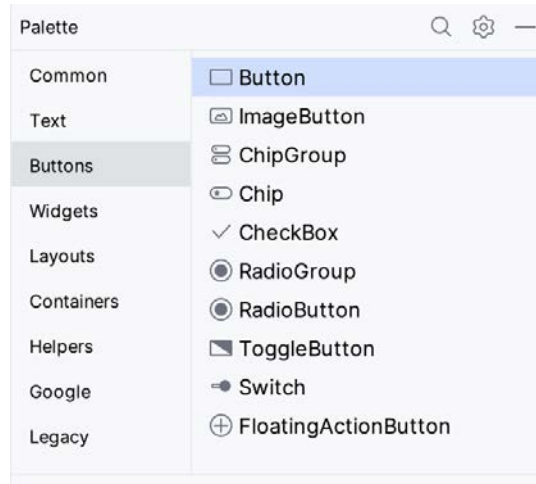


Figure 3-9

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:

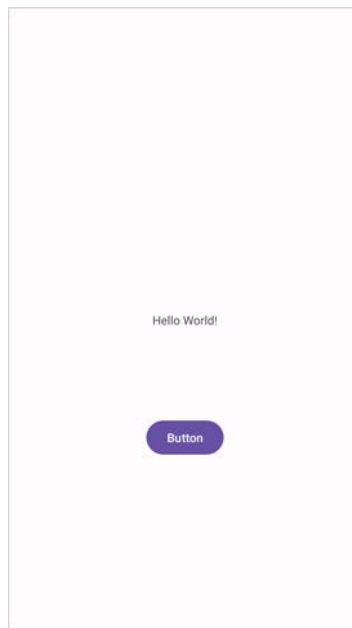


Figure 3-10

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:

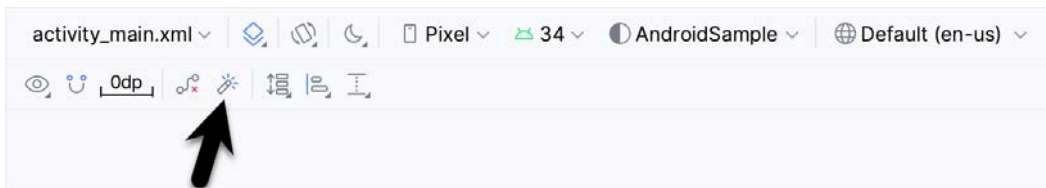


Figure 3-12

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:

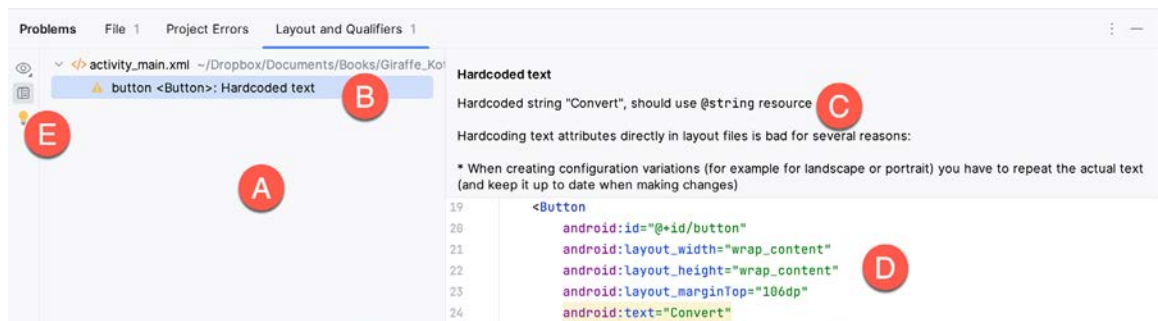


Figure 3-14

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:

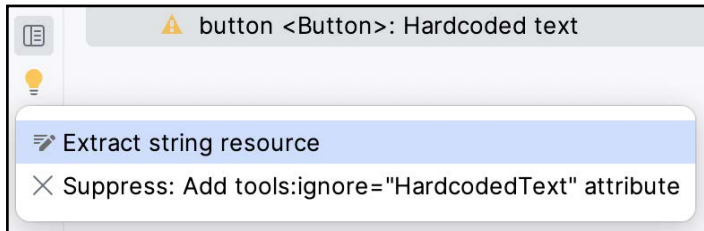


Figure 3-15

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:

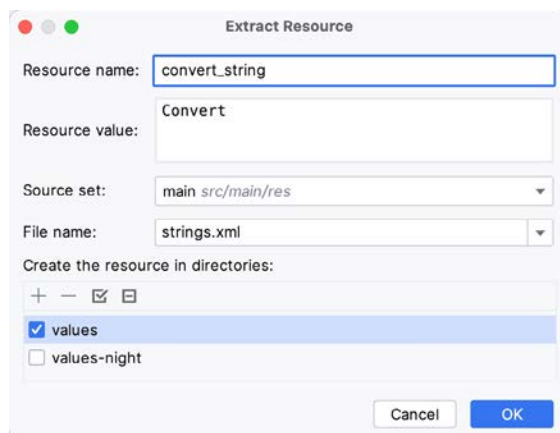


Figure 3-16

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:

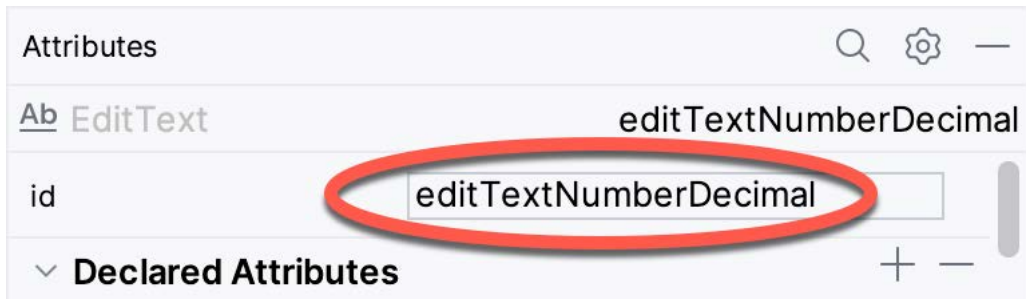


Figure 3-17

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:

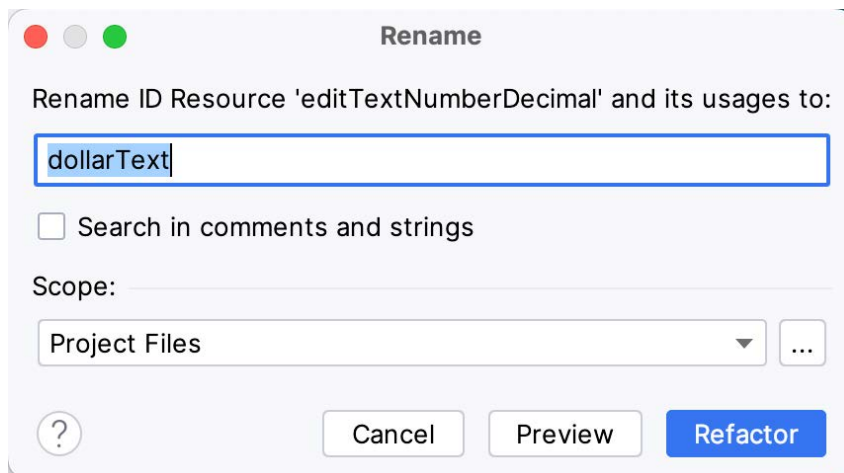


Figure 3-18

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:

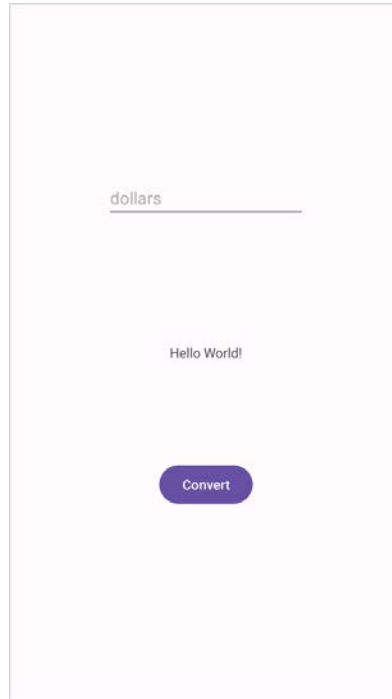


Figure 3-19

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:

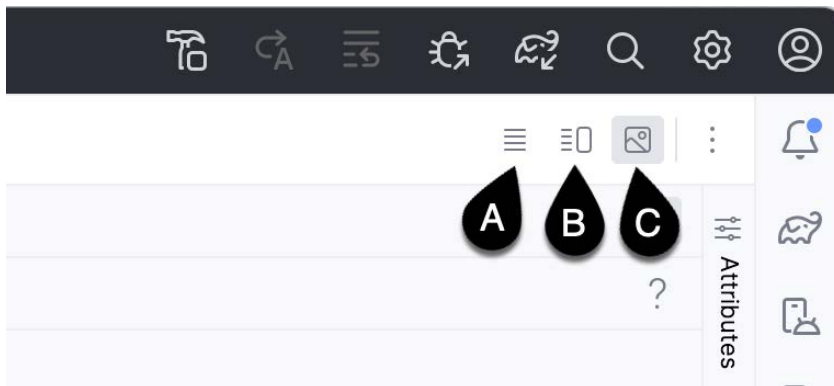


Figure 3-20

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:

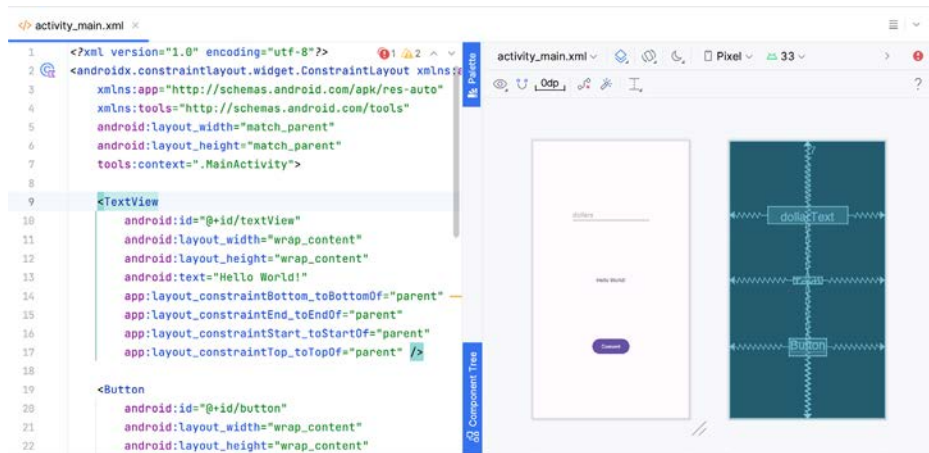


Figure 3-21

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:

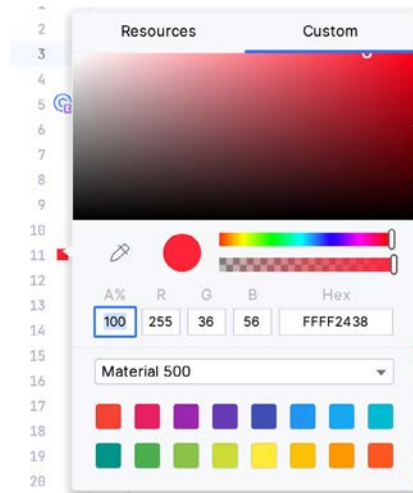


Figure 3-22

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:

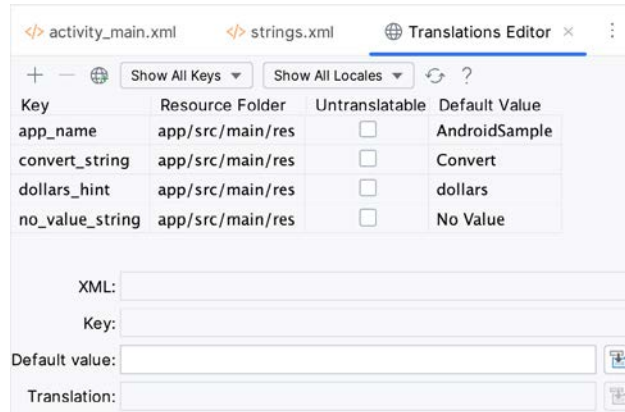


Figure 3-23

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:

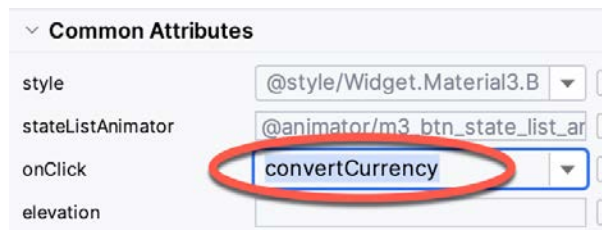


Figure 3-24

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

```
package com.example.androidsample

import android.os.Bundle
import androidx.activity.enableEdgeToEdge
import androidx.appcompat.app.AppCompatActivity
import androidx.core.view.ViewCompat
import androidx.core.view.WindowInsetsCompat
import android.view.View
import android.widget.EditText
import android.widget.TextView
```

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        .
        .
    }
}

fun convertCurrency(view: View) {
    val dollarText: EditText = findViewById(R.id.dollarText)
    val textView: TextView = findViewById(R.id.textView)

    if (dollarText.text.isNotEmpty()) {
        val dollarValue = dollarText.text.toString().toFloat()
        val euroValue = dollarValue * 0.85f
        textView.text = euroValue.toString()
    } else {
        textView.text = getString(R.string.no_value_string)
    }
}
```

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

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

3.9 Summary

While not excessively complex, 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.

6. A Tour of the Android Studio User Interface

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

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

6.1 The Welcome Screen

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

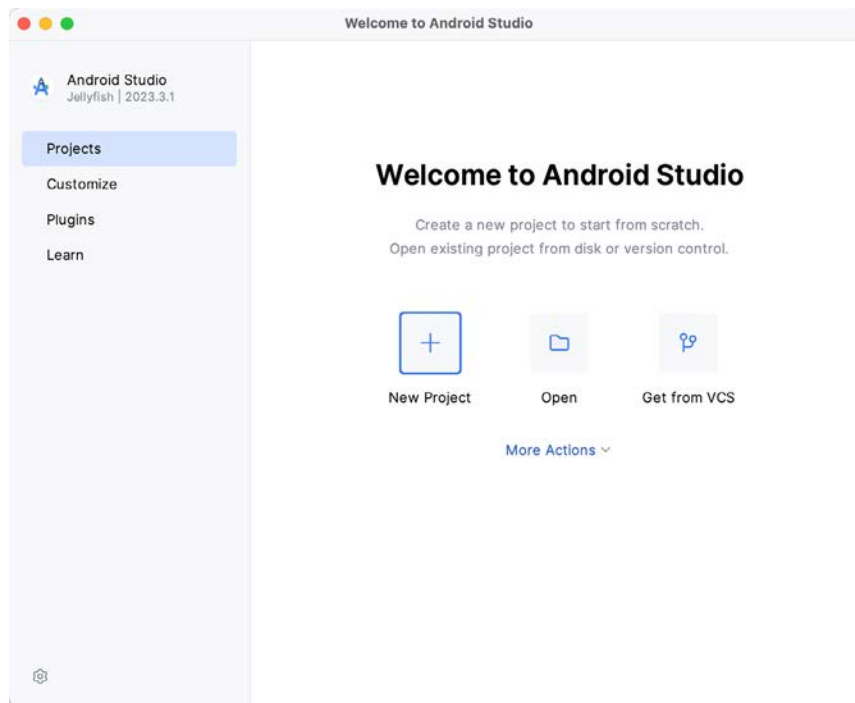


Figure 6-1

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

A Tour of the Android Studio User Interface

Studio plugins may be viewed, installed, and managed using the *Plugins* option.

Additional options are available by selecting the More Actions link or using the menu shown in Figure 6-2 when the list of recent projects replaces the More Actions link:

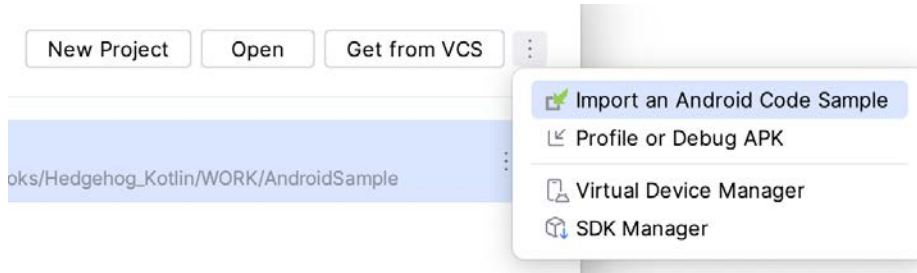


Figure 6-2

6.2 The Menu Bar

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

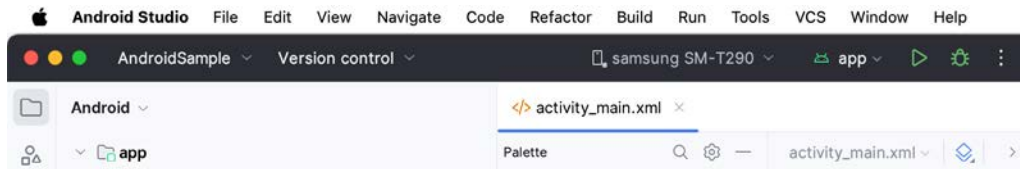


Figure 6-3

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

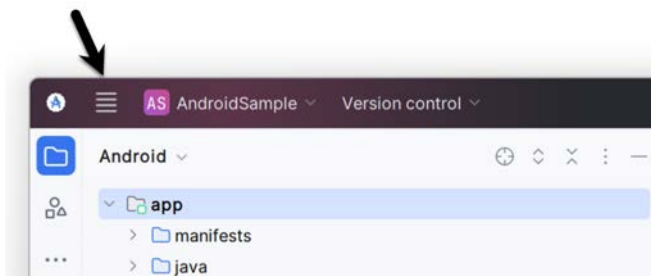


Figure 6-4

6.3 The Main Window

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

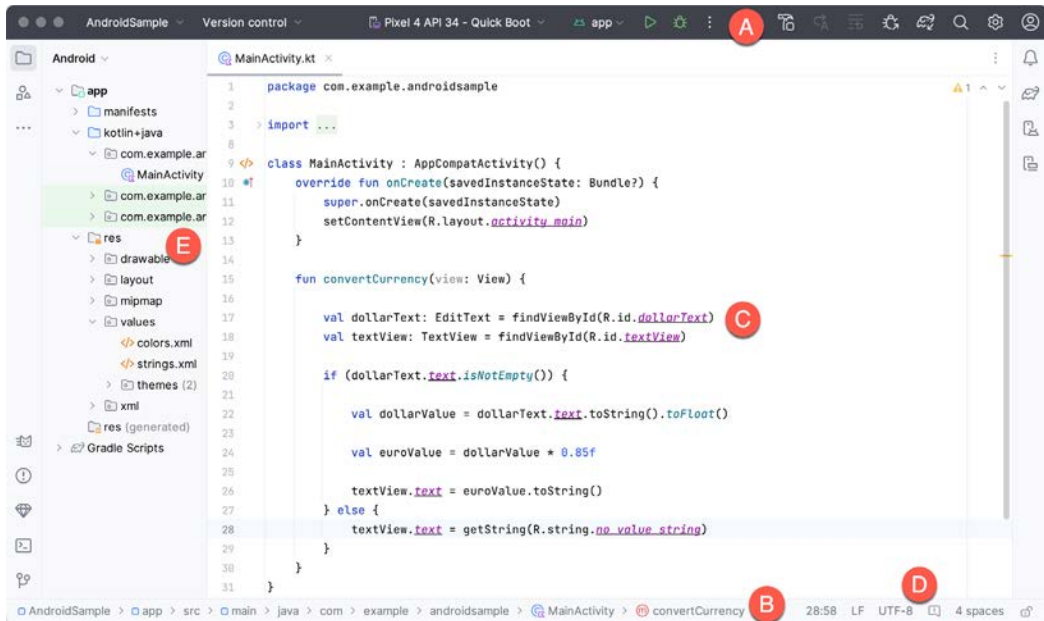


Figure 6-5

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

A – Toolbar – A selection of shortcuts to frequently performed actions. The toolbar buttons provide quick access to a select group of menu bar actions. The toolbar can be customized by right-clicking on the bar and selecting the *Customize Toolbar...* menu option. The toolbar menu shown in Figure 6-6 provides a convenient way to perform tasks such as creating and opening projects and switching between windows when multiple projects are open:

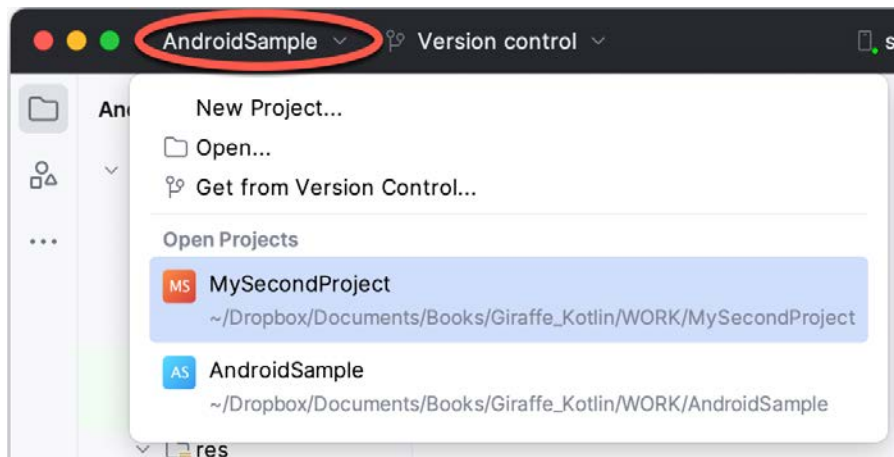


Figure 6-6

B – Navigation Bar – The navigation bar provides a convenient way to move around the files and folders that make up the project. Clicking on an element in the navigation bar will drop down a menu listing the sub-folders and files at that location, ready for selection. Similarly, clicking on a class name displays a menu listing methods contained within that class:



Figure 6-7

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

C – Editor Window – The editor window displays the content of the file on which the developer is currently working. When multiple files are open, each file is represented by a tab located along the top edge of the editor, as shown in Figure 6-8:

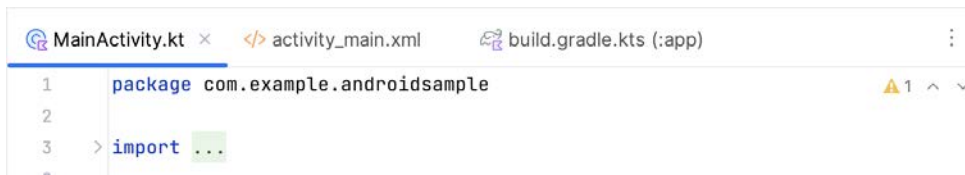


Figure 6-8

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

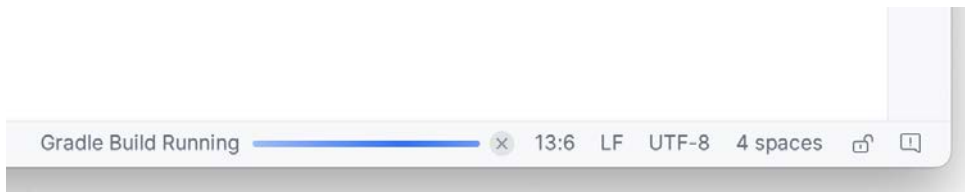


Figure 6-9

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

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

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

6.4 The Tool Windows

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

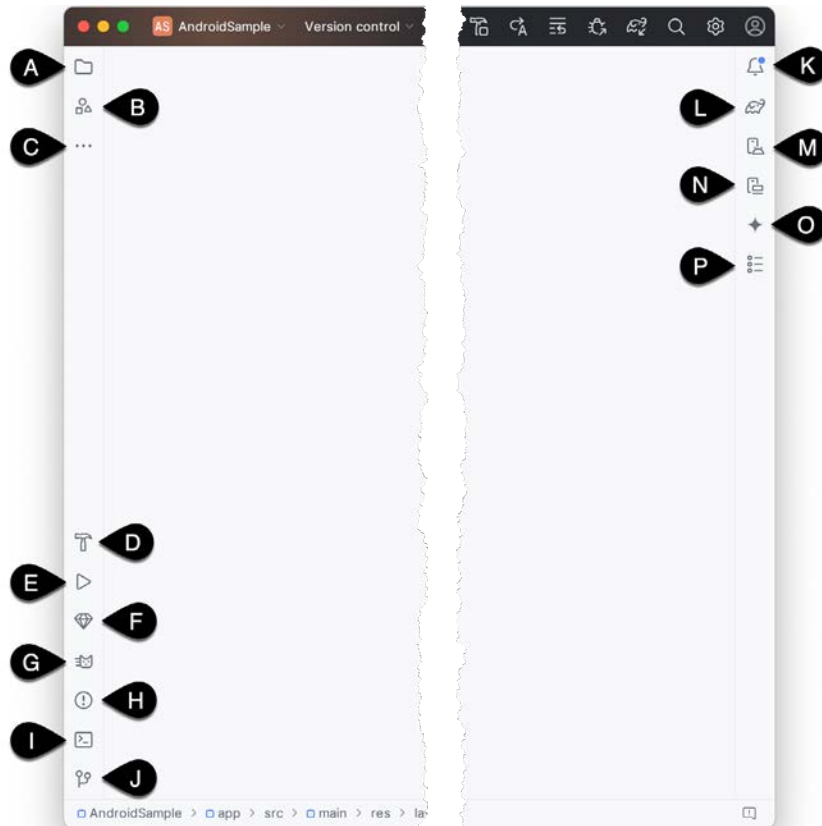


Figure 6-10

Clicking on a button will display the corresponding tool window, while a second click will hide the window. The location of a button in a tool window bar indicates the side of the window against which the window will appear when displayed. These positions can be changed by clicking and dragging the buttons to different locations in other window toolbars.

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

- **Project (A)** – The project view provides an overview of the file structure that makes up the project allowing for quick navigation between files. Generally, double-clicking on a file in the project view will cause that file to be loaded into the appropriate editing tool.
- **Resource Manager (B)** - A tool for adding and managing resources and assets within the project, such as images, colors, and layout files.
- **More Tool Windows (C)** - Displays a menu containing additional tool windows not currently displayed in a tool window bar. When a tool window is selected from this menu, it will appear as a button in a tool window bar.
- **Build (D)** - Displays a real-time view of each process step while Android Studio builds the current project.
- **Run (E)** – The run tool window becomes available when an application is currently running and provides a view of the results of the run together with options to stop or restart a running process. If an application fails to install and run on a device or emulator, this window typically provides diagnostic information about the problem.

- **App Quality Insights (F)** - Provides access to the cloud-based Firebase app quality and crash analytics platform.
- **Logcat (G)** - The Logcat tool window provides access to the monitoring log output from a running application and options for taking screenshots and videos of the application and stopping and restarting a process.
- **Problems (H)** - A central location to view all of the current errors or warnings within the project. Double-clicking on an item in the problem list will take you to the problem file and location.
- **Terminal (I)** - Provides access to a terminal window on the system on which Android Studio is running. On Windows systems, this is the Command Prompt interface, while on Linux and macOS systems, this takes the form of a Terminal prompt.
- **Version Control (J)** - This tool window is used when the project files are under source code version control, allowing access to Git repositories and code change history.
- **Notifications (K)** - Android Studio occasionally displays notification popups for events such as project build completion or the successful launch of an app on a device or emulator. The Notifications tool window provides a central location to review the notification history.
- **Gradle (L)** - The Gradle tool window provides a view of the Gradle tasks that make up the project build configuration. The window lists the tasks involved in compiling the various elements of the project into an executable application. Right-click on a top-level Gradle task and select the *Open Gradle Config* menu option to load the Gradle build file for the current project into the editor. Gradle will be covered in greater detail later in this book.
- **Device Manager (M)** - Provides access to the Device Manager tool window where physical Android device connections and emulators may be added, removed, and managed.
- **Running Devices (N)** - Contains the AVD emulator if the option has been enabled to run the emulator in a tool window as outlined in the chapter entitled “*Creating an Android Virtual Device (AVD) in Android Studio*”.
- **Gemini (O)** - Android Studio’s AI powered coding assistant. Currently in preview, this tool helps you develop your app by providing coding suggestions and solutions.
- **Assistant (P)** - Display the Assistant panel, the content of which will differ depending on which Android Studio feature you are currently using.
- **App Inspection** - Provides access to the Database and Background Task inspectors. The Database Inspector allows you to inspect, query, and modify your app’s databases while running. The Background Task Inspector allows background worker tasks created using WorkManager to be monitored and managed.
- **Bookmarks** - The Bookmarks tool window provides quick access to bookmarked files and code lines. For example, right-clicking on a file in the project view allows access to an Add to Bookmarks menu option. Similarly, you can bookmark a line of code in a source file by moving the cursor to that line and pressing the F11 key (F3 on macOS). All bookmarked items can be accessed through this tool window.
- **Build Variants** - The build variants tool window provides a quick way to configure different build targets for the current application project (for example, different builds for debugging and release versions of the application or multiple builds to target different device categories).
- **Device File Explorer** - Available via the *View -> Tool Windows -> Device File Explorer* menu, this tool window provides direct access to the filesystem of the currently connected Android device or emulator, allowing the filesystem to be browsed and files copied to the local filesystem.

- **Layout Inspector** - Provides a visual 3D rendering of the hierarchy of components that make up a user interface layout.
- **Structure** – The structure tool provides a high-level view of the structure of the source file currently displayed in the editor. This information includes a list of items such as classes, methods, and variables in the file. Selecting an item from the structure list will take you to that location in the source file in the editor window.
- **TODO** – As the name suggests, this tool provides a place to review items that have yet to be completed on the project. Android Studio compiles this list by scanning the source files that make up the project to look for comments that match specified TODO patterns. These patterns can be reviewed and changed by opening the Settings dialog and navigating to the *TODO* entry listed under *Editor*.

6.5 The Tool Window Menus

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

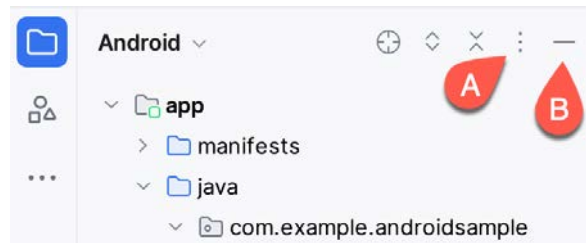


Figure 6-11

The Options menu allows various aspects of the window to be changed. Figure 6-12, for example, shows the Options menu for the Project tool window. Settings are available, for example, to undock a window and to allow it to float outside of the boundaries of the Android Studio main window, and to move and resize the tool panel:

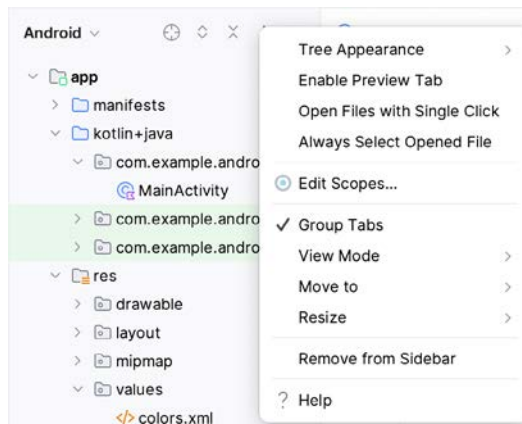


Figure 6-12

All tool windows also include a far-right button on the toolbar (marked B in Figure 6-11 above), providing an additional way to hide the tool window from view. A search of the items within a tool window can be performed by giving that window focus by clicking on it and then typing the search term (for example, the name of a file in the Project tool window). A search box will appear in the window's toolbar, and items matching the search highlighted.

6.6 Android Studio Keyboard Shortcuts

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

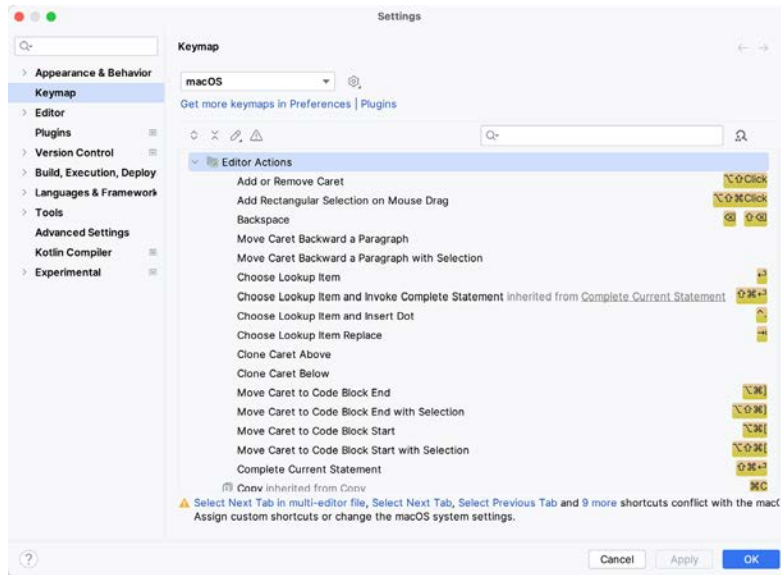


Figure 6-13

6.7 Switcher and Recent Files Navigation

Another useful mechanism for navigating within the Android Studio main window involves using the *Switcher*. Accessed via the Ctrl-Tab keyboard shortcut, the switcher appears as a panel listing both the tool windows and currently open files (Figure 6-14).

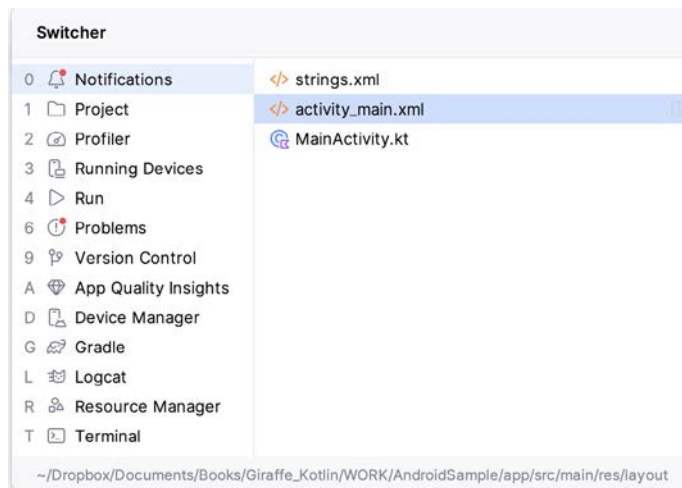


Figure 6-14

Once displayed, the switcher will remain visible as long as the Ctrl key remains depressed. Repeatedly tapping the Tab key while holding down the Ctrl key will cycle through the various selection options while releasing the

Ctrl key causes the currently highlighted item to be selected and displayed within the main window.

In addition to the Switcher, the Recent Files panel provides navigation to recently opened files (Figure 6-15). This can be accessed using the Ctrl-E keyboard shortcut (Cmd-E on macOS). Once displayed, either the mouse pointer can be used to select an option, or the keyboard arrow keys can be used to scroll through the file name and tool window options. Pressing the Enter key will select the currently highlighted item:

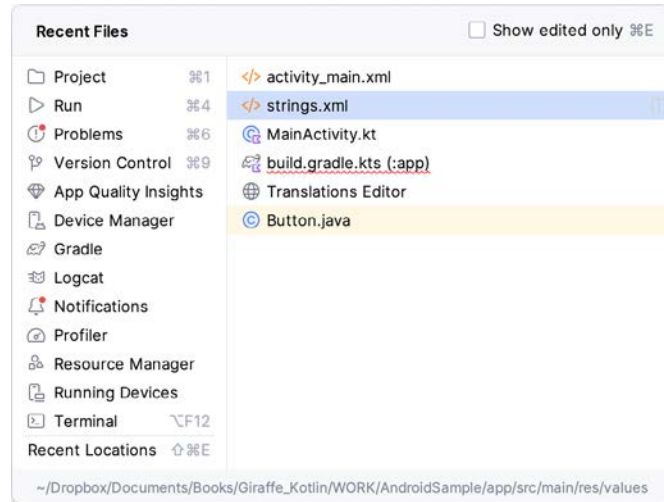


Figure 6-15

6.8 Changing the Android Studio Theme

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

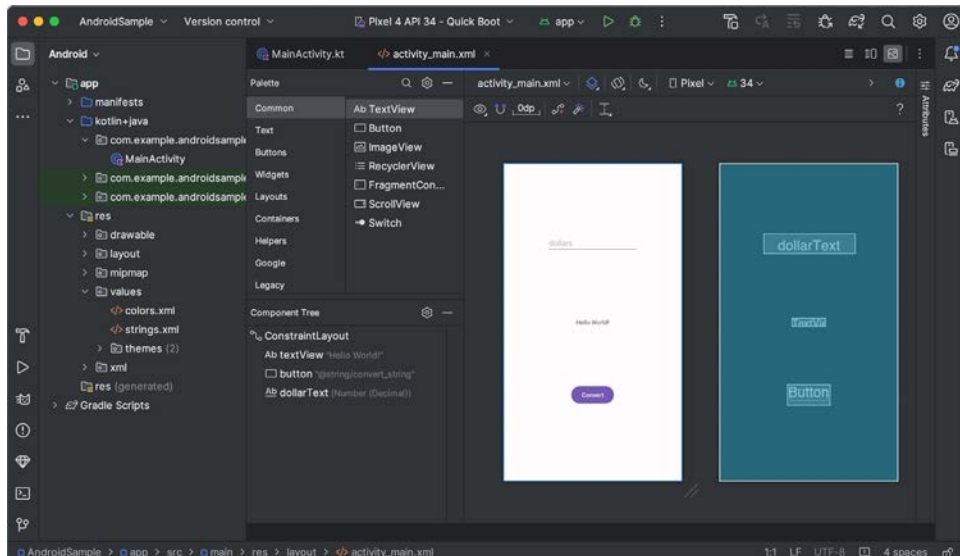


Figure 6-16

A Tour of the Android Studio User Interface

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

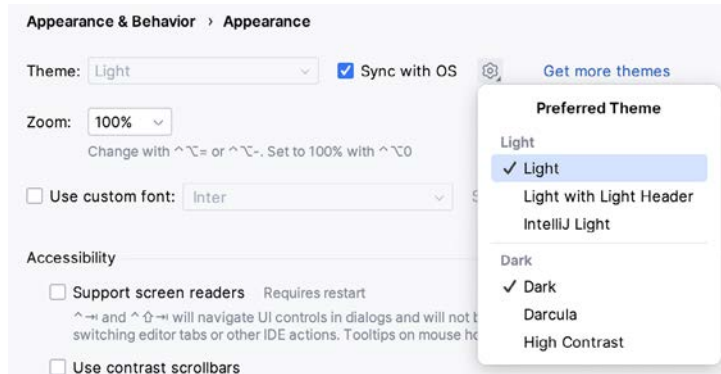


Figure 6-17

Hundreds of additional themes are available for download in the Android Studio Marketplace, which can be accessed by clicking on the *Get more themes* link.

6.9 Summary

The primary elements of the Android Studio environment consist of the welcome screen and main window. Each open project is assigned its own main window, which, in turn, consists of a menu bar, toolbar, editing and design area, status bar, and a collection of tool windows. Tool windows appear on the sides of the main window.

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

41. 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.

41.1 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.

41.2 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.

41.3 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.

41.4 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.

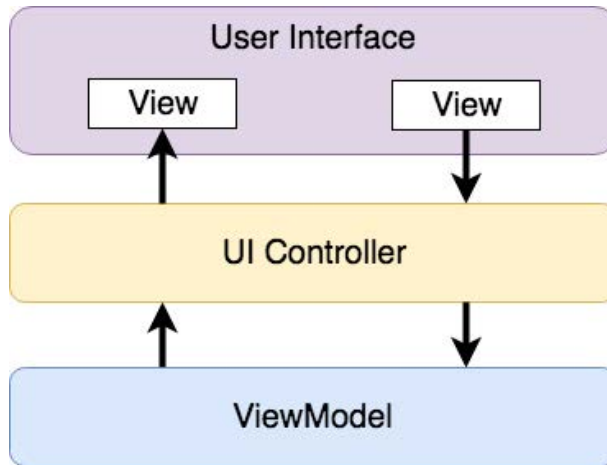


Figure 41-1

41.5 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.

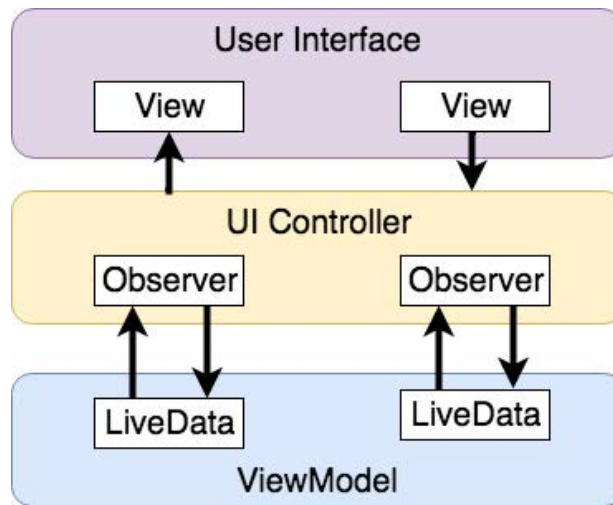


Figure 41-2

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.

41.6 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.

41.7 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.

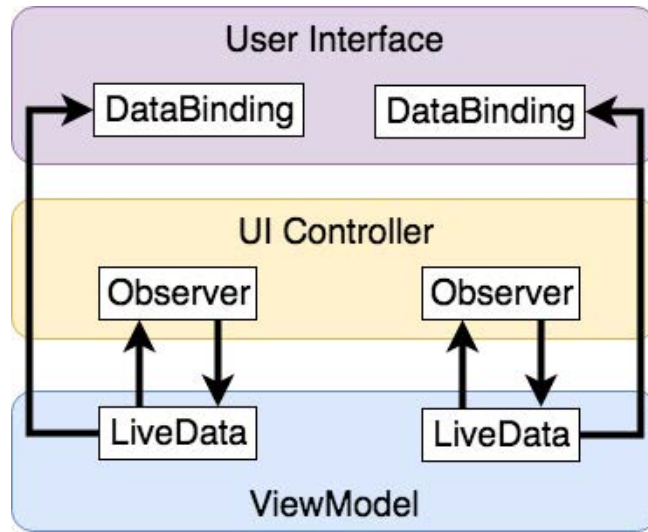


Figure 41-3

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

41.8 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.

41.9 Repository Modules

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

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

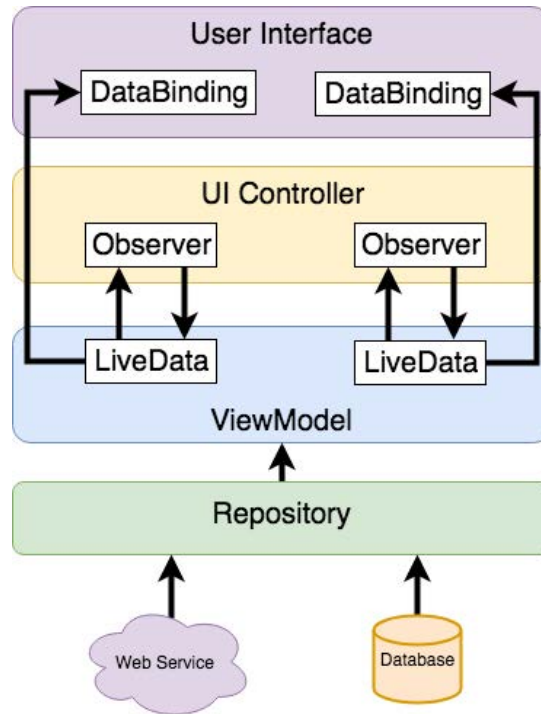


Figure 41-4

41.10 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.

42. 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.

42.1 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.

42.2 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 Kotlin.

42.3 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.kt* file and remove the floating action button code from the onCreate method as follows:

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

```

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


```

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

```

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

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


```

42.4 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 42-1, and accept the default settings in the resulting dialog:

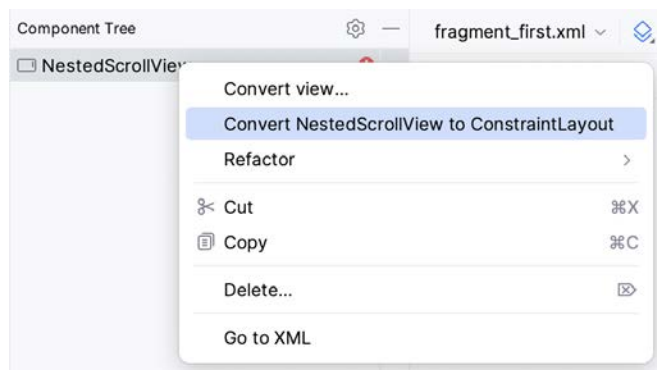


Figure 42-1

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 42-2 (note that the three views have been constrained using a vertical chain):

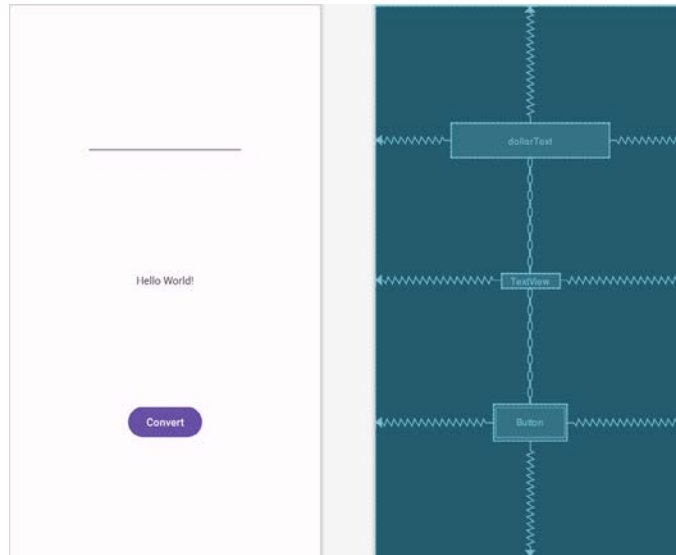


Figure 42-2

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

42.5 Implementing the View Model

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

```
package com.ebookfrenzy.viewmodeldemo

import androidx.lifecycle.ViewModel

class MainViewModel : ViewModel() {

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

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

```

    fun getResult(): Float {
        return result
    }
}

```

The class declares variables to store the current dollar string value and the converted amount together with getter and setter methods to provide access to those data values. When called, the `setAmount()` method takes 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.

42.6 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:

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

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

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

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

```

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

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

```

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

42.7 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.kt` file as outlined below:

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

    binding.convertButton.setOnClickListener {

    }
}
```

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

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

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

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

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

42.9 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.

42.10 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.

63. An Introduction to Kotlin Coroutines

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

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

63.1 What are Coroutines?

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

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

63.2 Threads vs. Coroutines

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

To avoid these overheads, instead of starting a new thread for each coroutine and destroying it when the coroutine exits, Kotlin maintains a pool of active threads and manages how coroutines are assigned to those threads. When an active coroutine is suspended, the Kotlin runtime saves it, and another coroutine resumes to take its place. When the coroutine is resumed, it is restored to an existing unoccupied thread within the pool to continue executing until it either completes or is suspended. Using this approach, a limited number of threads are used efficiently to execute asynchronous tasks with the potential to perform large numbers of concurrent

tasks without the inherent performance degeneration that would occur using standard multi-threading.

63.3 Coroutine Scope

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

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

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

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

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

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

```
myCoroutineScope.cancel()
```

63.4 Suspend Functions

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

The following is an example suspend function:

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

63.5 Coroutine Dispatchers

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

- **Dispatchers.Main** – Runs the coroutine on the main thread and is suitable for coroutines that need to make changes to the UI and as a general-purpose option for performing lightweight tasks.
- **Dispatchers.IO** – Recommended for coroutines that perform network, disk, or database operations.

- **Dispatchers.Default** – Intended for CPU-intensive tasks such as sorting data or performing complex calculations.

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

63.6 Coroutine Builders

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

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

63.7 Jobs

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

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

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

method.

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

63.8 Coroutines – Suspending and Resuming

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

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

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

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

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

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

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

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

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

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

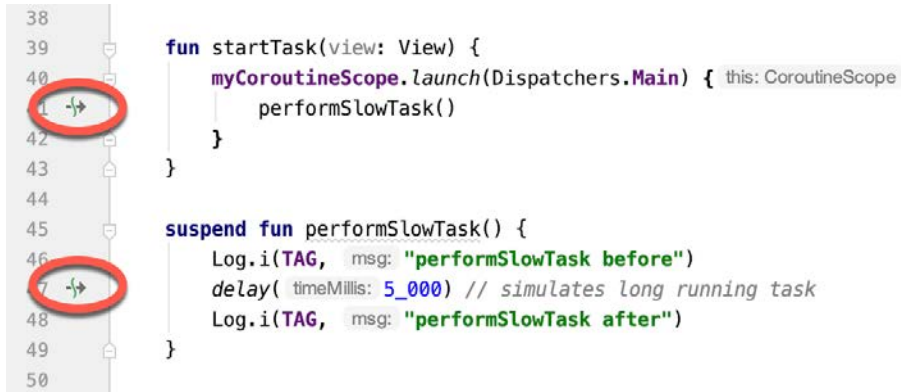


Figure 63-1

63.9 Returning Results from a Coroutine

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

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

```

fun startTask(view: View) {

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

```

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

```

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

```

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

63.10 Using withContext

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

An Introduction to Kotlin Coroutines

designed to call multiple functions from within a suspend function:

```
fun startTask(view: View) {

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

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

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

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

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

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

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

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

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

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

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

Task 3 main

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

```
fun startTask(view: View) {

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

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

        return@withContext "Finished"
    }
}
```

63.11 Coroutine Channel Communication

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

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

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

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

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

An Introduction to Kotlin Coroutines

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

When executed, the following logcat output will be generated:

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

63.12 Summary

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

80. An Android Picture-in-Picture Tutorial

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

80.1 Adding Picture-in-Picture Support to the Manifest

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

```

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

```

80.2 Adding a Picture-in-Picture Button

As currently designed, the layout for the VideoPlayer activity consists solely of a VideoView instance. As currently designed, the layout for the VideoPlayer activity consists solely of a VideoView instance. A button will now be added to the layout to switch to PiP mode. Load the *activity_main.xml* file into the layout editor and drag a Button object from the palette onto the layout so that it is positioned as shown in Figure 80-1:

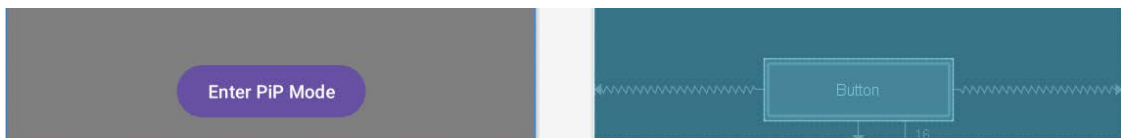


Figure 80-1

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

80.3 Entering Picture-in-Picture Mode

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

```

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

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

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

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

```

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

Compile and run the app on a device or emulator running Android version 8 or newer and wait for video playback to begin before clicking on the PiP mode button. The video playback should minimize and appear in the PiP window as shown in Figure 80-2:

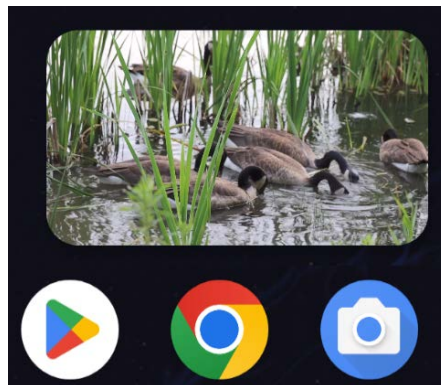


Figure 80-2

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

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

80.4 Detecting Picture-in-Picture Mode Changes

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

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

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

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

80.5 Adding a Broadcast Receiver

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

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

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

```

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

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

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

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

```

80.6 Adding the PiP Action

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

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

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

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

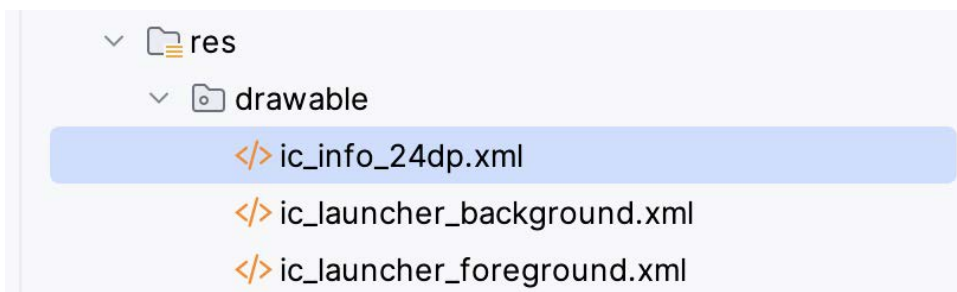


Figure 80-3

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

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

```

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

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

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

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

```

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

```

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

    val actions = ArrayList<RemoteAction>()

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

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

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

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

```

An Android Picture-in-Picture Tutorial

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

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

```
private fun createPipAction() {

    val actions = ArrayList<RemoteAction>()

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

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

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

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

    actions.add(remoteAction)

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

    setPictureInPictureParams(params)
}
```

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

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

80.7 Testing the Picture-in-Picture Action

Rerun the app and place the activity into PiP mode. Tap on the PiP window so that the new action button appears, as shown in Figure 80-4:

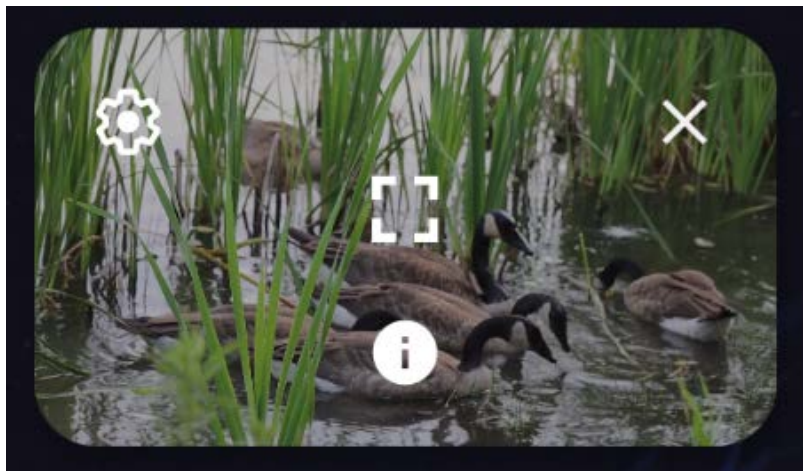


Figure 80-4

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

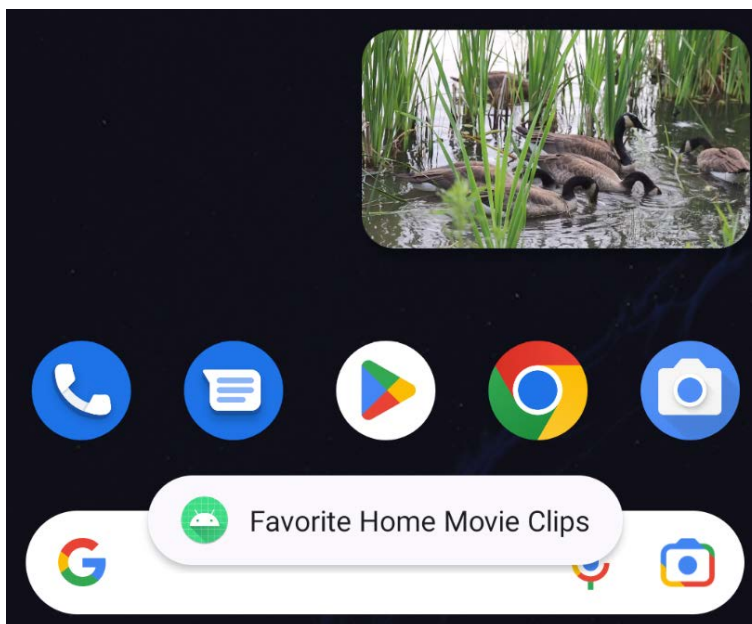


Figure 80-5

80.8 Summary

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

93. An Overview of Android In-App Billing

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

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

93.1 Preparing a Project for In-App Purchasing

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

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

The app must then be uploaded to the console and enabled for in-app purchasing. However, the console will not activate in-app purchasing support for an app unless the Google Play Billing Library has been added to the module-level *build.gradle.kts* file:

```
dependencies {  
    .  
    .  
    implementation(libs.billingclient.ktx)  
    .  
    .  
}
```

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

93.2 Creating In-App Products and Subscriptions

Products and subscriptions are created and managed using the options listed beneath the Monetize section of the Play Console navigation panel, as highlighted in Figure 93-1 below:

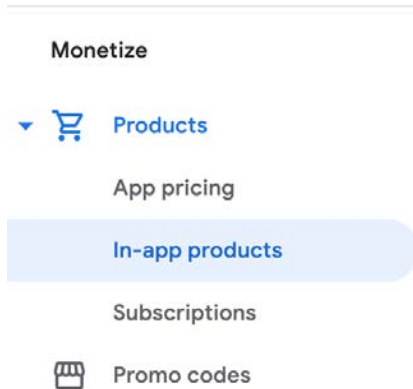


Figure 93-1

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

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

93.3 Billing Client Initialization

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

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

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

```

        // Handle errors here
    }
}

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

```

93.4 Connecting to the Google Play Billing Library

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

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

```

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

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

```

93.5 Querying Available Products

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

```

val queryProductDetailsParams = QueryProductDetailsParams.newBuilder()

```

An Overview of Android In-App Billing

```
.setProductList(  
    ImmutableList.of(  
        QueryProductDetailsParams.Product.newBuilder()  
            .setProductId(productId)  
            .setProductType(  
                BillingClient.ProductType.INAPP  
            )  
            .build()  
    )  
)  
.build()
```

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

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

93.6 Starting the Purchase Process

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

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

```
billingClient.launchBillingFlow(this, billingFlowParams)
```

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

93.7 Completing the Purchase

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

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

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

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

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

val acknowledgePurchaseResponseListener = AcknowledgePurchaseResponseListener {
    // Check acknowledgement result
}

billingClient.acknowledgePurchase(
    acknowledgePurchaseParams,
    acknowledgePurchaseResponseListener
)
```

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

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

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

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

}

93.8 Querying Previous Purchases

When working with in-app billing, checking whether a user has already purchased a product or subscription is a common requirement. A list of all the user's previous purchases of a specific type can be generated by calling the `queryPurchasesAsync()` method of the `BillingClient` instance and implementing a `PurchaseResponseListener`. The following code, for example, obtains a list of all previously purchased items that have not yet been consumed:

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

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

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

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

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

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

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

93.9 Summary

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

Index

Symbols

? . 97
 <application> 510
 <fragment> 301
 <fragment> element 301
 <provider> 567
 <receiver> 488
 <service> 510, 516, 523
 :: operator 99
 .well-known folder 461, 484, 726

A

AbsoluteLayout 178
 ACCESS_COARSE_LOCATION permission 636
 ACCESS_FINE_LOCATION permission 636
 acknowledgePurchase() method 765
 ACTION_CREATE_DOCUMENT 787
 ACTION_CREATE_INTENT 788
 ACTION_DOWN 278
 ACTION_MOVE 278
 ACTION_OPEN_DOCUMENT intent 780
 ACTION_POINTER_DOWN 278
 ACTION_POINTER_UP 278
 ACTION_UP 278
 ACTION_VIEW 479
 Active / Running state 154
 Activity 83, 157
 adding views in Java code 255
 class 157
 creation 14
 Entire Lifetime 161
 Foreground Lifetime 161
 lifecycle methods 159
 lifecycles 151
 returning data from 458

 state change example 165
 state changes 157
 states 154
 Visible Lifetime 161
 Activity Lifecycle 153
 Activity Manager 82
 ActivityResultLauncher 459
 Activity Stack 153
 Actual screen pixels 246
 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 487
 addMarker() method 689
 addView() method 249
 ADD_VOICEMAIL permission 636
 android
 exported 511
 gestureColor 294
 layout_behavior property 451
 onClick 303
 process 511, 523
 uncertainGestureColor 294
 Android
 Activity 83
 architecture 79
 events 271

Index

- intents 84
- onClick Resource 271
- runtime 80
- SDK Packages 5
- android.app 80
- Android Architecture Components 317
- android.content 80
- android.content.Intent 457
- android.database 80
- Android Debug Bridge. *See* ADB
- Android Development
 - System Requirements 3
- Android Devices
 - designing for different 177
- android.graphics 81
- android.hardware 81
- android.intent.action 493
- android.intent.action.BOOT_COMPLETED 511
- android.intent.action.MAIN 479
- android.intent.category.LAUNCHER 479
- Android Libraries 80
- android.media 81
- Android Monitor tool window 32
- Android Native Development Kit 281
- android.net 81
- android.opengl 81
- android.os 81
- android.permission.RECORD_AUDIO 645
- android.print 81
- Android Project
 - create new 13
- android.provider 81
- 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 79
- Android Storage Access Framework 780
- 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 81
- android.util 81
- android.view 81
- android.view.View 180
- android.view.ViewGroup 177, 180
- Android Virtual Device. *See* AVD
 - overview 27
- Android Virtual Device Manager 27
- android.webkit 81
- android.widget 81
- AndroidX libraries 816
- API Key 681
- APK analyzer 758
- APK file 751
- APK File
 - analyzing 758
- APK Signing 816
- APK Wizard dialog 750
- App Architecture
 - modern 317
- AppBar
 - anatomy of 449
- AppBarScrollingViewBehavior 451

- App Bundles 747
 - creating 751
 - overview 747
 - revisions 757
 - uploading 754
- AppCompatActivity class 158
- App Inspector 53
- Application
 - stopping 32
- Application Context 85
- Application Framework 82
- Application Manifest 85
- Application Resources 85
- App Link
 - Adding Intent Filter 734
 - Digital Asset Links file 726, 461
 - Intent Filter Handling 734
 - Intent Filters 725
 - Intent Handling 726
 - Testing 738
 - URL Mapping 731
- App Links 725
 - auto verification 460
 - autoVerify 461
 - overview 725
- Apply Changes 263
 - Apply Changes and Restart Activity 263
 - Apply Code Changes 263
 - fallback settings 265
 - options 263
 - Run App 263
 - tutorial 265
- applyToActivitiesIfAvailable() method 812
- Architecture Components 317
- ART 80
- as 99
- as? 99
- asFlow() builder 529
- assetlinks.json , 726, 461
- asSharedFlow() 538
- asStateFlow() 537
- async 497

- Attribute Keyframes 388
- Audio
 - supported formats 643
- Audio Playback 643
- Audio Recording 643
- Auto Blocker 60
- Autoconnect Mode 211
- Automatic Link Verification 460, 483
- autoVerify 461, 734
- 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 152
- Barriers 204
 - adding 223
 - constrained views 204
- Baseline Alignment 203
- beginTransaction() method 302
- BillingClient 766
 - acknowledgePurchase() method 765
 - consumeAsync() method 765
 - getPurchaseState() method 765
 - initialization 762, 771
 - launchBillingFlow() method 764
 - queryProductDetailsAsync() method 763
 - queryPurchasesAsync() method 766
- BillingResult 778

Index

- getDebugMessage() 778
 - Binding Expressions 337
 - one-way 337
 - two-way 338
 - BIND_JOB_SERVICE permission 511
 - bindService() method 509, 513, 517
 - Biometric Authentication 739
 - callbacks 743
 - overview 739
 - tutorial 739
 - Biometric Prompt 744
 - BitmapFactory 782
 - Bitwise AND 105
 - Bitwise Inversion 104
 - Bitwise Left Shift 106
 - Bitwise OR 105
 - Bitwise Right Shift 106
 - Bitwise XOR 105
 - black activity 14
 - Blank template 181
 - Blueprint view 209
 - BODY_SENSORS permission 636
 - Boolean 92
 - Bound Service 509, 513
 - adding to a project 514
 - Implementing the Binder 514
 - Interaction options 513
 - BoundService class 515
 - Broadcast Intent 487
 - example 489
 - overview 84, 487
 - sending 490
 - Sticky 489
 - Broadcast Receiver 487
 - adding to manifest file 492
 - creation 491
 - overview 84, 488
 - BroadcastReceiver class 488
 - BroadcastReceiver superclass 491
 - BufferedReader object 790
 - buffer() operator 531
 - Build Variants , 54
 - tool window 54
 - Bundle class 174
 - Bundled Notifications 664
- ## C
- Calendar permissions 636
 - CALL_PHONE permission 636
 - CAMERA permission 636
 - Camera permissions 636
 - CameraUpdateFactory class
 - methods 690
 - cancelAndJoin() 497
 - cancelChildren() 497
 - CancellationSignal 744
 - Canvas class 720
 - CardView
 - layout file 439
 - responding to selection of 447
 - CardView class 439
 - CATEGORY_OPENABLE 780
 - C/C++ Libraries 81
 - Chain bias 232
 - chain head 202
 - chains 202
 - Chains
 - creation of 229
 - Chain style
 - changing 231
 - chain styles 202
 - Char 92
 - CheckBox 177
 - checkSelfPermission() method 640
 - Circle class 677
 - Code completion 70
 - Code Editor
 - basics 67
 - Code completion 70
 - Code Generation 72
 - 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 72
- Code Reformatting 75
- code samples
 - download 1
- cold boot 44
- Cold flows 537
- CollapsingToolBarLayout
 - example 452
 - introduction 452
 - parallax mode 452
 - pin mode 452
 - setting scrim color 455
 - setting title 455
 - with image 452
- collectLatest() operator 530
- Color class 721
- COLOR_MODE_COLOR 696, 716
- COLOR_MODE_MONOCHROME 696, 716
- combine() operator 536
- Common Gestures 283
 - detection 283
- Communicating Sequential Processes 495
- Companion Objects 129
- Component tree 17
- conflate() operator 531
- Constraint Bias 201
 - adjusting 215
- ConstraintLayout
 - advantages of 207
 - Availability 208
 - Barriers 204
 - Baseline Alignment 203
 - chain bias 232
 - chain head 202
 - chains 202
 - chain styles 202
 - Constraint Bias 201
- Constraints 199
 - conversion to 227
 - convert to MotionLayout 395
 - deleting constraints 214
 - guidelines 221
 - Guidelines 204
 - manual constraint manipulation 211
 - Margins 200, 215
 - Opposing Constraints 200, 217
 - overview of 199
 - Packed chain 203, 232
 - ratios 207, 233
 - Spread chain 202
 - Spread inside 232
 - Spread inside chain 202
 - tutorial 237
 - using in Android Studio 209
 - Weighted chain 202, 232
 - Widget Dimensions 203, 219
 - Widget Group Alignment 225
- ConstraintLayout chains
 - creation of 229
 - in layout editor 229
- ConstraintLayout Chain style
 - changing 231
- Constraints
 - deleting 214
- ConstraintSet
 - addToHorizontalChain() method 252
 - addToVerticalChain() method 252
 - alignment constraints 251
 - apply to layout 250
 - applyTo() method 250
 - centerHorizontally() method 251
 - centerVertically() method 251
 - chains 251
 - clear() method 252
 - clone() method 251
 - connect() method 250
 - connect to parent 250
 - constraint bias 251
 - copying constraints 251
 - create 250

Index

- create connection 250
- createHorizontalChain() method 251
- createVerticalChain() method 251
- guidelines 252
- removeFromHorizontalChain() method 252
- removeFromVerticalChain() method 252
- removing constraints 252
- rotation 253
- scaling 252
- setGuidelineBegin() method 252
- setGuidelineEnd() method 252
- setGuidelinePercent() method 252
- setHorizontalBias() method 251
- setRotationX() method 253
- setRotationY() method 253
- setScaleX() method 252
- setScaleY() method 252
- setTransformPivot() method 253
- setTransformPivotX() method 253
- setTransformPivotY() method 253
- setVerticalBias() method 251
- sizing constraints 251
- tutorial 255
- view IDs 257
- ConstraintSet class 249, 250
- Constraint Sets 250
- ConstraintSets
 - configuring 384
- consumeAsync() method 765
- ConsumeParams 775
- Contacts permissions 636
- container view 177
- Content Provider 82, 565, 581
 - <provider> 567
 - accessing 581
 - Authority 571
 - client tutorial 581
 - ContentProvider class 565
 - Content Resolver 566
 - ContentResolver 578
 - content URI 566
 - Content URI 571, 581
 - ContentValues 573
 - delete() 566, 576
 - getType() 566
 - insert() 565, 573
 - onCreate() 565, 573
 - overview 85
 - query() 565, 574
 - tutorial 569
 - update() 566, 575
 - UriMatcher 572
 - UriMatcher class 566
- ContentProvider class 565
- Content Resolver 566
 - getContentResolver() 566
- ContentResolver 578
 - getContentResolver() 566
- content URI 566
- Content URI 566, 571
- ContentValues 573
- Context class 85
- CoordinatorLayout 178, 451
- Coroutine Builders 497
 - async 497
 - coroutineScope 497
 - launch 497
 - runBlocking 497
 - supervisorScope 497
 - withContext 497
- Coroutine Dispatchers 496
- Coroutines 495, 527
 - channel communication 501
 - GlobalScope 496
 - returning results 499
 - Suspend Functions 496
 - suspending 498
 - tutorial 503
 - ViewModelScope 496
 - vs. Threads 495
- coroutineScope 497
- Coroutine Scope 496
- createPrintDocumentAdapter() method 711
- Custom Accessors 127

- Custom Attribute 385
- Custom Document Printing 699, 711
- Custom Gesture
 - recognition 289
- Custom Print Adapter
 - implementation 713
- Custom Print Adapters 711
- Custom Theme
 - building 803
- Cycle Editor 413
- Cycle Keyframe 393
- Cycle Keyframes
 - overview 409

D

- dangerous permissions
 - list of 636
- Data Access Object (DAO) 586
- Database Inspector 592, 616
 - live updates 616
 - SQL query 616
- Database Rows 552
- Database Schema 551
- Database Tables 551
- Data binding
 - binding expressions 337
- Data Binding 319
 - binding classes 336
 - enabling 342
 - event and listener binding 338
 - key components 333
 - overview 333
 - tutorial 341
 - variables 336
 - with LiveData 319
- DDMS 32
- Debugging
 - enabling on device 59
- debug.keystore file 461, 483
- Default Function Parameters 119
- DefaultLifecycleObserver 354, 357
- deltaRelative 390

- Density-independent pixels 245
- Density Independent Pixels
 - converting to pixels 260
- Device Definition
 - custom 195
- Device File Explorer 54
- device frame 35
- Device Mirroring 65
 - enabling 65
- device pairing 63
- Digital Asset Links file 726, 461, 461
- Direct Reply Input 673
- Dispatchers.Default 497
- Dispatchers.IO 496
- Dispatchers.Main 496
- document provider 779
- dp 245
- DROP_LATEST 539
- DROP_OLDEST 539
- Dynamic Colors
 - applyToActivitiesIfAvailable() method 812
 - enabling in Android 811
- Dynamic State 159
 - saving 173

E

- Elvis Operator 99
- Empty Process 153
- Empty template 181
- 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

Index

- 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 765
- enabling ADB support 59
- Escape Sequences 93
- Event Handling 271
 - example 272
- Event Listener 273
- Event Listeners 272
- Events
 - consuming 275
- execSQL() 560
- explicit
 - intent 84
- explicit intent 457
- Explicit Intent 457
- Extended Control
 - options 41
- F**
- Files
 - switching between 68
- filter() operator 532
- findPointerIndex() method 278
- findViewById() 139
- Fingerprint
 - emulation 44
- Fingerprint authentication
 - device configuration 740
 - permission 740
 - steps to implement 739
- Fingerprint Authentication
 - overview 739
 - tutorial 739
- FLAG_INCLUDE_STOPPED_PACKAGES 487
- flatMapConcat() operator 535
- flatMapMerge() operator 535
- flexible space area 449
- Float 92
- floating action button 14, 182
 - changing appearance of 424
 - margins 422
 - removing 183
 - sizes 422
- Flow 527
 - asFlow() builder 529
 - asSharedFlow() 538
 - asStateFlow() 537
 - background handling 547
 - buffering 531
 - buffer() operator 531
 - cold 537
 - collect() 529
 - collecting data 529
 - collectLatest() operator 530
 - combine() operator 536
 - conflate() operator 531
 - declaring 528
 - emit() 529
 - emitting data 529
 - filter() operator 532
 - flatMapConcat() operator 535
 - flatMapMerge() operator 535
 - flattening 534
 - flowOf() builder 529
 - flow of flows 534
 - fold() operator 534
 - hot 537
 - intermediate operators 532
 - library requirements 528
 - map() operator 532
 - MutableSharedFlow 538
 - MutableStateFlow 537
 - onEach() operator 536
 - reduce() operator 534
 - repeatOnLifecycle 548
 - SharedFlow 538

- single() operator 531
 - StateFlow 537
 - terminal flow operators 534
 - transform() operator 533
 - try/finally 530
 - zip() operator 536
 - flowOf() builder 529
 - flow of flows 534
 - Flow operators 532
 - Flows
 - combining 536
 - Introduction to 527
 - Foldable Devices 162
 - multi-resume 162
 - Foreground Process 152
 - Forward-geocoding 683
 - Fragment
 - creation 299
 - event handling 303
 - XML file 300
 - FragmentActivity class 158
 - Fragment Communication 303
 - Fragments 299
 - adding in code 302
 - duplicating 430
 - example 307
 - overview 299
 - FragmentManager class 433
 - FrameLayout 178
 - Function Parameters
 - variable number of 119
 - Functions 117
- G**
- Gemini 145
 - asking questions 148
 - configuration 147
 - enabling 145
 - in Android Studio 145
 - inline code completion 149
 - overview 145
 - playground 148
 - question context 149
 - tool window 146
 - Geocoder object 684
 - Geocoding 682
 - Gesture Builder Application 289
 - building and running 289
 - Gesture Detector class 283
 - GestureDetectorCompat
 - instance creation 286
 - GestureDetectorCompat class 283
 - GestureDetector.OnDoubleTapListener 283, 284
 - GestureDetector.OnGestureListener 284
 - GestureLibrary 289
 - GestureOverlayView 289
 - configuring color 294
 - configuring multiple strokes 294
 - GestureOverlayView class 289
 - GesturePerformedListener 289
 - Gestures
 - interception of 294
 - Gestures File
 - creation 290
 - extract from SD card 290
 - loading into application 292
 - GET_ACCOUNTS permission 636
 - getAction() method 493
 - getContentResolver() 566
 - getDebugMessage() 778
 - getFromLocation() method 684
 - getId() method 250
 - getIntent() method 458
 - getPointerCount() method 278
 - getPointerId() method 278
 - getPurchaseState() method 765
 - getService() method 517
 - getWritableDatabase() 560
 - GlobalScope 496
 - GNU/Linux 80
 - Google Cloud
 - billing account 678
 - new project 679
 - Google Cloud Print 694

Index

Google Drive 780
 printing to 694
GoogleMap 677
 map types 687
GoogleMap.MAP_TYPE_HYBRID 687
GoogleMap.MAP_TYPE_NONE 687
GoogleMap.MAP_TYPE_NORMAL 687
GoogleMap.MAP_TYPE_SATELLITE 687
GoogleMap.MAP_TYPE_TERRAIN 687
Google Maps Android API 677
 Controlling the Map Camera 690
 displaying controls 688
 Map Markers 689
 overview 677
Google Maps SDK 677
 API Key 681
 Credentials 681
 enabling 680
 Maps SDK for Android 681
Google Play App Signing 750
Google Play Console 769
 Creating an in-app product 769
 License Testers 770
Google Play Developer Console 748
Gradle
 APK signing settings 820
 Build Variants 816
 command line tasks 821
 dependencies 815
 Manifest Entries 816
 overview 815
 sensible defaults 815
Gradle Build File
 top level 817
Gradle Build Files
 module level 818
gradle.properties file 816
GridLayout 178
GridLayoutManager 437

H

HAL 80

Handler class 522
Hardware Abstraction Layer 80
Higher-order Functions 121
Hot flows 537
HP Print Services Plugin 693
HTML printing 697
HTML Printing
 example 701

I

IBinder 509, 515
IBinder object 513, 522
Image Printing 696
Immutable Variables 94
implicit
 intent 84
implicit intent 457
Implicit Intent 459
Implicit Intents
 example 475
importance hierarchy 151
in 245
INAPP 766
In-App Products 761
In-App Purchasing 767
 acknowledgePurchase() method 765
 BillingClient 762
 BillingResult 778
 consumeAsync() method 765
 ConsumeParams 775
 Consuming purchases 775
 enablePendingPurchases() method 765
 getPurchaseState() method 765
 launchBillingFlow() method 764
 Libraries 767
 newBuilder() method 762
 onBillingServiceDisconnected() callback 772
 onBillingServiceDisconnected() method 763
 onBillingSetupFinished() listener 772
 onProductDetailsResponse() callback 772
 Overview 761
 ProductDetail 764

- ProductDetails 773
- products 761
- ProductType 766
- Purchase Flow 773
- PurchaseResponseListener 766
- PurchasesUpdatedListener 765
- PurchaseUpdatedListener 774
- purchase updates 774
- queryProductDetailsAsync() 772
- queryProductDetailsAsync() method 763
- queryPurchasesAsync() 776
- queryPurchasesAsync() method 766
- runOnUiThread() 773
- subscriptions 761
- tutorial 767
- Initializer Blocks 127
- In-Memory Database 592
- Inner Classes 128
- IntelliJ IDEA 87
- Intent 84
 - explicit 84
 - implicit 84
- Intent Availability
 - checking for 464
- Intent.CATEGORY_OPENABLE 788
- Intent Filters 460
 - App Link 725
- Intents 457
 - ActivityResultLauncher 459
 - overview 457
 - registerForActivityResult() 459, 472
- Intent Service 509
- Intent URL 477
- intermediate flow operators 532
- is 99
- isInitialized property 99

J

- Java
 - convert to Kotlin 87
- Java Native Interface 81
- JetBrains 87

- Jetpack 317
 - overview 317
- JobIntentService 509
 - BIND_JOB_SERVICE permission 511
 - onHandleWork() method 509
- join() 497

K

- KeyAttribute 388
- Keyboard Shortcuts 56
- KeyCycle 409
 - Cycle Editor 413
 - tutorial 409
- Keyframe 402
- Keyframes 388
- KeyFrameSet 418
- KeyPosition 389
 - deltaRelative 390
 - parentRelative 389
 - pathRelative 390
- Keystore File
 - creation 750
- KeyTimeCycle 409
- keytool 461
- KeyTrigger 392
- Killed state 154
- Kotlin
 - accessing class properties 127
 - and Java 87
 - arithmetic operators 101
 - assignment operator 101
 - augmented assignment operators 102
 - bitwise operators 104
 - Boolean 92
 - break 112
 - breaking from loops 111
 - calling class methods 127
 - Char 92
 - class declaration 123
 - class initialization 124
 - class properties 124
 - Companion Objects 129

Index

- conditional control flow 113
- continue labels 112
- continue statement 112
- control flow 109
- convert from Java 87
- Custom Accessors 127
- data types 91
- decrement operator 102
- Default Function Parameters 119
- defining class methods 124
- do ... while loop 111
- Elvis Operator 99
- equality operators 103
- Escape Sequences 93
- expression syntax 101
- Float 92
- Flow 527
- for-in statement 109
- function calling 118
- Functions 117
- Higher-order Functions 121
- if ... else ... expressions 114
- if expressions 113
- Immutable Variables 94
- increment operator 102
- inheritance 133
- Initializer Blocks 127
- Inner Classes 128
- introduction 87
- Lambda Expressions 120
- let Function 97
- Local Functions 118
- logical operators 103
- looping 109
- Mutable Variables 94
- Not-Null Assertion 97
- Nullable Type 96
- Overriding inherited methods 136
- playground 88
- Primary Constructor 124
- properties 127
- range operator 104

- Safe Call Operator 96
- Secondary Constructors 124
- Single Expression Functions 118
- String 92
- subclassing 133
- Type Annotations 95
- Type Casting 99
- Type Checking 99
- Type Inference 95
- variable parameters 119
- when statement 114
- while loop 110

L

- Lambda Expressions 120
- Large Language Model 145
- lateinit 98
- Late Initialization 98
- launch 497
- launchBillingFlow() method 764
- layout_collapseMode
 - parallax 454
 - pin 454
- layout_constraintDimensionRatio 234
- layout_constraintHorizontal_bias 232
- layout_constraintVertical_bias 232
- layout editor
 - ConstraintLayout chains 229
- Layout Editor 16, 237
 - Autoconnect Mode 211
 - code mode 188
 - Component Tree 185
 - design mode 185
 - device screen 185
 - example project 237
 - Inference Mode 211
 - palette 185
 - properties panel 186
 - Sample Data 194
 - Setting Properties 189
 - toolbar 186
 - user interface design 237

- view conversion 193
- Layout Editor Tool
 - changing orientation 17
 - overview 185
- Layout Inspector 55
- Layout Managers 177
- LayoutResultCallback object 717
- Layouts 177
- layout_scrollFlags
 - enterAlwaysCollapsed mode 451
 - enterAlways mode 451
 - exitUntilCollapsed mode 451
 - scroll mode 451
- Layout Validation 196
- let Function 97
- libc 81
- libs.versions.toml file 268
- License Testers 770
- Lifecycle
 - awareness 353
 - components 320
 - observers 354
 - owners 353
 - states and events 354
 - tutorial 357
- Lifecycle-Aware Components 353
- Lifecycle library 528
- Lifecycle Methods 159
- Lifecycle Observer 357
 - creating a 357
- Lifecycle Owner
 - creating a 359
- Lifecycles
 - modern 320
- Lifecycle.State.CREATED 549
- Lifecycle.State.DESTROYED 549
- Lifecycle.State.INITIALIZED 549
- Lifecycle.State.RESUMED 549
- Lifecycle.State.STARTED 549
- LinearLayout 178
- LinearLayoutManager 437
- LinearLayoutManager layout 445

- Linux Kernel 80
- list devices 59
- LiveData 318, 329
 - adding to ViewModel 329
 - observer 331
 - tutorial 329
- Live Templates 76
- LLM 145
- Local Bound Service 513
 - example 513
- Local Functions 118
- Location Manager 82
- Location permission 636
- Logcat
 - tool window 54
- LogCat
 - enabling 169

M

- MANAGE_EXTERNAL_STORAGE 637
 - adb enabling 637
 - testing 637
- Manifest File
 - permissions 479
- map() operator 532
- Maps 677
- MapView 677
 - adding to a layout 684
- Marker class 677
- Master/Detail Flow
 - creation 794
 - two pane mode 793
- match_parent properties 245
- Material design 421
- Material Design 2 801
- Material Design 2 Theming 801
- Material Design 3 801
- Material Theme Builder 803
- Material You 801
- measureTimeMillis() function 531
- MediaController
 - adding to VideoView instance 621

Index

- MediaController class 618
 - methods 618
 - MediaPlayer class 643
 - methods 643
 - MediaRecorder class 643
 - methods 644
 - recording audio 644
 - Memory Indicator 69
 - Messenger object 522
 - Microphone
 - checking for availability 646
 - Microphone permissions 636
 - mm 245
 - MotionEvent 277, 278, 297
 - getActionMasked() 278
 - MotionLayout 383
 - arc motion 388
 - Attribute Keyframes 388
 - ConstraintSets 384
 - Custom Attribute 404
 - Custom Attributes 385
 - Cycle Editor 413
 - Editor 395
 - KeyAttribute 388
 - KeyCycle 409
 - Keyframes 388
 - KeyFrameSet 418
 - KeyPosition 389
 - KeyTimeCycle 409
 - KeyTrigger 392
 - OnClick 387, 400
 - OnSwipe 387
 - overview 383
 - Position Keyframes 389
 - previewing animation 400
 - Trigger Keyframe 392
 - Tutorial 395
 - MotionScene
 - ConstraintSets 384
 - Custom Attributes 385
 - file 384
 - overview 383
 - transition 384
 - moveCamera() method 690
 - multiple devices
 - testing app on 31
 - Multiple Touches
 - handling 278
 - multi-resume 162
 - Multi-Touch
 - example 279
 - Multi-touch Event Handling 277
 - multi-window support 162
 - MutableSharedFlow 538
 - MutableStateFlow 537
 - Mutable Variables 94
 - My Location Layer 677
- ## N
- Navigation 363
 - adding destinations 372
 - overview 363
 - pass data with safeargs 379
 - passing arguments 368
 - stack 363
 - tutorial 369
 - Navigation Action
 - triggering 367
 - Navigation Architecture Component 363
 - Navigation Component
 - tutorial 369
 - Navigation Controller
 - accessing 367
 - Navigation Graph 366, 370
 - adding actions 376
 - creating a 370
 - Navigation Host 364
 - declaring 371
 - newBuilder() method 762
 - normal permissions 635
 - Notification
 - adding actions 664
 - Direct Reply Input 673
 - issuing a basic 660

- launch activity from a 662
- PendingIntent 670
- Reply Action 672
- updating direct reply 674
- Notifications
 - bundled 664
 - overview 653
- Notifications Manager 82
- Not-Null Assertion 97
- Nullable Type 96

O

- Observer
 - implementing a LiveData 331
- onAttach() method 304
- onBillingServiceDisconnected() callback 772
- onBillingServiceDisconnected() method 763
- onBillingSetupFinished() listener 772
- onBind() method 510, 513, 521
- onBindViewHolder() method 445
- OnClick 387
- onClickListener 272, 273, 276
- onClick() method 271
- onCreateContextMenuListener 272
- onCreate() method 152, 159, 510
- onCreateView() method 160
- onDestroy() method 160, 510
- onDoubleTap() method 283
- onDown() method 283
- onEach() operator 536
- onFling() method 283
- onFocusChangeListener 272
- OnFragmentInteractionListener
 - implementation 377
- onGesturePerformed() method 289
- onHandleWork() method 510
- onKeyListener 272
- onLayoutFailed() method 717
- onLayoutFinished() method 717
- onLongClickListener 272
- onLongPress() method 283
- onMapReady() method 686

- onPageFinished() callback 702
- onPause() method 160
- onProductDetailsResponse() callback 772
- onReceive() method 152, 488, 489, 491
- onRequestPermissionsResult() method 639, 650, 658, 668
- onRestart() method 159
- onRestoreInstanceState() method 160
- onResume() method 152, 160
- onSaveInstanceState() method 160
- onScaleBegin() method 295
- onScaleEnd() method 295
- onScale() method 295
- onScroll() method 283
- OnSeekBarChangeListener 314
- onServiceConnected() method 513, 516, 523
- onServiceDisconnected() method 513, 516, 523
- onShowPress() method 283
- onSingleTapUp() method 283
- onStartCommand() method 510
- onStart() method 160
- onStop() method 160
- onTouchEvent() method 283, 295
- onTouchListener 272
- onTouch() method 278
- onUpgrade() 560
- onViewCreated() method 160
- onViewStatusRestored() method 160
- openFileDescriptor() method 780
- OpenJDK 3

P

- Package Explorer 15
- Package Manager 82
- PackageManager class 646
- PackageManager.FEATURE_MICROPHONE 646
- PackageManager.PERMISSION_DENIED 637
- PackageManager.PERMISSION_GRANTED 637
- Package Name 14
- Packed chain 203, 232
- PageRange 718, 719
- Paint class 721
- parentRelative 389

Index

- parent view 179
- pathRelative 390
- Paused state 154
- PdfDocument 699
- PdfDocument.Page 711, 718
- PendingIntent class 670
- Permission
 - checking for 637
- permissions
 - normal 635
- Persistent State 159
- Phone permissions 636
- picker 779
- Pinch Gesture
 - detection 295
 - example 295
- Pinch Gesture Recognition 289
- Position Keyframes 389
- POST_NOTIFICATIONS permission 636, 668
- Primary Constructor 124
- PrintAttributes 716
- PrintDocumentAdapter 699, 711
- Printing
 - color 696
 - monochrome 696
- Printing framework
 - architecture 693
- Printing Framework 693
- Print Job
 - starting 722
- PrintManager service 703
- Problems
 - tool window 54, 55
- process
 - priority 151
 - state 151
- PROCESS_OUTGOING_CALLS permission 636
- Process States 151
- ProductDetail 764
- ProductDetails 773
- ProductType 766
- Profiler

- tool window 55
- ProgressBar 177
- proguard-rules.pro file 820
- ProGuard Support 816
- Project Name 14
- Project tool window 15, 53
- pt 245
- PurchaseResponseListener 766
- PurchasesUpdatedListener 765
- PurchaseUpdatedListener 774
- putExtra() method 457, 487
- px 246

Q

- queryProductDetailsAsync() 772
- queryPurchasesAsync() 776
- quickboot snapshot 44
- Quick Documentation 75

R

- RadioButton 177
- Range Operator 104
- ratios 233
- READ_CALENDAR permission 636
- READ_CALL_LOG permission 636
- READ_CONTACTS permission 636
- READ_EXTERNAL_STORAGE permission 637
- READ_PHONE_STATE permission 636
- READ_SMS permission 636
- RECEIVE_MMS permission 636
- RECEIVE_SMS permission 636
- RECEIVE_WAP_PUSH permission 636
- Recent Files Navigation 56
- RECORD_AUDIO permission 636
- Recording Audio
 - permission 645
- RecyclerView 437
 - adding to layout file 438
 - GridLayoutManager 437
 - initializing 445
 - LinearLayoutManager 437
 - StaggeredGridLayoutManager 437

- RecyclerView Adapter
 - creation of 443
 - RecyclerView.Adapter 438, 444
 - getItemCount() method 438
 - onBindViewHolder() method 438
 - onCreateViewHolder() method 438
 - RecyclerView.ViewHolder
 - getAdapterPosition() method 448
 - reduce() operator 534
 - registerForActivityResult() 459
 - registerForActivityResult() method 458, 472
 - registerReceiver() method 489
 - RelativeLayout 178
 - releasePersistableUriPermission() method 783
 - Release Preparation 747
 - Remote Bound Service 521
 - client communication 521
 - implementation 521
 - manifest file declaration 523
 - RemoteInput.Builder() method 670
 - RemoteInput Object 670
 - Remote Service
 - launching and binding 523
 - sending a message 525
 - repeatOnLifecycle 548
 - Repository
 - tutorial 603
 - Repository Modules 320
 - Resizable Emulator 47
 - Resource
 - string creation 20
 - Resource File 22
 - Resource Management 151
 - Resource Manager 53, 82
 - result receiver 489
 - Reverse-geocoding 683
 - Reverse Geocoding 682
 - Room
 - Data Access Object (DAO) 586
 - entities 586, 587
 - In-Memory Database 592
 - Repository 586
 - Room Database 586
 - tutorial 603
 - Room Database Persistence 585
 - Room Persistence Library 556, 585
 - root element 177
 - root view 179
 - Run
 - tool window 53
 - runBlocking 497
 - Running Devices
 - tool window 65
 - runOnUiThread() 773
- ## S
- safeargs 379
 - Safe Call Operator 96
 - Sample Data 194
 - Saved State 319, 349
 - SavedStateHandle 350
 - contains() method 351
 - keys() method 351
 - remove() method 351
 - Saved State module 349
 - SavedStateViewModelFactory 350
 - ScaleGestureDetector class 295
 - Scale-independent 245
 - SDK Packages 5
 - Secondary Constructors 124
 - Secure Sockets Layer (SSL) 81
 - SeekBar 307
 - sendBroadcast() method 487, 489
 - sendOrderedBroadcast() method 487, 489
 - SEND_SMS permission 636
 - sendStickyBroadcast() method 487
 - Sensor permissions 636
 - Service
 - anatomy 510
 - launch at system start 511
 - manifest file entry 510
 - overview 84
 - run in separate process 511
 - ServiceConnection class 523

Index

- Service Process 152
- Service Restart Options 510
- setAudioEncoder() method 644
- setAudioSource() method 644
- setBackground-color() 250
- setCompassEnabled() method 688
- setContentView() method 249, 255
- setId() method 250
- setMyLocationButtonEnabled() method 688
- setOnClickListener() method 271, 273
- setOnDoubleTapListener() method 283, 286
- setOutputFile() method 644
- setOutputFormat() method 644
- setResult() method 459
- setText() method 176
- settings.gradle file 816
- settings.gradle.kts file 816
- setTransition() 393
- setVideoSource() method 644
- SHA-256 certificate fingerprint 461
- SharedFlow 538, 541
 - background handling 547
 - DROP_LATEST 539
 - DROP_OLDEST 539
 - in ViewModel 543
 - repeatOnLifecycle 548
 - SUSPEND 539
 - tutorial 541
- shouldOverrideUrlLoading() method 702
- SimpleOnScaleGestureListener 295
- SimpleOnScaleGestureListener class 296
- single() operator 531
- SMS permissions 636
- Snackbar 421, 422, 423
- Snapshots
 - emulator 43
- sp 245
- Spread chain 202
- Spread inside 232
- Spread inside chain 202
- SQL 552
- SQL CREATE 560
- SQLite 551
 - AVD command-line use 553
 - Columns and Data Types 551
 - overview 552
 - Primary keys 552
 - tutorial 557
- SQLiteDatabase 560
- SQLiteOpenHelper 558
- SQL SELECT 561
- StaggeredGridLayoutManager 437
- startActivity() method 457
- startForeground() method 152
- START_NOT_STICKY 510
- START_REDELIVER_INTENT 510
- START_STICKY 510
- State
 - restoring 176
- State Change
 - handling 155
- StateFlow 537
- Statement Completion 72
- Status Bar Widgets 69
 - Memory Indicator 69
- Sticky Broadcast Intents 489
- Stopped state 154
- Storage Access Framework 779
 - ACTION_CREATE_DOCUMENT 780
 - ACTION_OPEN_DOCUMENT 780
 - deleting a file 783
 - example 785
 - file creation 787
 - file filtering 780
 - file reading 781
 - file writing 782
 - intents 780
 - MIME Types 781
 - Persistent Access 783
 - picker 779
- Storage permissions 637
- String 92
- StringBuilder object 790
- strings.xml file 24

- Structure
 - tool window 55
 - Structured Query Language 552
 - Structure tool window 55
 - SUBS 766
 - subscriptions 761
 - supervisorScope 497
 - SupportMapFragment class 677
 - SUSPEND 539
 - Suspend Functions 496
 - Switcher 56
 - System Broadcasts 493
 - system requirements 3
- ## T
- TabLayout
 - adding to layout 431
 - app
 - tabGravity property 436
 - tabMode property 436
 - example 428
 - fixed mode 435
 - getItemCount() method 427
 - overview 427
 - TableLayout 178, 595
 - TableRow 595
 - Telephony Manager 82
 - Templates
 - blank vs. empty 181
 - Terminal
 - tool window 54
 - terminal flow operators 534
 - Theme
 - building a custom 803
 - Theme Builder 803
 - Theming 801
 - tutorial 807
 - Time Cycle Keyframes 393
 - TODO
 - tool window 55
 - ToolBarListener 304
 - tools
 - layout 301
 - Tool window bars 52
 - Tool windows 52
 - Touch Actions 278
 - Touch Event Listener
 - implementation 279
 - Touch Events
 - intercepting 277
 - Touch handling 277
 - transform() operator 533
 - try/finally 530
 - Type Annotations 95
 - Type Casting 99
 - Type Checking 99
 - Type Inference 95
- ## U
- UiSettings class 677
 - unbindService() method 509
 - unregisterReceiver() method 489
 - upload key 750
 - UriMatcher 566, 572
 - UriMatcher class 566
 - URL Mapping 731
 - USB connection issues
 - resolving 62
 - USE_BIOMETRIC 740
 - user interface state 159
 - USE_SIP permission 636
- ## V
- Version catalog 267
 - dependencies 269
 - libraries 269
 - libs.versions.toml file 268
 - plugins 269
 - versions 269
 - Video Playback 617
 - VideoView class 617
 - methods 617
 - supported formats 617
 - view bindings

Index

- enabling 140
- using 140
- View class
 - setting properties 256
- view conversion 193
- ViewGroup 177
- View Groups 177
- View Hierarchy 179
- ViewHolder class 438
 - sample implementation 444
- ViewModel
 - adding LiveData 329
 - data access 327
 - overview 318
 - saved state 349
 - Saved State 319, 349
 - tutorial 323
- ViewModelProvider 326
- ViewModel Saved State 349
- ViewModelScope 496
- ViewPager
 - adding to layout 431
 - example 428
- Views 177
 - Java creation 249
- View System 82
- Virtual Device Configuration dialog 28
- Virtual Sensors 43
- Visible Process 152

W

- WebViewClient 697, 702
- WebView view 477
- Weighted chain 202, 232
- Welcome screen 49
- while Loop 110
- Widget Dimensions 203
- Widget Group Alignment 225
- Widgets palette 238
- WiFi debugging 63
- Wireless debugging 63
- Wireless pairing 63

- withContext 497, 499
- wrap_content properties 247
- WRITE_CALENDAR permission 636
- WRITE_CALL_LOG permission 636
- WRITE_CONTACTS permission 636
- WRITE_EXTERNAL_STORAGE permission 637

X

- XML Layout File
 - manual creation 245
 - vs. Java Code 249

Z

- zip() operator 536