

# **Jetpack Compose 1.4 Essentials**

---

Jetpack Compose 1.4 Essentials

ISBN-13: 978-1-951442-78-1

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

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

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

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

Rev: 1.0

## Table of Contents

<b>1. Start Here.....</b>	<b>1</b>
1.1 For Kotlin programmers .....	1
1.2 For new Kotlin programmers .....	1
1.3 Downloading the code samples.....	1
1.4 Feedback.....	2
1.5 Errata.....	2
<b>2. Setting up an Android Studio Development Environment.....</b>	<b>3</b>
2.1 System requirements.....	3
2.2 Downloading the Android Studio package .....	3
2.3 Installing Android Studio.....	4
2.3.1 Installation on Windows .....	4
2.3.2 Installation on macOS .....	4
2.3.3 Installation on Linux.....	5
2.4 The Android Studio setup wizard .....	5
2.5 Installing additional Android SDK packages .....	6
2.6 Installing the Android SDK Command-line Tools.....	9
2.6.1 Windows 8.1 .....	10
2.6.2 Windows 10 .....	11
2.6.3 Windows 11 .....	11
2.6.4 Linux .....	11
2.6.5 macOS.....	11
2.7 Android Studio memory management .....	11
2.8 Updating Android Studio and the SDK .....	12
2.9 Summary .....	13
<b>3. A Compose Project Overview .....</b>	<b>15</b>
3.1 About the project.....	15
3.2 Creating the project .....	16
3.3 Creating an activity .....	16
3.4 Defining the project and SDK settings .....	17
3.5 Enabling the New Android Studio UI .....	18
3.6 Previewing the example project .....	19
3.7 Reviewing the main activity.....	22
3.8 Preview updates.....	25
3.9 Bill of Materials and the Compose version .....	26
3.10 Summary .....	27
<b>4. An Example Compose Project .....</b>	<b>29</b>
4.1 Getting started .....	29
4.2 Removing the template Code .....	29
4.3 The Composable hierarchy .....	30
4.4 Adding the DemoText composable .....	30
4.5 Previewing the DemoText composable.....	32

## Table of Contents

4.6 Adding the DemoSlider composable.....	32
4.7 Adding the DemoScreen composable .....	33
4.8 Previewing the DemoScreen composable.....	35
4.9 Adjusting preview settings .....	35
4.10 Testing in interactive mode.....	36
4.11 Completing the project.....	37
4.12 Summary .....	38
<b>5. Creating an Android Virtual Device (AVD) in Android Studio .....</b>	<b>39</b>
5.1 About Android Virtual Devices .....	39
5.2 Starting the Emulator.....	41
5.3 Running the Application in the AVD .....	42
5.4 Real-time updates with Live Edit .....	43
5.5 Running on Multiple Devices.....	44
5.6 Stopping a Running Application .....	45
5.7 Supporting Dark Theme.....	45
5.8 Running the Emulator in a Separate Window.....	46
5.9 Enabling the Device Frame.....	49
5.10 Summary .....	50
<b>6. Using and Configuring the Android Studio AVD Emulator .....</b>	<b>51</b>
6.1 The Emulator Environment .....	51
6.2 Emulator Toolbar Options .....	51
6.3 Working in Zoom Mode .....	53
6.4 Resizing the Emulator Window.....	53
6.5 Extended Control Options.....	53
6.5.1 Location.....	54
6.5.2 Displays.....	54
6.5.3 Cellular .....	54
6.5.4 Battery.....	54
6.5.5 Camera.....	54
6.5.6 Phone .....	54
6.5.7 Directional Pad.....	54
6.5.8 Microphone.....	54
6.5.9 Fingerprint .....	54
6.5.10 Virtual Sensors .....	55
6.5.11 Snapshots.....	55
6.5.12 Record and Playback .....	55
6.5.13 Google Play.....	55
6.5.14 Settings .....	55
6.5.15 Help.....	55
6.6 Working with Snapshots.....	55
6.7 Configuring Fingerprint Emulation .....	56
6.8 The Emulator in Tool Window Mode.....	58
6.9 Creating a Resizable Emulator.....	58
6.10 Summary .....	60
<b>7. A Tour of the Android Studio User Interface .....</b>	<b>61</b>
7.1 The Welcome Screen .....	61
7.2 The Menu Bar .....	62
7.3 The Main Window .....	62

7.4 The Tool Windows .....	64
7.5 The Tool Window Menus .....	67
7.6 Android Studio Keyboard Shortcuts .....	67
7.7 Switcher and Recent Files Navigation .....	68
7.8 Changing the Android Studio Theme .....	69
7.9 Summary .....	70
<b>8. Testing Android Studio Apps on a Physical Android Device.....</b>	<b>71</b>
8.1 An Overview of the Android Debug Bridge (ADB).....	71
8.2 Enabling USB Debugging ADB on Android Devices.....	71
8.2.1 macOS ADB Configuration .....	72
8.2.2 Windows ADB Configuration.....	73
8.2.3 Linux adb Configuration.....	74
8.3 Resolving USB Connection Issues .....	74
8.4 Enabling Wireless Debugging on Android Devices .....	75
8.5 Testing the adb Connection .....	77
8.6 Device Mirroring.....	77
8.7 Summary .....	77
<b>9. The Basics of the Android Studio Code Editor.....</b>	<b>79</b>
9.1 The Android Studio Editor.....	79
9.2 Splitting the Editor Window .....	82
9.3 Code Completion .....	82
9.4 Statement Completion .....	84
9.5 Parameter Information .....	84
9.6 Parameter Name Hints .....	84
9.7 Code Generation .....	84
9.8 Code Folding.....	86
9.9 Quick Documentation Lookup .....	87
9.10 Code Reformatting.....	87
9.11 Finding Sample Code .....	88
9.12 Live Templates .....	88
9.13 Summary .....	89
<b>10. An Overview of the Android Architecture .....</b>	<b>91</b>
10.1 The Android software stack .....	91
10.2 The Linux kernel.....	92
10.3 Android runtime – ART .....	92
10.4 Android libraries .....	92
10.4.1 C/C++ libraries.....	92
10.5 Application framework.....	93
10.6 Applications .....	93
10.7 Summary .....	93
<b>11. An Introduction to Kotlin.....</b>	<b>95</b>
11.1 What is Kotlin? .....	95
11.2 Kotlin and Java.....	95
11.3 Converting from Java to Kotlin .....	95
11.4 Kotlin and Android Studio .....	96
11.5 Experimenting with Kotlin .....	96
11.6 Semi-colons in Kotlin .....	97

11.7 Summary .....	97
<b>12. Kotlin Data Types, Variables and Nullability .....</b>	<b>99</b>
12.1 Kotlin data types.....	99
12.1.1 Integer data types .....	100
12.1.2 Floating point data types.....	100
12.1.3 Boolean data type.....	100
12.1.4 Character data type.....	100
12.1.5 String data type.....	100
12.1.6 Escape sequences.....	101
12.2 Mutable variables .....	102
12.3 Immutable variables.....	102
12.4 Declaring mutable and immutable variables .....	102
12.5 Data types are objects .....	102
12.6 Type annotations and type inference.....	103
12.7 Nullable type .....	104
12.8 The safe call operator .....	104
12.9 Not-null assertion .....	105
12.10 Nullable types and the let function.....	105
12.11 Late initialization (lateinit) .....	106
12.12 The Elvis operator .....	107
12.13 Type casting and type checking.....	107
12.14 Summary.....	108
<b>13. Kotlin Operators and Expressions .....</b>	<b>109</b>
13.1 Expression syntax in Kotlin .....	109
13.2 The Basic assignment operator.....	109
13.3 Kotlin arithmetic operators.....	109
13.4 Augmented assignment operators .....	110
13.5 Increment and decrement operators .....	110
13.6 Equality operators .....	111
13.7 Boolean logical operators.....	111
13.8 Range operator .....	112
13.9 Bitwise operators .....	112
13.9.1 Bitwise inversion .....	112
13.9.2 Bitwise AND .....	113
13.9.3 Bitwise OR.....	113
13.9.4 Bitwise XOR.....	113
13.9.5 Bitwise left shift .....	114
13.9.6 Bitwise right shift .....	114
13.10 Summary.....	115
<b>14. Kotlin Control Flow .....</b>	<b>117</b>
14.1 Looping control flow.....	117
14.1.1 The Kotlin <i>for-in</i> Statement.....	117
14.1.2 The <i>while</i> loop .....	118
14.1.3 The <i>do ... while</i> loop .....	119
14.1.4 Breaking from Loops .....	119
14.1.5 The <i>continue</i> statement .....	120
14.1.6 Break and continue labels .....	120
14.2 Conditional control flow .....	121

14.2.1 Using the <i>if</i> expressions .....	121
14.2.2 Using <i>if ... else ...</i> expressions .....	122
14.2.3 Using <i>if ... else if ...</i> Expressions .....	122
14.2.4 Using the <i>when</i> statement .....	122
14.3 Summary .....	123
<b>15. An Overview of Kotlin Functions and Lambdas .....</b>	<b>125</b>
15.1 What is a function? .....	125
15.2 How to declare a Kotlin function.....	125
15.3 Calling a Kotlin function.....	126
15.4 Single expression functions.....	126
15.5 Local functions .....	126
15.6 Handling return values.....	127
15.7 Declaring default function parameters .....	127
15.8 Variable number of function parameters .....	127
15.9 Lambda expressions.....	128
15.10 Higher-order functions .....	129
15.11 Summary .....	130
<b>16. The Basics of Object-Oriented Programming in Kotlin.....</b>	<b>131</b>
16.1 What is an object? .....	131
16.2 What is a class? .....	131
16.3 Declaring a Kotlin class.....	131
16.4 Adding properties to a class.....	132
16.5 Defining methods.....	132
16.6 Declaring and initializing a class instance .....	132
16.7 Primary and secondary constructors .....	132
16.8 Initializer blocks .....	135
16.9 Calling methods and accessing properties.....	135
16.10 Custom accessors .....	135
16.11 Nested and inner classes.....	136
16.12 Companion objects.....	137
16.13 Summary .....	139
<b>17. An Introduction to Kotlin Inheritance and Subclassing.....</b>	<b>141</b>
17.1 Inheritance, classes, and subclasses .....	141
17.2 Subclassing syntax.....	141
17.3 A Kotlin inheritance example.....	142
17.4 Extending the functionality of a subclass.....	143
17.5 Overriding inherited methods .....	144
17.6 Adding a custom secondary constructor .....	145
17.7 Using the SavingsAccount class.....	145
17.8 Summary .....	145
<b>18. An Overview of Compose .....</b>	<b>147</b>
18.1 Development before Compose.....	147
18.2 Compose declarative syntax .....	147
18.3 Compose is data-driven .....	148
18.4 Summary .....	148
<b>19. Composable Functions Overview .....</b>	<b>149</b>

## Table of Contents

19.1 What is a composable function? .....	149
19.2 Stateful vs. stateless composables .....	149
19.3 Composable function syntax .....	150
19.4 Foundation and Material composables .....	152
19.5 Summary .....	153
<b>20. An Overview of Compose State and Recomposition.....</b>	<b>155</b>
20.1 The basics of state .....	155
20.2 Introducing recomposition .....	155
20.3 Creating the StateExample project.....	156
20.4 Declaring state in a composable.....	156
20.5 Unidirectional data flow.....	159
20.6 State hoisting.....	161
20.7 Saving state through configuration changes.....	163
20.8 Summary .....	164
<b>21. An Introduction to Composition Local.....</b>	<b>167</b>
21.1 Understanding CompositionLocal .....	167
21.2 Using CompositionLocal .....	168
21.3 Creating the CompLocalDemo project.....	169
21.4 Designing the layout.....	169
21.5 Adding the CompositionLocal state .....	170
21.6 Accessing the CompositionLocal state.....	171
21.7 Testing the design.....	171
21.8 Summary .....	174
<b>22. An Overview of Compose Slot APIs .....</b>	<b>175</b>
22.1 Understanding slot APIs .....	175
22.2 Declaring a slot API.....	176
22.3 Calling slot API composables.....	176
22.4 Summary .....	178
<b>23. A Compose Slot API Tutorial.....</b>	<b>179</b>
23.1 About the project.....	179
23.2 Creating the SlotApiDemo project .....	179
23.3 Preparing the MainActivity class file .....	179
23.4 Creating the mainScreen composable.....	180
23.5 Adding the ScreenContent composable .....	181
23.6 Creating the Checkbox composable .....	182
23.7 Implementing the ScreenContent slot API.....	183
23.8 Adding an Image drawable resource .....	184
23.9 Coding the TitleImage composable.....	185
23.10 Completing the mainScreen composable.....	186
23.11 Previewing the project.....	188
23.12 Summary.....	189
<b>24. Using Modifiers in Compose.....</b>	<b>191</b>
24.1 An overview of modifiers.....	191
24.2 Creating the ModifierDemo project.....	191
24.3 Creating a modifier .....	192
24.4 Modifier ordering.....	194



24.5 Adding modifier support to a composable .....	194
24.6 Common built-in modifiers .....	198
24.7 Combining modifiers.....	198
24.8 Summary .....	199
<b>25. Annotated Strings and Brush Styles.....</b>	<b>201</b>
25.1 What are annotated strings? .....	201
25.2 Using annotated strings.....	201
25.3 Brush Text Styling .....	202
25.4 Creating the example project.....	203
25.5 An example SpanStyle annotated string.....	203
25.6 An example ParagraphStyle annotated string .....	204
25.7 A Brush style example .....	207
25.8 Summary .....	208
<b>26. Composing Layouts with Row and Column .....</b>	<b>209</b>
26.1 Creating the RowColDemo project .....	209
26.2 Row composable.....	210
26.3 Column composable.....	210
26.4 Combining Row and Column composables.....	211
26.5 Layout alignment .....	212
26.6 Layout arrangement positioning.....	214
26.7 Layout arrangement spacing.....	216
26.8 Row and Column scope modifiers.....	217
26.9 Scope modifier weights .....	221
26.10 Summary .....	221
<b>27. Box Layouts in Compose.....</b>	<b>223</b>
27.1 An introduction to the Box composable.....	223
27.2 Creating the BoxLayout project .....	223
27.3 Adding the TextCell composable .....	223
27.4 Adding a Box layout.....	224
27.5 Box alignment.....	225
27.6 BoxScope modifiers .....	227
27.7 Using the clip() modifier .....	227
27.8 Summary .....	229
<b>28. An Introduction to FlowRow and FlowColumn.....</b>	<b>231</b>
28.1 FlowColumn and FlowRow .....	231
28.2 Maximum number of items .....	232
28.3 Working with main axis arrangement.....	232
28.4 Understanding cross-axis arrangement .....	234
28.5 Item alignment .....	235
28.6 Controlling item size.....	236
28.7 Summary .....	237
<b>29. A FlowRow and FlowColumn Tutorial.....</b>	<b>239</b>
29.1 Creating the FlowLayoutDemo project.....	239
29.2 Generating random height and color values .....	240
29.3 Adding the Box Composable.....	241
29.4 Modifying the Flow arrangement .....	242

## Table of Contents

29.5 Modifying item alignment .....	242
29.6 Switching to FlowColumn .....	244
29.7 Using cross-axis arrangement.....	245
29.8 Adding item weights .....	245
29.9 Summary .....	246
<b>30. Custom Layout Modifiers.....</b>	<b>247</b>
30.1 Compose layout basics .....	247
30.2 Custom layouts .....	247
30.3 Creating the LayoutModifier project.....	247
30.4 Adding the ColorBox composable.....	248
30.5 Creating a custom layout modifier .....	249
30.6 Understanding default position.....	249
30.7 Completing the layout modifier .....	249
30.8 Using a custom modifier .....	250
30.9 Working with alignment lines .....	251
30.10 Working with baselines .....	253
30.11 Summary .....	253
<b>31. Building Custom Layouts.....</b>	<b>255</b>
31.1 An overview of custom layouts .....	255
31.2 Custom layout syntax .....	255
31.3 Using a custom layout.....	256
31.4 Creating the CustomLayout project .....	257
31.5 Creating the CascadeLayout composable .....	257
31.6 Using the CascadeLayout composable .....	259
31.7 Summary .....	260
<b>32. A Guide to ConstraintLayout in Compose.....</b>	<b>261</b>
32.1 An introduction to ConstraintLayout .....	261
32.2 How ConstraintLayout works.....	261
32.2.1 Constraints.....	261
32.2.2 Margins.....	262
32.2.3 Opposing constraints.....	262
32.2.4 Constraint bias.....	263
32.2.5 Chains.....	264
32.2.6 Chain styles.....	264
32.3 Configuring dimensions.....	265
32.4 Guideline helper .....	265
32.5 Barrier helper.....	266
32.6 Summary .....	267
<b>33. Working with ConstraintLayout in Compose .....</b>	<b>269</b>
33.1 Calling ConstraintLayout.....	269
33.2 Generating references .....	269
33.3 Assigning a reference to a composable.....	269
33.4 Adding constraints .....	270
33.5 Creating the ConstraintLayout project .....	270
33.6 Adding the ConstraintLayout library .....	271
33.7 Adding a custom button composable.....	271
33.8 Basic constraints.....	272

33.9 Opposing constraints.....	273
33.10 Constraint bias.....	274
33.11 Constraint margins .....	275
33.12 The importance of opposing constraints and bias .....	276
33.13 Creating chains.....	279
33.14 Working with guidelines .....	280
33.15 Working with barriers .....	281
33.16 Decoupling constraints with constraint sets.....	284
33.17 Summary.....	286
<b>34. Working with IntrinsicSize in Compose.....</b>	<b>287</b>
34.1 Intrinsic measurements.....	287
34.2 Max. vs Min. Intrinsic Size measurements.....	287
34.3 About the example project.....	288
34.4 Creating the IntrinsicSizeDemo project.....	289
34.5 Creating the custom text field.....	289
34.6 Adding the Text and Box components.....	290
34.7 Adding the top-level Column.....	290
34.8 Testing the project.....	291
34.9 Applying IntrinsicSize.Max measurements .....	291
34.10 Applying IntrinsicSize.Min measurements .....	292
34.11 Summary.....	292
<b>35. Coroutines and LaunchedEffects in Jetpack Compose.....</b>	<b>293</b>
35.1 What are coroutines? .....	293
35.2 Threads vs. coroutines .....	293
35.3 Coroutine Scope .....	294
35.4 Suspend functions.....	294
35.5 Coroutine dispatchers.....	294
35.6 Coroutine builders .....	295
35.7 Jobs.....	295
35.8 Coroutines – suspending and resuming .....	296
35.9 Coroutine channel communication.....	297
35.10 Understanding side effects .....	298
35.11 Summary.....	299
<b>36. An Overview of Lists and Grids in Compose .....</b>	<b>301</b>
36.1 Standard vs. lazy lists .....	301
36.2 Working with Column and Row lists .....	301
36.3 Creating lazy lists .....	302
36.4 Enabling scrolling with ScrollState .....	303
36.5 Programmatic scrolling.....	303
36.6 Sticky headers .....	304
36.7 Responding to scroll position.....	306
36.8 Creating a lazy grid .....	306
36.9 Summary.....	309
<b>37. A Compose Row and Column List Tutorial .....</b>	<b>311</b>
37.1 Creating the ListDemo project.....	311
37.2 Creating a Column-based list.....	311
37.3 Enabling list scrolling .....	313

## Table of Contents

37.4 Manual scrolling.....	313
37.5 A Row list example.....	316
37.6 Summary .....	316
<b>38. A Compose Lazy List Tutorial .....</b>	<b>317</b>
38.1 Creating the LazyListDemo project.....	317
38.2 Adding list data to the project.....	317
38.3 Reading the XML data.....	319
38.4 Handling image loading.....	320
38.5 Designing the list item composable.....	322
38.6 Building the lazy list.....	323
38.7 Testing the project.....	324
38.8 Making list items clickable.....	324
38.9 Summary .....	326
<b>39. Lazy List Sticky Headers and Scroll Detection .....</b>	<b>327</b>
39.1 Grouping the list item data .....	327
39.2 Displaying the headers and items .....	327
39.3 Adding sticky headers.....	328
39.4 Reacting to scroll position .....	329
39.5 Adding the scroll button .....	331
39.6 Testing the finished app.....	333
39.7 Summary .....	333
<b>40. A Compose Lazy Staggered Grid Tutorial .....</b>	<b>335</b>
40.1 Lazy Staggered Grids .....	335
40.2 Creating the StaggeredGridDemo project .....	336
40.3 Adding the Box composable.....	337
40.4 Generating random height and color values .....	337
40.5 Creating the Staggered List.....	338
40.6 Testing the project.....	339
40.7 Switching to a horizontal staggered grid.....	340
40.8 Summary .....	341
<b>41. VerticalPager and HorizontalPager in Compose .....</b>	<b>343</b>
41.1 The Pager composables.....	343
41.2 Working with pager state .....	345
41.3 About the PagerDemo project.....	345
41.4 Creating the PagerDemo project.....	345
41.5 Modifying the build configuration .....	346
41.6 Adding the book cover images.....	346
41.7 Adding the HorizontalPager.....	347
41.8 Creating the page content .....	348
41.9 Testing the pager .....	349
41.10 Adding the arrow buttons .....	350
41.11 Summary.....	353
<b>42. Compose Visibility Animation .....</b>	<b>355</b>
42.1 Creating the AnimateVisibility project .....	355
42.2 Animating visibility .....	355
42.3 Defining enter and exit animations .....	358

42.4 Animation specs and animation easing .....	359
42.5 Repeating an animation .....	361
42.6 Different animations for different children .....	361
42.7 Auto-starting an animation .....	362
42.8 Implementing crossfading .....	363
42.9 Summary .....	365
<b>43. Compose State-Driven Animation.....</b>	<b>367</b>
43.1 Understanding state-driven animation .....	367
43.2 Introducing animate as state functions .....	367
43.3 Creating the AnimateState project.....	368
43.4 Animating rotation with animateFloatAsState.....	368
43.5 Animating color changes with animateColorAsState.....	371
43.6 Animating motion with animateDpAsState .....	373
43.7 Adding spring effects .....	376
43.8 Working with keyframes .....	377
43.9 Combining multiple animations .....	378
43.10 Using the Animation Inspector.....	381
43.11 Summary .....	382
<b>44. Canvas Graphics Drawing in Compose .....</b>	<b>383</b>
44.1 Introducing the Canvas component .....	383
44.2 Creating the CanvasDemo project.....	383
44.3 Drawing a line and getting the canvas size .....	383
44.4 Drawing dashed lines.....	385
44.5 Drawing a rectangle .....	385
44.6 Applying rotation .....	389
44.7 Drawing circles and ovals.....	390
44.8 Drawing gradients.....	391
44.9 Drawing arcs .....	394
44.10 Drawing paths .....	395
44.11 Drawing points .....	396
44.12 Drawing an image .....	397
44.13 Drawing text .....	399
44.14 Summary .....	401
<b>45. Working with ViewModels in Compose .....</b>	<b>403</b>
45.1 What is Android Jetpack? .....	403
45.2 The “old” architecture .....	403
45.3 Modern Android architecture .....	403
45.4 The ViewModel component.....	403
45.5 ViewModel implementation using state.....	404
45.6 Connecting a ViewModel state to an activity.....	405
45.7 ViewModel implementation using LiveData.....	406
45.8 Observing ViewModel LiveData within an activity .....	407
45.9 Summary .....	407
<b>46. A Compose ViewModel Tutorial.....</b>	<b>409</b>
46.1 About the project.....	409
46.2 Creating the ViewModelDemo project .....	410
46.3 Adding the ViewModel .....	410

## Table of Contents

46.4	Accessing DemoViewModel from MainActivity .....	411
46.5	Designing the temperature input composable .....	412
46.6	Designing the temperature input composable .....	414
46.7	Completing the user interface design.....	416
46.8	Testing the app.....	418
46.9	Summary .....	418
<b>47.</b>	<b>An Overview of Android SQLite Databases .....</b>	<b>419</b>
47.1	Understanding database tables.....	419
47.2	Introducing database schema .....	419
47.3	Columns and data types .....	419
47.4	Database rows .....	420
47.5	Introducing primary keys .....	420
47.6	What is SQLite? .....	420
47.7	Structured Query Language (SQL).....	420
47.8	Trying SQLite on an Android Virtual Device (AVD) .....	421
47.9	The Android Room persistence library.....	423
47.10	Summary .....	423
<b>48.</b>	<b>Room Databases and Compose .....</b>	<b>425</b>
48.1	Revisiting modern app architecture .....	425
48.2	Key elements of Room database persistence .....	425
48.2.1	Repository .....	425
48.2.2	Room database .....	426
48.2.3	Data Access Object (DAO) .....	426
48.2.4	Entities.....	426
48.2.5	SQLite database .....	426
48.3	Understanding entities .....	427
48.4	Data Access Objects.....	429
48.5	The Room database.....	430
48.6	The Repository.....	431
48.7	In-Memory databases .....	432
48.8	Database Inspector.....	433
48.9	Summary .....	433
<b>49.</b>	<b>A Compose Room Database and Repository Tutorial .....</b>	<b>435</b>
49.1	About the RoomDemo project.....	435
49.2	Creating the RoomDemo project.....	436
49.3	Modifying the build configuration .....	436
49.4	Building the entity.....	437
49.5	Creating the Data Access Object.....	438
49.6	Adding the Room database.....	439
49.7	Adding the repository.....	440
49.8	Adding the ViewModel .....	442
49.9	Designing the user interface .....	444
49.10	Writing a ViewModelProvider Factory class.....	445
49.11	Completing the MainScreen function.....	447
49.12	Testing the RoomDemo app.....	450
49.13	Using the Database Inspector.....	451
49.14	Summary .....	452

<b>50. An Overview of Navigation in Compose .....</b>	<b>453</b>
50.1 Understanding navigation.....	453
50.2 Declaring a navigation controller.....	455
50.3 Declaring a navigation host .....	455
50.4 Adding destinations to the navigation graph .....	455
50.5 Navigating to destinations.....	456
50.6 Passing arguments to a destination.....	458
50.7 Working with bottom navigation bars .....	459
50.8 Summary .....	461
<b>51. A Compose Navigation Tutorial .....</b>	<b>463</b>
51.1 Creating the NavigationDemo project .....	463
51.2 About the NavigationDemo project .....	463
51.3 Declaring the navigation routes .....	464
51.4 Adding the home screen .....	464
51.5 Adding the welcome screen .....	466
51.6 Adding the profile screen .....	466
51.7 Creating the navigation controller and host.....	467
51.8 Implementing the screen navigation .....	468
51.9 Passing the user name argument.....	468
51.10 Testing the project.....	469
51.11 Summary.....	470
<b>52. A Compose Navigation Bar Tutorial.....</b>	<b>471</b>
52.1 Creating the BottomBarDemo project .....	471
52.2 Declaring the navigation routes .....	471
52.3 Designing bar items .....	472
52.4 Creating the bar item list.....	472
52.5 Adding the destination screens .....	473
52.6 Creating the navigation controller and host.....	475
52.7 Designing the navigation bar.....	476
52.8 Working with the Scaffold component.....	477
52.9 Testing the project.....	478
52.10 Summary.....	479
<b>53. Detecting Gestures in Compose.....</b>	<b>481</b>
53.1 Compose gesture detection.....	481
53.2 Creating the GestureDemo project.....	481
53.3 Detecting click gestures.....	481
53.4 Detecting taps using PointerInputScope.....	483
53.5 Detecting drag gestures .....	484
53.6 Detecting drag gestures using PointerInputScope.....	486
53.7 Scrolling using the scrollable modifier .....	487
53.8 Scrolling using the scroll modifiers .....	488
53.9 Detecting pinch gestures .....	490
53.10 Detecting rotation gestures.....	491
53.11 Detecting translation gestures .....	492
53.12 Summary .....	493
<b>54. An Introduction to Kotlin Flow .....</b>	<b>495</b>
54.1 Understanding Flows.....	495

## Table of Contents

54.2	Creating the sample project .....	495
54.3	Adding a view model to the project.....	496
54.4	Declaring the flow .....	497
54.5	Emitting flow data .....	497
54.6	Collecting flow data as state.....	498
54.7	Transforming data with intermediaries .....	499
54.8	Collecting flow data .....	501
54.9	Adding a flow buffer .....	502
54.10	More terminal flow operators.....	503
54.11	Flow flattening.....	504
54.12	Combining multiple flows .....	506
54.13	Hot and cold flows .....	507
54.14	StateFlow .....	507
54.15	SharedFlow.....	508
54.16	Converting a flow from cold to hot .....	510
54.17	Summary .....	510
<b>55.</b>	<b>A Jetpack Compose SharedFlow Tutorial .....</b>	<b>511</b>
55.1	About the project.....	511
55.2	Creating the SharedFlowDemo project.....	511
55.3	Adding a view model to the project.....	512
55.4	Declaring the SharedFlow .....	512
55.5	Collecting the flow values .....	513
55.6	Testing the SharedFlowDemo app .....	515
55.7	Handling flows in the background.....	515
55.8	Summary .....	517
<b>56.</b>	<b>Creating, Testing, and Uploading an Android App Bundle .....</b>	<b>519</b>
56.1	The Release Preparation Process.....	519
56.2	Android App Bundles.....	519
56.3	Register for a Google Play Developer Console Account.....	520
56.4	Configuring the App in the Console .....	521
56.5	Enabling Google Play App Signing.....	522
56.6	Creating a Keystore File .....	522
56.7	Creating the Android App Bundle.....	523
56.8	Generating Test APK Files .....	525
56.9	Uploading the App Bundle to the Google Play Developer Console.....	526
56.10	Exploring the App Bundle .....	527
56.11	Managing Testers .....	528
56.12	Rolling the App Out for Testing.....	528
56.13	Uploading New App Bundle Revisions.....	529
56.14	Analyzing the App Bundle File .....	530
56.15	Summary .....	531
<b>57.</b>	<b>An Overview of Android In-App Billing .....</b>	<b>533</b>
57.1	Preparing a project for In-App purchasing.....	533
57.2	Creating In-App products and subscriptions.....	533
57.3	Billing client initialization.....	534
57.4	Connecting to the Google Play Billing library .....	535
57.5	Querying available products.....	535
57.6	Starting the purchase process .....	536



57.7	Completing the purchase .....	536
57.8	Querying previous purchases .....	537
57.9	Summary .....	538
<b>58.</b>	<b>An Android In-App Purchasing Tutorial .....</b>	<b>539</b>
58.1	About the In-App purchasing example project .....	539
58.2	Creating the InAppPurchase project .....	539
58.3	Adding libraries to the project .....	539
58.4	Adding the App to the Google Play Store .....	540
58.5	Creating an In-App product .....	540
58.6	Enabling license testers .....	541
58.7	Creating a purchase helper class .....	542
58.8	Adding the StateFlow streams .....	543
58.9	Initializing the billing client .....	543
58.10	Querying the product .....	544
58.11	Handling purchase updates .....	545
58.12	Launching the purchase flow .....	545
58.13	Consuming the product .....	546
58.14	Restoring a previous purchase .....	546
58.15	Completing the MainActivity .....	547
58.16	Testing the app .....	549
58.17	Troubleshooting .....	551
58.18	Summary .....	552
<b>59.</b>	<b>Working with Compose Theming .....</b>	<b>553</b>
59.1	Material Design 2 vs. Material Design 3 .....	553
59.2	Material Design 3 theming .....	553
59.3	Building a custom theme .....	557
59.4	Summary .....	558
<b>60.</b>	<b>A Material Design 3 Theming Tutorial .....</b>	<b>559</b>
60.1	Creating the ThemeDemo project .....	559
60.2	Designing the user interface .....	559
60.3	Building a new theme .....	561
60.4	Adding the theme to the project .....	562
60.5	Enabling dynamic colors .....	563
60.6	Summary .....	564
<b>61.</b>	<b>An Overview of Gradle in Android Studio .....</b>	<b>565</b>
61.1	An Overview of Gradle .....	565
61.2	Gradle and Android Studio .....	565
61.2.1	Sensible Defaults .....	565
61.2.2	Dependencies .....	565
61.2.3	Build Variants .....	566
61.2.4	Manifest Entries .....	566
61.2.5	APK Signing .....	566
61.2.6	ProGuard Support .....	566
61.3	The Property and Settings Gradle Build File .....	566
61.4	The Top-level Gradle Build File .....	567
61.5	Module Level Gradle Build Files .....	568
61.6	Configuring Signing Settings in the Build File .....	571

Table of Contents

61.7 Running Gradle Tasks from the Command Line .....	572
61.8 Summary .....	572
<b>Index .....</b>	<b>573</b>

## 1. Start Here

This book teaches you how to build Android applications using Jetpack Compose 1.4, Android Studio Giraffe (2023.2.1), Material Design 3, and the Kotlin programming language.

The book begins with the basics by explaining how to set up an Android Studio development environment.

The book also includes in-depth chapters introducing the Kotlin programming language, including data types, operators, control flow, functions, lambdas, coroutines, and object-oriented programming.

An introduction to the key concepts of Jetpack Compose and Android project architecture is followed by a guided tour of Android Studio in Compose development mode. The book also covers the creation of custom Composables and explains how functions are combined to create user interface layouts, including row, column, box, flow, pager, and list components.

Other topics covered include data handling using state properties, key user interface design concepts such as modifiers, navigation bars, and user interface navigation. Additional chapters explore building your own reusable custom layout components.

The book covers graphics drawing, user interface animation, transitions, Kotlin Flows, and gesture handling.

Chapters also cover view models, SQLite databases, Room database access, the Database Inspector, live data, and custom theme creation. Using in-app billing, you will also learn to generate extra revenue from your app.

Finally, the book explains how to package up a completed app and upload it to the Google Play Store for publication.

Along the way, the topics covered in the book are put into practice through detailed tutorials, the source code for which is also available for download.

Assuming you already have some rudimentary programming experience, are ready to download Android Studio and the Android SDK, and have access to a Windows, Mac, or Linux system, you are ready to start.

### 1.1 For Kotlin programmers

This book addresses the needs of existing Kotlin programmers and those new to Kotlin and Jetpack Compose app development. If you are familiar with the Kotlin programming language, you can probably skip the Kotlin-specific chapters.

### 1.2 For new Kotlin programmers

If you are new to Kotlin programming, the entire book is appropriate for you. Just start at the beginning and keep going.

### 1.3 Downloading the code samples

The source code and Android Studio project files for the examples contained in this book are available for download at:

<https://www.ebookfrenzy.com/retail/compose14/index.php>

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

## Start Here

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

## 1.4 Feedback

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

## 1.5 Errata

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

<https://www.ebookfrenzy.com/errata/compose14.html>

If you find an error not listed in the errata, email our technical support team at [feedback@ebookfrenzy.com](mailto:feedback@ebookfrenzy.com).

## 2. Setting up an Android Studio Development Environment

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

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

### 2.1 System requirements

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

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

### 2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio Giraffe 2022.3.1 using the Android API 33 SDK (Tiramisu), which, at the time of writing, are the latest stable releases.

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

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

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

If this page provides instructions for downloading a newer version of Android Studio, there may be differences between this book and the software. A web search for “Android Studio Giraffe” should provide the option to download the older version if these differences become a problem. Alternatively, visit the following web page to find Android Studio Giraffe 2022.3.1 in the archives:

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

## 2.3 Installing Android Studio

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

### 2.3.1 Installation on Windows

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

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

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

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

### 2.3.2 Installation on macOS

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

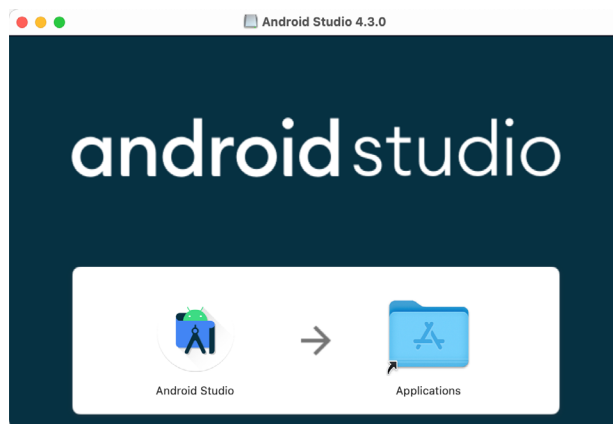


Figure 2-1

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

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

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

### 2.3.3 Installation on Linux

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

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

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

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

```
./studio.sh
```

## 2.4 The Android Studio setup wizard

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

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

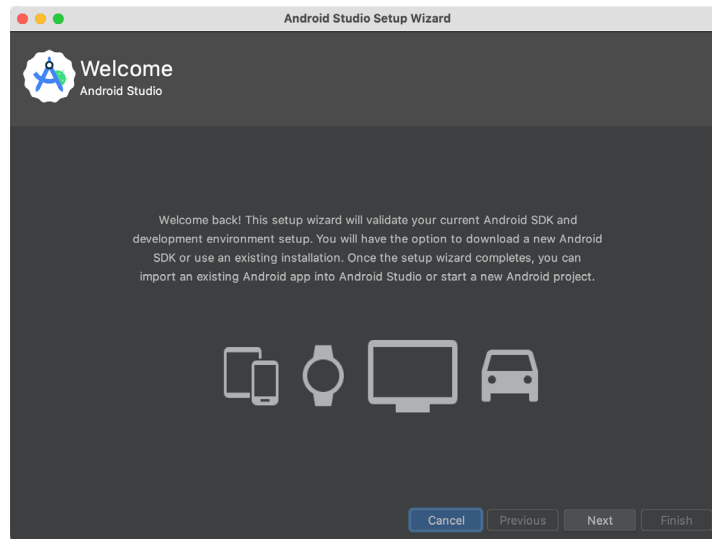


Figure 2-2

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

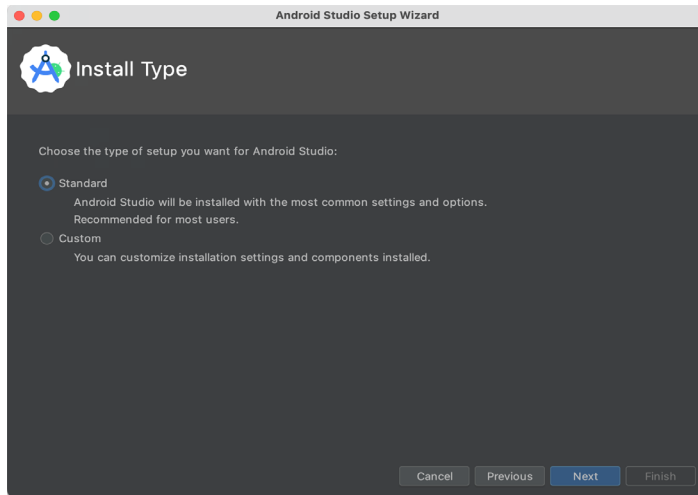


Figure 2-3

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

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

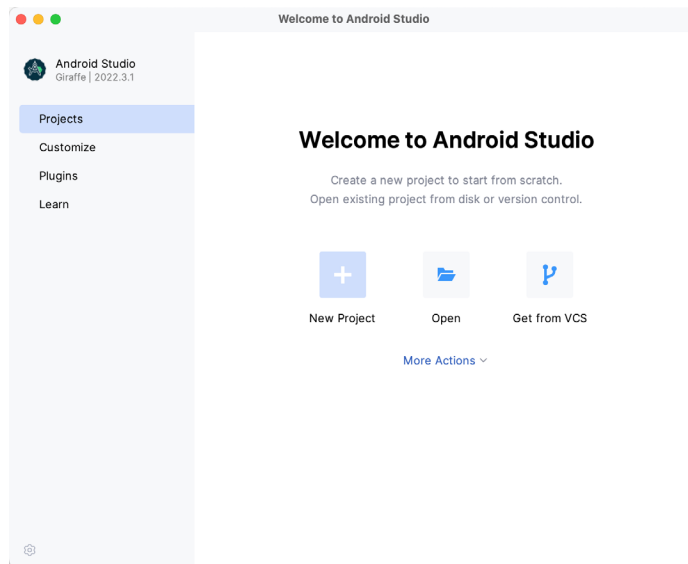


Figure 2-4

## 2.5 Installing additional Android SDK packages

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



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

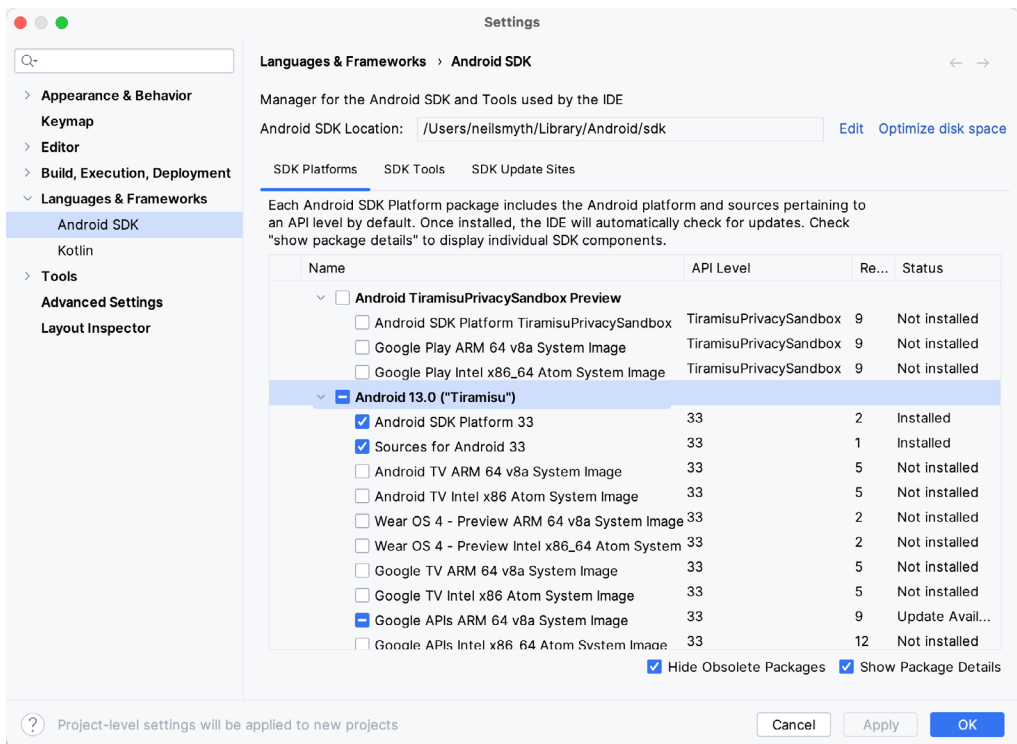


Figure 2-5

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

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

Immediately after installing Android Studio for the first time, it is likely that only the latest supported version of the Android SDK has been installed. To install older versions of the Android SDK, select the checkboxes corresponding to the versions and click the *Apply* button. The rest of this book assumes that the Android Tiramisu (API Level 33) SDK is installed.

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

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

## Setting up an Android Studio Development Environment

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

Figure 2-6

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

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

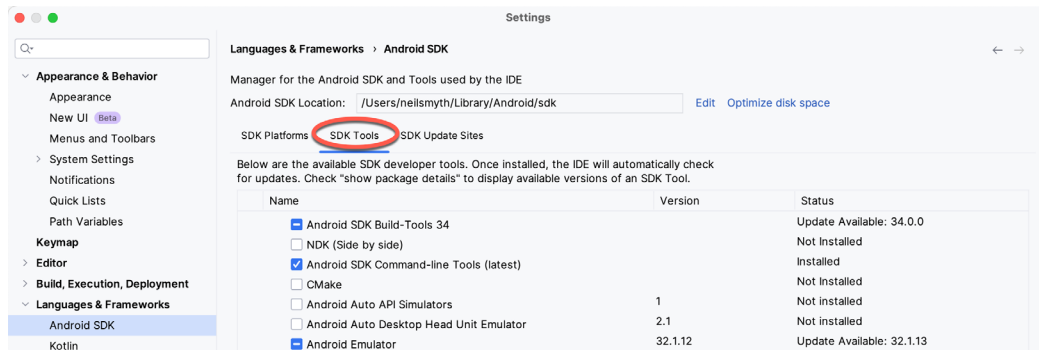


Figure 2-7

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

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

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

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

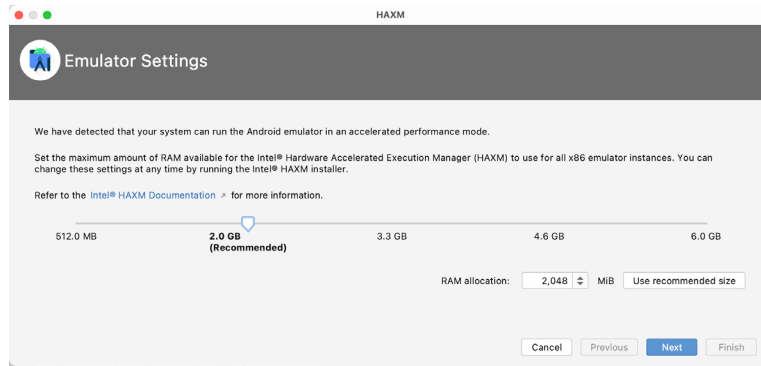


Figure 2-8

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

## 2.6 Installing the Android SDK Command-line Tools

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

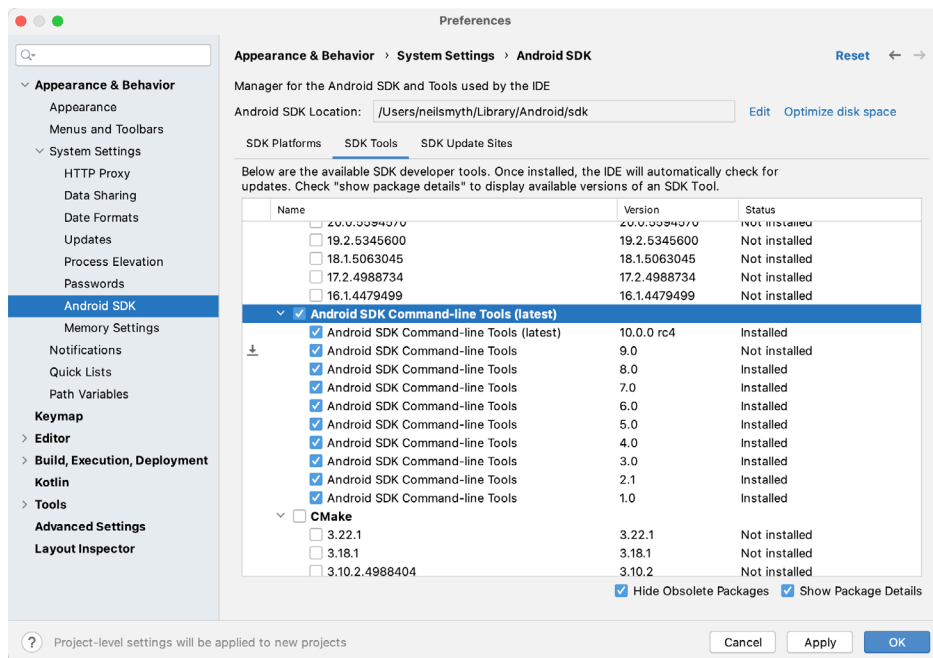


Figure 2-9

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

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

## Setting up an Android Studio Development Environment

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

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

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

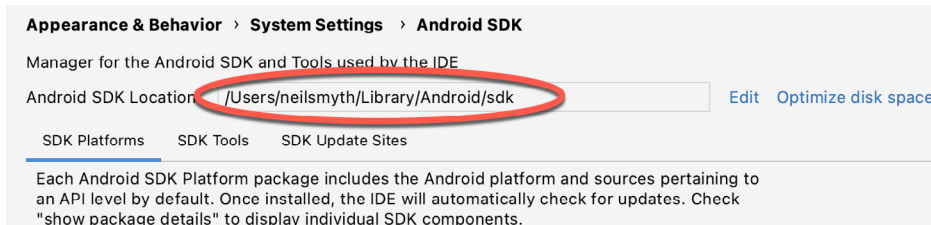


Figure 2-10

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

### 2.6.1 Windows 8.1

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

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

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

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

```
echo %Path%
```

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

```
adb
```

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

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

```
avdmanager
```

If a message similar to the following message appears for one or both of the commands, it is most likely that an

incorrect path was appended to the Path environment variable:

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

## 2.6.2 Windows 10

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

## 2.6.3 Windows 11

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

## 2.6.4 Linux

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

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

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

## 2.6.5 macOS

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

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

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

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

## 2.7 Android Studio memory management

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

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

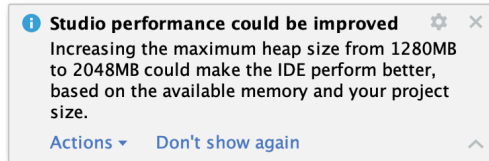


Figure 2-11

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

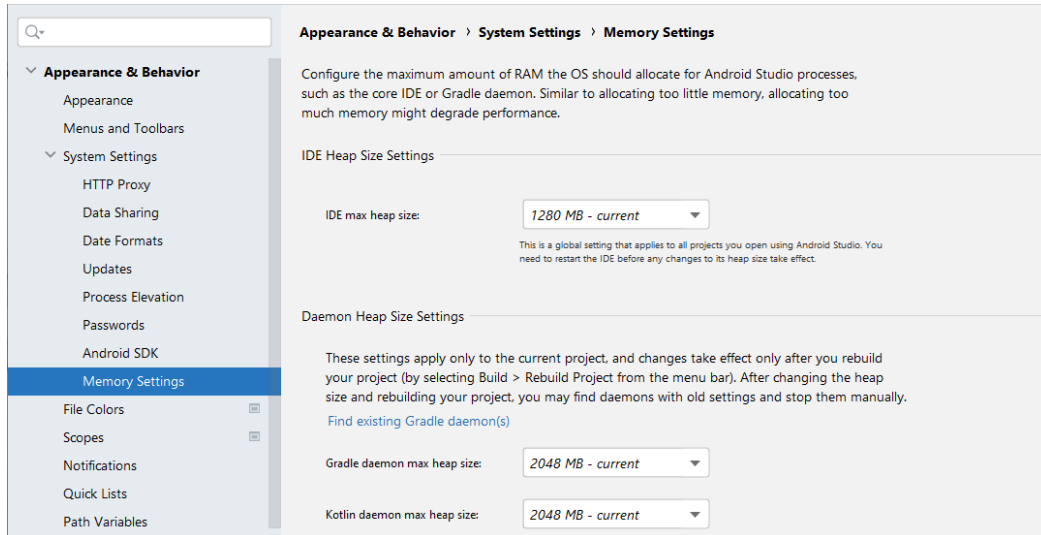


Figure 2-12

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

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

## 2.8 Updating Android Studio and the SDK

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

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

## 2.9 Summary

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





## 3. A Compose Project Overview

Now that we have installed Android Studio, the next step is to create an Android app using Jetpack Compose. Although this project will use several Compose features, it is an intentionally simple example intended to provide an early demonstration of Compose in action and an initial success on which to build as you work through the remainder of the book. The project will also verify that your Android Studio environment is correctly installed and configured.

This chapter will create a new project using the Android Studio Compose project template and explore both the basic structure of a Compose-based Android Studio project and some of the key areas of Android Studio. The next chapter will use this project to create a simple Android app.

Both chapters will briefly explain key features of Compose as they are introduced within the project. If anything is unclear when you have completed the project, rest assured that all the areas covered in the tutorial will be explored in greater detail in later chapters of the book.

### 3.1 About the project

The completed project will consist of two text components and a slider. When the slider is moved, the current value will be displayed on one of the text components, while the font size of the second text instance will adjust to match the current slider position. Once completed, the user interface for the app will appear as shown in Figure 3-1:

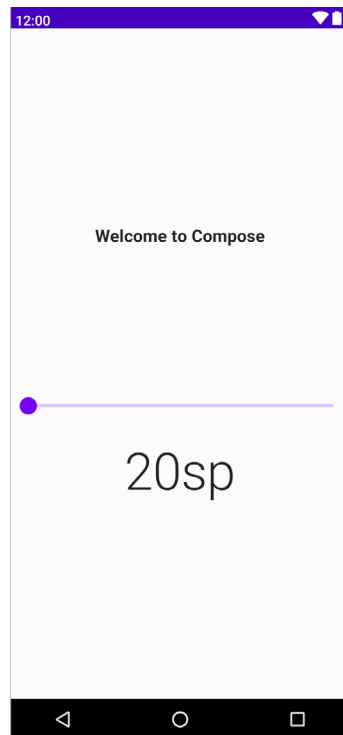


Figure 3-1

## 3.2 Creating the project

The first step in building an app is to create a new project within Android Studio. Begin, therefore, by launching Android Studio so that the “Welcome to Android Studio” screen appears as illustrated in Figure 3-2:

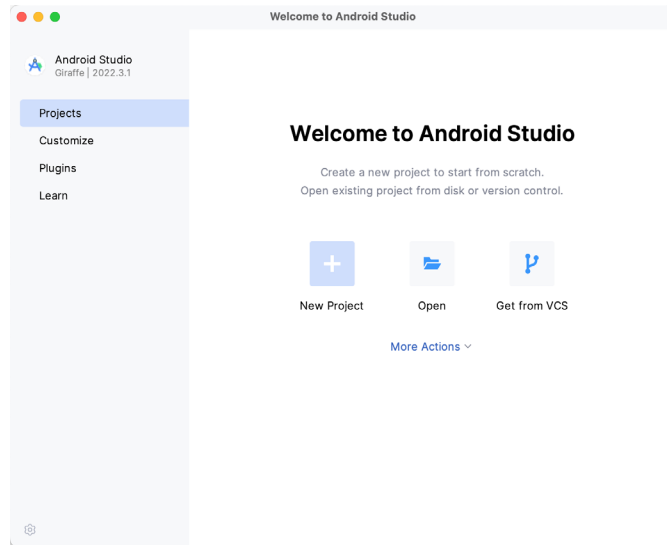


Figure 3-2

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* button 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 that is to be created for the application. The left-hand panel provides a list of platform categories from which the *Phone and Tablet* option must be selected. Although various activity types are available when developing Android applications, only the *Empty Activity* template provides a pre-configured project ready to work with Compose. Select this option before clicking on the *Next* button:

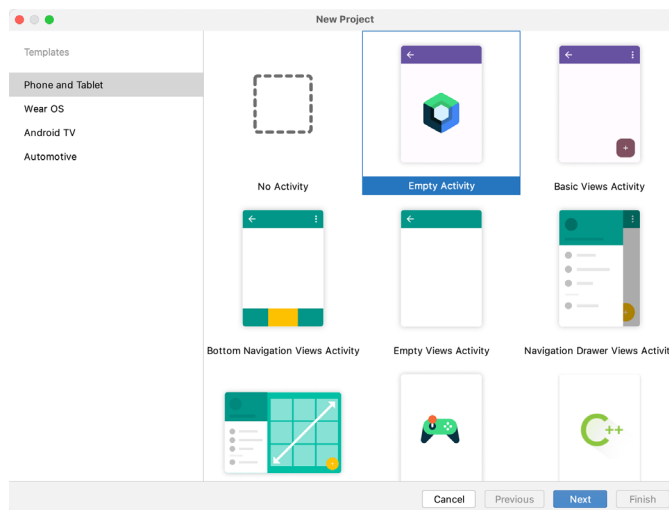


Figure 3-3

### 3.4 Defining the project and SDK settings

In the project configuration window (Figure 3-4), set the *Name* field to *ComposeDemo*. 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:

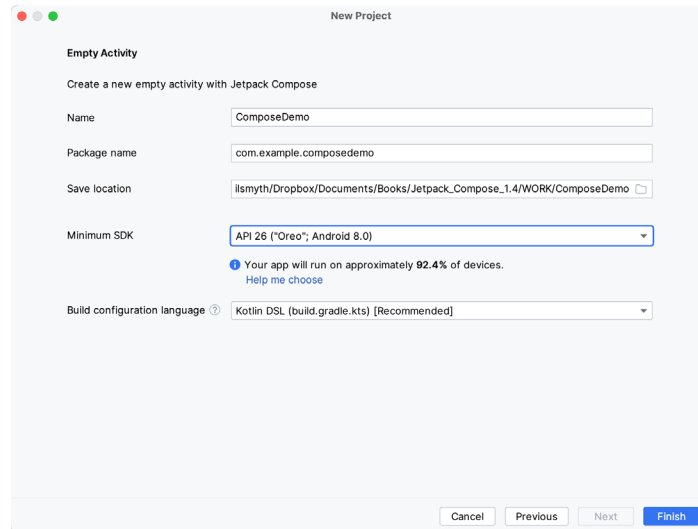


Figure 3-4

The *Package name* uniquely identifies the application within the Google Play app store 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 *ComposeDemo*, then the package name might be specified as follows:

```
com.mycompany.composedemo
```

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.composedemo
```

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: Android 8.0 (Oreo). This is the minimum SDK that 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 also 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* link to see a full breakdown of the various Android versions still in use:

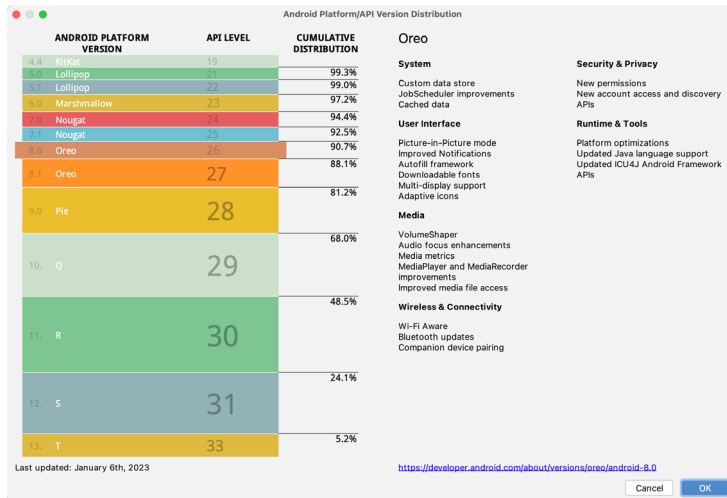


Figure 3-5

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

### 3.5 Enabling the New Android Studio UI

Android Studio is transitioning to a new, modern user interface that is not enabled by default in the Giraffe version. If your installation of Android Studio resembles Figure 3-6 below, then you will need to enable the new UI before proceeding:

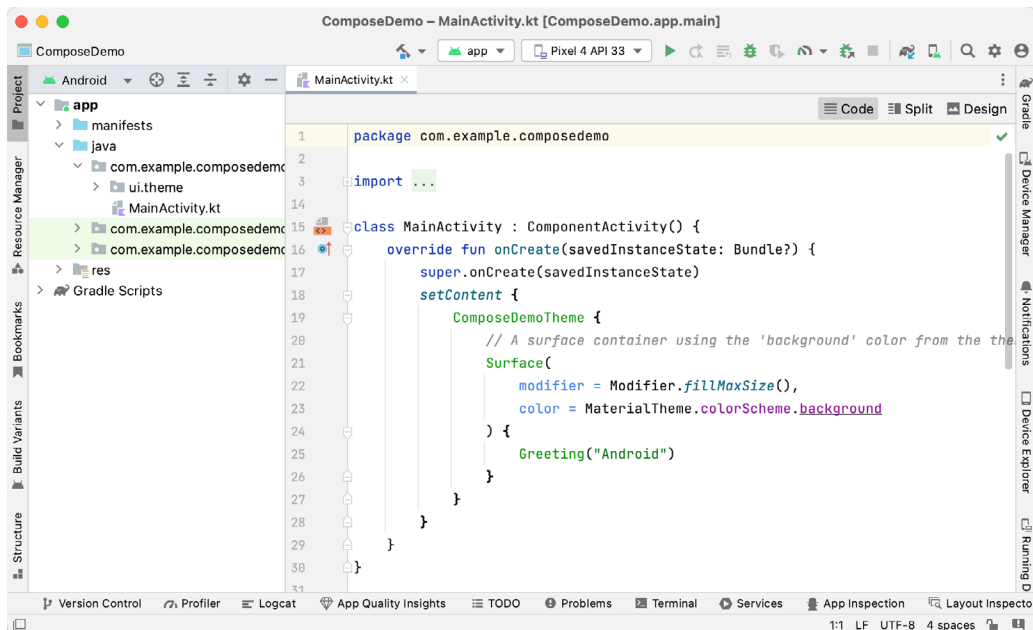


Figure 3-6

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

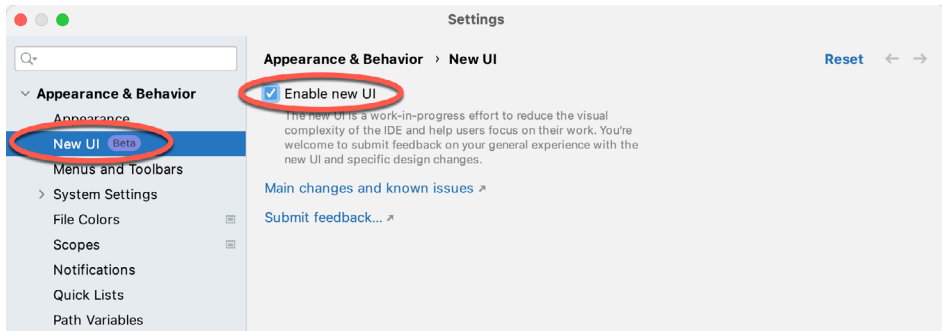


Figure 3-7

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

### 3.6 Previewing the example project

Once Android Studio has restarted, the main window will reappear using the new UI and containing our `AndroidSample` project as illustrated in Figure 3-8 below:

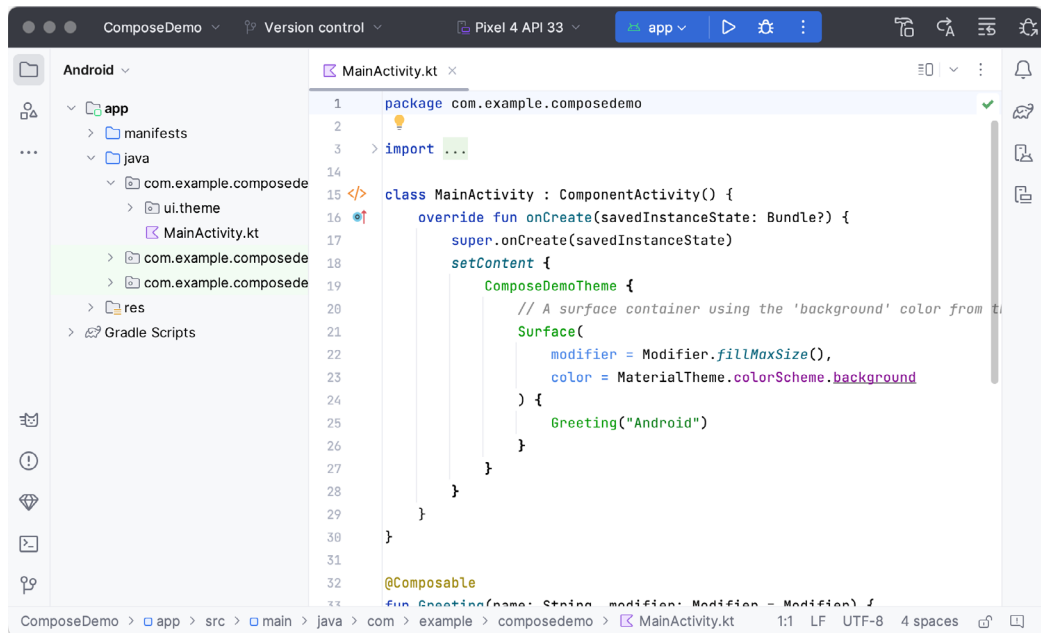


Figure 3-8

The newly created project and references to associated files are listed in the *Project* tool window located on the left-hand 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 3-9. If the panel is not currently in Android mode, use the menu to switch mode:

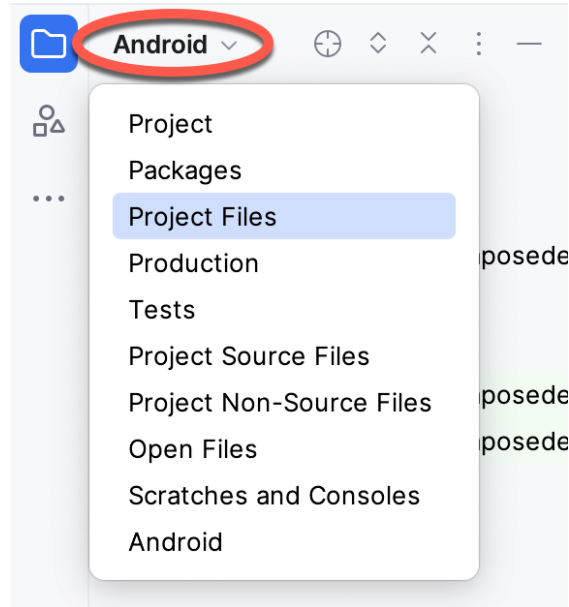


Figure 3-9

The code for the main activity of the project (an activity corresponds to a single user interface screen or module within an Android app) is contained within the *MainActivity.kt* file located under *app* -> *java* -> *com.example.composedemo* within the Project tool window as indicated in Figure 3-10:

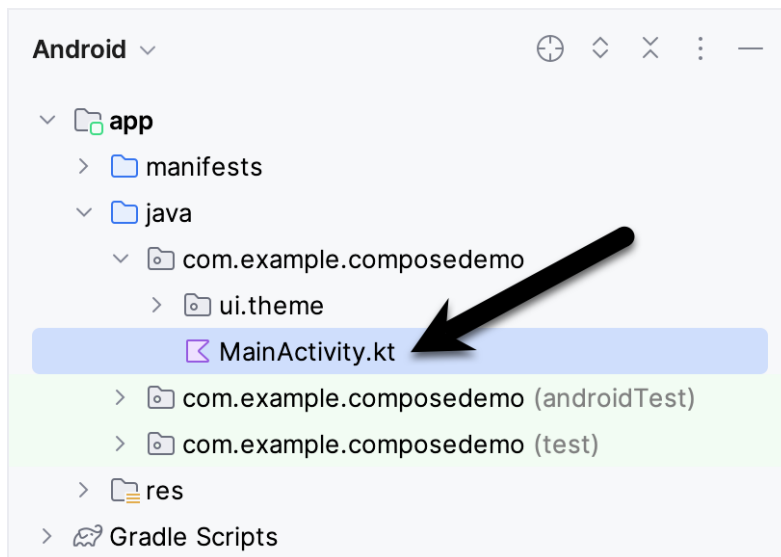


Figure 3-10

Double-click on this file to load it into the main code editor panel. The editor can be used in different view modes. Only the source code of the currently selected file is visible when the editor is in Code mode (as it is in Figure 3-8 above). However, the most useful when working with Compose is Split mode. To switch between Code and Split modes, click on the button marked A in Figure 3-11 below:

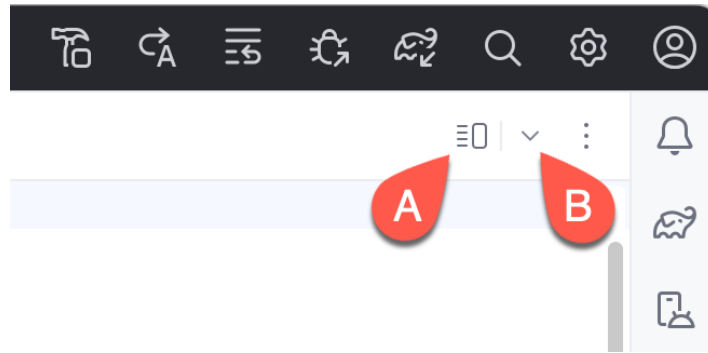


Figure 3-11

The button marked B above displays a menu that may also be used to change editor view modes:

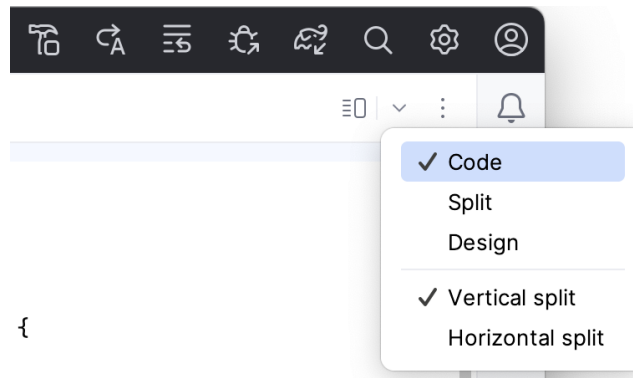


Figure 3-12

Split mode displays the code editor (A) alongside the Preview panel (B) in which the current user interface design will appear:

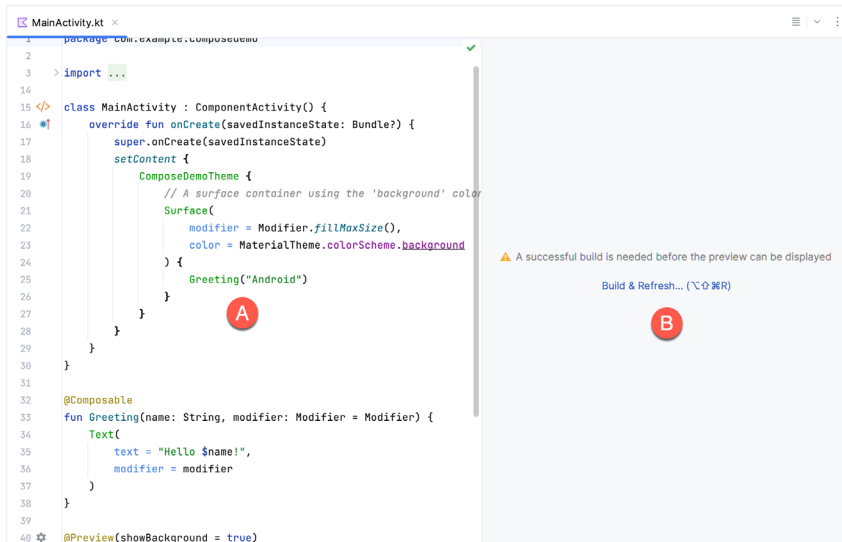


Figure 3-13

## A Compose Project Overview

Only the Preview panel is displayed when the editor is in Design mode.

To get us started, Android Studio has already added some code to the *MainActivity.kt* file to display a Text component configured to display a message which reads “Hello Android”.

If the project has not yet been built, the Preview panel will display the message shown in Figure 3-14:

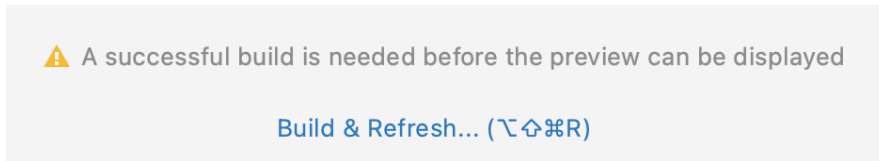


Figure 3-14

If you see this notification, click on the *Build & Refresh* link to rebuild the project. After the build is complete, the Preview panel should update to display the user interface defined by the code in the *MainActivity.kt* file:

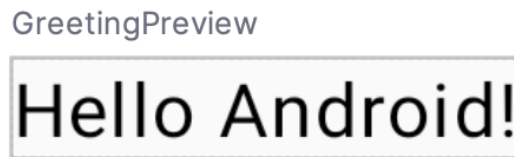


Figure 3-15

### 3.7 Reviewing the main activity

Android applications are created by combining one or more elements known as *Activities*. An activity is a single, standalone module of application functionality that either correlates directly to a single user interface screen and its corresponding functionality, or acts as a container for a collection of related screens. An appointments application might, for example, contain an activity screen that displays appointments set up for the current day. The application might also utilize a second activity consisting of multiple screens where new appointments may be entered by the user and existing appointments edited.

When we created the ComposeDemo project, Android Studio created a single initial activity for our app, named it MainActivity, and generated some code for it in the *MainActivity.kt* file. This activity contains the first screen that will be displayed when the app is run on a device. Before we modify the code for our requirements in the next chapter, it is worth taking some time to review the code currently contained within the *MainActivity.kt* file.

The file begins with the following line (keep in mind that this may be different if you used your own domain name instead of *com.example*):

```
package com.example.composedemo
```

This tells the build system that the classes and functions declared in this file belong to the *com.example.composedemo* package which we configured when we created the project.

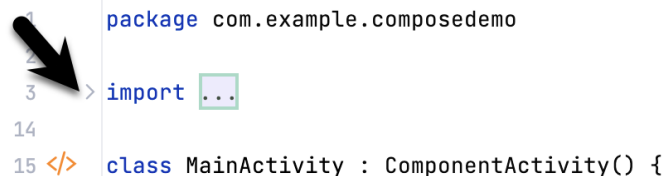
Next are a series of *import* directives. The Android SDK comprises a vast collection of libraries that provide the foundation for building Android apps. If all of these libraries were included within an app the resulting app bundle would be too large to run efficiently on a mobile device. To avoid this problem an app only imports the libraries that it needs to be able to run:

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
```



```
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
.
.
```

Initially, the list of import directives will most likely be “folded” to save space. To unfold the list, click on the small disclosure button indicated by the arrow in Figure 3-16 below:



```
1 package com.example.composedemo
3 > import ..
14
15 </> class MainActivity : ComponentActivity() {
```

Figure 3-16

The MainActivity class is then declared as a subclass of the Android ComponentActivity class:

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

The MainActivity class implements a single method in the form of *onCreate()*. This is the first method that is called when an activity is launched by the Android runtime system and is an artifact of the way apps used to be developed before the introduction of Compose. The *onCreate()* method is used here to provide a bridge between the containing activity and the Compose-based user interfaces that are to appear within it:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
        ComposeDemoTheme {
.
.
        }
    }
}
```

The method declares that the content of the activity’s user interface will be provided by a composable function named *ComposeDemoTheme*. This composable function is declared in the *Theme.kt* file located under the *app* -> *<package name>* -> *ui.theme* folder in the Project tool window. This, along with the other files in the *ui.theme* folder defines the colors, fonts, and shapes to be used by the activity and provides a central location from which to customize the overall theme of the app’s user interface.

The call to the *ComposeDemoTheme* composable function is configured to contain a *Surface* composable. *Surface* is a built-in Compose component designed to provide a background for other composables:

```
ComposeDemoTheme {
    // A surface container using the 'background' color from the theme
    Surface(
```

## A Compose Project Overview

```
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    }
}
```

In this case, the `Surface` component is configured to fill the entire screen and with the background set to the standard background color defined by the Android Material Design theme. Material Design is a set of design guidelines developed by Google to provide a consistent look and feel across all Android apps. It includes a theme (including fonts and colors), a set of user interface components (such as button, text, and a range of text fields), icons, and generally defines how an Android app should look, behave and respond to user interactions.

Finally, the `Surface` is configured to contain a composable function named `Greeting` which is passed a string value that reads “Android”:

```
ComposeDemoTheme {
    // A surface container using the 'background' color from the theme
    Surface(
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    ) {
        Greeting("Android")
    }
}
```

Outside of the scope of the `MainActivity` class, we encounter our first composable function declaration within the activity. The function is named `Greeting` and is, unsurprisingly, marked as being composable by the `@Composable` annotation:

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

The function accepts a `String` parameter (labeled *name*) and calls the built-in `Text` composable, passing through a string value containing the word “Hello” concatenated with the name parameter. The function also accepts an optional modifier parameter (a topic covered in the chapter titled “*Using Modifiers in Compose*”). As will soon become evident as you work through the book, composable functions are the fundamental building blocks for developing Android apps using `Compose`.

The second composable function declared in the `MainActivity.kt` file reads as follows:

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    ComposeDemoTheme {
        Greeting("Android")
    }
}
```

Earlier in the chapter, we looked at how the Preview panel allows us to see how the user interface will appear without having to compile and run the app. At first glance, it would be easy to assume that the preview rendering is generated by the code in the `onCreate()` method. In fact, that method only gets called when the app runs on a device or emulator. Previews are generated by preview composable functions. The `@Preview` annotation associated with the function tells Android Studio that this is a preview function and that the content emitted by the function is to be displayed in the Preview panel. As we will see later in the book, a single activity can contain multiple preview composable functions configured to preview specific sections of a user interface using different data values.

In addition, each preview may be configured by passing parameters to the `@Preview` annotation. For example, to view the preview with the rest of the standard Android screen decorations, modify the preview annotation so that it reads as follows:

```
@Preview(showSystemUi = true)
```

Once the preview has been updated, it should now be rendered as shown in Figure 3-17:

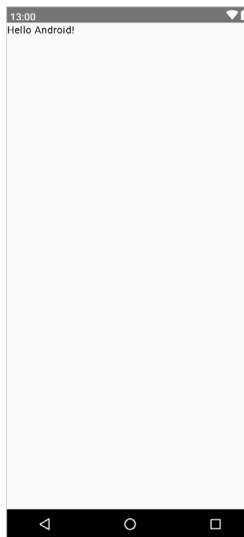


Figure 3-17

### 3.8 Preview updates

One final point worth noting is that the Preview panel is live and will automatically reflect minor changes made to the composable functions that make up a preview. To see this in action, edit the call to the Greeting function in the `GreetingPreview()` preview composable function to change the name from “Android” to “Compose”. Note that as you make the change in the code editor, it is reflected in the preview.

More significant changes will require a build and refresh before being reflected in the preview. When this is required, Android Studio will display the following “Out of date” notice at the top of the Preview panel and a *Build & Refresh* button (indicated by the arrow in Figure 3-18):

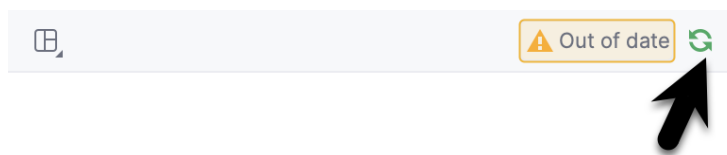


Figure 3-18

## A Compose Project Overview

Simply click on the button to update the preview for the latest changes. Occasionally, Android Studio will fail to update the preview after code changes. If you believe that the preview no longer matches your code, hover the mouse pointer over the Up-to-date status text and select Build & Refresh from the resulting menu, as illustrated in Figure 3-19:

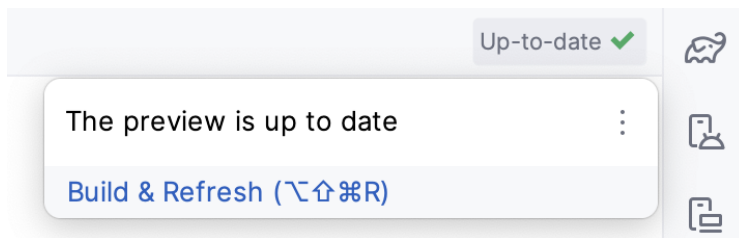


Figure 3-19

The Preview panel also includes an interactive mode that allows you to trigger events on the user interface components (for example, clicking buttons, moving sliders, scrolling through lists, etc.). Since ComposeDemo contains only an inanimate Text component at this stage, it makes more sense to introduce interactive mode in the next chapter.

### 3.9 Bill of Materials and the Compose version

Although Jetpack Compose and Android Studio appear to be tightly integrated, they are two separate products developed by different teams at Google. As a result, there is no guarantee that the most recent Android Studio version will default to using the latest version of Jetpack Compose. It can, therefore, be helpful to know which version of Jetpack Compose is being used by Android Studio. This is declared in a *Bill of Materials* (BOM) setting within the build configuration files of your Android Studio projects.

To identify the BOM for a project, locate the *Gradle Scripts* -> *build.gradle.kts* (*Module: app*) file (highlighted in the figure below) and double-click on it to load it into the editor:

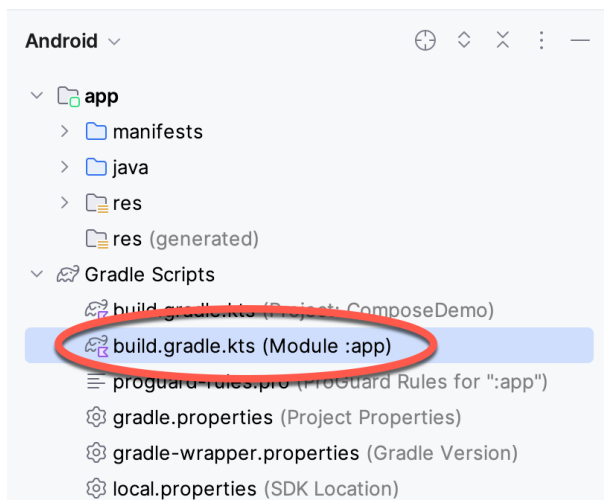


Figure 3-20

With the file loaded into the editor, locate the *compose-bom* entry in the dependencies section:

```
dependencies {
```

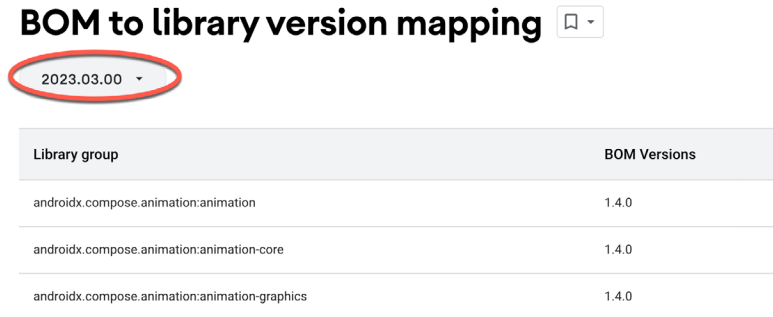
```
.  
.
```

```
implementation(platform("androidx.compose:compose-bom:2023.03.00"))
```

In the above example, we can see that the project is using BOM 2023.03.00. With this information, we can use the *BOM to library version mapping* web page at the following URL to identify the library versions being used to build our app:

<https://developer.android.com/jetpack/compose/bom/bom-mapping>

Once the web page has loaded, select the BOM version from the menu highlighted in Figure 3-21 below. For example, the figure shows that BOM 2023.03.00 uses version 1.4.0 of the Compose libraries:



Library group	BOM Versions
androidx.compose.animation:animation	1.4.0
androidx.compose.animation:animation-core	1.4.0
androidx.compose.animation:animation-graphics	1.4.0

Figure 3-21

The BOM does not currently define the versions of all the dependencies listed in the build file. Therefore, you will see some library dependencies in the *build.gradle.kts* file that include a specific version number, as is the case with the *core-ktx* and *lifecycle-runtime-ktx* libraries:

```
dependencies {
    implementation 'androidx.core:core-ktx:1.9.0'
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.6.1'
    .
    .
}
```

You can add specific version numbers to any libraries you add to the dependencies, though it is recommended to rely on the BOM settings whenever possible to ensure library compatibility. However, a version number declaration will be required when adding libraries not listed in the BOM. You can also override the BOM version of a library by appending a version number to the declaration. The following declaration, for example, overrides the version number in the BOM for the *compose.ui* library:

```
implementation 'androidx.compose.ui:ui:1.3.3'
```

### 3.10 Summary

In this chapter, we have created a new project using Android Studio's *Empty Activity* template and explored some of the code automatically generated for the project. We have also introduced several features of Android Studio designed to make app development with Compose easier. The most useful features, and the places where you will spend most of your time while developing Android apps, are the code editor and Preview panel.

While the default code in the *MainActivity.kt* file provides an interesting example of a basic user interface, it bears no resemblance to the app we want to create. In the next chapter, we will modify and extend the app by removing some of the template code and writing our own composable functions.



## 4. An Example Compose Project

In the previous chapter, we created a new Compose-based Android Studio project named ComposeDemo and took some time to explore both Android Studio and some of the project code that it generated to get us started. With those basic steps covered, this chapter will use the ComposeDemo project as the basis for a new app. This will involve the creation of new composable functions, introduce the concept of state, and make use of the Preview panel in interactive mode. As with the preceding chapter, key concepts explained in basic terms here will be covered in significantly greater detail in later chapters.

### 4.1 Getting started

Start Android Studio if it is not already running and open the ComposeDemo project created in the previous chapter. Once the project has loaded, double-click on the *MainActivity.kt* file (located in Project tool window under *app* -> *java* -> *<package name>*) to open it in the code editor. If necessary, switch the editor into Split mode so that both the editor and Preview panel are visible.

### 4.2 Removing the template Code

Within the *MainActivity.kt* file, delete some of the template code so that the file reads as follows:

```
package com.example.composedemo
.
.
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeDemoTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Greeting("Android")
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

†

```
@Preview(showSystemUi = true)
@Composable
fun GreetingPreview() {
    ComposeDemoTheme {
        Greeting("Android")
    }
}
```

†

### 4.3 The Composable hierarchy

Before we write the composable functions that will make up our user interface, it helps to visualize the relationships between these components. The ability of one composable to call other composables essentially allows us to build a hierarchy tree of components. Once completed, the composable hierarchy for our ComposeDemo main activity can be represented as shown in Figure 4-1:

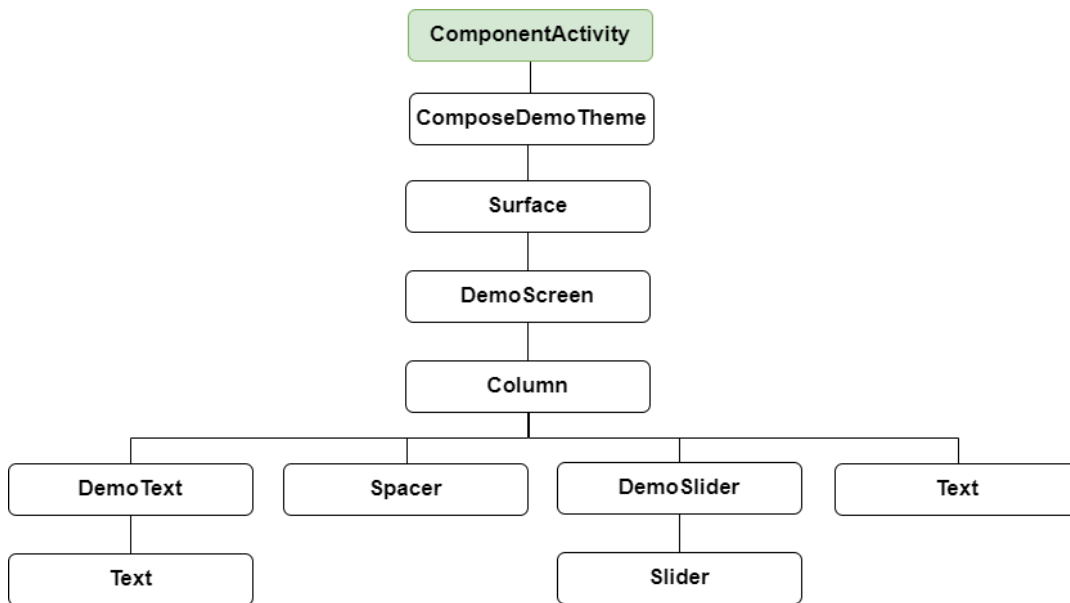


Figure 4-1

All of the elements in the above diagram, except for `ComponentActivity`, are composable functions. Of those functions, the `Surface`, `Column`, `Spacer`, `Text`, and `Slider` functions are built-in composables provided by Compose. The `DemoScreen`, `DemoText`, and `DemoSlider` composables, on the other hand, are functions that we will create to provide both structure to the design and the custom functionality we require for our app. You can find the `ComposeDemoTheme` composable declaration in the `ui.theme -> Theme.kt` file.

### 4.4 Adding the DemoText composable

We are now going to add a new composable function to the activity to represent the `DemoText` item in the hierarchy tree. The purpose of this composable is to display a text string using a font size value that adjusts in real-time as the slider moves. Place the cursor beneath the final closing brace `}` of the `MainActivity` declaration and add the following function declaration:

```
@Composable
```



```
fun DemoText() {
}
```

The `@Composable` annotation notifies the build system that this is a composable function. When the function is called, the plan is for it to be passed both a text string and the font size at which that text is to be displayed. This means that we need to add some parameters to the function:

```
@Composable
fun DemoText(message: String, fontSize: Float) {
}
```

The next step is to make sure the text is displayed. To achieve this, we will make a call to the built-in `Text` composable, passing through as parameters the message string, font size and, to make the text more prominent, a bold font weight setting:

```
@Composable
fun DemoText(message: String, fontSize: Float) {
    Text(
        text = message,
        fontSize = fontSize.sp,
        fontWeight = FontWeight.Bold
    )
}
```

Note that after making these changes, the code editor indicates that “`sp`” and “`FontWeight`” are undefined. This happens because these are defined and implemented in libraries that have not yet been imported into the `MainActivity.kt` file. One way to resolve this is to click on an undefined declaration so that it highlights as shown below, and then press `Alt+Enter` (`Opt+Enter` on macOS) on the keyboard to import the missing library automatically:



```
32     @Composable
33     fun DemoText(message: String, fontSize: Float) {
34         Text(
35             text = message,
36             fontSize = fontSize.sp,
37             fontWeight = FontWeight.Bold
38         )
39     }
```

Figure 4-2

Alternatively, you may add the missing import statements manually to the list at the top of the file:

```
.
.
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.sp
.
.
```

In the remainder of this book, all code examples will include any required library import statements.

We have now finished writing our first composable function. Notice that, except for the font weight, all the other

properties are passed to the function when it is called (a function that calls another function is generally referred to as the *caller*). This increases the flexibility, and therefore re-usability, of the `DemoText` composable and is a key goal to keep in mind when writing composable functions.

### 4.5 Previewing the `DemoText` composable

At this point, the Preview panel will most likely be displaying a message which reads “No preview found”. The reason for this is that our `MainActivity.kt` file does not contain any composable functions prefixed with the `@Preview` annotation. Add a preview composable function for `DemoText` to the `MainActivity.kt` file as follows:

```
@Preview
@Composable
fun DemoTextPreview() {
    ComposeDemoTheme {
        DemoText(message = "Welcome to Android", fontSize = 12f)
    }
}
```

After adding the preview composable, the Preview panel should have detected the change and displayed the link to build and refresh the preview rendering. Click the link and wait for the rebuild to complete, at which point the `DemoText` composable should appear as shown in Figure 4-3:



Figure 4-3

Minor changes made to the code in the `MainActivity.kt` file such as changing values will be instantly reflected in the preview without the need to build and refresh. For example, change the “Welcome to Android” text literal to “Welcome to Compose” and note that the text in the Preview panel changes as you type. Similarly, increasing the font size literal will instantly change the size of the text in the preview. This feature is referred to as Live Edit.

### 4.6 Adding the `DemoSlider` composable

The `DemoSlider` composable is a little more complicated than `DemoText`. It will need to be passed a variable containing the current slider position and an event handler function or lambda to call when the slider is moved by the user so that the new position can be stored and passed to the two `Text` composables. With these requirements in mind, add the function as follows:

```
.
.
import androidx.compose.foundation.layout.*
import androidx.compose.material3.Slider
import androidx.compose.ui.unit.dp
.
.
@Composable
fun DemoSlider(sliderPosition: Float, onPositionChange: (Float) -> Unit ) {
    Slider(
        modifier = Modifier.padding(10.dp),
```

```

        valueRange = 20f..38f,
        value = sliderPosition,
        onChange = { onPositionChange(it) }
    )
}

```

The DemoSlider declaration contains a single Slider composable which is, in turn, passed four parameters. The first is a Modifier instance configured to add padding space around the slider. Modifier is a Kotlin class built into Compose which allows a wide range of properties to be set on a composable within a single object. Modifiers can also be created and customized in one composable before being passed to other composables where they can be further modified before being applied.

The second value passed to the Slider is a range allowed for the slider value (in this case the slider is limited to values between 20 and 38).

The next parameter sets the value of the slider to the position passed through by the caller. This ensures that each time DemoSlider is recomposed it retains the last position value.

Finally, we set the *onChange* parameter of the Slider to call the function or lambda we will be passing to the DemoSlider composable when we call it later. Each time the slider position changes, the call will be made and passed the current value which we can access via the Kotlin *it* keyword. We can further simplify this by assigning just the event handler parameter name (*onPositionChange*) and leaving the compiler to handle the passing of the current value for us:

```
onChange = onPositionChange
```

## 4.7 Adding the DemoScreen composable

The next step in our project is to add the DemoScreen composable. This will contain a variable named *sliderPosition* in which to store the current slider position and the implementation of the *handlePositionChange* event handler to be passed to the DemoSlider. This lambda will be responsible for storing the current position in the *sliderPosition* variable each time it is called with an updated value. Finally, DemoScreen will contain a Column composable configured to display the DemoText, Spacer, DemoSlider and the second, as yet to be added, Text composable in a vertical arrangement.

Start by adding the DemoScreen function as follows:

```

.
.
import androidx.compose.runtime.*
.
.
@Composable
fun DemoScreen() {

    var sliderPosition by remember { mutableStateOf(20f) }

    val handlePositionChange = { position : Float ->
        sliderPosition = position
    }
}

```

## An Example Compose Project

The `sliderPosition` variable declaration requires some explanation. As we will learn later, the Compose system repeatedly and rapidly *recomposes* user interface layouts in response to data changes. The change of slider position will, therefore, cause `DemoScreen` to be recomposed along with all of the composables it calls. Consider if we had declared and initialized our `sliderPosition` variable as follows:

```
var sliderPosition = 20f
```

Suppose the user slides the slider to position 21. The `handlePositionChange` event handler is called and stores the new value in the `sliderPosition` variable as follows:

```
val handlePositionChange = { position : Float ->
    sliderPosition = position
}
```

The Compose runtime system detects this data change and recomposes the user interface, including a call to the `DemoScreen` function. This will, in turn, reinitialize the `sliderPosition` target state causing the previous value of 21 to be lost. Declaring the `sliderPosition` variable in this way informs Compose that the current value needs to be remembered during recompositions:

```
var sliderPosition by remember { mutableStateOf(20f) }
```

The only remaining work within the `DemoScreen` implementation is to add a `Column` containing the required composable functions:

```
.
.
import androidx.compose.ui.Alignment
.
.
@Composable
fun DemoScreen() {

    var sliderPosition by remember { mutableStateOf(20f) }

    val handlePositionChange = { position : Float ->
        sliderPosition = position
    }

    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center,
        modifier = Modifier.fillMaxSize()
    ) {

        DemoText(message = "Welcome to Compose", fontSize = sliderPosition)

        Spacer(modifier = Modifier.height(150.dp))

        DemoSlider(
            sliderPosition = sliderPosition,
            onPositionChange = handlePositionChange
        )
    }
}
```

```

    )

    Text(
        style = MaterialTheme.typography.headlineMedium,
        text = sliderPosition.toInt().toString() + "sp"
    )
}

```

Points to note regarding these changes may be summarized as follows:

- When `DemoSlider` is called, it is passed a reference to our `handlePositionChange` event handler as the `onPositionChange` parameter.
- The `Column` composable accepts parameters that customize layout behavior. In this case, we have configured the column to center its children both horizontally and vertically.
- A `Modifier` has been passed to the `Spacer` to place a 150dp vertical space between the `DemoText` and `DemoSlider` components.
- The second `Text` composable is configured to use the `headlineMedium` style of the `Material` theme. In addition, the `sliderPosition` value is converted from a `Float` to an integer so that only whole numbers are displayed and then converted to a string value before being displayed to the user.

## 4.8 Previewing the `DemoScreen` composable

To confirm that the `DemoScreen` layout meets our expectations, we need to modify the `DemoTextPreview` composable:

```

.
.
@Preview(showSystemUi = true)
@Composable
fun DemoTextPreview() {
    ComposeDemoTheme {
        DemoScreen()
    }
}

```

Note that we have enabled the `showSystemUi` property of the preview so that we will experience how the app will look when running on an Android device.

After performing a preview rebuild and refresh, the user interface should appear as originally shown in Figure 3-1.

## 4.9 Adjusting preview settings

The `showSystemUi` preview property is only one of many preview configuration options provided by Android Studio. In addition, properties are available to change configuration settings, such as the device type, screen size, orientation, API level, and locale. To access these configuration settings, click on the `Preview` configuration picker button located in the gutter to the left of the `@Preview` line in the code editor, as shown in Figure 4-4:

```

90
91 ⚙️ @Preview(showSystemUi = true)
92 @Composable
93 ▶️ fun DemoTextPreview() {
94     ComposeDemoTheme() {
95         DemoScreen()
96     }
97 }

```

Figure 4-4

When the button is clicked, the panel shown in Figure 4-5 will appear, from which the full range of preview configuration settings is available:

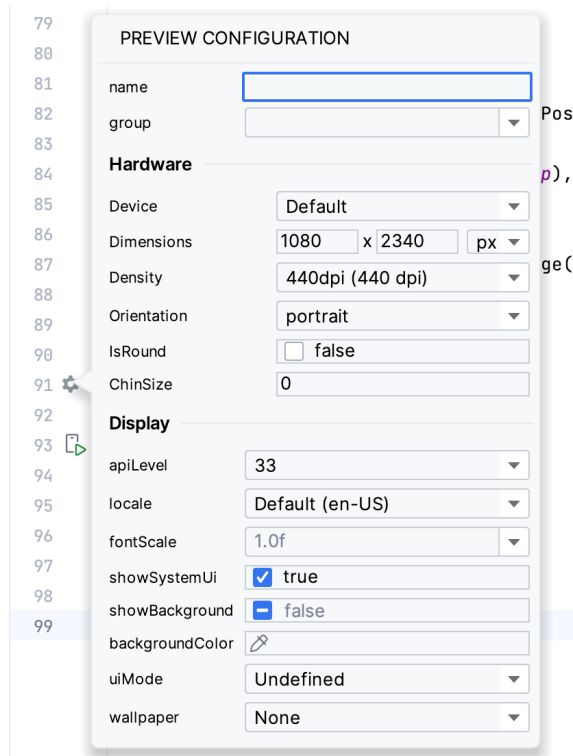


Figure 4-5

## 4.10 Testing in interactive mode

At this stage, we know that the user interface layout for our activity looks how we want it to, but we don't know if it will behave as intended. One option is to run the app on an emulator or physical device (topics covered in later chapters). A quicker option, however, is to switch the preview panel into interactive mode. To start interactive mode, hover the mouse pointer over the area above the preview canvas so that the two buttons shown in Figure 4-6 appear and click on the left-most button:

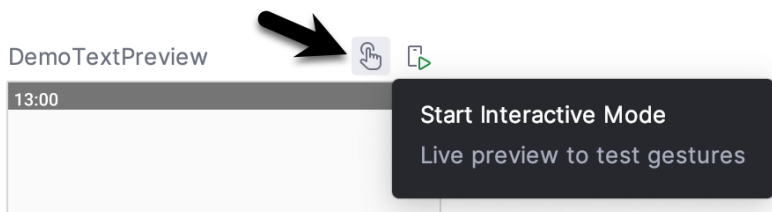


Figure 4-6

When clicked, there will be a short delay when interactive mode starts, after which it should be possible to move the slider and watch the two Text components update:

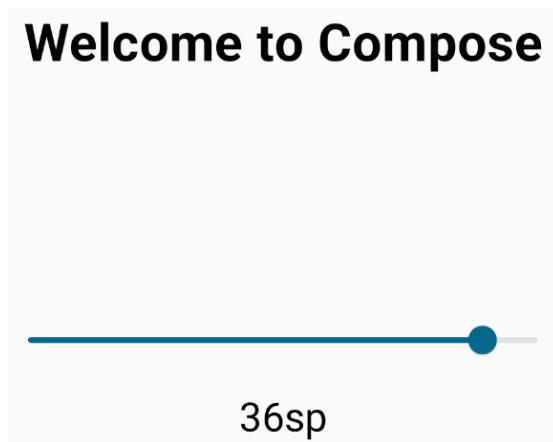


Figure 4-7

Click the button (highlighted in Figure 4-8 below) to exit interactive mode:

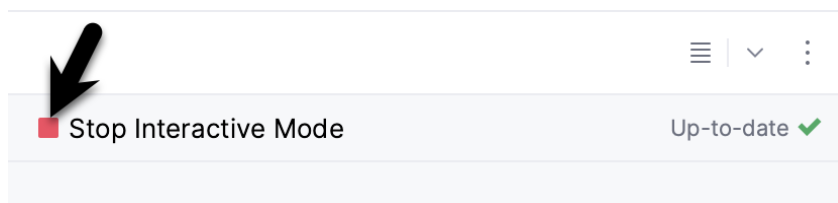


Figure 4-8

## 4.11 Completing the project

The final step is to make sure that the `DemoScreen` composable is called from within the `Surface` function located in the `onCreate()` method of the `MainActivity` class. Locate this method and modify it as follows:

```

·
·
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeDemoTheme {
                Surface (

```

## An Example Compose Project

```
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    ) {
        DemoScreen()
    }
}
}
```

This will ensure that, in addition to appearing in the preview panel, our user interface will also be displayed when the app runs on a device or emulator (a topic that will be covered in later chapters).

## 4.12 Summary

In this chapter, we have extended our ComposeDemo project to include some additional user interface elements in the form of two Text composables, a Spacer, and a Slider. These components were arranged vertically using a Column composable. We also introduced the concept of mutable state variables and explained how they are used to ensure that the app remembers state when the Compose runtime performs recompositions. The example also demonstrated how to use event handlers to respond to user interaction (in this case, the user moving a slider). Finally, we made use of the Preview panel in interactive mode to test the app without the need to compile and run it on an emulator or physical device.



## 6. Using and Configuring the Android Studio AVD Emulator

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

### 6.1 The Emulator Environment

When launched in standalone mode, the emulator displays an initial splash screen during the loading process. Once loaded, the main emulator window appears, containing a representation of the chosen device type (in the case of Figure 6-1, this is a Pixel 4 device):

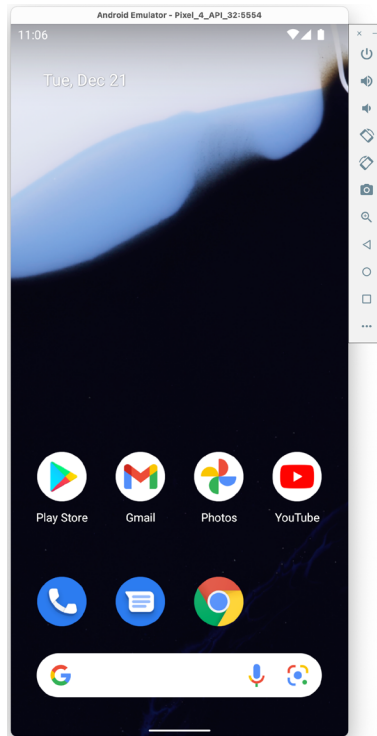


Figure 6-1

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

### 6.2 Emulator Toolbar Options

The emulator toolbar (Figure 6-2) provides access to a range of options relating to the appearance and behavior of the emulator environment.

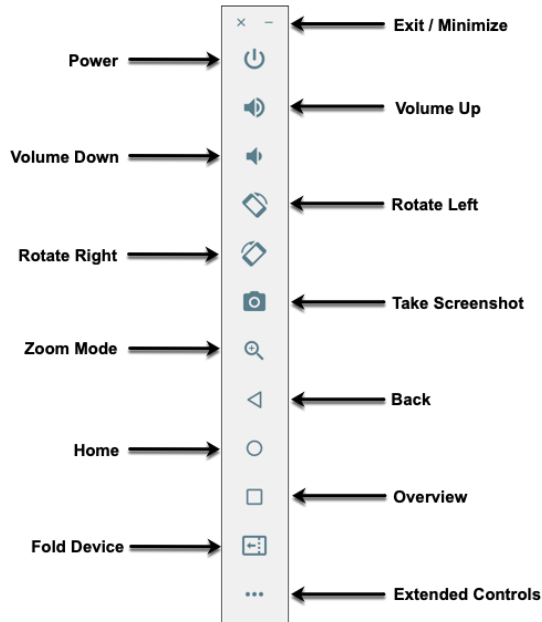


Figure 6-2

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

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

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

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

## 6.3 Working in Zoom Mode

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

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

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

## 6.4 Resizing the Emulator Window

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

## 6.5 Extended Control Options

The extended controls toolbar button displays the panel illustrated in Figure 6-3. By default, the location settings will be displayed. Selecting a different category from the left-hand panel will display the corresponding group of controls:

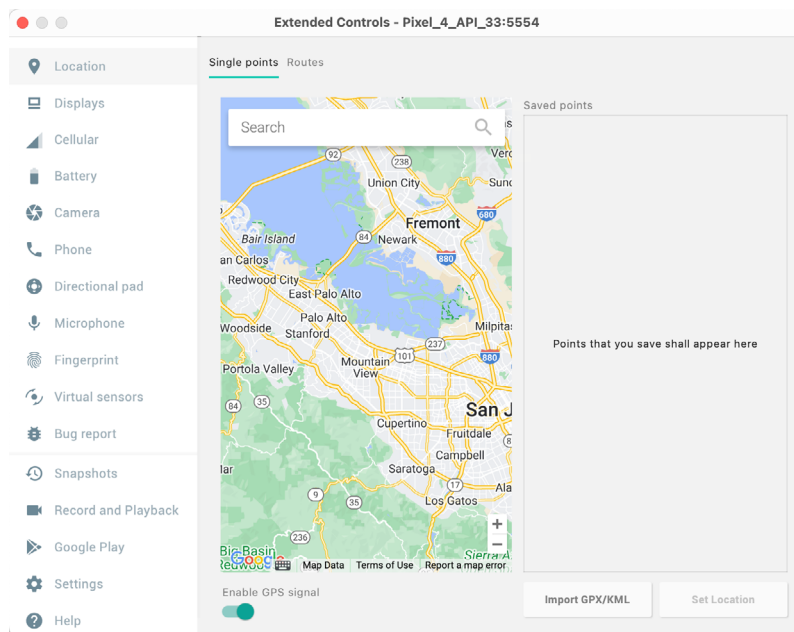


Figure 6-3

### 6.5.1 Location

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

### 6.5.2 Displays

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

### 6.5.3 Cellular

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

### 6.5.4 Battery

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

### 6.5.5 Camera

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

### 6.5.6 Phone

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

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

### 6.5.7 Directional Pad

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

### 6.5.8 Microphone

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

### 6.5.9 Fingerprint

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

### 6.5.10 Virtual Sensors

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

### 6.5.11 Snapshots

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

### 6.5.12 Record and Playback

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

### 6.5.13 Google Play

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

### 6.5.14 Settings

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

### 6.5.15 Help

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

## 6.6 Working with Snapshots

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

The Snapshots screen of the extended controls panel can store additional snapshots at any point during the execution of the emulator. This saves the exact state of the entire emulator allowing the emulator to be restored to the exact point in time that the snapshot was taken. From within the screen, snapshots can be taken using the *Take Snapshot* button (marked A in Figure 6-4). To restore an existing snapshot, select it from the list (B) and click the run button (C) located at the bottom of the screen. Options are also provided to edit (D) the snapshot name and description and to delete (E) the currently selected snapshot:

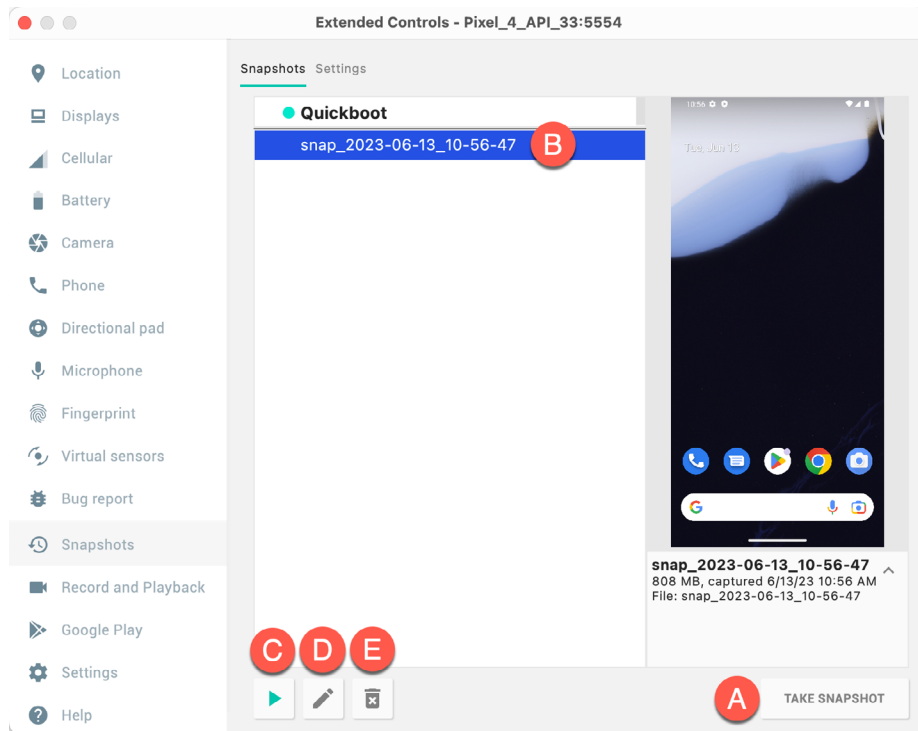


Figure 6-4

You can also choose whether to start an emulator using either a cold boot, the most recent quick-boot snapshot, or a previous snapshot by making a selection from the run target menu in the main toolbar, as illustrated in Figure 6-5:

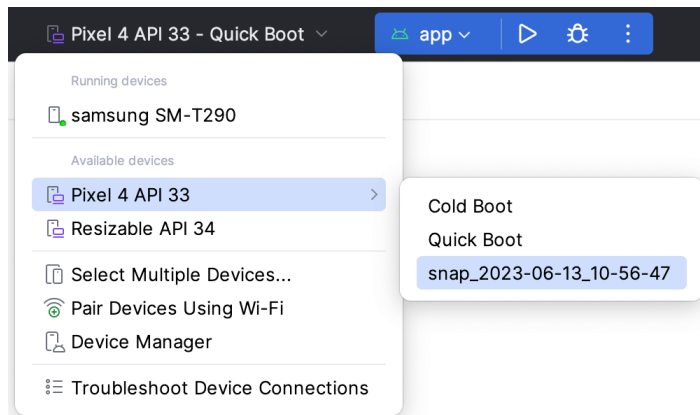


Figure 6-5

## 6.7 Configuring Fingerprint Emulation

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

Within the Security settings screen, select the fingerprint option. On the resulting information screen, click on

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

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

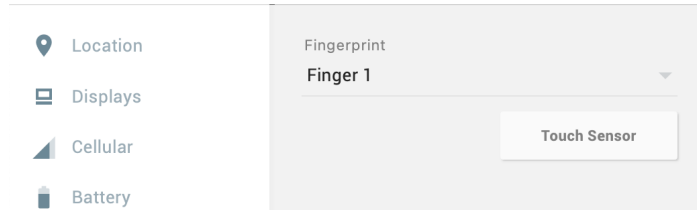


Figure 6-6

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

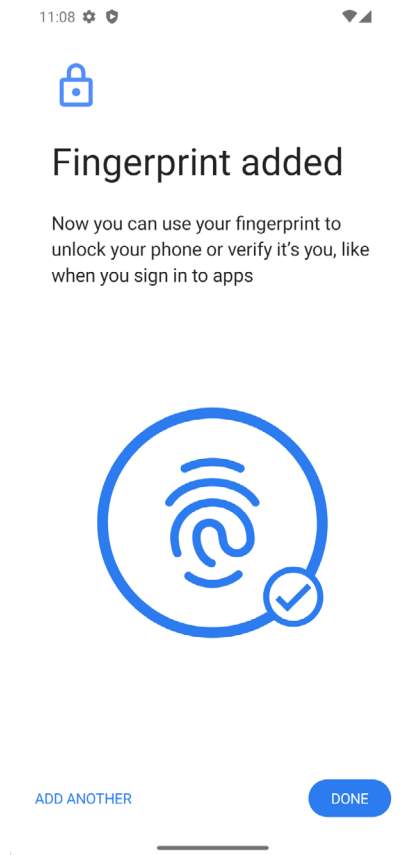


Figure 6-7

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

## 6.8 The Emulator in Tool Window Mode

As outlined in the previous chapter (“*Creating an Android Virtual Device (AVD) in Android Studio*”), Android Studio can be configured to launch the emulator in an embedded tool window so that it does not appear in a separate window. When running in this mode, the same controls available in standalone mode are provided in the toolbar, as shown in Figure 6-8:

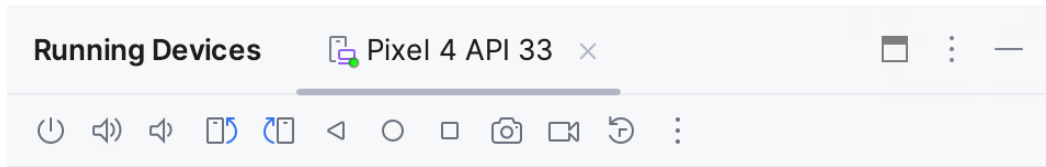


Figure 6-8

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

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

## 6.9 Creating a Resizable Emulator

In addition to emulators configured to match specific Android device models, Android Studio also provides a resizable AVD that allows you to switch between phone, tablet, and foldable device sizes. To create a resizable emulator, open the Device Manager and click the *Create device* button. Next, select the Resizable device definition illustrated in Figure 6-9, and follow the usual steps to create a new AVD:



Figure 6-9



When you run an app on the new emulator within a tool window, the *Display mode* option will appear in the toolbar, allowing you to switch between emulator configurations as shown in Figure 6-10:

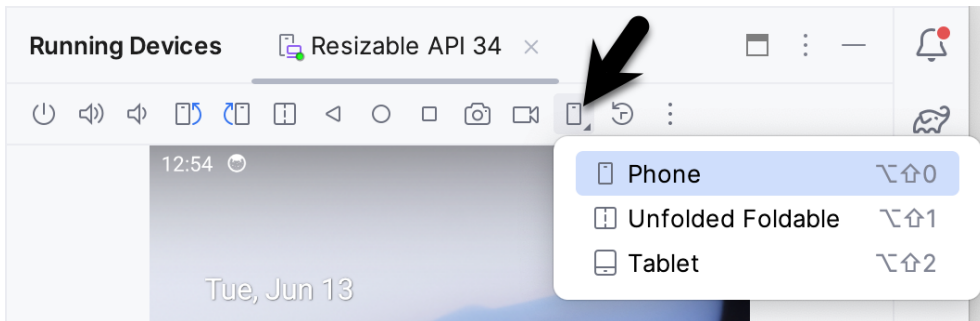


Figure 6-10

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

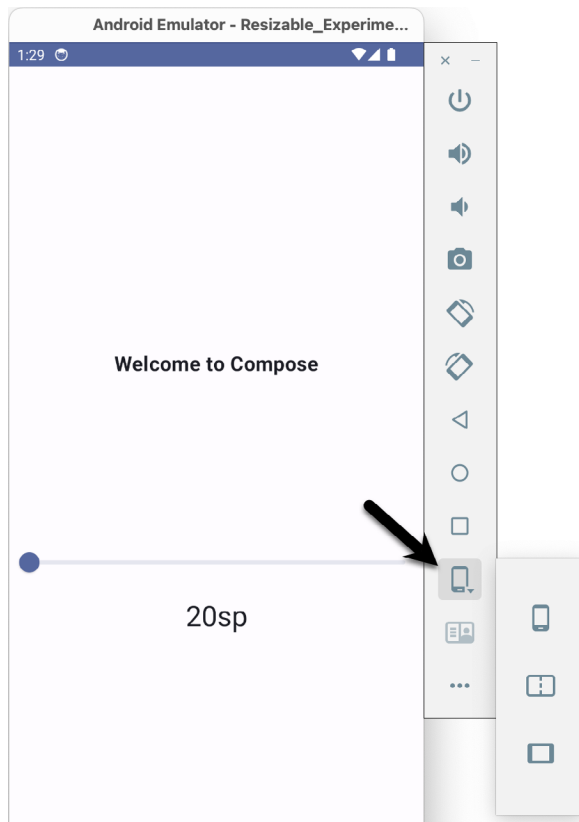


Figure 6-11

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



Figure 6-12

## 6.10 Summary

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

## 11. An Introduction to Kotlin

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

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

### 11.1 What is Kotlin?

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

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

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

### 11.2 Kotlin and Java

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

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

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

### 11.3 Converting from Java to Kotlin

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

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

### 11.4 Kotlin and Android Studio

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

### 11.5 Experimenting with Kotlin

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

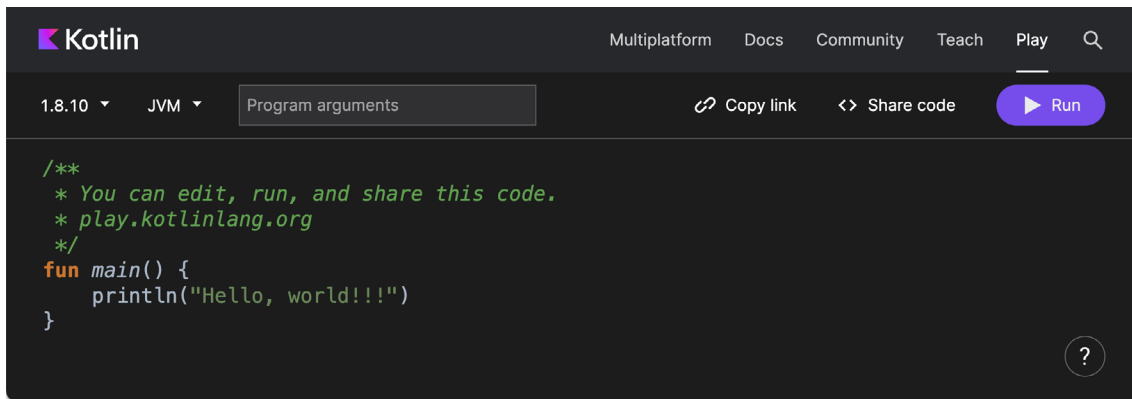


Figure 11-1

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

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

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

    println("Welcome to Kotlin")

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

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

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

Figure 11-2

## 11.6 Semi-colons in Kotlin

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

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

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

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

## 11.7 Summary

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

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

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



# 13. Kotlin Operators and Expressions

So far we have looked at using variables and constants in Kotlin and also described the different data types. Being able to create variables is only part of the story, however. The next step is to learn how to use these variables in Kotlin code. The primary method for working with data is in the form of *expressions*.

## 13.1 Expression syntax in Kotlin

The most basic expression consists of an *operator*, two *operands*, and an *assignment*. The following is an example of an expression:

```
val myresult = 1 + 2
```

In the above example, the (+) operator is used to add two operands (1 and 2) together. The *assignment operator* (=) subsequently assigns the result of the addition to a variable named *myresult*. The operands could just have easily been variables (or a mixture of values and variables) instead of the actual numerical values used in the example.

In the remainder of this chapter, we will look at the basic types of operators available in Kotlin.

## 13.2 The Basic assignment operator

We have already looked at the most basic of assignment operators, the = operator. This assignment operator simply assigns the result of an expression to a variable. In essence, the = assignment operator takes two operands. The left-hand operand is the variable to which a value is to be assigned and the right-hand operand is the value to be assigned. The right-hand operand is, more often than not, an expression that performs some type of arithmetic or logical evaluation or a call to a function, the result of which will be assigned to the variable. The following examples are all valid uses of the assignment operator:

```
var x: Int // Declare a mutable Int variable
val y = 10 // Declare and initialize an immutable Int variable

x = 10 // Assign a value to x
x = x + y // Assign the result of x + y to x
x = y // Assign the value of y to x
```

## 13.3 Kotlin arithmetic operators

Kotlin provides a range of operators for creating mathematical expressions. These operators primarily fall into the category of *binary operators* in that they take two operands. The exception is the *unary negative operator* (-) which serves to indicate that a value is negative rather than positive. This contrasts with the *subtraction operator* (-) which takes two operands (i.e. one value to be subtracted from another). For example:

```
var x = -10 // Unary - operator used to assign -10 to variable x
x = x - 5 // Subtraction operator. Subtracts 5 from x
```

The following table lists the primary Kotlin arithmetic operators:

Operator	Description
-(unary)	Negates the value of a variable or expression
*	Multiplication

/	Division
+	Addition
-	Subtraction
%	Remainder/Modulo

Table 13-1

Note that multiple operators may be used in a single expression.

For example:

```
x = y * 10 + z - 5 / 4
```

### 13.4 Augmented assignment operators

In an earlier section, we looked at the basic assignment operator (=). Kotlin provides several operators designed to combine an assignment with a mathematical or logical operation. These are primarily of use when performing an evaluation where the result is to be stored in one of the operands. For example, one might write an expression as follows:

```
x = x + y
```

The above expression adds the value contained in variable *x* to the value contained in variable *y* and stores the result in variable *x*. This can be simplified using the addition augmented assignment operator:

```
x += y
```

The above expression performs the same task as  $x = x + y$  but saves the programmer some typing.

Numerous augmented assignment operators are available in Kotlin. The most frequently used of which are outlined in the following table:

Operator	Description
<code>x += y</code>	Add <i>x</i> to <i>y</i> and place result in <i>x</i>
<code>x -= y</code>	Subtract <i>y</i> from <i>x</i> and place result in <i>x</i>
<code>x *= y</code>	Multiply <i>x</i> by <i>y</i> and place result in <i>x</i>
<code>x /= y</code>	Divide <i>x</i> by <i>y</i> and place result in <i>x</i>
<code>x %= y</code>	Perform Modulo on <i>x</i> and <i>y</i> and place result in <i>x</i>

Table 13-2

### 13.5 Increment and decrement operators

Another useful shortcut can be achieved using the Kotlin increment and decrement operators (also referred to as unary operators because they operate on a single operand). Consider the code fragment below:

```
x = x + 1 // Increase value of variable x by 1
x = x - 1 // Decrease value of variable x by 1
```

These expressions increment and decrement the value of *x* by 1. Instead of using this approach, however, it is quicker to use the ++ and -- operators. The following examples perform the same tasks as the examples above:

```
x++ // Increment x by 1
x-- // Decrement x by 1
```

These operators can be placed either before or after the variable name. If the operator is placed before the variable name, the increment or decrement operation is performed before any other operations are performed on the variable. For example, in the following code, *x* is incremented before it is assigned to *y*, leaving *y* with a



value of 10:

```
var x = 9
val y = ++x
```

In the next example, however, the value of  $x$  (9) is assigned to variable  $y$  before the decrement is performed. After the expression is evaluated the value of  $y$  will be 9 and the value of  $x$  will be 8.

```
var x = 9
val y = x--
```

## 13.6 Equality operators

Kotlin also includes a set of logical operators useful for performing comparisons. These operators all return a Boolean result depending on the result of the comparison. These operators are *binary operators* in that they work with two operands.

Equality operators are most frequently used in constructing program control flow logic. For example an *if* statement may be constructed based on whether one value matches another:

```
if (x == y) {
    // Perform task
}
```

The result of a comparison may also be stored in a Boolean variable. For example, the following code will result in a *true* value being stored in the variable `result`:

```
var result: Boolean
val x = 10
val y = 20
```

```
result = x < y
```

Clearly 10 is less than 20, resulting in a *true* evaluation of the  $x < y$  expression. The following table lists the full set of Kotlin comparison operators:

Operator	Description
$x == y$	Returns true if $x$ is equal to $y$
$x > y$	Returns true if $x$ is greater than $y$
$x >= y$	Returns true if $x$ is greater than or equal to $y$
$x < y$	Returns true if $x$ is less than $y$
$x <= y$	Returns true if $x$ is less than or equal to $y$
$x != y$	Returns true if $x$ is not equal to $y$

Table 13-3

## 13.7 Boolean logical operators

Kotlin also provides a set of so-called logical operators designed to return Boolean *true* or *false* values. These operators both return Boolean results and take Boolean values as operands. The key operators are NOT (!), AND (&&), and OR (||).

The NOT (!) operator simply inverts the current value of a Boolean variable or the result of an expression. For example, if a variable named *flag* is currently true, prefixing the variable with a '!' character will invert the value to false:

```
val flag = true // variable is true
```

## Kotlin Operators and Expressions

```
val secondFlag = !flag // secondFlag set to false
```

The OR (||) operator returns true if one of its two operands evaluates to true, otherwise, it returns false. For example, the following code evaluates to true because at least one of the expressions on either side of the OR operator is true:

```
if ((10 < 20) || (20 < 10)) {  
    print("Expression is true")  
}
```

The AND (&&) operator returns true only if both operands evaluate to be true. The following example will return false because only one of the two operand expressions evaluates to true:

```
if ((10 < 20) && (20 < 10)) {  
    print("Expression is true")  
}
```

### 13.8 Range operator

Kotlin includes a useful operator that allows a range of values to be declared. As will be seen in later chapters, this operator is invaluable when working with looping in program logic.

The syntax for the range operator is as follows:

```
x..y
```

This operator represents the range of numbers starting at x and ending at y where both x and y are included within the range (referred to as a closed range). The range operator 5..8, for example, specifies the numbers 5, 6, 7, and 8.

### 13.9 Bitwise operators

As previously discussed, computer processors work in binary. These are essentially streams of ones and zeros, each one referred to as a bit. Bits are formed into groups of 8 to form bytes. As such, it is not surprising that we, as programmers, will occasionally end up working at this level in our code. To facilitate this requirement, Kotlin provides a range of *bit operators*.

Those familiar with bitwise operators in other languages such as C, C++, C#, Objective-C, and Java will find nothing new in this area of the Kotlin language syntax. For those unfamiliar with binary numbers, now may be a good time to seek out reference materials on the subject to understand how ones and zeros are formed into bytes to form numbers. Other authors have done a much better job of describing the subject than we can do within the scope of this book.

For this exercise, we will be working with the binary representation of two numbers. First, the decimal number 171 is represented in binary as:

```
10101011
```

Second, the number 3 is represented by the following binary sequence:

```
00000011
```

Now that we have two binary numbers with which to work, we can begin to look at the Kotlin bitwise operators:

#### 13.9.1 Bitwise inversion

The Bitwise inversion (also referred to as NOT) is performed using the *inv()* operation and has the effect of inverting all of the bits in a number. In other words, all the zeros become ones and all the ones become zeros. Taking our example 3 number, a Bitwise NOT operation has the following result:

```
00000011 NOT
```

```
=====
```

```
11111100
```

The following Kotlin code, therefore, results in a value of -4:

```
val y = 3
val z = y.inv()
```

```
print("Result is $z")
```

### 13.9.2 Bitwise AND

The Bitwise AND is performed using the *and()* operation. It makes a bit-by-bit comparison of two numbers. Any corresponding position in the binary sequence of each number where both bits are 1 results in a 1 appearing in the same position of the resulting number. If either bit position contains a 0 then a zero appears in the result. Taking our two example numbers, this would appear as follows:

```
10101011 AND
00000011
=====
00000011
```

As we can see, the only locations where both numbers have 1s are the last two positions. If we perform this in Kotlin code, therefore, we should find that the result is 3 (00000011):

```
val x = 171
val y = 3
val z = x.and(y)
```

```
print("Result is $z")
```

### 13.9.3 Bitwise OR

The bitwise OR also performs a bit-by-bit comparison of two binary sequences. Unlike the AND operation, the OR places a 1 in the result if there is a 1 in the first or second operand. Using our example numbers, the result will be as follows:

```
10101011 OR
00000011
=====
10101011
```

If we perform this operation in Kotlin using the *or()* operation the result will be 171:

```
val x = 171
val y = 3
val z = x.or(y)
```

```
print("Result is $z")
```

### 13.9.4 Bitwise XOR

The bitwise XOR (commonly referred to as *exclusive OR* and performed using the *xor()* operation) performs a similar task to the OR operation except that a 1 is placed in the result if one or other corresponding bit positions in the two numbers is 1. If both positions are a 1 or a 0 then the corresponding bit in the result is set to a 0. For example:

```
10101011 XOR
```

## Kotlin Operators and Expressions

```
00000011
=====
10101000
```

The result, in this case, is 10101000 which converts to 168 in decimal. To verify this we can, once again, try some Kotlin code:

```
val x = 171
val y = 3
val z = x.xor(y)

print("Result is $z")
```

When executed, we get the following output from print:

```
Result is 168
```

### 13.9.5 Bitwise left shift

The bitwise left shift moves each bit in a binary number a specified number of positions to the left. Shifting an integer one position to the left has the effect of doubling the value.

As the bits are shifted to the left, zeros are placed in the vacated rightmost (low order) positions. Note also that once the leftmost (high order) bits are shifted beyond the size of the variable containing the value, those high order bits are discarded:

```
10101011 Left Shift one bit
=====
101010110
```

In Kotlin the bitwise left shift operator is performed using the *shl()* operation, passing through the number of bit positions to be shifted. For example, to shift left by 1 bit:

```
val x = 171
val z = x.shl(1)

print("Result is $z")
```

When compiled and executed, the above code will display a message stating that the result is 342 which, when converted to binary, equates to 101010110.

### 13.9.6 Bitwise right shift

A bitwise right shift is, as you might expect, the same as a left except that the shift takes place in the opposite direction. Shifting an integer one position to the right has the effect of halving the value.

Note that since we are shifting to the right there is no opportunity to retain the lowermost bits regardless of the data type used to contain the result. As a result, the low order bits are discarded. Whether or not the vacated high order bit positions are replaced with zeros or ones depends on whether the *sign bit* used to indicate positive and negative numbers is set or not.

```
10101011 Right Shift one bit
=====
01010101
```

The bitwise right shift is performed using the *shr()* operation passing through the shift count:

```
val x = 171
```

```
val z = x.shr(1)

print("Result is $z")
```

When executed, the above code will report the result of the shift as being 85, which equates to binary 01010101.

## 13.10 Summary

Operators and expressions provide the underlying mechanism by which variables and constants are manipulated and evaluated within Kotlin code. This can take the simplest of forms whereby two numbers are added using the addition operator in an expression and the result stored in a variable using the assignment operator. Operators fall into a range of categories, details of which have been covered in this chapter.



# 25. Annotated Strings and Brush Styles

The previous chapter explored how we use modifiers to change the appearance and behavior of composables. Many examples used to demonstrate modifiers involved the Text composable, performing tasks such as changing the font type, size, and weight. This chapter will introduce another powerful text-related feature of Jetpack Compose, known as annotated strings. We will also look at brush styles and how they can be used to add more effects to the text in a user interface.

## 25.1 What are annotated strings?

The previous chapter's modifier examples changed the appearance of the entire string displayed by a Text composable. For instance, we could not display part one part of the text in bold while another section was in italics. It is for this reason that Jetpack Compose includes the annotated strings.

Annotated strings allow a text to be divided into multiple sections, each with its own style.

## 25.2 Using annotated strings

An AnnotatedString instance is created by calling the *buildAnnotatedString* function and passing it the text and styles to be displayed. These string sections are combined via calls to the *append* function to create the complete text to be displayed.

Two style types are supported, the first of which, SpanStyle, is used to apply styles to a span of individual characters within a string. The syntax for building an annotated string using SpanStyle is as follows:

```
buildAnnotatedString {
    withStyle(style = SpanStyle( /* style settings */) ) {
        append( /* text string */ )
    }

    withStyle(style = SpanStyle( /* style settings */) ) {
        append( /* more text */ )
    }

    .
    .
}
```

A SpanStyle instance can be initialized with any combination of the following style options:

- color
- fontSize
- fontWeight
- fontStyle

## Annotated Strings and Brush Styles

- fontSynthesis
- fontFamily
- fontFeatureSettings
- letterSpacing
- baselineShift,
- textGeometricTransform
- localeList
- background
- textDecoration
- shadow

ParagraphStyle, on the other hand, applies a style to paragraphs and can be used to modify the following properties:

- textAlign
- textDirection
- lineHeight
- textIndent

The following is the basic syntax for using paragraph styles in annotated strings:

```
buildAnnotatedString {
    withStyle(style = ParagraphStyle( /* style settings */) ) {
        append( /* text string */ )
    }

    withStyle(style = ParagraphStyle( /* style settings */) )
        append( /* more text */ )
}
.
```

## 25.3 Brush Text Styling

Additional effects may be added to any text by using the Compose Brush styling. Brush effects can be applied directly to standard text strings or selectively to segments of an annotated string. For example, the following syntax applies a radial color gradient to a Text composable (color gradients will be covered in the chapter entitled “*Canvas Graphics Drawing in Compose*”):

```
val myColors = listOf( /* color list */)
```

```
Text(
    text = "text here",
```



```

        style = TextStyle(
            brush = Brush.radialGradient(
                colors = myColors
            )
        )
    )
}

```

## 25.4 Creating the example project

Launch Android Studio and select the New Project option from the welcome screen. Choose the Empty Activity template within the New Project dialog before clicking the Next button.

Enter *StringsDemo* into the Name field and specify *com.example.stringsdemo* as the package name. Before clicking the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo). Once the project has been created, the StringsDemo project should be listed in the Project tool window along the left-hand edge of the Android Studio main window.

Within the *MainActivity.kt* file, delete the Greeting function and add a new empty composable named *MainScreen*:

```

@Composable
fun MainScreen() {

}

```

Next, edit the *onCreate()* method and *GreetingPreview* function to call *MainScreen* instead of *Greeting*.

## 25.5 An example SpanStyle annotated string

The first example we will create uses *SpanStyle* to build an annotated string consisting of multiple color and font styles.

Begin by editing the *MainActivity.kt* file and modifying the *MainScreen* function to read as follows:

```

.
.
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.buildAnnotatedString
import androidx.compose.ui.text.withStyle
import androidx.compose.ui.text.SpanStyle
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.font.FontStyle
import androidx.compose.foundation.layout.Column
import androidx.compose.ui.unit.sp
.
.
@Composable
fun MainScreen() {
    Column {
        SpanString()
    }
}

```

## Annotated Strings and Brush Styles

Next, add the `SpanString` declaration to the `MainActivity.kt` file as follows:

```
@Composable
fun SpanString() {
    Text(
        buildAnnotatedString {
            withStyle(
                style = SpanStyle(fontWeight = FontWeight.Bold,
                    fontSize = 30.sp)) {
                append("T")
            }

            withStyle(style = SpanStyle(color = Color.Gray)) {
                append("his")
            }
            append(" is ")
            withStyle(
                style = SpanStyle(
                    fontWeight = FontWeight.Bold,
                    fontStyle = FontStyle.Italic,
                    color = Color.Blue
                )
            ) {
                append("great!")
            }
        }
    )
}
```

The example code creates an annotated string in three parts using several span styles for each section. After making these changes, refer to the Preview panel, where the text should appear as shown in Figure 25-1:

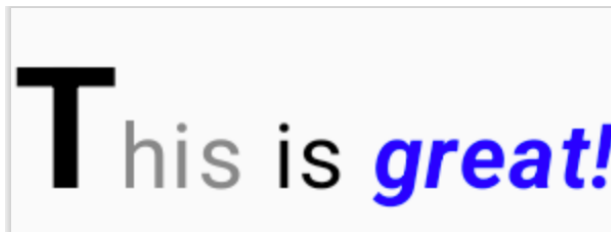


Figure 25-1

### 25.6 An example `ParagraphStyle` annotated string

Now that we have seen how to create a span-style annotated string, the next step is to build a paragraph-style string. Remaining in the `MainActivity.kt` file, make the following changes to add a new function named `ParaString` and to call it from the `MainScreen` function:

```
.
.
import androidx.compose.ui.text.ParagraphStyle
```

```

import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextIndent
.
.
@Composable
fun MainScreen() {
    Column {
        SpanString()
        ParaString()
    }
}

@Composable
fun ParaString() {

    Text(
        buildAnnotatedString {
            append(
                "\nThis is some text that doesn't have any style applied to
it.\n")
            })
}

```

The above code gives us an unmodified paragraph against which we can compare the additional paragraphs we will add. Next, modify the function to add an indented paragraph with an increased line height:

```

@Composable
fun ParaString() {

    Text(
        buildAnnotatedString {

            append("\nThis is some text that doesn't have any style applied to
it.\n")

            withStyle(style = ParagraphStyle(
                lineHeight = 30.sp,
                textIndent = TextIndent(
                    firstLine = 60.sp,
                    restLine = 25.sp))
            ) {
                append("This is some text that is indented more on the first lines
than the rest of the lines. It also has an increased line height.\n")
            }
        })
}

```

When the preview is rendered, it should resemble Figure 25-2 (note that we specified different indents for the

first and remaining lines):

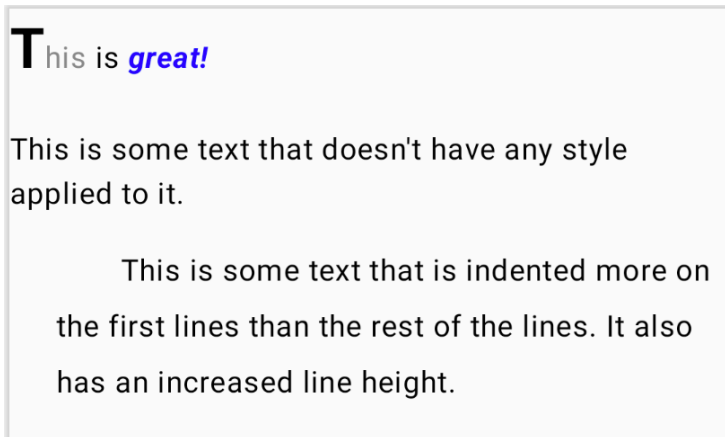


Figure 25-2

Next, add a third paragraph that uses right alignment as follows:

```
@Composable
fun ParaString() {
    .
    .
        append("This is some text that is indented more on the first lines
than the rest of the lines. It also has an increased line height.\n")
    }

    withStyle(style = ParagraphStyle(textAlign = TextAlign.End)) {
        append("This is some text that is right aligned.")
    }
}
})
}
```

This change should result in the following preview:

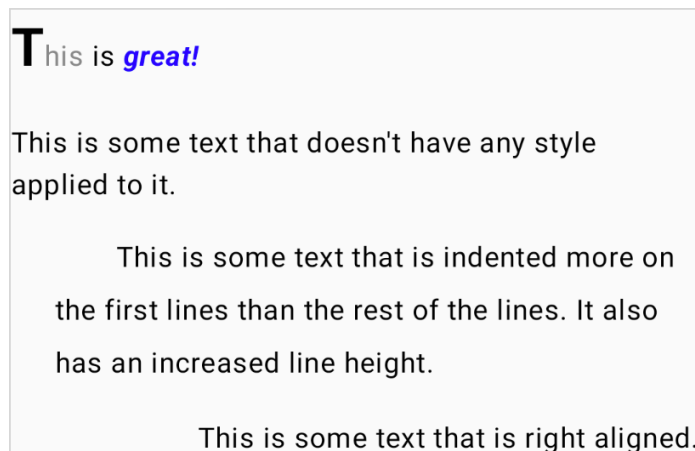


Figure 25-3

## 25.7 A Brush style example

The final example in this tutorial involves using the Brush style to change the text's appearance. First, add another function to the *MainActivity.kt* file and call it from within the *MainScreen* function:

```

.
.
import androidx.compose.ui.graphics.Brush
import androidx.compose.ui.text.ExperimentalTextApi
.
.
@Composable
fun MainScreen() {
    Column {
        SpanString()
        ParaString()
        BrushStyle()
    }
}

@OptIn(ExperimentalTextApi::class)
@Composable
fun BrushStyle() {

}

```

We will begin by declaring a list of colors and use a span style to display large, bold text as follows:

```

@OptIn(ExperimentalTextApi::class)
@Composable
fun BrushStyle() {

    val colorList: List<Color> = listOf(Color.Red, Color.Blue,
        Color.Magenta, Color.Yellow, Color.Green, Color.Red)

    Text(
        text = buildAnnotatedString {

            withStyle(
                style = SpanStyle(
                    fontWeight = FontWeight.Bold,
                    fontSize = 70.sp
                )
            ) {
                append("COMPOSE!")
            }
        }
    )
}

```

## Annotated Strings and Brush Styles

```
}
```

All that remains is to apply a `linearGradient` brush to the style, using the previously declared color list:

```
@OptIn(ExperimentalTextApi::class)
@Composable
fun BrushStyle() {

    Text(
        text = buildAnnotatedString {

            withStyle(
                style = SpanStyle(
                    fontWeight = FontWeight.Bold,
                    fontSize = 70.sp,
                    brush = Brush.linearGradient(colors = colorList)
                )
            ) {
                append("COMPOSE!")
            }
        }
    )
}
```

After completing the above changes, check that the new text appears in the preview panel as illustrated in Figure 39-3:



Figure 25-4

## 25.8 Summary

While modifiers provide a quick and convenient way to make changes to the appearance of text in a user interface, they do not support multiple styles within a single string. On the other hand, annotated strings provide greater flexibility in changing the appearance of text. Annotated strings are built using the `buildAnnotatedString` function and can be configured using either span or paragraph styles. Another option for altering how text appears is using the Brush style to change the text foreground creatively, such as using color gradients.

## 27. Box Layouts in Compose

Now that we have an understanding of the Compose Row and Column composables, we will move on to look at the third layout type provided by Compose in the form of the Box component. This chapter will introduce the Box layout and explore some key parameters and modifiers available when designing user interface layouts.

### 27.1 An introduction to the Box composable

Unlike the Row and Column, which organize children in a horizontal row or vertical column, the Box layout stacks its children on top of each other. The stacking order is defined by the order in which the children are called within the Box declaration, with the first child positioned at the bottom of the stack. As with the Row and Column layouts, Box is provided with several parameters and modifiers we can use to customize the layout.

### 27.2 Creating the BoxLayout project

Begin by launching Android Studio and, if necessary, closing any currently open projects using the *File -> Close Project* menu option so that the Welcome screen appears.

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

Enter *BoxLayout* into the Name field and specify *com.example.boxlayout* as the package name. Before clicking the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo). On completion of the project creation process, the BoxLayout project should be listed in the Project tool window located along the left-hand edge of the Android Studio main window.

Within the *MainActivity.kt* file, delete the Greeting function and add a new empty composable named *MainScreen*:

```
@Composable
fun MainScreen() {

}
```

Next, change the *OnCreate()* method and *GreetingPreview* function to call *MainScreen* instead of *Greeting*.

### 27.3 Adding the TextCell composable

In this chapter, we will again use our *TextCell* composable, though to best demonstrate the features of the Box layout, we will modify the declaration slightly so that it can be passed an optional font size when called. Remaining within the *MainActivity.kt* file, add this composable function so that it reads as follows:

```
.
.
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.padding
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
```

## Box Layouts in Compose

```
import androidx.compose.ui.unit.sp
.
.
@Composable
fun TextCell(text: String, modifier: Modifier = Modifier, fontSize: Int = 150 ) {

    val cellModifier = Modifier
        .padding(4.dp)
        .border(width = 5.dp, color = Color.Black)

    Text(
        text = text, cellModifier.then(modifier),
        fontSize = fontSize.sp,
        fontWeight = FontWeight.Bold,
        textAlign = TextAlign.Center
    )
}
```

## 27.4 Adding a Box layout

Next, modify the `MainScreen` function to include a `Box` layout with three `TextCell` children:

```
.
.
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.size
.
.
@Composable
fun MainScreen() {
    Box {
        val height = 200.dp
        val width = 200.dp

        TextCell("1", Modifier.size(width = width, height = height))
        TextCell("2", Modifier.size(width = width, height = height))
        TextCell("3", Modifier.size(width = width, height = height))
    }
}
```

After modifying the function, update the Preview panel to reflect these latest changes. Once the layout appears it should resemble Figure 27-1:





Figure 27-1

The transparent nature of the `Text` composable allows us to see that the three children have, indeed, been stacked directly on top of each other. While this transparency is useful to show that the children have been stacked, this isn't the behavior we are looking for in this example. To give the `TextCell` an opaque background, we need to call the `Text` composable from within a `Surface` component. To achieve this, edit the `TextCell` function so that it now reads as follows:

```
@Composable
fun TextCell(text: String, modifier: Modifier = Modifier,  fontSize: Int = 150 ) {
    .
    .
    Surface {
        Text (
            text = text, cellModifier.then(modifier),
            fontSize = fontSize.sp,
            fontWeight = FontWeight.Bold,
            textAlign = TextAlign.Center
        )
    }
}
```

When the preview is updated, only the last composable to be called by the `Box` will be visible because it is the uppermost child of the stack.

## 27.5 Box alignment

The `Box` composable includes support for an alignment parameter to customize the positioning of the group of children within the content area of the box. The parameter is named *contentAlignment* and may be set to any one of the following values:

- **Alignment.TopStart**
- **Alignment.TopCenter**
- **Alignment.TopEnd**
- **Alignment.CenterStart**
- **Alignment.Center**
- **Alignment.CenterEnd**
- **Alignment.BottomCenter**
- **Alignment.BottomEnd**
- **Alignment.BottomStart**

The diagram in Figure 27-2 illustrates the positioning of the Box content for each of the above settings:

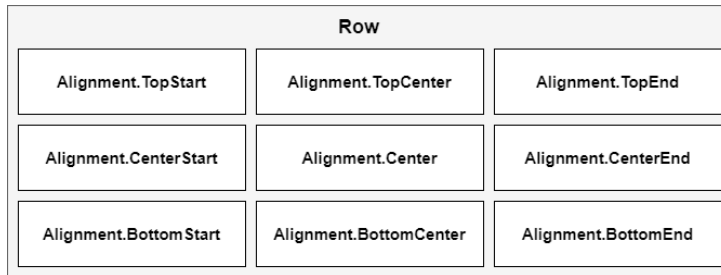


Figure 27-2

To try out some of these alignments options, edit the Box declaration in the MainScreen function both to increase its size and to add a *contentAlignment* parameter:

```

.
.
import androidx.compose.ui.Alignment
.
.
@Composable
fun MainScreen() {
.
.
    Box(contentAlignment = Alignment.CenterEnd,
        modifier = Modifier.size(400.dp, 400.dp)) {
        val height = 200.dp
        val width = 200.dp
        TextCell("1", Modifier.size(width = width, height = height))
        TextCell("2", Modifier.size(width = width, height = height))
        TextCell("3", Modifier.size(width = width, height = height))
    }
}

```

Refresh the preview and verify that the Box content now appears at the CenterEnd position within the Box content area:

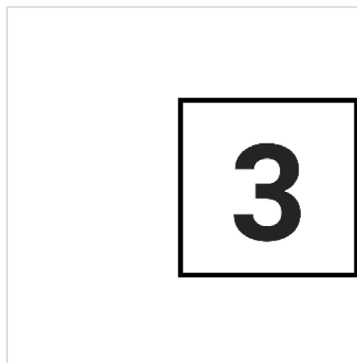


Figure 27-3

## 27.6 BoxScope modifiers

In the chapter entitled “*Composing Layouts with Row and Column*”, we introduced `ColumnScope` and `RowScope` and explored how these provide additional modifiers that can be applied individually to child components. In the case of the `Box` layout, the following `BoxScope` modifiers are available to be applied to child composables:

- **`align()`** - Aligns the child within the `Box` content area using the specified `Alignment` value.
- **`matchParentSize()`** - Sizes the child on which the modifier is applied to match the size of the parent `Box`.

The set of `Alignment` values accepted by the `align` modifier is the same as those listed above for `Box` alignment. The following changes to the `MainScreen` function demonstrate the `align()` modifier in action:

```
@Composable
fun MainScreen() {
    .
    .
    Box(modifier = Modifier.size(height = 90.dp, width = 290.dp)) {
        Text("TopStart", Modifier.align(Alignment.TopStart))
        Text("TopCenter", Modifier.align(Alignment.TopCenter))
        Text("TopEnd", Modifier.align(Alignment.TopEnd))

        Text("CenterStart", Modifier.align(Alignment.CenterStart))
        Text("Center", Modifier.align(Alignment.Center))
        Text(text = "CenterEnd", Modifier.align(Alignment.CenterEnd))

        Text("BottomStart", Modifier.align(Alignment.BottomStart))
        Text("BottomCenter", Modifier.align(Alignment.BottomCenter))
        Text("BottomEnd", Modifier.align(Alignment.BottomEnd))
    }
}
```

When previewed, the above `Box` layout will appear as shown in Figure 27-4 below:

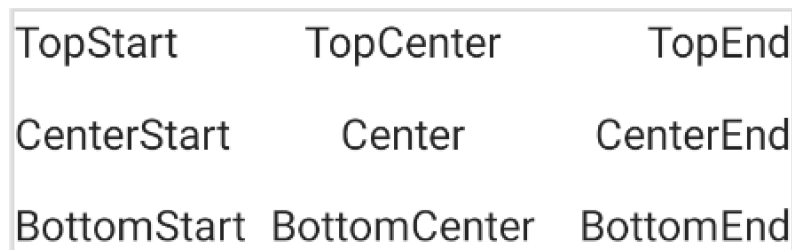


Figure 27-4

## 27.7 Using the `clip()` modifier

The compose `clip()` modifier allows composables to be rendered to conform to specific shapes. Though not specific to `Box`, the `Box` component provides perhaps the best example of clipping shapes. To define the shape of a composable, the `clip()` modifier is called and passed a `Shape` value which can be `RectangleShape`, `CircleShape`, `RoundedCornerShape`, or `CutCornerShape`.

The following code, for example, draws a `Box` clipped to appear as a circle:

## Box Layouts in Compose

```
.  
.   
import androidx.compose.foundation.background  
import androidx.compose.ui.draw.clip  
import androidx.compose.foundation.shape.CircleShape  
.   
.   
Box(Modifier.size(200.dp).clip(CircleShape).background(Color.Blue))  
.   
.
```

When rendered, the Box will appear as shown in Figure 27-5:

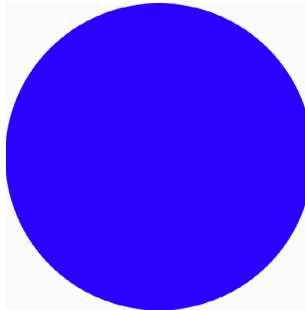


Figure 27-5

To draw a composable with rounded corners call `RoundedCornerShape`, passing through the radius for each corner. If a single radius value is provided, it will be applied to all four corners:

```
.   
.   
import androidx.compose.foundation.shape.RoundedCornerShape  
.   
.   

```

The above composable will appear as shown below:

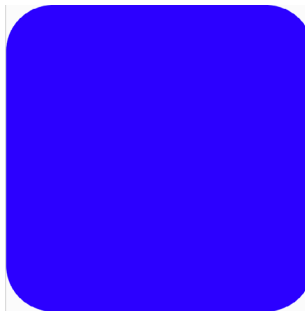


Figure 27-6

As an alternative to rounded corners, composables may also be rendered with cut corners. In this case, `CutCornerShape` is passed the cut length for the corners. Once again, we may specify different values for each corner, or all corners cut equally with a single length parameter:

```
.  
.br/>import androidx.compose.foundation.shape.CutCornerShape  
.br/>Box(Modifier.size(200.dp).clip(CutCornerShape(30.dp)).background(Color.Blue))  
.br/.
```

The following figure shows the Box rendered by the above code:

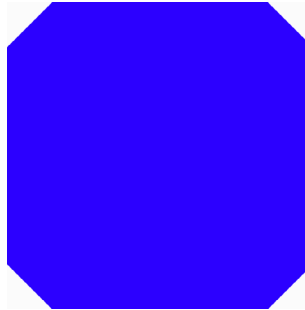


Figure 27-7

## 27.8 Summary

The Compose Box layout positions all of its children on top of each other in a stack arrangement, with the first child positioned at the bottom of the stack. By default, this stack will be placed in the top left-hand corner of the content area, though this can be changed using the *contentAlignment* parameter when calling the Box composable.

Direct children of a Box layout have access to additional modifiers via RowScope. These modifiers allow individual children to be positioned independently within the Box content using a collection of nine pre-defined position settings.



## 36. An Overview of Lists and Grids in Compose

It is a common requirement when designing user interface layouts to present information in either scrollable list or grid configurations. For basic list requirements, the Row and Column components can be re-purposed to provide vertical and horizontal lists of child composables. Extremely large lists, however, are likely to cause degraded performance if rendered using the standard Row and Column composables. For lists containing large numbers of items, Compose provides the LazyColumn and LazyRow composables. Similarly, grid-based layouts can be presented using the LazyVerticalGrid composable.

This chapter will introduce the basics of list and grid creation and management in Compose in preparation for the tutorials in subsequent chapters.

### 36.1 Standard vs. lazy lists

Part of the popularity of lists is that they provide an effective way to present large amounts of items in a scrollable format. Each item in a list is represented by a composable which may, itself, contain descendant composables. When a list is created using the Row or Column component, all of the items it contains are also created at initialization, regardless of how many are visible at any given time. While this does not necessarily pose a problem for smaller lists, it can be an issue for lists containing many items.

Consider, for example, a list that is required to display 1000 photo images. It can be assumed with a reasonable degree of certainty that only a small percentage of items will be visible to the user at any one time. If the application was permitted to create each of the 1000 items in advance, however, the device would very quickly run into memory and performance limitations.

When working with longer lists, the recommended course of action is to use LazyColumn, LazyRow, and LazyVerticalGrid. These components only create those items that are visible to the user. As the user scrolls, items that move out of the viewable area are destroyed to free up resources while those entering view are created just in time to be displayed. This allows lists of potentially infinite length to be displayed with no performance degradation.

Since there are differences in approach and features when working with Row and Column compared to the lazy equivalents, this chapter will provide an overview of both types.

### 36.2 Working with Column and Row lists

Although lacking some of the features and performance advantages of the LazyColumn and LazyRow, the Row and Column composables provide a good option for displaying shorter, basic lists of items. Lists are declared in much the same way as regular rows and columns with the exception that each list item is usually generated programmatically. The following declaration, for example, uses the Column component to create a vertical list containing 100 instances of a composable named MyListItem:

```
Column {
    repeat(100) {
        MyListItem()
    }
}
```

## An Overview of Lists and Grids in Compose

```
}
```

Similarly, the following example creates a horizontal list containing the same items:

```
Row {  
    repeat(100) {  
        MyListItem()  
    }  
}
```

The `MyListItem` composable can be anything from a single `Text` composable to a complex layout containing multiple composables.

### 36.3 Creating lazy lists

Lazy lists are created using the `LazyColumn` and `LazyRow` composables. These layouts place children within a `LazyListScope` block which provides additional features for managing and customizing the list items. For example, individual items may be added to a lazy list via calls to the `item()` function of the `LazyListScope`:

```
LazyColumn {  
    item {  
        MyListItem()  
    }  
}
```

Alternatively, multiple items may be added in a single statement by calling the `items()` function:

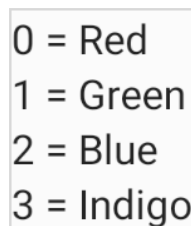
```
LazyColumn {  
    items(1000) { index ->  
        Text("This is item $index");  
    }  
}
```

`LazyListScope` also provides the `itemsIndexed()` function which associates the item content with an index value, for example:

```
val colorNamesList = listOf("Red", "Green", "Blue", "Indigo")
```

```
LazyColumn {  
    itemsIndexed(colorNamesList) { index, item ->  
        Text("$index = $item")  
    }  
}
```

When rendered, the above lazy column will appear as shown in Figure 36-1 below:



```
0 = Red  
1 = Green  
2 = Blue  
3 = Indigo
```

Figure 36-1



Lazy lists also support the addition of headers to groups of items in a list using the `stickyHeader()` function. This topic will be covered in more detail later in the chapter.

## 36.4 Enabling scrolling with ScrollState

While the above Column and Row list examples will display a list of items, only those that fit into the viewable screen area will be accessible to the user. This is because lists are not scrollable by default. To make Row and Column-based lists scrollable, some additional steps are needed. `LazyList` and `LazyRow`, on the other hand, support scrolling by default.

The first step in enabling list scrolling when working with Row and Column-based lists is to create a `ScrollState` instance. This is a special state object designed to allow Row and Column parents to remember the current scroll position through recompositions. A `ScrollState` instance is generated via a call to the `rememberScrollState()` function, for example:

```
val scrollState = rememberScrollState()
```

Once created, the scroll state is passed as a parameter to the Column or Row composable using the `verticalScroll()` and `horizontalScroll()` modifiers. In the following example, vertical scrolling is being enabled in a Column list:

```
Column(Modifier.verticalScroll(scrollState)) {
    repeat(100) {
        MyListItem()
    }
}
```

Similarly, the following code enables horizontal scrolling on a `LazyRow` list:

```
Row(Modifier.horizontalScroll(scrollState)) {
    repeat(1000) {
        MyListItem()
    }
}
```

## 36.5 Programmatic scrolling

We generally think of scrolling as being something a user performs through dragging or swiping gestures on the device screen. It is also important to know how to change the current scroll position from within code. An app screen might, for example, contain buttons which can be tapped to scroll to the start and end of a list. The steps to implement this behavior differ between Row and Columns lists and the lazy list equivalents.

When working with Row and Column lists, programmatic scrolling can be performed by calling the following functions on the `ScrollState` instance:

- **animateScrollTo(value: Int)** - Scrolls smoothly to the specified pixel position in the list using animation.
- **scrollTo(value: Int)** - Scrolls instantly to the specified pixel position.

Note that the value parameters in the above function represent the list position in pixels instead of referencing a specific item number. It is safe to assume that the start of the list is represented by pixel position 0, but the pixel position representing the end of the list may be less obvious. Fortunately, the maximum scroll position can be identified by accessing the `maxValue` property of the scroll state instance:

```
val maxScrollPosition = scrollState.maxValue
```

To programmatically scroll `LazyColumn` and `LazyRow` lists, functions need to be called on a `LazyListState` instance which can be obtained via a call to the `rememberLazyListState()` function as follows:

## An Overview of Lists and Grids in Compose

```
val listState = rememberLazyListState()
```

Once the list state has been obtained, it must be applied to the `LazyRow` or `LazyColumn` declaration as follows:

```
.  
.br/>LazyColumn(  
    state = listState,  
{  
    .  
    .
```

Scrolling can then be performed via calls to the following functions on the list state instance:

- **`animateScrollToItem(index: Int)`** - Scrolls smoothly to the specified list item (where 0 is the first item).
- **`scrollToItem(index: Int)`** - Scrolls instantly to the specified list item (where 0 is the first item).

In this case, the scrolling position is referenced by the index of the item instead of pixel position.

One complication is that all four of the above scroll functions are *coroutine* functions. As outlined in the chapter titled “*Coroutines and LaunchedEffects in Jetpack Compose*”, coroutines are a feature of Kotlin that allows blocks of code to execute asynchronously without blocking the thread from which they are launched (in this case the *main thread* which is responsible for making sure the app remains responsive to the user). Coroutines can be implemented without having to worry about building complex 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. One of the key requirements of coroutine functions is that they must be launched from within a *coroutine scope*.

As with `ScrollState` and `LazyListState`, we need access to a `CoroutineScope` instance that will be remembered through recompositions. This requires a call to the `rememberCoroutineScope()` function as follows:

```
val coroutineScope = rememberCoroutineScope()
```

Once we have a coroutine scope, we can use it to launch the scroll functions. The following code, for example, declares a `Button` component configured to launch the `animateScrollTo()` function within the coroutine scope. In this case, the button will cause the list to scroll to the end position when clicked:

```
.  
.br/>Button(onClick = {  
    coroutineScope.launch {  
        scrollState.animateScrollTo(scrollState.maxValue)  
    }  
})
```

## 36.6 Sticky headers

Sticky headers is a feature only available within lazy lists that allows list items to be grouped under a corresponding header. Sticky headers are created using the `LazyListScope.stickyHeader()` function.

The headers are referred to as being sticky because they remain visible on the screen while the current group is scrolling. Once a group scrolls from view, the header for the next group takes its place. Figure 36-2, for example,

shows a list with sticky headers. Note that although the Apple group is scrolled partially out of view, the header remains in position at the top of the screen:



Figure 36-2

When working with sticky headers, the list content must be stored in an Array or List which has been mapped using the Kotlin `groupBy()` function. The `groupBy()` function accepts a lambda which is used to define the *selector* which defines how data is to be grouped. This selector then serves as the key to access the elements of each group. Consider, for example, the following list which contains mobile phone models:

```
val phones = listOf("Apple iPhone 12", "Google Pixel 4", "Google Pixel 6",
    "Samsung Galaxy 6s", "Apple iPhone 7", "OnePlus 7", "OnePlus 9 Pro",
    "Apple iPhone 13", "Samsung Galaxy Z Flip", "Google Pixel 4a",
    "Apple iPhone 8")
```

Now suppose that we want to group the phone models by manufacturer. To do this we would use the first word of each string (in other words, the text before the first space character) as the selector when calling `groupBy()` to map the list:

```
val groupedPhones = phones.groupBy { it.substringBefore(' ') }
```

Once the phones have been grouped by manufacturer, we can use the `forEach` statement to create a sticky header for each manufacture name, and display the phones in the corresponding group as list items:

```
groupedPhones.forEach { (manufacturer, models) ->
    stickyHeader {
        Text(
            text = manufacturer,
            color = Color.White,
            modifier = Modifier
                .background(Color.Gray)
```

## An Overview of Lists and Grids in Compose

```
                .padding(5.dp)
                .fillMaxWidth()
            )
        }

        items(models) { model ->
            MyListItem(model)
        }
    }
}
```

In the above *forEach* lambda, *manufacturer* represents the selector key (for example “Apple”) and *models* an array containing the items in the corresponding manufacturer group (“Apple iPhone 12”, “Apple iPhone 7”, and so on for the Apple selector):

```
groupedPhones.forEach { (manufacturer, models) ->
```

The selector key is then used as the text for the sticky header, and the *models* list is passed to the *items()* function to display all the group elements, in this case using a custom composable named *MyListItem* for each item:

```
items(models) { model ->
    MyListItem(model)
}
}
```

When rendered, the above code will display the list shown in Figure 36-2 above.

### 36.7 Responding to scroll position

Both *LazyRow* and *LazyColumn* allow actions to be performed when a list scrolls to a specified item position. This can be particularly useful for displaying a “scroll to top” button that appears only when the user scrolls towards the end of the list.

The behavior is implemented by accessing the *firstVisibleItemIndex* property of the *LazyListState* instance which contains the index of the item that is currently the first visible item in the list. For example, if the user scrolls a *LazyColumn* list such that the third item in the list is currently the topmost visible item, *firstVisibleItemIndex* will contain a value of 2 (since indexes start counting at 0). The following code, for example, could be used to display a “scroll to top” button when the first visible item index exceeds 8:

```
val firstVisible = listState.firstVisibleItemIndex

if (firstVisible > 8) {
    // Display scroll to top button
}
```

### 36.8 Creating a lazy grid

Grid layouts may be created using the *LazyVerticalGrid* composable. The appearance of the grid is controlled by the *cells* parameter that can be set to either *adaptive* or *fixed* mode. In adaptive mode, the grid will calculate the number of rows and columns that will fit into the available space, with even spacing between items and subject to a minimum specified cell size. Fixed mode, on the other hand, is passed the number of rows to be displayed and sizes each column width equally to fill the width of the available space.

The following code, for example, declares a grid containing 30 cells, each with a minimum width of 60dp:

```
LazyVerticalGrid(GridCells.Adaptive(minSize = 60.dp),
    state = rememberLazyGridState(),
```

```

        contentPadding = PaddingValues(10.dp)
    ) {
        items(30) { index ->
            Card(
                colors = CardDefaults.cardColors(
                    containerColor = MaterialTheme.colorScheme.primary
                ),
                modifier = Modifier.padding(5.dp).fillMaxSize() {

                    Text(
                        "$index",
                        textAlign = TextAlign.Center,
                        fontSize = 30.sp,
                        color = Color.White,
                        modifier = Modifier.width(120.dp)
                    )
                }
            )
        }
    }
}

```

When called, the `LazyVerticalGrid` composable will fit as many items as possible into each row without making the column width smaller than 60dp as illustrated in the figure below:

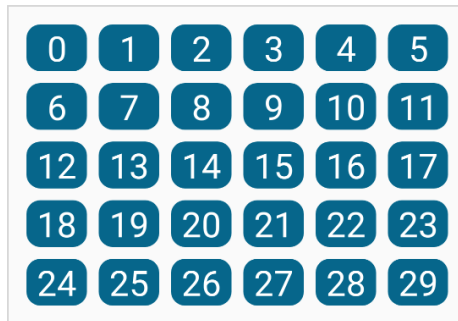


Figure 36-3

The following code organizes items in a grid containing three columns:

```

LazyVerticalGrid(
    GridCells.Fixed(3),
    state = rememberLazyGridState(),
    contentPadding = PaddingValues(10.dp)
) {

    items(15) { index ->
        Card(colors = CardDefaults.cardColors(
            containerColor = MaterialTheme.colorScheme.primary
        ),
            modifier = Modifier.padding(5.dp).fillMaxSize() {
                Text(

```

## An Overview of Lists and Grids in Compose

```
        "$index",  
        fontSize = 35.sp,  
        color = Color.White,  
        textAlign = TextAlign.Center,  
        modifier = Modifier.width(120.dp))  
    }  
}
```

The layout from the above code will appear as illustrated in Figure 36-4 below:

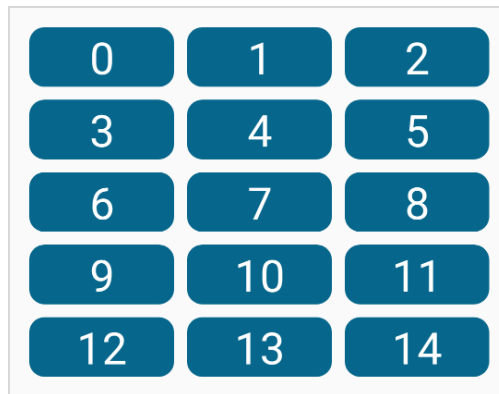


Figure 36-4

Both the above grid examples used a Card composable containing a Text component for each cell item. The Card component provides a surface into which to group content and actions relating to a single content topic and is often used as the basis for list items. Although we provided a Text composable as the child, the content in a card can be any composable, including containers such as Row, Column, and Box layouts. A key feature of Card is the ability to create a shadow effect by specifying an elevation:

```
Card(  
    modifier = Modifier  
        .fillMaxWidth()  
        .padding(15.dp),  
    elevation = CardDefaults.cardElevation(  
        defaultElevation = 10.dp  
    )  
) {  
    Column(horizontalAlignment = Alignment.CenterHorizontally,  
        modifier = Modifier.padding(15.dp).fillMaxWidth()  
    ) {  
        Text("Jetpack Compose", fontSize = 30.sp, )  
        Text("Card Example", fontSize = 20.sp)  
    }  
}
```

When rendered, the above Card component will appear as shown in Figure 36-5:

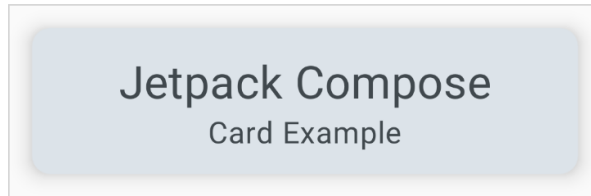


Figure 36-5

## 36.9 Summary

Lists in Compose may be created using either standard or lazy list components. The lazy components have the advantage that they can present large amounts of content without impacting the performance of the app or the device on which it is running. This is achieved by creating list items only when they become visible and destroying them as they scroll out of view. Lists can be presented in row, column, and grid formats and can be static or scrollable. It is also possible to programmatically scroll lists to specific positions and to trigger events based on the current scroll position.





## 45. Working with ViewModels in Compose

Until a few years ago, 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 became part of Android Jetpack when it was released in 2018. Jetpack has of course, since been expanded with the addition of Compose.

This chapter will provide an overview of the concepts of Jetpack, Android app architecture recommendations, and the ViewModel component.

### 45.1 What is Android Jetpack?

Android Jetpack consists of Android Studio, the Android Architecture Components, Android Support Library, and the Compose framework together with a set of guidelines that recommend how an Android App should be structured. The Android Architecture Components were designed to make it quicker and easier both to perform common tasks when developing Android apps while also conforming to the key principle of the architectural guidelines. While many of these components have been superseded by features built into Compose, the ViewModel architecture component remains relevant today. Before exploring the ViewModel component, it first helps to understand both the old and new approaches to Android app architecture.

### 45.2 The “old” architecture

In the chapter entitled *“An Example Compose Project”*, 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. Up 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.

### 45.3 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 called “separation of concerns”). One of the keys to this approach is the ViewModel component.

### 45.4 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 in 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 needed by those controllers.

A ViewModel is implemented as a separate class and contains *state* values containing the model data and functions that can be called to manage that data. The activity containing the user interface *observes* the model state values such that any value changes trigger a recomposition. User interface events relating to the model data such as a button click are configured to call the appropriate function within the ViewModel. This is, in fact, a direct implementation of the *unidirectional data flow* concept described in the chapter entitled “*An Overview of Compose State and Recomposition*”. The diagram in Figure 45-1 illustrates this concept as it relates to activities and ViewModels:

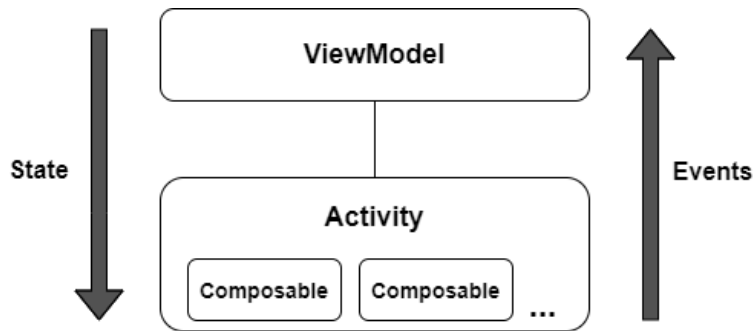


Figure 45-1

This separation of responsibility addresses the issues relating to the lifecycle of activities. Regardless of how many times an activity is recreated during the lifecycle of an app, the ViewModel instances remain in memory thereby maintaining data consistency. A ViewModel used by an activity, for example, will remain in memory until the activity finishes which, in the single activity app, is not until the app exits.

In addition to using ViewModels, 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. This topic will be covered in detail beginning with the chapter entitled “*Room Databases and Compose*”.

### 45.5 ViewModel implementation using state

The main purpose of a ViewModel is to store data that can be observed by the user interface of an activity. This allows the user interface to react when changes occur to the ViewModel data. There are two ways to declare the data within a ViewModel so that it is observable. One option is to use the Compose state mechanism which has been used extensively throughout this book. An alternative approach is to use the Jetpack LiveData component, a topic that will be covered later in this chapter.

Much like the state declared within composables, ViewModel state is declared using the *mutableStateOf* group of functions. The following ViewModel declaration, for example, declares a state containing an integer count value with an initial value of 0:

```
class MyViewModel : ViewModel() {  
  
    var customerCount by mutableStateOf(0)  
  
}
```

With some data encapsulated in the model, the next step is to add a function that can be called from within the UI to change the counter value:

```
class MyViewModel : ViewModel() {

    var customerCount by mutableStateOf(0)

    fun increaseCount() {
        customerCount++
    }
}
```

Even complex models are nothing more than a continuation of these two basic state and function building blocks.

## 45.6 Connecting a ViewModel state to an activity

A ViewModel is of little use unless it can be used within the composables that make up the app user interface. All this requires is to pass an instance of the ViewModel as a parameter to a composable from which the state values and functions can be accessed. Programming convention recommends that these steps be performed in a composable dedicated solely for this task and located at the top of the screen's composable hierarchy. The model state and event handler functions can then be passed to child composables as necessary. The following code shows an example of how a ViewModel might be accessed from within an activity:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ViewModelWorkTheme {
                Surface(color = MaterialTheme.colorScheme.background) {
                    TopLevel()
                }
            }
        }
    }
}
```

```
@Composable
fun TopLevel(model: MyViewModel = viewModel()) {
    MainScreen(model.customerCount) { model.increaseCount() }
}
```

```
@Composable
fun MainScreen(count: Int, addCount: () -> Unit = {}) {
    Column(horizontalAlignment = Alignment.CenterHorizontally,
        modifier = Modifier.fillMaxWidth()) {
        Text("Total customers = $count",
            Modifier.padding(10.dp))
        Button(
            onClick = addCount,
        ) {
            Text(text = "Add a Customer")
        }
    }
}
```

```
        }  
    }  
}
```

In the above example, the first function call is made by the `onCreate()` method to the `TopLevel` composable which is declared with a default `ViewModel` parameter initialized via a call to the `viewModel()` function:

```
@Composable  
fun TopLevel(model: MyViewModel = viewModel()) {  
    .  
    .  
}
```

The `viewModel()` function is provided by the Compose view model lifecycle library which needs to be added to the project's build dependencies when working with view models as follows:

```
dependencies {  
    .  
    .  
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.6.1")  
    .  
    .  
}
```

If an instance of the view model has already been created within the current scope, the `viewModel()` function will return a reference to that instance. Otherwise, a new view model instance will be created and returned.

With access to the `ViewModel` instance, the `TopLevel` function is then able to obtain references to the view model `customerCount` state variable and `increaseCount()` function which it passes to the `MainScreen` composable:

```
MainScreen(model.customerCount) { model.increaseCount() }
```

As implemented, Button clicks will result in calls to the view model `increaseCount()` function which, in turn, increments the `customerCount` state. This change in state triggers a recomposition of the user interface, resulting in the new customer count value appearing in the `Text` composable.

The use of state and view models will be demonstrated in the chapter entitled “A Compose *ViewModel* Tutorial”.

## 45.7 ViewModel implementation using LiveData

The Jetpack `LiveData` component predates the introduction of Compose and can be used as a wrapper around data values within a view model. Once contained in a `LiveData` instance, those variables become observable to composables within an activity. `LiveData` instances can be declared as being mutable using the `MutableLiveData` class, allowing the `ViewModel` functions to make changes to the underlying data value. An example view model designed to store a customer name could, for example, be implemented as follows using `MutableLiveData` instead of state:

```
class MyViewModel : ViewModel() {  
  
    var customerName: MutableLiveData<String> = MutableLiveData("")  
  
    fun setName(name: String) {  
        customerName.value = name  
    }  
}
```

Note that new values must be assigned to the live data variable via the `value` property.

## 45.8 Observing ViewModel LiveData within an activity

As with state, the first step when working with LiveData is to obtain an instance of the view model within an initialization composable:

```
@Composable
fun TopLevel(model: MyViewModel = viewModel()) {

}
```

Once we have access to a view model instance, the next step is to make the live data observable. This is achieved by calling the *observeAsState()* method on the live data object:

```
@Composable
fun TopLevel(model: MyViewModel = viewModel()) {
    var customerName: String by model.customerName.observeAsState("")
}
```

In the above code, the *observeAsState()* call converts the live data value into a state instance and assigns it to the *customerName* variable. Once converted, the state will behave in the same way as any other state object, including triggering recompositions whenever the underlying value changes.

The use of LiveData and view models will be demonstrated in the chapter entitled “*A Compose Room Database and Repository Tutorial*”.

## 45.9 Summary

Until recently, Google has tended not to recommend any particular approach to structuring an Android app. That changed with the introduction of Android Jetpack which consists of a set of tools, components, libraries, and architecture guidelines. These architectural guidelines recommend that an app project be divided into separate modules, each being 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. This is achieved using the ViewModel component. In this chapter, we have covered ViewModel-based architecture and demonstrated how this is implemented when developing with Compose. We have also explored how to observe and access view model data from within an activity using both state and LiveData.



## 47. An Overview of Android SQLite Databases

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

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

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

### 47.1 Understanding database tables

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

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

### 47.2 Introducing database schema

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

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

### 47.3 Columns and data types

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

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

Each column, in turn, is defined to contain a certain type of data. A column designed to store numbers would,

therefore, be defined as containing numerical data.

## 47.4 Database rows

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

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

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

## 47.5 Introducing primary keys

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

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

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

## 47.6 What is SQLite?

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

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

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

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

## 47.7 Structured Query Language (SQL)

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



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

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

## 47.8 Trying SQLite on an Android Virtual Device (AVD)

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

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

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

```
adb -e shell
```

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

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

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

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

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

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

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

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

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

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

## An Overview of Android SQLite Databases

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

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

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

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

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

```
sqlite> .tables
contacts
```

To insert records into the table:

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

To retrieve all rows from a table:

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

To extract a row that meets specific criteria:

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

To exit from the *sqlite3* interactive environment:

```
sqlite> .exit
```

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

It is also important to note that, while it is possible to connect with an *adb* shell to a physical Android device, the shell is not granted sufficient privileges by default to create and manage SQLite databases. Debugging of database problems is, therefore, best performed using an AVD session. Alternatively, databases can be inspected on both emulators and devices using the Android Studio Database Inspector, a topic that will be covered later.

## 47.9 The Android Room persistence library

SQLite is, as previously mentioned, written in the C programming language while Android applications are primarily developed using Java or Kotlin. To bridge this “language gap” in the past, the Android SDK included a set of classes that provide a layer on top of the SQLite database management system. Although still available in the SDK, the use of these classes involves writing a considerable amount of code and does not take advantage of the new architecture guidelines and features such as view models and LiveData. To address these shortcomings, the Android Jetpack Architecture Components include the Room persistent library. This library provides a high-level interface on top of the SQLite database system that makes it easy to store data locally on Android devices with minimal coding while also conforming to the recommendations for modern application architecture.

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

### 47.10 Summary

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

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



# 50. An Overview of Navigation in Compose

Very few Android apps today consist of just a single screen. In reality, most apps comprise multiple screens through which the user navigates using screen gestures, button clicks, and menu selections. Before the introduction of Android Jetpack, the implementation of navigation within an app was primarily a manual coding process with no easy way to view and organize potentially complex navigation paths. This situation improved considerably, however, with the introduction of the Android Navigation Architecture Component, which has now been extended to support navigation in Compose-based apps. This chapter will provide an overview of navigation within Compose, including explanations of routes, navigation graphs, the navigation back stack, passing arguments, and the `NavHostController` and `NavHost` classes.

## 50.1 Understanding navigation

Every app has a home screen that appears after the app has launched and after any splash screen has appeared (a splash screen being the app branding screen that appears temporarily while the app loads). From this home screen, the user will typically perform tasks that will result in other screens appearing. These screens will usually take the form of other composables within the project. A messaging app, for example, might have a home screen listing current messages from which the user can navigate to another screen to access a contact list or a settings screen. The contacts list screen, in turn, might allow the user to navigate to other screens where new users can be added or existing contacts updated. Graphically, the app's *navigation graph* might be represented as shown in Figure 50-1:

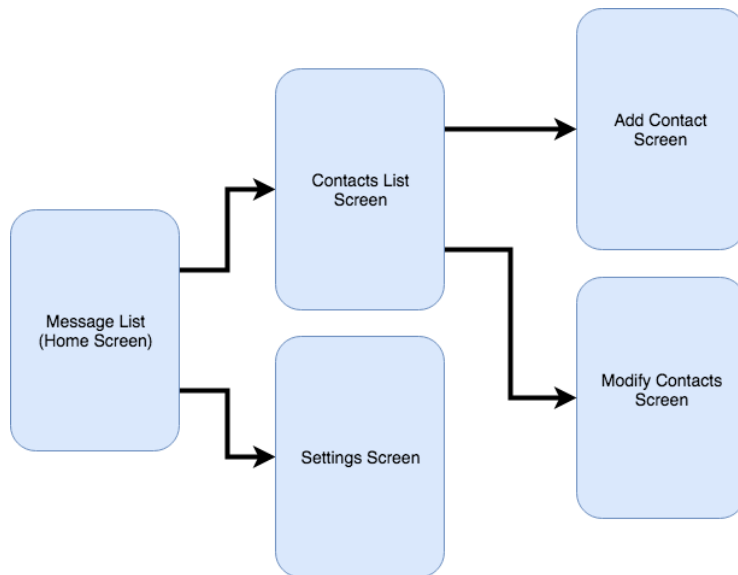


Figure 50-1

Each screen that makes up an app, including the home screen, is referred to as a *destination* and is usually a composable or activity. The Android navigation architecture uses a *navigation back stack* to track the user's path

through the destinations within the app. When the app first launches, the home screen is the first destination placed onto the stack and becomes the *current destination*. When the user navigates to another destination, that screen becomes the current destination and is *pushed* onto the back stack above the home destination. As the user navigates to other screens, they are also pushed onto the stack. Figure 50-2, for example, shows the current state of the navigation stack for the hypothetical messaging app after the user has launched the app and is navigating to the “Add Contact” screen:

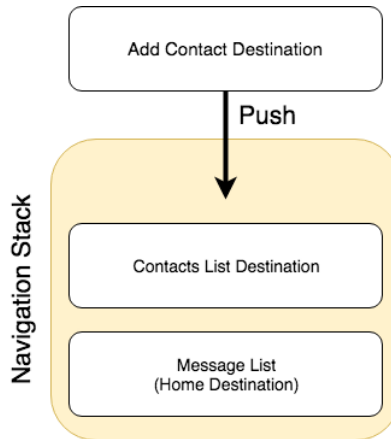


Figure 50-2

As the user navigates back through the screens using the system back button, each destination composable is *popped* off the stack until the home screen is once again the only destination on the stack. In Figure 50-3, the user has navigated back from the Add Contact screen, popping it off the stack and making the Contact List screen composable the current destination:

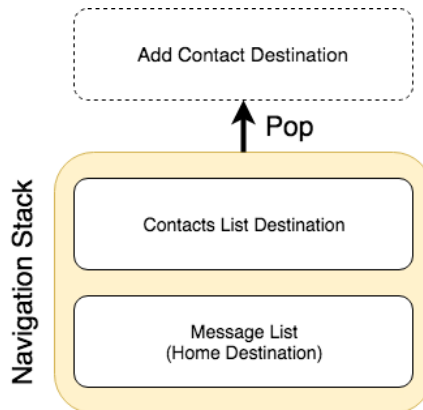


Figure 50-3

All the work involved in navigating between destinations and managing the navigation stack is handled by a *navigation controller*, represented by the `NavHostController` class. It is also possible to manually pop composables off the stack so that the app returns to a screen lower down the stack when the user navigates backward from the current screen.

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

## 50.2 Declaring a navigation controller

The first step in adding navigation to an app project is to create a `NavHostController` instance. This is responsible for managing the back stack and keeping track of which composable is the current destination. So that the integrity of the back stack is maintained through recomposition, `NavHostController` is a stateful object and is created via a call to the `rememberNavController()` method as follows:

```
val navController = rememberNavController()
```

Once a navigation controller has been created it needs to be assigned to a `NavHost` instance.

## 50.3 Declaring a navigation host

The navigation host (`NavHost`) is a special component that is added to the user interface layout of an activity and serves as a placeholder for the destinations through which the user will navigate. Figure 50-4, for example, shows a typical activity screen and highlights the area represented by the navigation host:



Figure 50-4

When it is called, `NavHost` must be passed a `NavHostController` instance, a composable to serve as the *start destination*, and a *navigation graph*. The navigation graph consists of all the composables that are to be available as navigation destinations within the context of the navigation controller. These destinations are declared in the form of *routes*:

```
NavHost(navController = navController, startDestination = <start route>) {
    // Navigation graph destinations
}
```

## 50.4 Adding destinations to the navigation graph

Destinations are added to the navigation graph by making calls to the `composable()` method and providing a *route* and destination. The route is simply a string value that uniquely identifies the destination within the context of the current navigation controller. The destination is the composable to be called when the navigation is performed. The following `NavHost` declaration includes a navigation graph consisting of three destinations,

## An Overview of Navigation in Compose

with the “home” route configured as the start destination:

```
NavHost(navController = navController, startDestination = "home") {  
  
    composable("home") {  
        Home()  
    }  
  
    composable("customers") {  
        Customers()  
    }  
  
    composable("purchases") {  
        Purchases()  
    }  
}
```

A more flexible alternative to hard-coding the route strings into the *composable()* method calls is to define the routes in a sealed class:

```
sealed class Routes(val route: String) {  
    object Home : Routes("home")  
    object Customers : Routes("customers")  
    object Purchases : Routes("purchases")  
}
```

With the class declared, the NavHost will now reference the routes as follows:

```
NavHost(navController = navController, startDestination = Routes.Home.route) {  
  
    composable(Routes.Home.route) {  
        Home()  
    }  
  
    composable(Routes.Customers.route) {  
        Customers()  
    }  
  
    composable(Routes.Purchases.route) {  
        Purchases()  
    }  
}
```

The use of the sealed class approach gives us the advantage of a single location in which to make changes to the routes. Also, it adds syntax validation to avoid mistyping a route string when creating a NavHost or performing navigation.

## 50.5 Navigating to destinations

The primary mechanism for triggering navigation is via calls to the *navigate()* method of the navigation controller instance, specifying the route for the destination composable. The following code, for example, configures a



Button component to navigate to the Customers screen when clicked:

```
Button(onClick = {
    navController.navigate(Routes.Customers.route)
}) {
    Text(text = "Navigate to Customers")
}
```

The *navigate()* method also accepts a trailing lambda containing navigation options, one of which is the *popUpTo()* function. Consider, for example, a scenario where the user starts on the home screen and then navigates to the customer screen. The customer screen displays a list of customer names which, when clicked navigates to the purchases screen populated with a list of the selected customer's previous purchases. At this point, the back stack contains the customer and home destinations. If the user were to tap the back button located at the bottom of the screen, the app will navigate back to the customer screen. The *popUpTo()* navigation option allows us to pop items off the stack back to the specific destination. We could, for example, pop all destinations off the stack before navigating to the purchases screen so that only the home destination remains on the back stack as follows:

```
Button(onClick = {
    navController.navigate(Routes.Customers.route) {
        popUpTo(Routes.Home.route)
    }
}) {
    Text(text = "Navigate to Customers")
}
```

Now when the user clicks the back button on the purchases screen, the app will navigate directly to the home screen. The *popUpTo()* method also accepts options. The following, for example, uses the *inclusive* option to also pop the home destination off the stack before performing the navigation:

```
Button(onClick = {
    navController.navigate(Routes.Customers.route) {
        popUpTo(Routes.Home.route) {
            inclusive = true
        }
    }
}) {
    Text(text = "Navigate to Customers")
}
```

By default, an attempt to navigate from the current destination to itself will push an additional destination instance onto the stack. In most situations, this is unlikely to be the desired behavior. To prevent the addition of multiple instances of the same destination to the top of the stack, set the *launchSingleTop* option to true when calling the *navigate()* method:

```
Button(onClick = {
    navController.navigate(Routes.Customers.route) {
        launchSingleTop = true
    }
}) {
    Text(text = "Navigate to Customers")
}
```

The `saveState` and `restoreState` options, if set to true, will automatically save and restore the state of back stack entries when the user reselects a destination that has been selected previously.

### 50.6 Passing arguments to a destination

It is a common requirement when navigating from one screen to another to need to pass an argument to the destination. Compose supports the passing of arguments of a wide range of types from one screen to another and involves several steps. In our hypothetical example, we would probably need to pass the name of the selected customer from the customer screen to the purchases screen so that the correct purchase history can be displayed.

The first step in navigating with arguments involves adding the argument name to the destination route. We can, for example, add an argument named “customerName” to the purchases route as follows:

```
NavHost(navController = navController, startDestination = Routes.Home.route) {  
    .  
    .  
    composable(Routes.Purchases.route + "{customerName}") {  
        Purchases()  
    }  
    .  
    .  
}
```

When the app triggers navigation to the customer destination, the value to be assigned to the argument will be stored within the corresponding back stack entry. The back stack entry for the current navigation is passed as a parameter to the trailing lambda of the `composable()` method where it can be extracted and passed to the Customer composable:

```
composable(Routes.Purchases.route + "{customerName}") { backStackEntry ->  
  
    val customerName = backStackEntry.arguments?.getString("customerName")  
  
    Purchases(customerName)  
}
```

By default, the navigation argument is assumed to be of String type. To pass arguments of different types, the type must be specified using the `NavType` enumeration via the `composable()` method `arguments` parameter. In the following example, the parameter type is declared as being of type `Int`. Note also that the argument now needs to be extracted from the back stack entry using `getInt()` instead of `getString()`:

```
composable(Routes.Purchases.route + "{customerId}",  
    arguments = listOf(navArgument("customerId") { type = NavType.IntType })) {  
    navBackStack ->  
    Customers(navBackStack.arguments?.getInt("customerId"))  
}
```

Returning to the original string argument example, the Purchases composable now needs to be modified to expect a String parameter:

```
@Composable  
fun Customers(customerName: String?) {  
    .  
    .  
}
```

```
}
```

The final step is to pass a value for the argument when making the `navigate()` method call. We do this by appending the argument value to the end of the destination route. Assuming that the value we need to pass to the purchases screen is stored as a state variable named `selectedCustomer`, the `navigate()` call would be written as follows:

```
var selectedCustomer by remember {
    mutableStateOf("")
}

// Code to identify selected customer here

Button(onClick = {
    navController.navigate(Routes.Customers.route + " /$selectedCustomer")
}) {
    Text(text = "Navigate to Customers")
}
```

When the button is clicked, the following sequence of events will occur:

1. A back stack entry is created for the current destination.
2. The current `selectedCustomer` state value is stored in the back stack entry.
3. The back stack entry is pushed onto the back stack.
4. The `composable()` method for the purchase route in the `NavHost` declaration is called.
5. The trailing lambda of the `composable()` method extracts the argument value from the back stack entry and passes it to the Purchases composable.

## 50.7 Working with bottom navigation bars

So far in this chapter, we have focused on navigation in response to click events on `Button` components. Another common form of navigation involves the bottom navigation bar.

The bottom navigation bar appears at the bottom of the screen and displays a list of navigation items, usually comprising an icon and a label. Clicking on an item navigates to a different screen within the current activity. An example bottom navigation bar is illustrated in Figure 50-5 below:

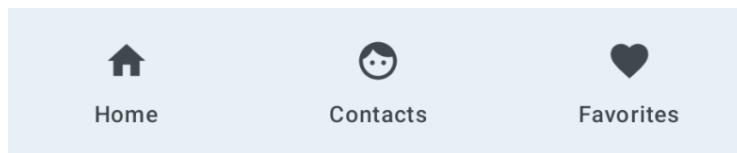


Figure 50-5

The core components of bottom bar navigation are the Compose `BottomNavigation` and `BottomNavigationItem` components. Implementation typically involves a parent `BottomNavigationBar` containing a *forEach* loop which iterates through a list creating each `BottomNavigationItem` child. Each child is configured with the label and icon to be displayed and an `onClick` handler to perform the navigation to the corresponding destination. Typical syntax will read as follows:

```
BottomNavigation {
```

## An Overview of Navigation in Compose

```
<items list>.forEach { navItem ->

    BottomNavigationItem(
        selected = <true | false>,
        onClick = {
            navController.navigate(navItem.route) {
                popUpTo(navController.graph.findStartDestination().id) {
                    saveState = true
                }
                launchSingleTop = true
                restoreState = true
            }
        },

        icon = {
            <icon>
        },
        label = {
            <text>
        },
    )
}
```

Note that the *PopUpTo()* method is called to ensure that if the user clicks the back button the navigation returns to the start destination. We can identify the start destination by calling the *findStartDestination()* method on the navigation graph:

```
navController.graph.findStartDestination()
```

Also, the *launchSingleTop*, *saveState*, and *restoreState* options must be enabled when working with bottom bar navigation.

Each *BottomNavigationItem* needs to be told whether it is the currently selected item via the *selected* property. When working with bottom bar navigation, you will need to write code to compare the route associated with the item against the current route selection. We can obtain the current route selection by gaining access to the back stack via the *currentBackStackEntryAsState()* method of the navigation controller and accessing the destination route property, for example:

```
BottomNavigation {
    val backStackEntry by navController.currentBackStackEntryAsState()
    val currentRoute = backStackEntry?.destination?.route

    NavBarItems.BarItems.forEach { navItem ->

        BottomNavigationItem(
            selected = currentRoute == navItem.route
        )
    }
}
```

The two routes are then compared and the result assigned to the selected property. A more detailed example of bottom bar navigation will be demonstrated in the chapter entitled “*A Compose Navigation Bar Tutorial*”.

## 50.8 Summary

This chapter has covered the addition of navigation to Android apps using the Compose support built into the Jetpack Navigation Architecture Component. Navigation is implemented by creating an instance of the `NavHostController` class and associating it with a `NavHost` instance. The `NavHost` instance is configured with the starting destination and the navigation routes that make up the navigation graph for the current activity. Navigation is then performed by making calls to the `navigate()` method of the navigation controller instance, passing through the path of the destination composable. Compose also supports the passing of arguments to the destination composable. Navigation may also be added to screens using the `Compose BottomNavigation` and `BottomNavigationItem` components.



# 57. 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. However, Google soon introduced another revenue opportunity by embedding advertising within applications. Perhaps the most common and lucrative option is now to charge the user for purchasing items from within the application after it has been installed. 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.

## 57.1 Preparing a project for In-App purchasing

Building in-app purchasing into an app will require a Google Play Developer Console account, which was covered previously in the “*Creating, Testing and Uploading an Android App Bundle*” chapter. In addition, you must also register a Google merchant account and configure your payment settings. You can find these settings 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 will then need to be uploaded to the console and enabled for in-app purchasing. The console will not activate in-app purchasing support for an app, however, unless the Google Play Billing Library has been added to the module-level *build.gradle.kts* file. When working with Kotlin, the Google Play Kotlin Extensions Library is also recommended:

```
dependencies {
    .
    .
    implementation("com.android.billingclient:billing:<latest version>")
    implementation("com.android.billingclient:billing-ktx:<latest version>")
    .
    .
}
```

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

## 57.2 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 57-1 below:

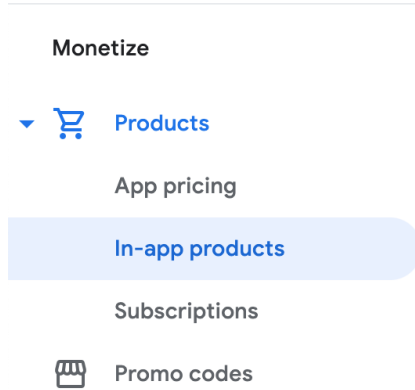


Figure 57-1

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 on a regular schedule such as access to news content or the premium features of an app. When creating a subscription, a *base plan* is defined specifying the price, renewal period (monthly, annually, etc.), and whether the subscription auto-renews. Users can also be provided with discount *offers* and given the option of pre-purchasing a subscription.

### 57.3 Billing client initialization

A `BillingClient` instance handles communication between your app and the Google Play Billing Library. 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 cancelled by user
        } else {
```



```

        // Handle errors here
    }
}

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

```

## 57.4 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. To establish this connection, a call needs to be made to the *startConnection()* method of the billing client instance. Since the connection is performed asynchronously, a *BillingClientStateListener* handler needs to 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
    }
})

```

## 57.5 Querying available products

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

```

val queryProductDetailsParams = QueryProductDetailsParams.newBuilder()
    .setProductList(
        ImmutableList.of(
            QueryProductDetailsParams.Product.newBuilder()

```

## An Overview of Android In-App Billing

```
        .setProductId(productId)
        .setProductType(
            BillingClient.ProductType.INAPP
        )
        .build()
    )
)
.build()

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

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

### 57.6 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 item being purchased.

```
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.

### 57.7 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 will need to 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 together with 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
    }
}

```

## 57.8 Querying previous purchases

When working with in-app billing it is a common requirement to check whether a user has already purchased a product or subscription. A list of all the user's previous purchases of a specific type can be generated by calling

## An Overview of Android In-App Billing

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 =
    PurchaseResponseListener { billingResult, purchases ->

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

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

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

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

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

## 57.9 Summary

In-app purchases provide a way to generate revenue from within Android apps by selling virtual products and subscriptions to users. In this chapter, we have 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 and 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

- ?. 105
- 2D graphics 381
- @Composable 24, 149
- @ExperimentalFoundationApi 326
- :: operator 107
- @Preview 25
  - showSystemUi 25

## A

- acknowledgePurchase() method 535
- Activity Manager 93
- adb
  - command-line tool 71
  - connection testing 77
  - device pairing 75
  - enabling on Android devices 71
  - Linux configuration 74
  - list devices 71
  - macOS configuration 72
  - overview 71
  - restart server 72
  - testing connection 77
  - WiFi debugging 75
  - Windows configuration 73
  - Wireless debugging 75
  - Wireless pairing 75
- AlertDialog 153
- align() 227
- alignByBaseline() 219
- Alignment.Bottom 213, 217
- Alignment.BottomCenter 225
- Alignment.BottomEnd 225
- Alignment.BottomStart 225
- Alignment.Center 225

- Alignment.CenterEnd 225
- Alignment.CenterHorizontally 213
- Alignment.CenterStart 225
- Alignment.CenterVertically 213, 217
- Alignment.End 213
- alignment lines 249
- Alignment.Start 213
- Alignment.Top 213, 217
- Alignment.TopCenter 225
- Alignment.TopEnd 225
- Alignment.TopStart 225
- Android
  - architecture 91
  - runtime 92
  - SDK Packages 6
- Android Architecture Components 401
- Android Debug Bridge. *See* ADB
- Android Development
  - System Requirements 3
- Android Jetpack 401
- Android Libraries 92
- Android Monitor tool window 45
- Android Native Development Kit 93
- Android SDK Location
  - identifying 10
- Android SDK Manager 8, 10
- Android SDK Packages
  - version requirements 8
- Android SDK Tools
  - command-line access 9
  - Linux 11
  - macOS 11
  - Windows 7 10
  - Windows 8 10
- Android Software Stack 91
- Android Studio
  - Animation Inspector 379
  - Asset Studio 184
  - changing theme 69

## Index

- Database Inspector 451
- downloading 3
- Editor Window 64
- installation 4
- Layout Editor 147
- Linux installation 5
- macOS installation 4
- Navigation Bar 63
- Project tool window 64
- setup wizard 5
- Status Bar 64
- Toolbar 63
- Tool window bars 64
- tool windows 64
- updating 12
- Welcome Screen 61
- Windows installation 4
- Android Support Library, 401
- Android Virtual Device. *See* AVD
  - overview 39
- Android Virtual Device Manager 39
- AndroidX libraries 564
- animate as state functions 365
- animateColorAsState() function 365, 369, 371
- animateDpAsState() function 373, 378
- AnimatedVisibility 353
  - animation specs 357
  - enter and exit animations 356
  - expandHorizontally() 356
  - expandIn() 356
  - expandVertically() 356
  - fadeIn() 356
  - fadeOut() 359
  - MutableTransitionState 363
  - scaleIn() 359
  - scaleOut() 359
  - shrinkHorizontally() 359
  - shrinkOut() 359
  - shrinkVertically() 359
  - slideIn() 357
  - slideInHorizontally() 357
  - slideInVertically() 357
  - slideOut() 357
  - slideOutHorizontally() 357
  - slideOutVertically() 357
- animateEnterExit() modifier 362
- animateFloatAsState() function 366
- animateScrollTo() function 304, 313
- animateScrollToItem(index: Int) 302
- animateScrollTo(value: Int) 301
- Animation
  - auto-starting 360
  - combining animations 376
  - inspector 379
  - keyframes 375
  - KeyframesSpec 375
  - motion 373
  - spring effects 374
  - state-based 365
  - visibility 353
- Animation damping
  - DampingRatioHighBouncy 374
  - DampingRatioLowBouncy 374
  - DampingRatioMediumBouncy 374
  - DampingRatioNoBouncy 374
- Animation Inspector 379
- AnimationSpec 357
  - tween() function 360
- Animation specs 357
- Animation stiffness
  - StiffnessHigh 375
  - StiffnessLow 375
  - StiffnessMedium 375
  - StiffnessMediumLow 375
  - StiffnessVeryLow 375
- annotated strings 201, 397
  - append function 201
  - buildAnnotatedString function 201
  - ParagraphStyle 202
  - SpanStyle 201
- APK analyzer 528
- APK file 523
- APK File
  - analyzing 528

- APK Signing 564
- APK Wizard dialog 520
- App Bundles 517
  - creating 523
  - overview 517
  - revisions 527
  - uploading 524
- append function 201
- App Inspector 65
- Application
  - stopping 45
- Application Framework 93
- Arrangement.Bottom 215
- Arrangement.Center 214, 215
- Arrangement.End 214
- Arrangement.SpaceAround 216
- Arrangement.SpaceBetween 216
- Arrangement.SpaceEvenly 216
- Arrangement.Start 214
- Arrangement.Top 215
- ART 92
- as 107
- as? 107
- asFlow() builder 495
- Asset Studio 184
- asSharedFlow() 508
- asStateFlow() 505
- async 293
- AVD
  - Change posture 59
  - cold boot 56
  - command-line creation 39
  - creation 39
  - device frame 49
  - Display mode 59
  - launch in tool window 49
  - overview 39
  - quickboot 56
  - Resizable 58
  - running an application 42
  - Snapshots 55
  - standalone 46
  - starting 41
  - Startup size and orientation 42
- B**
- background modifier 198
- barriers 279
- Barriers 264
  - constrained views 264
- baseline
  - alignment 217
- baselines 251
- BaseTextField 152
- BillingClient 536
  - acknowledgePurchase() method 535
  - consumeAsync() method 535
  - getPurchaseState() method 536
  - initialization 532, 543
  - launchBillingFlow() method 534
  - queryProductDetailsAsync() method 535
  - queryPurchasesAsync() method 536
  - startConnection() method 533
- BillingResult 551
  - getDebugMessage() 551
- Bill of Materials. *See* BOM
- Bitwise AND 113
- Bitwise Inversion 112
- Bitwise Left Shift 114
- Bitwise OR 113
- Bitwise Right Shift 114
- Bitwise XOR 113
- BOM 26
  - build.gradle.kts 26
  - compose-bom 26
  - library version mapping 27
  - override library version 27
- Boolean 100
- BottomNavigation 153, 457
- BottomNavigationItem 457
- Box 152
  - align() 227
  - alignment 225
  - Alignment.BottomCenter 225

## Index

- Alignment.BottomEnd 225
- Alignment.BottomStart 225
- Alignment.Center 225
- Alignment.CenterEnd 225
- Alignment.CenterStart 225
- Alignment.TopCenter 225
- Alignment.TopEnd 225
- Alignment.TopStart 225
- BoxScope 227
- contentAlignment 225
- matchParentSize() 227
- overview 223
- tutorial 223
- BoxScope
  - align() 227
  - matchParentSize() 227
  - modifiers 227
- BoxWithConstraints 152
- Brush Text Styling 202
- buffer() operator 502
- buildAnnotatedString function 201
- Build tool window 66
- Build Variants , 66
  - tool window 66
- Button 153
- by keyword 158
  
- C**
- cancelAndJoin() 294
- cancelChildren() 296
- Canvas 152
  - DrawScope 381
  - inset() function 385
  - overview 381
  - size 381
- Card 153
  - example 306
- C/C++ Libraries 92
- centerAround() function 268
- chain head 262
- chaining modifiers 193
- chains 262
  - chain styles 262
- Char 100
- Checkbox 153, 182
- CircleShape 227
- CircularProgressIndicator 153
- clickable 198
- clip 198
- Clip Art 185
- clip() modifier 227
  - CircleShape 227
  - CutCornerShape 227
  - RectangleShape 227
  - RoundedCornerShape 227
- close() function 393
- Code completion 82
- Code editor 20
  - Split mode 20
- Code Editor
  - basics 79
  - Code completion 82
  - Code Generation 84
  - Code Reformatting 87
  - Document Tabs 80
  - Editing area 80
  - Gutter Area 80
  - Live Templates 88
  - Splitting 82
  - Statement Completion 84
  - Status Bar 81
- Code Generation 84
- Code Reformatting 87
- code samples
  - download 1
- Coil
  - library 318
  - rememberImagePainter() function 319
- cold boot 56
- Cold flow 505
  - convert to hot 508
- collectLatest() operator 500
- collect() operator 496
- ColorFilter 398



- color filtering 398
- Column 152
  - Alignment.CenterHorizontally 213
  - Alignment.End 213
  - Alignment.Start 213
  - Arrangement.Bottom 215
  - Arrangement.Center 215
  - Arrangement.SpaceAround 216
  - Arrangement.SpaceBetween 216
  - Arrangement.SpaceEvenly 216
  - Arrangement.Top 215
  - Layout alignment 212
  - list 299
  - list tutorial 309
  - overview 210
  - scope 217
  - scope modifiers 217
  - spacing 216
  - tutorial 209
  - verticalArrangement 214
- Column lists 299
- ColumnScope 217
  - Modifier.align() 217
  - Modifier.alignBy() 217
  - Modifier.weight() 217
- combine() operator 504
- combining modifiers 198
- Communicating Sequential Processes 291
- Companion Objects 137
- components 149
- Composable
  - adding a 30
  - previewing 32
- Composable function
  - syntax 150
- composable functions 149
- composables
  - add modifier support 194
- Composables
  - Foundation 152
  - Material 152
- Compose
  - before 147
  - components 149
  - data-driven 148
  - declarative syntax 147
  - functions 149
  - layout overview 245
  - modifiers 191
  - overview 147
  - state 148
- compose-bom 26
- compose() method 453
- CompositionLocal
  - example 169
  - overview 167
  - state 170, 171
  - syntax 168
- compositionLocalOf() function 168
- conflate() operator 500
- constrainAs() modifier function 267
- constrain() function 283
- Constraint bias 272
- Constraint Bias 261
- ConstraintLayout 152
  - adding constraints 268
  - barriers 279
  - Barriers 264
  - basic constraints 270
  - centerAround() function 268
  - chain head 262
  - chains 262
  - chain styles 262
  - constrainAs() function 267
  - constrain() function 283
  - Constraint bias 272
  - Constraint Bias 261
  - Constraint margins 273
  - Constraints 259
  - constraint sets 282
  - createEndBarrier() 279
  - createHorizontalChain() 277
  - createRefFor() function 283
  - createRef() function 267

## Index

- createRefs() function 267
- createStartBarrier() 279
- createTopBarrier() 279
- createVerticalChain() 277
- creating chains 277
- generating references 267
- guidelines 278
- Guidelines 265
- how to call 267
- layout() modifier 286
- library 269
- linkTo() function 268
- Margins 260
- Opposing constraints 271
- Opposing Constraints 260, 276
- overview of 259
- Packed chain 263
- reference assignment 267
- Spread chain 262
- Spread inside chain 262
- Weighted chain 262
- Widget Dimensions 263
- Constraint margins 273
- constraints 254
- constraint sets 282
- consumeAsync() method 535
- ConsumeParams 546
- contentAlignment 225
- Content Provider 93
- Coroutine Builders 293
  - async 293
  - coroutineScope 293
  - launch 293
  - runBlocking 293
  - supervisorScope 293
  - withContext 293
- Coroutine Dispatchers 292
- Coroutines 302, 493
  - channel communication 297
  - coroutine scope 302
  - CoroutineScope 302
  - GlobalScope 292
  - LaunchedEffect 296
  - rememberCoroutineScope() 304
  - rememberCoroutineScope() function 292
  - SideEffect 296
  - Side Effects 296
  - Suspend Functions 292
  - suspending 294
  - ViewModelScope 292
  - vs Threads 291
  - vs. Threads 291
- coroutineScope 293
- CoroutineScope 292, 302
  - rememberCoroutineScope() 304
- createEndBarrier() 279
- createHorizontalChain() 277
- createRefFor() function 283
- createRef() function 267
- createRefs() 267
- createStartBarrier() 279
- createTopBarrier() 279
- createVerticalChain() 277
- cross axis arrangement 243
- Crossfading 361
- currentBackStackEntryAsState() method 458, 476
- Custom Accessors 135
- Custom layout 253
  - building 253
  - constraints 254
  - Layout() composable 254
  - measurables 254
  - overview 253
  - Placeable 254
  - syntax 253
- custom layout modifiers 245
  - alignment lines 249
  - baselines 251
  - creating 247
  - default position 247
- Custom layouts
  - overview 245
  - tutorial 245
- Custom Theme

building 555  
 CutCornerShape 227

## D

DampingRatioHighBouncy 374  
 DampingRatioLowBouncy 374  
 DampingRatioMediumBouncy 374  
 DampingRatioNoBouncy 374  
 Dark Theme 45
 

- enable on device 45

 dashPathEffect() method 383  
 Data Access Object (DAO) 424, 436  
 Data Access Objects 427  
 Database Inspector 433, 451
 

- live updates 451
- SQL query 451

 Database Rows 418  
 Database Schema 417  
 Database Tables 417  
 data-driven 148  
 DDMS 45  
 Debugging
 

- enabling on device 71

 declarative syntax 147  
 Default Function Parameters 127  
 default position 247  
 derivedStateOf 329  
 Device File Explorer 66  
 device frame 49  
 Device Mirroring 77
 

- enabling 77

 device pairing 75  
 Dispatchers.Default 293  
 Dispatchers.IO 295  
 Dispatchers.Main 292  
 drag gestures 482  
 drawable
 

- folder 184

 drawArc() function 392  
 drawCircle() function 388  
 drawImage() function 395  
 Drawing

arcs 392  
 circle 388  
 close() 393  
 dashed lines 383  
 dashPathEffect() 383  
 drawArc() 392  
 drawImage() 395  
 drawPath() 393  
 drawPoints() 394  
 drawRect() 383  
 drawRoundRect() 386  
 gradients 389  
 images 395  
 line 381  
 oval 388  
 points 394  
 rectangle 383  
 rotate() 387  
 rotation 387  
 Drawing text 397  
 drawLine() function 382  
 drawPath() function 393  
 drawPoints() function 394  
 drawRect() function 383  
 drawRoundRect() function 386  
 DrawScope 381  
 drawText() function 397, 398  
 DropdownMenu 153  
 DROP\_LATEST 507  
 DROP\_OLDEST 507  
 DurationBasedAnimationSpec 357  
 Dynamic colors
 

- enabling in Android 561

## E

Elvis Operator 107  
 emit 149  
 Empty Compose Activity
 

- template 16

 Emulator
 

- battery 54
- cellular configuration 54

## Index

- configuring fingerprints 56
- directional pad 54
- extended control options 53
- Extended controls 53
- fingerprint 54
- location configuration 54
- phone settings 54
- Resizable 58
- resize 53
- rotate 52
- Screen Record 55
- Snapshots 55
- starting 41
- take screenshot 52
- toolbar 51
- toolbar options 51
- tool window mode 58
- Virtual Sensors 55
- zoom 52
- enablePendingPurchases() method 535
- enabling ADB support 71
- enter animations 356
- EnterTransition.None 362
- Errata 2
- Escape Sequences 101
- ettings.gradle file 564
- exit animations 356
- ExitTransition.None 362
- expandHorizontally() 356
- expandIn() 356
- expandVertically() 356
- Extended Control
  - options 53
- F**
- fadeIn() 356
- fadeOut() 359
- Files
  - switching between 80
- fillMaxHeight 198
- fillMaxSize 198
- fillMaxWidth 198
- filter() operator 498
- findStartDestination() method 458
- Fingerprint
  - emulation 56
- firstVisibleItemIndex 304
- flatMapConcat() operator 503
- flatMapMerge() operator 503
- Float 100
- FloatingActionButton 153
- Flow 493
  - asFlow() builder 495
  - asSharedFlow() 508
  - asStateFlow() 505
  - backgroundn handling 513
  - buffering 500
  - buffer() operator 502
  - builder 495
  - cold 505
  - collect() 499
  - collecting data 499
  - collectLatest() operator 500
  - combine() operator 504
  - conflate() operator 500
  - emit() 495
  - emitting data 495
  - filter() operator 498
  - flatMapConcat() operator 503
  - flatMapMerge() operator 503
  - flattening 502
  - flowOf() builder 495
  - flow of flows 502
  - fold() operator 502
  - hot 505
  - MutableSharedFlow 508
  - MutableStateFlow 505
  - onEach() operator 504
  - reduce() operator 501, 502
  - repeatOnLifecycle 514
  - SharedFlow 506
  - shareIn() function 508
  - single() operator 500
  - StateFlow 505

- transform() operator 498
- try/finally 499
- zip() operator 504
- flow builder 495
- FlowColumn 231, 237, 242
  - cross axis arrangement 243
  - maxItemsInEachColumn 232
  - tutorial 237
- Flow layout
  - arrangement 240
- Flow layouts
  - cross axis arrangement 234
  - fillMaxHeight() 236
  - fillMaxWidth() 236
  - Fractional sizing 236
  - horizontalArrangement 243
  - Item alignment 235
  - item weights 243
  - main axis arrangement 232
  - verticalArrangement 243
  - weight 236
- flowOf() builder 495
- flow of flows 502
- FlowRow 231, 237, 239
  - cross axis arrangement 243
  - horizontalArrangement 240
  - item alignment 240
  - maxItemsInEachRow 232
  - tutorial 237
- Flows
  - combining 504
  - Introduction to 493
- FontWeight 31
- forEachIndexed 243
- forEach loop 256
- Foundation components 152
- Foundation Composables 152
- Foundation libraries 346
- Foundation library 237
- Function Parameters
  - variable number of 127
- Functions 125

## G

- Gestures 479
  - click 479
  - drag 482
  - horizontalScroll() 486
  - overview 479
  - pinch gestures 490
  - PointerInputScope 483
  - rememberScrollableState() function 485
  - rememberScrollState() 486
  - rememberTransformableState() 490
  - rotation gestures 489
  - scrollable() modifier 485
  - scroll modifiers 486
  - taps 483
  - translation gestures 490
  - tutorial 479
  - verticalScroll() 486
- getDebugMessage() 551
- getPurchaseState() method 536
- getStringArray() method 317
- GlobalScope 292
- GNU/Linux 92
- Google Play App Signing 520
- Google Play Billing Library 531
- Google Play Console 538
  - Creating an in-app product 538
  - License Testers 539
- Google Play Developer Console 518
- Google Play store 17
- Gradient drawing 389
- Gradle
  - APK signing settings 571
  - Build Variants 564
  - command line tasks 572
  - dependencies 563
  - Manifest Entries 564
  - overview 563
  - sensible defaults 563
- Gradle Build File
  - top level 567
- Gradle Build Files

## Index

- module level 566
- gradle.properties file 564
- Graphics
  - drawing 381
- Grid
  - overview 299
- groupBy() function 303
- guidelines 278

## H

- Higher-order Functions 129
- horizontalArrangement 214, 216, 243
- HorizontalPager 341
  - animateScrollToPage() 343
  - scrollToPage() 343
  - state 342
  - syntax 341
- horizontalScroll() 486
- Hot flows 505

## I

- Image 152
  - add drawable resource 184
  - painterResource method 186
- Immutable Variables 102
- INAPP 536
- In-App Products 531
- In-App Purchasing 537
  - acknowledgePurchase() method 535
  - BillingClient 532
  - BillingResult 551
  - consumeAsync() method 535
  - ConsumeParams 546
  - Consuming purchases 544
  - enablePendingPurchases() method 535
  - getPurchaseState() method 536
  - Google Play Billing Library 531
  - launchBillingFlow() method 534
  - Libraries 537
  - newBuilder() method 532
  - onBillingServiceDisconnected() callback 542
  - onBillingServiceDisconnected() method 533

- onBillingSetupFinished() listener 544
- onProductDetailsResponse() callback 542
- Overview 531
- ProductDetail 534
- ProductDetails 543
- products 531
- ProductType 536
- Purchase Flow 545
- PurchaseResponseListener 536
- PurchasesUpdatedListener 534
- PurchaseUpdatedListener 543
- purchase updates 543
- queryProductDetailsAsync() 542
- queryProductDetailsAsync() method 535
- queryPurchasesAsync() 546
- queryPurchasesAsync() method 536
- startConnection() method 533
- subscriptions 531
- tutorial 537

- Initializer Blocks 135

- In-Memory Database 430

- Inner Classes 136

- inset() function 385

- IntrinsicSize.Max 289

- IntrinsicSize.Min 289, 290

- intelligent recomposition 155

- IntelliJ IDEA 95

- Interactive mode 36

- Intrinsic measurements 285

- IntrinsicSize 285

- intrinsic measurements 285

- Max 285

- Min 285

- tutorial 287

- is 107

- isInitialized property 107

- isSystemInDarkTheme() function 170

- item() function 300

- items() function 300

- itemsIndexed() function 300

## J

- Java
  - convert to Kotlin 95
- Java Native Interface 93
- JetBrains 95
- Jetpack Compose
  - see Compose 147
- join() 296
  
- K**
- keyboardOptions 413
- Keyboard Shortcuts 67
- keyframe 360
- keyframes 375
  - KeyframesSpec 375
- keyframes() function 375
- KeyframesSpec 375
- Keystore File
  - creation 520
- Kotlin
  - accessing class properties 135
  - and Java 95
  - arithmetic operators 109
  - assignment operator 109
  - augmented assignment operators 110
  - bitwise operators 112
  - Boolean 100
  - break 120
  - breaking from loops 119
  - calling class methods 135
  - Char 100
  - class declaration 131
  - class initialization 132
  - class properties 132
  - Companion Objects 137
  - conditional control flow 121
  - continue labels 120
  - continue statement 120
  - control flow 117
  - convert from Java 95
  - Custom Accessors 135
  - data types 99
  - decrement operator 110
  - Default Function Parameters 127
  - defining class methods 132
  - do ... while loop 119
  - Elvis Operator 107
  - equality operators 111
  - Escape Sequences 101
  - expression syntax 109
  - Float 100
  - Flow 493
  - for-in statement 117
  - function calling 126
  - Functions 125
    - groupBy() function 303
  - Higher-order Functions 129
  - if ... else ... expressions 122
  - if expressions 121
  - Immutable Variables 102
  - increment operator 110
  - inheritance 141
  - Initializer Blocks 135
  - Inner Classes 136
  - introduction 95
  - Lambda Expressions 128
  - let Function 105
  - Local Functions 126
  - logical operators 111
  - looping 117
  - Mutable Variables 102
  - Not-Null Assertion 105
  - Nullable Type 104
  - Overriding inherited methods 144
  - playground 96
  - Primary Constructor 132
  - properties 135
  - range operator 112
  - Safe Call Operator 104
  - Secondary Constructors 132
  - Single Expression Functions 126
  - String 100
  - subclassing 141
  - substringBefore() method 319
  - Type Annotations 103

## Index

- Type Casting 107
- Type Checking 107
- Type Inference 103
- variable parameters 127
- when statement 122
- while loop 118

## L

- Lambda Expressions 128
- lateinit 106
- Late Initialization 106
- launch 293
- launchBillingFlow() method 534
- LaunchedEffect 296
- launchSingleTop 455
- Layout alignment 212
- Layout arrangement 214
- Layout arrangement spacing 216
- Layout components 152
- Layout() composable 254
- Layout Editor 147
- Layout Inspector 66
- layout modifier 198
- layout() modifier 286
- LazyColumn 152, 299
  - creation 300
  - scroll position detection 304
- LazyHorizontalStaggeredGrid 333, 336
  - syntax 334
- LazyList
  - tutorial 315
- Lazy lists 299
  - Scrolling 301
- LazyListScope 300
  - item() function 300
  - items() function 300
  - itemsIndexed() function 300
  - stickyHeader() function 304
- LazyListState 304
  - firstVisibleItemIndex 304
- LazyRow 152, 299
  - creation 300
  - scroll position detection 304
- LazyVerticalGrid 299
  - adaptive mode 306
  - fixed mode 306
- LazyVerticalStaggeredGrid 333, 336
  - syntax 333
- let Function 105
- libc 92
- License Testers 539
- Lifecycle.State.CREATED 514
- Lifecycle.State.DESTROYED 514
- Lifecycle.State.INITIALIZED 514
- Lifecycle.State.RESUMED 514
- Lifecycle.State.STARTED 514
- LinearProgressIndicator 153
- lineTo() 393
- lineTo() function 393
- linkTo() function 268
- Linux Kernel 92
- list devices 71
- Lists
  - clickable items 322
  - enabling scrolling 301
  - overview 299
- literals
  - live editing 32
- LiveData 404
  - observeAsState() 405
- Live Edit 43
  - disabling 32
  - enabling 32
  - of literals 32
- Live Templates 88
- Local Functions 126
- Location Manager 93
- Logcat
  - tool window 65

## M

- MainActivity.kt file 20
  - template code 29
- map method 254



- matchParentSize() 227
- Material Composables 152
- Material Design 2 551
- Material Design 2 Theming 551
- Material Design 3 551
- Material Design components 153
- Material Theme Builder 557
- Material You 551
- maxValue property 313
- measurables 254
- measure() function 400
- measureTimeMillis() function 500
- Memory Indicator 81
- Minimum SDK
  - setting 17
- ModalDrawer 153
- Modern Android architecture 401
- modifier
  - adding to composable 194
  - chaining 193
  - combining 198
  - creating a 192
  - ordering 194
  - tutorial 191
- Modifier.align() 217
- Modifier.alignBy() 217
- modifiers
  - build-in 198
  - overview 191
- Modifier.weight() 217
- multiple devices
  - testing app on 44
- MutableLiveData 404
- MutableSharedFlow 508
- MutableState 156
- MutableStateFlow 505
- mutableStateOf function 149
- mutableStateOf() function 157
- MutableTransitionState 363
- Mutable Variables 102

## N

- NavHost 453, 467, 473
- NavHostController 451, 467, 473
- navigate() method 455
- Navigation 451
  - BottomNavigation 457
  - BottomNavigationItem 457
  - compose() method 453
  - currentBackStackEntryAsState() method 458
  - declaring routes 464
  - findStartDestination() method 458
  - graph 453
  - launchSingleTop 455
  - NavHost 453, 467
  - NavHostController 451, 467
  - navigate() method 455
  - navigation graph 451
  - NavType 456
  - overview 451
  - passing arguments
  - popUpTo() method 455
  - route 453
  - stack 451, 452
  - start destination 453
  - tutorial 461
- Navigation Architecture Component 451
- NavigationBar 474
- NavigationBarItem 474
- Navigation bars 457
- navigation graph 451, 453
- Navigation Host 455
- NavType 456
- newBuilder() method 532
- Notifications Manager 93
- Not-Null Assertion 105
- Nullable Type 104

## O

- observeAsState() 405
- Offset() function 382
- offset modifier 198
- onBillingServiceDisconnected() callback 542
- onBillingServiceDisconnected() method 533

## Index

onBillingSetupFinished() listener 544  
onCreate() method 25  
onEach() operator 504  
onProductDetailsResponse() callback 542  
OpenJDK 3  
Opposing constraints 271  
OutlinedButton 329  
OutlinedTextField 407

## P

Package Manager 93  
Package name 17  
Packed chain 263  
padding 198  
Pager 341  
    animateScrollToPage() 343  
    scrollToPage() 343  
    state 342  
    syntax , 232  
Pager state 342  
painterResource method 186  
ParagraphStyle 202  
PathEffect 383  
pinch gestures 490  
Placeable 254  
PointerInputScope 483  
    drag gestures 486  
    tap gestures 483  
popUpTo() method 455  
Preview configuration picker 35  
Preview panel 25  
    build and refresh 25  
    Interactive mode 36  
    settings 35  
Primary Constructor 132  
Problems  
    tool window 66  
ProductDetail 534  
ProductDetails 543  
ProductType 536  
Profiler  
    tool window 66

proguard-rules.pro file 568  
ProGuard Support 564  
project  
    create new 16  
    package name 17  
Project tool window 19, 65  
    Android mode 19  
PurchaseResponseListener 536  
PurchasesUpdatedListener 536, 543

## Q

queryProductDetailsAsync() 542  
queryProductDetailsAsync() method 535  
queryPurchaseHistoryAsync() method 536  
queryPurchasesAsync() 546  
queryPurchasesAsync() method 536  
quickboot snapshot 56  
Quick Documentation 87

## R

RadioButton 153  
Random  
    nextInt() 238  
Random.nextInt() method 337, 240  
Range Operator 112  
Recent Files Navigation 68  
recomposition 148  
    intelligent recomposition 155  
    overview 155  
RectangleShape 227  
reduce() operator 501, 502  
relativeLineTo() function 393  
Release Preparation 517  
rememberCoroutineScope() function 292, 304, 311  
rememberDraggableState() function 482  
rememberImagePainter() function 319  
remember keyword 157  
rememberPagerState 342  
rememberSaveable keyword 164  
rememberScrollableState() function 485  
rememberScrollState() 486  
rememberScrollState() function 301, 311

- rememberTextMeasurer() function 397
  - rememberTransformableState() 490
  - rememberTransformationState() function 488
  - repeatable() function 361
  - RepeatableSpec
    - repeatable() 361
  - RepeatMode.Reverse 361
  - repeatOnLifecycle 514
  - Repository
    - tutorial 433
  - Resizable Emulator 58
  - Resource Manager 93, 65
  - Room
    - Data Access Object (DAO) 424
    - entities 424, 425
    - In-Memory Database 430
    - Repository 425
  - Room Database 424
    - tutorial 433
  - Room Database Persistence 423
  - Room persistence library 434
  - Room Persistence Library 421
  - rotate modifier 198
  - rotation gestures 489
  - RoundedCornerShape 227
  - Row 152
    - Alignment.Bottom 213
    - Alignment.CenterVertically 213
    - Alignment.Top 213
    - Arrangement.Center 214
    - Arrangement.End 214
    - Arrangement.SpaceAround 216
    - Arrangement.SpaceBetween 216
    - Arrangement.SpaceEvenly 216
    - Arrangement.Start 214
    - horizontalArrangement 214
    - Layout alignment 212
    - Layout arrangement 214
    - list 299
    - list example 314
    - overview 210
    - scope 217
    - scope modifiers 217
    - spacing 216
    - tutorial 209
  - Row lists 299
  - RowScope 217
    - Modifier.align() 217
    - Modifier.alignBy() 217
    - Modifier.alignByBaseline() 217
    - Modifier.paddingFrom() 218
    - Modifier.weight() 218
  - Run
    - tool window 65
  - runBlocking 293
  - Running Devices
    - tool window 77
- ## S
- Safe Call Operator 104
  - Scaffold 153, 475
    - bottomBar 478
    - AppBar 476
  - scaleIn() 359
  - scale modifier 198
  - scaleOut() 359
  - Scope modifiers
    - weights 221
  - scrollable modifier 198
  - scrollable() modifier 485, 486
  - Scroll detection
    - example 325
  - scroll modifiers 486
  - ScrollState
    - maxValue property 313
    - rememberScrollState() function 301
  - scrollToItem(index: Int) 302
  - scrollToPage() 343
  - scrollTo(value: Int) 301
  - SDK Packages 6
  - SDK settings 17
  - Secondary Constructors 132
  - Secure Sockets Layer (SSL) 92
  - settings.gradle.kts file 564

## Index

- Shape 153
- Shapes
  - CircleShape 227
  - CutCornerShape 227
  - RectangleShape 227
  - RoundedCornerShape 227
- SharedFlow 506, 509
  - backgroundn handling 513
  - DROP\_LATEST 507
  - DROP\_OLDEST 507
  - in ViewModel 510
  - repeatOnLifecycle 514
  - SUSPEND 507
  - tutorial 509
- shareIn() function 508
- SharingStarted.Eagerly() 508
- SharingStarted.Lazily() 508
- SharingStarted.WhileSubscribed() 508
- showSystemUi 25, 310
- shrinkHorizontally() 359
- shrinkOut() 359
- shrinkVertically() 359
- SideEffect 296
- Side Effects 296
- single() operator 500
- size modifier 198
- slideIn() 357
- slideInHorizontally() 357
- slideInVertically() 357
- slideOut() 357
- slideOutHorizontally() 357
- slideOutVertically() 357
- Slider 153
- Slider component 33
- Slot APIs
  - calling 176
  - declaring 176
  - overview 175
  - tutorial 179
- Snackbar 153
- Snapshots
  - emulator 55
- SpanStyle 201
- Spread chain 262
- Spread inside chain 262
- Spring effects 374
- spring() function 374
- SQL 418
- SQLite 417
  - AVD command-line use 419
  - Columns and Data Types 417
  - overview 418
  - Primary keys 418
- Staggered Grids 333
- startConnection() method 533
- start destination 453
- state 148
  - basics of 155
  - by keyword 158
  - configuration changes 163
  - declaring 156
  - hoisting 161
  - MutableState 156
  - mutableStateOf() function 157
  - overview 155
  - remember keyword 157
  - rememberSaveable 164
  - Unidirectional data flow 159
- StateFlow 505
- stateful 155
- stateful composables 149
- State hoisting 161
- stateless composables 149
- Statement Completion 84
- staticCompositionLocalOf() function 168, 170
- Status Bar Widgets 81
  - Memory Indicator 81
- stickyHeader 326
- stickyHeader() function 304
- Sticky headers
  - adding 326
  - example 325
  - stickyHeader() function 304
- StiffnessHigh 375

- StiffnessLow 375
- StiffnessMedium 375
- StiffnessMediumLow 375
- StiffnessVeryLow 375
- String 100
- Structure
  - tool window 66
- Structured Query Language 418
- Structure tool window 66
- SUBS 536
- subscriptions 531
- subStringBefore() method 319
- supervisorScope 293
- Surface component 24, 225
- SUSPEND 507
- Suspend Functions 292
- Switch 153
- Switcher 68
- system requirements 3

## T

- Telephony Manager 93
- Terminal
  - tool window 66
- Text 153
- Text component 150
- TextField 153
- TextMeasurer 397
  - measure() function 400
- TextStyle 416
- Theme
  - building a custom 555
- Theming 551
  - tutorial 557
- TODO
  - tool window 67
- Tool window bars 64
- Tool windows 64
- TopAppBar 153, 476
- trailingIcon 414
- TransformableState 490
- transform() operator 498

- translation gestures 490
- try/finally 499
- tween() function 360
- Type Annotations 103
- Type Casting 107
- Type Checking 107
- Type Inference 103
- Type.kt file 556

## U

- UI Controllers 402
- UI\_NIGHT\_MODE\_YES 171
- Unidirectional data flow 159
- updateTransition() function 366, 373, 376
- upload key 520
- USB connection issues
  - resolving 74

## V

- Vector Asset
  - add to project 184
- verticalArrangement 214, 216
- VerticalPager
  - animateScrollToPage() 343
  - scrollToPage() 343
  - state 342
  - syntax , 232
- verticalScroll() 486
- verticalScroll() modifier 311
- ViewModel
  - example 408
  - lifecycle library 404, 408, 494, 509
  - LiveData 404
  - observeAsState() 405
  - overview 401
  - tutorial 407
  - using state 402
  - viewModel() 404, 410, 444
  - ViewModelProvider Factory 443
  - ViewModelStoreOwner 444
- viewModel() function 404, 410, 444
- ViewModelProvider Factory 443

ViewModelScope 292  
ViewModelStoreOwner 444  
View System 93  
Virtual Device Configuration dialog 40  
Virtual Sensors 55  
Visibility animation 353

## **W**

Weighted chain 262  
Welcome screen 61  
while Loop 118  
Widget Dimensions 263  
WiFi debugging 75  
Wireless debugging 75  
Wireless pairing 75  
withContext 293

## **X**

XML resource  
    reading an 315

## **Z**

zip() operator 504