

Jetpack Compose 1.5 Essentials



Jetpack Compose 1.5 Essentials

Jetpack Compose 1.5 Essentials

ISBN-13: 978-1-951442-83-5

© 2024 Neil Smyth / Payload Media, Inc. All Rights Reserved.

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

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

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

Rev: 1.0



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

Table of Contents

1. Start Here.....	1
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
2. Setting up an Android Studio Development Environment.....	3
2.1 System requirements.....	3
2.2 Downloading the Android Studio package	3
2.3 Installing Android Studio.....	4
2.3.1 Installation on Windows	4
2.3.2 Installation on macOS	4
2.3.3 Installation on Linux.....	5
2.4 The Android Studio setup wizard	5
2.5 Installing additional Android SDK packages	6
2.6 Installing the Android SDK Command-line Tools.....	9
2.6.1 Windows 8.1	10
2.6.2 Windows 10	11
2.6.3 Windows 11	11
2.6.4 Linux	11
2.6.5 macOS.....	11
2.7 Android Studio memory management	11
2.8 Updating Android Studio and the SDK	12
2.9 Summary	13
3. A Compose Project Overview	15
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
4. An Example Compose Project	29
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
5. Creating an Android Virtual Device (AVD) in Android Studio	39
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	45
5.6 Stopping a Running Application	46
5.7 Supporting Dark Theme.....	46
5.8 Running the Emulator in a Separate Window.....	47
5.9 Removing the Device Frame.....	50
5.10 Summary	51
6. Using and Configuring the Android Studio AVD Emulator	53
6.1 The Emulator Environment	53
6.2 Emulator Toolbar Options	53
6.3 Working in Zoom Mode	55
6.4 Resizing the Emulator Window.....	55
6.5 Extended Control Options.....	55
6.5.1 Location.....	56
6.5.2 Displays.....	56
6.5.3 Cellular	56
6.5.4 Battery.....	56
6.5.5 Camera.....	56
6.5.6 Phone	56
6.5.7 Directional Pad.....	56
6.5.8 Microphone.....	56
6.5.9 Fingerprint	56
6.5.10 Virtual Sensors	57
6.5.11 Snapshots.....	57
6.5.12 Record and Playback	57
6.5.13 Google Play	57
6.5.14 Settings	57
6.5.15 Help.....	57
6.6 Working with Snapshots.....	57
6.7 Configuring Fingerprint Emulation	58
6.8 The Emulator in Tool Window Mode.....	59
6.9 Creating a Resizable Emulator.....	60
6.10 Summary	62
7. A Tour of the Android Studio User Interface	63
7.1 The Welcome Screen	63
7.2 The Menu Bar	64
7.3 The Main Window	64

7.4 The Tool Windows	66
7.5 The Tool Window Menus	69
7.6 Android Studio Keyboard Shortcuts	69
7.7 Switcher and Recent Files Navigation	70
7.8 Changing the Android Studio Theme	71
7.9 Summary	72
8. Testing Android Studio Apps on a Physical Android Device.....	73
8.1 An Overview of the Android Debug Bridge (ADB).....	73
8.2 Enabling USB Debugging ADB on Android Devices.....	73
8.2.1 macOS ADB Configuration	74
8.2.2 Windows ADB Configuration.....	75
8.2.3 Linux adb Configuration.....	76
8.3 Resolving USB Connection Issues	76
8.4 Enabling Wireless Debugging on Android Devices	77
8.5 Testing the adb Connection	79
8.6 Device Mirroring.....	79
8.7 Summary	79
9. The Basics of the Android Studio Code Editor.....	81
9.1 The Android Studio Editor.....	81
9.2 Splitting the Editor Window	84
9.3 Code Completion	84
9.4 Statement Completion	86
9.5 Parameter Information	86
9.6 Parameter Name Hints	86
9.7 Code Generation	86
9.8 Code Folding.....	88
9.9 Quick Documentation Lookup	89
9.10 Code Reformatting.....	89
9.11 Finding Sample Code	90
9.12 Live Templates	90
9.13 Summary	91
10. An Overview of the Android Architecture	93
10.1 The Android software stack	93
10.2 The Linux kernel.....	94
10.3 Android runtime – ART	94
10.4 Android libraries	94
10.4.1 C/C++ libraries.....	94
10.5 Application framework.....	95
10.6 Applications	95
10.7 Summary	95
11. An Introduction to Kotlin.....	97
11.1 What is Kotlin?	97
11.2 Kotlin and Java.....	97
11.3 Converting from Java to Kotlin	97
11.4 Kotlin and Android Studio	98
11.5 Experimenting with Kotlin	98
11.6 Semi-colons in Kotlin	99

11.7 Summary	99
12. Kotlin Data Types, Variables and Nullability	101
12.1 Kotlin data types.....	101
12.1.1 Integer data types	102
12.1.2 Floating point data types.....	102
12.1.3 Boolean data type.....	102
12.1.4 Character data type.....	102
12.1.5 String data type.....	102
12.1.6 Escape sequences.....	103
12.2 Mutable variables	104
12.3 Immutable variables.....	104
12.4 Declaring mutable and immutable variables	104
12.5 Data types are objects	104
12.6 Type annotations and type inference.....	105
12.7 Nullable type	106
12.8 The safe call operator	106
12.9 Not-null assertion	107
12.10 Nullable types and the let function.....	107
12.11 Late initialization (lateinit)	108
12.12 The Elvis operator	109
12.13 Type casting and type checking.....	109
12.14 Summary.....	110
13. Kotlin Operators and Expressions	111
13.1 Expression syntax in Kotlin	111
13.2 The Basic assignment operator.....	111
13.3 Kotlin arithmetic operators.....	111
13.4 Augmented assignment operators	112
13.5 Increment and decrement operators	112
13.6 Equality operators	113
13.7 Boolean logical operators.....	113
13.8 Range operator	114
13.9 Bitwise operators	114
13.9.1 Bitwise inversion	114
13.9.2 Bitwise AND	115
13.9.3 Bitwise OR.....	115
13.9.4 Bitwise XOR.....	115
13.9.5 Bitwise left shift	116
13.9.6 Bitwise right shift	116
13.10 Summary.....	117
14. Kotlin Control Flow	119
14.1 Looping control flow.....	119
14.1.1 The Kotlin <i>for-in</i> Statement.....	119
14.1.2 The <i>while</i> loop	120
14.1.3 The <i>do ... while</i> loop	121
14.1.4 Breaking from Loops	121
14.1.5 The <i>continue</i> statement	122
14.1.6 Break and continue labels	122
14.2 Conditional control flow	123

14.2.1 Using the <i>if</i> expressions	123
14.2.2 Using <i>if ... else ...</i> expressions	124
14.2.3 Using <i>if ... else if ...</i> Expressions	124
14.2.4 Using the <i>when</i> statement	124
14.3 Summary	125
15. An Overview of Kotlin Functions and Lambdas	127
15.1 What is a function?	127
15.2 How to declare a Kotlin function.....	127
15.3 Calling a Kotlin function.....	128
15.4 Single expression functions.....	128
15.5 Local functions	128
15.6 Handling return values.....	129
15.7 Declaring default function parameters	129
15.8 Variable number of function parameters	129
15.9 Lambda expressions.....	130
15.10 Higher-order functions	131
15.11 Summary	132
16. The Basics of Object-Oriented Programming in Kotlin.....	133
16.1 What is an object?	133
16.2 What is a class?	133
16.3 Declaring a Kotlin class.....	133
16.4 Adding properties to a class.....	134
16.5 Defining methods.....	134
16.6 Declaring and initializing a class instance	134
16.7 Primary and secondary constructors	134
16.8 Initializer blocks	137
16.9 Calling methods and accessing properties.....	137
16.10 Custom accessors	137
16.11 Nested and inner classes.....	138
16.12 Companion objects.....	139
16.13 Summary	141
17. An Introduction to Kotlin Inheritance and Subclassing.....	143
17.1 Inheritance, classes, and subclasses	143
17.2 Subclassing syntax.....	143
17.3 A Kotlin inheritance example.....	144
17.4 Extending the functionality of a subclass.....	145
17.5 Overriding inherited methods	146
17.6 Adding a custom secondary constructor	147
17.7 Using the SavingsAccount class.....	147
17.8 Summary	147
18. An Overview of Compose	149
18.1 Development before Compose.....	149
18.2 Compose declarative syntax	149
18.3 Compose is data-driven	150
18.4 Summary	150
19. Composable Functions Overview	151

Table of Contents

19.1 What is a composable function?	151
19.2 Stateful vs. stateless composables	151
19.3 Composable function syntax	152
19.4 Foundation and Material composables	154
19.5 Summary	155
20. An Overview of Compose State and Recomposition.....	157
20.1 The basics of state	157
20.2 Introducing recomposition	157
20.3 Creating the StateExample project.....	158
20.4 Declaring state in a composable.....	158
20.5 Unidirectional data flow.....	161
20.6 State hoisting.....	163
20.7 Saving state through configuration changes.....	165
20.8 Summary	166
21. An Introduction to Composition Local.....	169
21.1 Understanding CompositionLocal	169
21.2 Using CompositionLocal	170
21.3 Creating the CompLocalDemo project.....	171
21.4 Designing the layout.....	171
21.5 Adding the CompositionLocal state	172
21.6 Accessing the CompositionLocal state.....	173
21.7 Testing the design.....	173
21.8 Summary	176
22. An Overview of Compose Slot APIs	177
22.1 Understanding slot APIs	177
22.2 Declaring a slot API.....	178
22.3 Calling slot API composables.....	178
22.4 Summary	180
23. A Compose Slot API Tutorial.....	181
23.1 About the project.....	181
23.2 Creating the SlotApiDemo project	181
23.3 Preparing the MainActivity class file	181
23.4 Creating the MainScreen composable.....	182
23.5 Adding the ScreenContent composable	183
23.6 Creating the Checkbox composable	184
23.7 Implementing the ScreenContent slot API.....	185
23.8 Adding an Image drawable resource	186
23.9 Coding the TitleImage composable.....	187
23.10 Completing the MainScreen composable.....	188
23.11 Previewing the project.....	190
23.12 Summary.....	191
24. Using Modifiers in Compose.....	193
24.1 An overview of modifiers.....	193
24.2 Creating the ModifierDemo project.....	193
24.3 Creating a modifier	194
24.4 Modifier ordering.....	196

24.5 Adding modifier support to a composable	196
24.6 Common built-in modifiers	200
24.7 Combining modifiers.....	200
24.8 Summary	201
25. Annotated Strings and Brush Styles.....	203
25.1 What are annotated strings?	203
25.2 Using annotated strings.....	203
25.3 Brush Text Styling	204
25.4 Creating the example project.....	205
25.5 An example SpanStyle annotated string.....	205
25.6 An example ParagraphStyle annotated string	206
25.7 A Brush style example	209
25.8 Summary	210
26. Composing Layouts with Row and Column	211
26.1 Creating the RowColDemo project	211
26.2 Row composable.....	212
26.3 Column composable.....	212
26.4 Combining Row and Column composables.....	213
26.5 Layout alignment	214
26.6 Layout arrangement positioning.....	216
26.7 Layout arrangement spacing.....	218
26.8 Row and Column scope modifiers.....	219
26.9 Scope modifier weights	223
26.10 Summary	223
27. Box Layouts in Compose.....	225
27.1 An introduction to the Box composable.....	225
27.2 Creating the BoxLayout project	225
27.3 Adding the TextCell composable	225
27.4 Adding a Box layout.....	226
27.5 Box alignment.....	227
27.6 BoxScope modifiers	229
27.7 Using the clip() modifier	229
27.8 Summary	231
28. An Introduction to FlowRow and FlowColumn.....	233
28.1 FlowColumn and FlowRow	233
28.2 Maximum number of items	234
28.3 Working with main axis arrangement.....	234
28.4 Understanding cross-axis arrangement	235
28.5 Item alignment	236
28.6 Controlling item size.....	236
28.7 Summary	238
29. A FlowRow and FlowColumn Tutorial.....	239
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
30. Custom Layout Modifiers.....	247
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
31. Building Custom Layouts.....	255
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
32. A Guide to ConstraintLayout in Compose.....	261
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
33. Working with ConstraintLayout in Compose	269
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
34. Working with IntrinsicSize in Compose.....	287
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
35. Coroutines and LaunchedEffects in Jetpack Compose.....	293
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
36. An Overview of Lists and Grids in Compose	301
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
37. A Compose Row and Column List Tutorial	311
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
38. A Compose Lazy List Tutorial	317
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
39. Lazy List Sticky Headers and Scroll Detection	327
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
40. A Compose Lazy Staggered Grid Tutorial	335
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
41. VerticalPager and HorizontalPager in Compose	343
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
42. Compose Visibility Animation	355
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
43. Compose State-Driven Animation.....	367
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
44. Canvas Graphics Drawing in Compose	383
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
45. Working with ViewModels in Compose	403
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
46. A Compose ViewModel Tutorial.....	409
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
47.	An Overview of Android SQLite Databases	419
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
48.	Room Databases and Compose	425
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
49.	A Compose Room Database and Repository Tutorial	435
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	451

50. An Overview of Navigation in Compose	453
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
51. A Compose Navigation Tutorial	463
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
52. A Compose Navigation Bar Tutorial.....	471
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
53. Detecting Gestures in Compose.....	481
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
54. An Introduction to Kotlin Flow	495
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
55.	A Jetpack Compose SharedFlow Tutorial	511
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
56.	An Android Biometric Authentication Tutorial.....	519
56.1	An overview of biometric authentication	519
56.2	Creating the biometric authentication project	519
56.3	Adding the biometric dependency	520
56.4	Configuring device fingerprint authentication	520
56.5	Adding the biometric permissions to the manifest file	521
56.6	Checking the security settings.....	521
56.7	Designing the user interface	522
56.8	Configuring the authentication callbacks	523
56.9	Starting the biometric prompt.....	524
56.10	Testing the project.....	524
56.11	Summary	525
57.	Working with the Google Maps Android API in Android Studio	527
57.1	The elements of the Google Maps Android API	527
57.2	Creating the Google Maps project	528
57.3	Creating a Google Cloud billing account.....	528
57.4	Creating a new Google Cloud project	529
57.5	Enabling the Google Maps SDK.....	530
57.6	Generating a Google Maps API key	531
57.7	Adding the API key to the Android Studio project.....	531
57.8	Adding the compose map dependency	532
57.9	Creating a map.....	532
57.10	Testing the application.....	532

57.11 Understanding geocoding and reverse geocoding	533
57.12 Specifying a map location	534
57.13 Changing the map type	536
57.14 Displaying map controls to the user	537
57.15 Handling map gesture interaction	539
57.15.1 Map zooming gestures.....	539
57.15.2 Map scrolling/panning gestures.....	539
57.15.3 Map tilt gestures	539
57.15.4 Map rotation gestures	540
57.16 Creating map markers.....	540
57.17 Controlling the map camera.....	541
57.18 Summary.....	543
58. Creating, Testing, and Uploading an Android App Bundle.....	545
58.1 The Release Preparation Process.....	545
58.2 Android App Bundles.....	545
58.3 Register for a Google Play Developer Console Account.....	546
58.4 Configuring the App in the Console	547
58.5 Enabling Google Play App Signing.....	548
58.6 Creating a Keystore File	548
58.7 Creating the Android App Bundle.....	549
58.8 Generating Test APK Files	551
58.9 Uploading the App Bundle to the Google Play Developer Console.....	552
58.10 Exploring the App Bundle	553
58.11 Managing Testers	554
58.12 Rolling the App Out for Testing.....	554
58.13 Uploading New App Bundle Revisions.....	555
58.14 Analyzing the App Bundle File	556
58.15 Summary.....	557
59. An Overview of Android In-App Billing	559
59.1 Preparing a project for In-App purchasing.....	559
59.2 Creating In-App products and subscriptions.....	559
59.3 Billing client initialization.....	560
59.4 Connecting to the Google Play Billing library	561
59.5 Querying available products.....	561
59.6 Starting the purchase process	562
59.7 Completing the purchase	562
59.8 Querying previous purchases	563
59.9 Summary.....	564
60. An Android In-App Purchasing Tutorial	565
60.1 About the In-App purchasing example project.....	565
60.2 Creating the InAppPurchase project	565
60.3 Adding libraries to the project.....	565
60.4 Adding the App to the Google Play Store	566
60.5 Creating an In-App product	566
60.6 Enabling license testers.....	567
60.7 Creating a purchase helper class	568
60.8 Adding the StateFlow streams	569
60.9 Initializing the billing client.....	569

Table of Contents

60.10 Querying the product.....	570
60.11 Handling purchase updates	571
60.12 Launching the purchase flow.....	571
60.13 Consuming the product	572
60.14 Restoring a previous purchase.....	572
60.15 Completing the MainActivity.....	573
60.16 Testing the app.....	575
60.17 Troubleshooting	577
60.18 Summary.....	578
61. Working with Compose Theming	579
61.1 Material Design 2 vs. Material Design 3	579
61.2 Material Design 3 theming	579
61.3 Building a custom theme	583
61.4 Summary	584
62. A Material Design 3 Theming Tutorial	585
62.1 Creating the ThemeDemo project	585
62.2 Designing the user interface	585
62.3 Building a new theme	587
62.4 Adding the theme to the project	588
62.5 Enabling dynamic colors.....	589
62.6 Summary	590
63. An Overview of Gradle in Android Studio.....	591
63.1 An overview of Gradle.....	591
63.2 Gradle and Android Studio	591
63.2.1 Sensible defaults	591
63.2.2 Dependencies.....	591
63.2.3 Build variants.....	592
63.2.4 Manifest entries.....	592
63.2.5 APK signing.....	592
63.2.6 ProGuard support	592
63.3 The Property and Settings Gradle build file.....	592
63.4 The top-level Gradle build file	593
63.5 Module level Gradle build files.....	594
63.6 Configuring signing settings in the build File.....	597
63.7 Running Gradle tasks from the command line.....	598
63.8 Summary	599
Index	601

1. Start Here

This book teaches you how to build Android applications using Jetpack Compose 1.5, Android Studio Hedgehog (2023.1.1), 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 and key user interface design concepts such as modifiers, navigation bars, and user interface navigation. Additional chapters explore building your own reusable custom layout components, securing your apps with Biometric authentication, and integrating Google Maps.

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. You will also learn to generate extra revenue from your app using in-app billing.

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.payloadbooks.com/product/compose15/>

Start Here

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

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 *info@payloadbooks.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.payloadbooks.com/compose15_errata

If you find an error not listed in the errata, email our technical support team at *info@payloadbooks.com*.

2. Setting up an Android Studio Development Environment

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

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

2.1 System requirements

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

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

2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio Hedgehog 2023.1.1 using the Android API 34 SDK (UpsideDownCake), which, at the time of writing, are the latest stable releases.

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

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

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

If this page provides instructions for downloading a newer version of Android Studio, there may be differences between this book and the software. A web search for “Android Studio Hedgehog” 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 Hedgehog 2023.1.1 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:

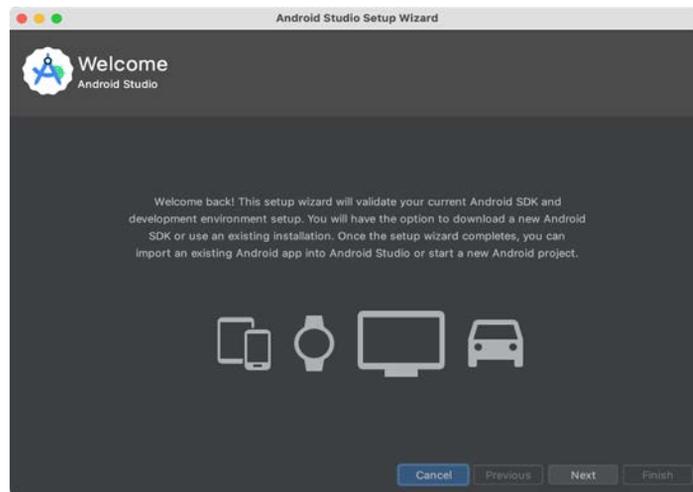


Figure 2-2

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.

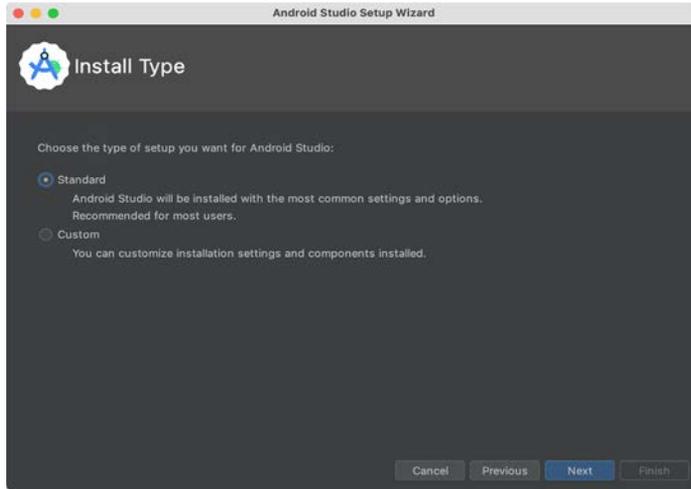


Figure 2-3

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:

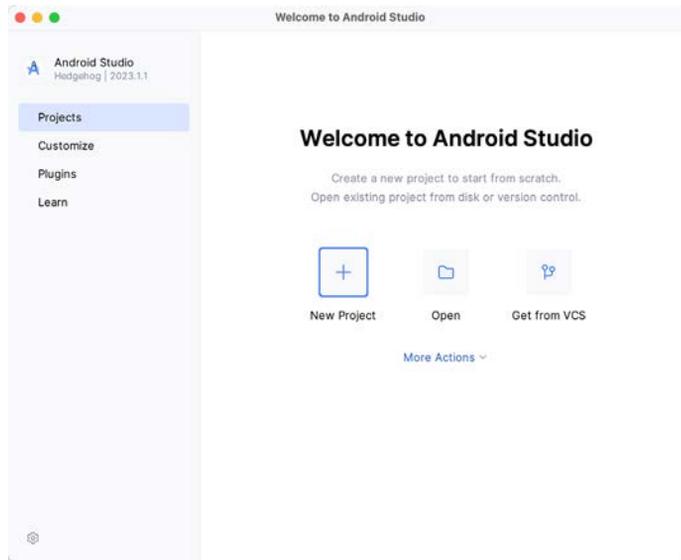


Figure 2-4

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:

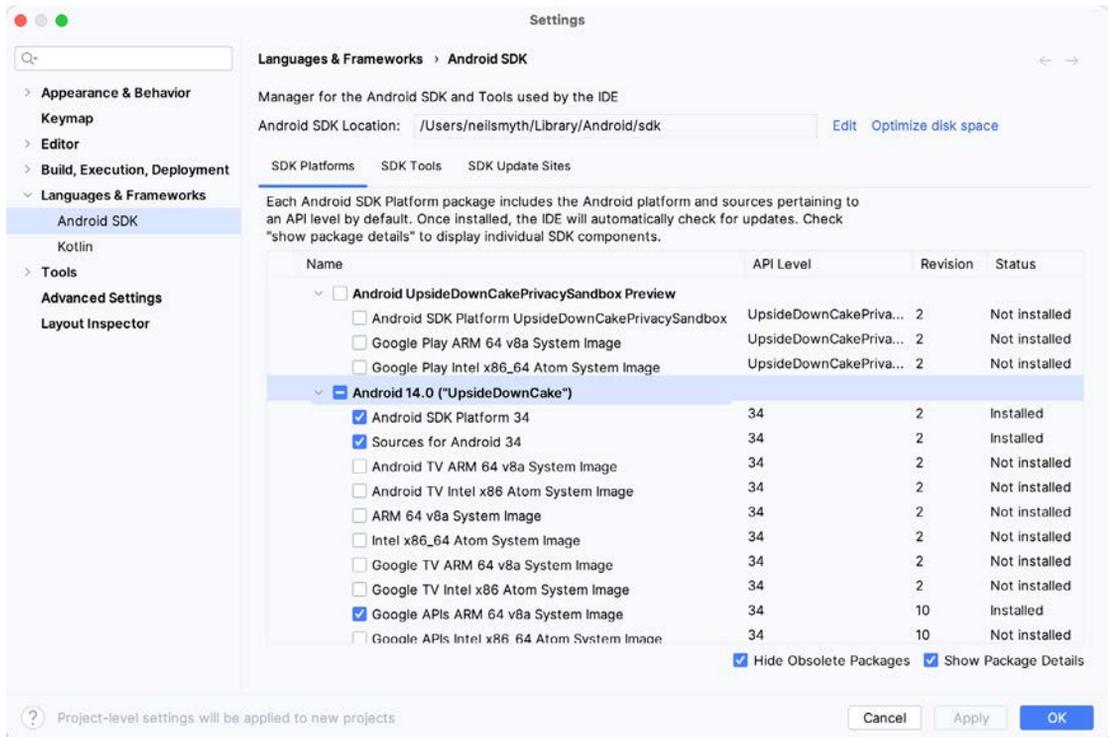


Figure 2-5

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 Hedgehog, this is Android UpsideDownCake (API Level 34). This information can be confirmed using the following link:

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

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

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

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

Setting up an Android Studio Development Environment

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

Figure 2-6

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

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



Figure 2-7

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

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

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

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



Figure 2-8

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

2.6 Installing the Android SDK Command-line Tools

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

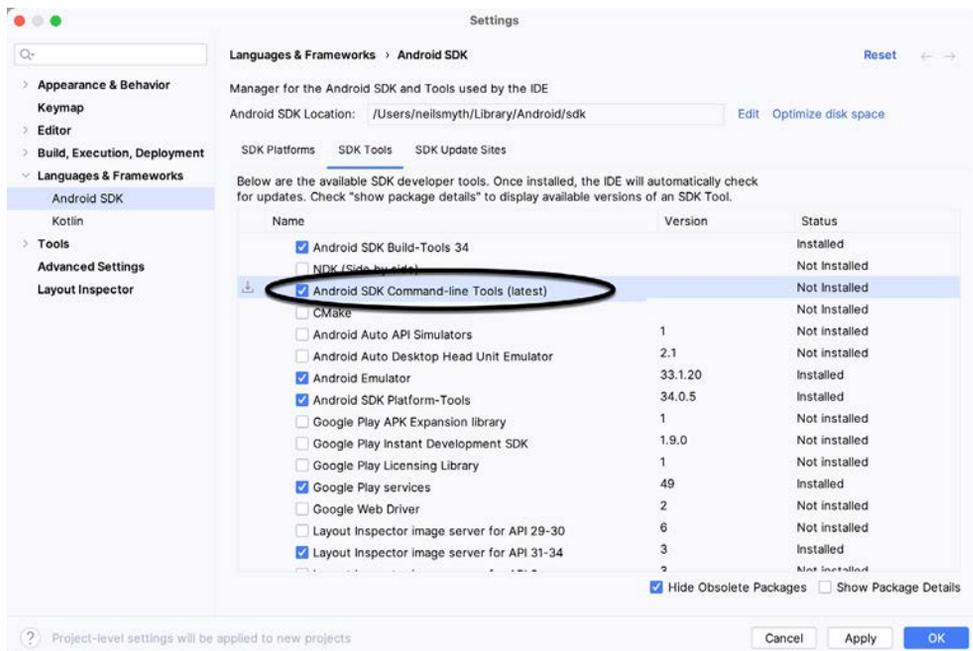


Figure 2-9

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

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

Setting up an Android Studio Development Environment

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

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

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



Figure 2-10

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

2.6.1 Windows 8.1

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

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

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

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

```
echo %Path%
```

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

```
adb
```

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

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

```
avdmanager
```

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

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

2.6.2 Windows 10

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

2.6.3 Windows 11

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

2.6.4 Linux

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

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

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

2.6.5 macOS

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

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

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

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

2.7 Android Studio memory management

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

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

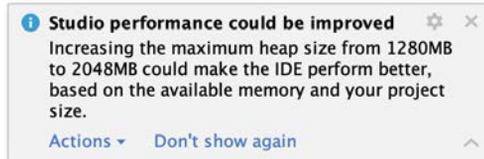


Figure 2-11

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

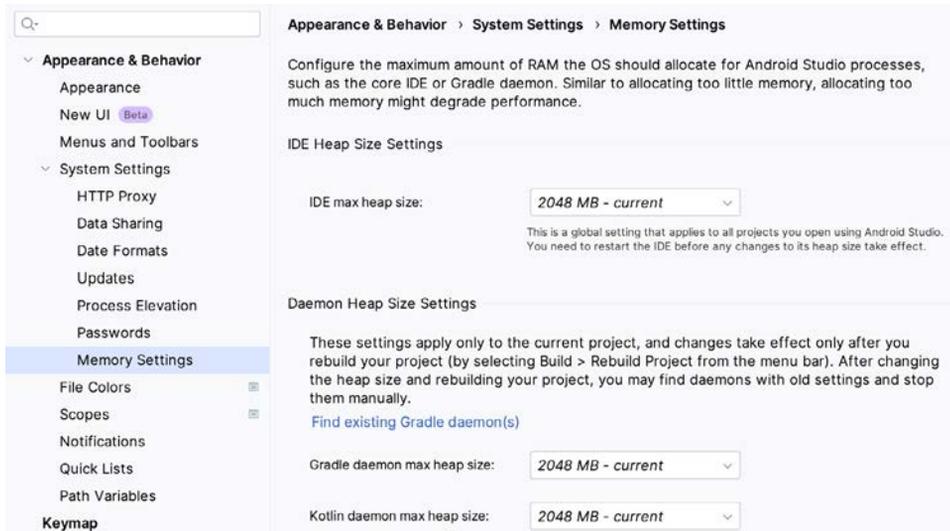


Figure 2-12

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

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

2.8 Updating Android Studio and the SDK

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

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

2.9 Summary

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

3. 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:

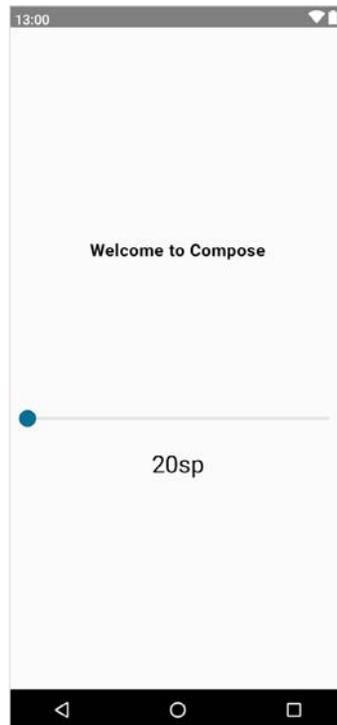


Figure 3-1

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:

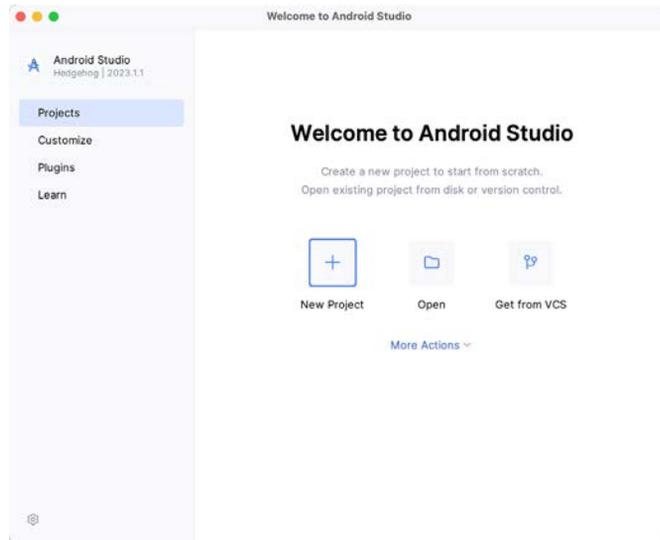


Figure 3-2

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:

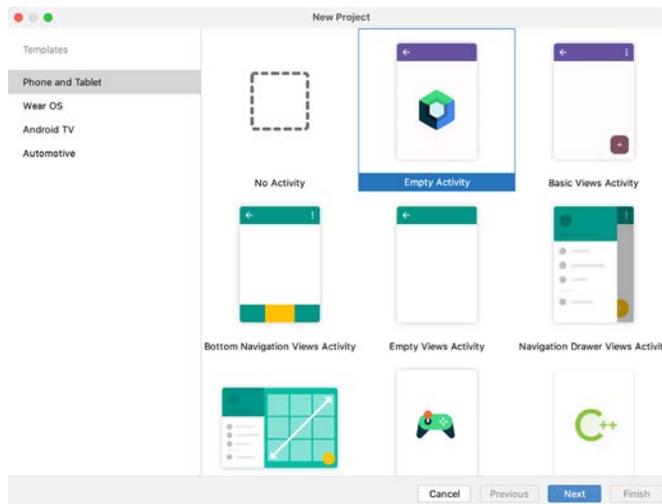


Figure 3-3

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:

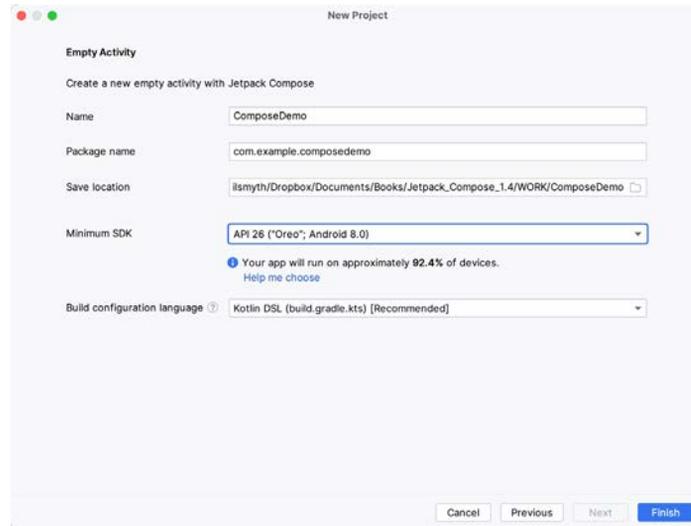


Figure 3-4

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:

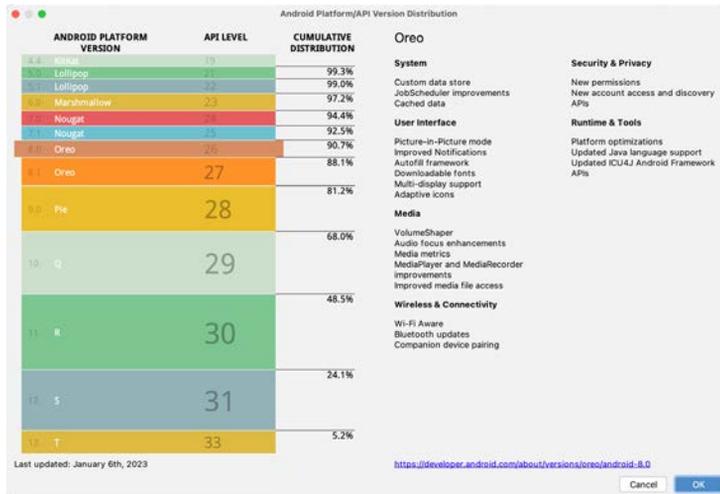


Figure 3-5

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:

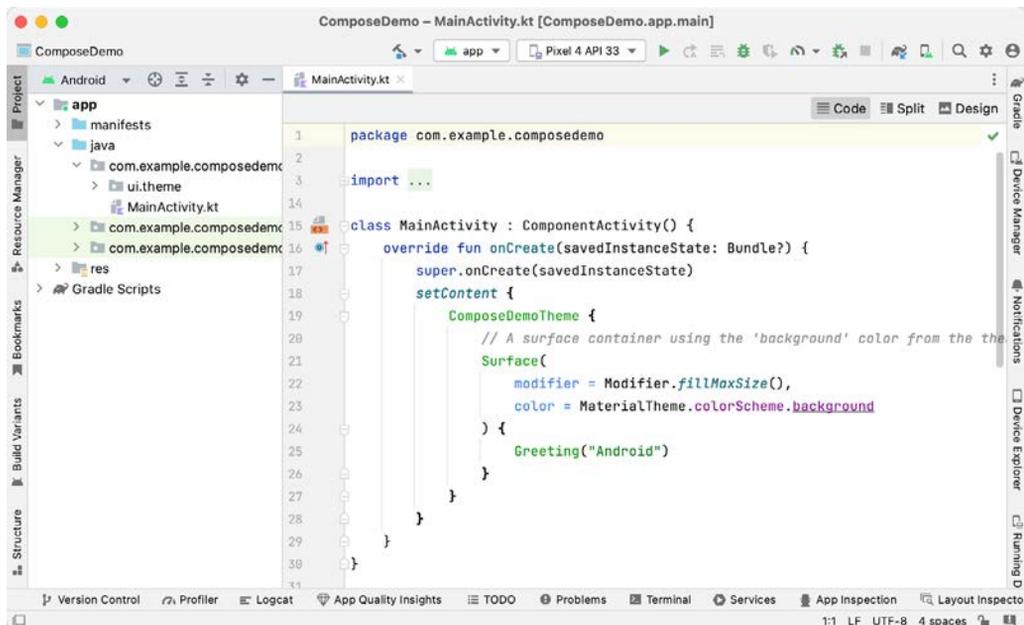


Figure 3-6

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:

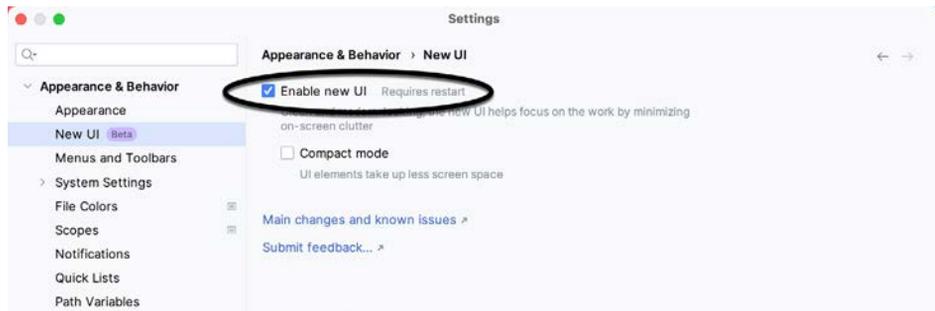


Figure 3-7

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:

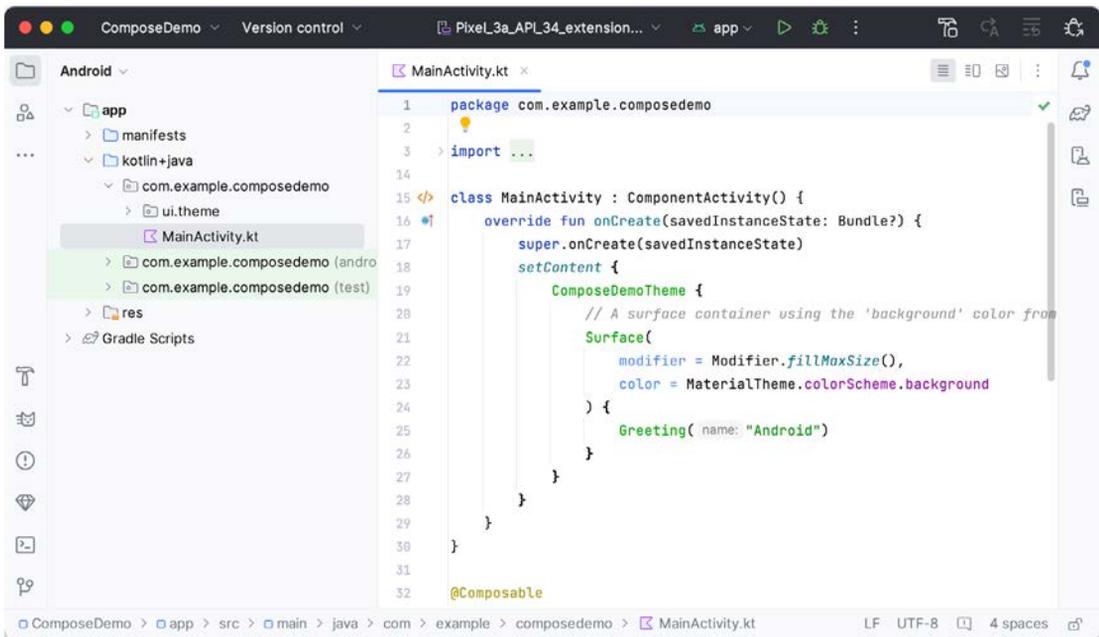


Figure 3-8

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:

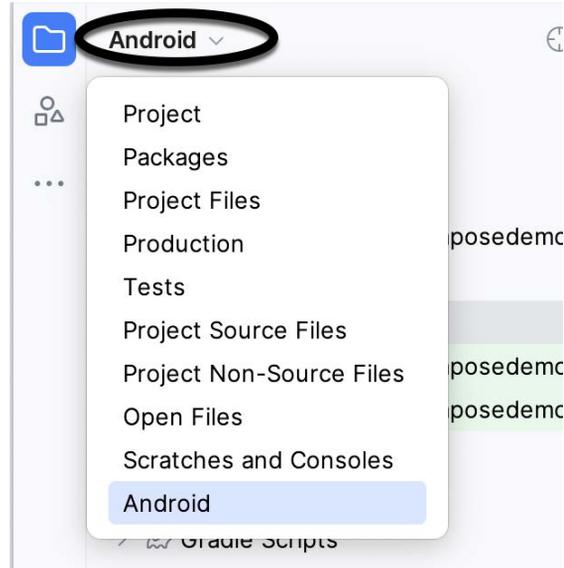


Figure 3-9

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:

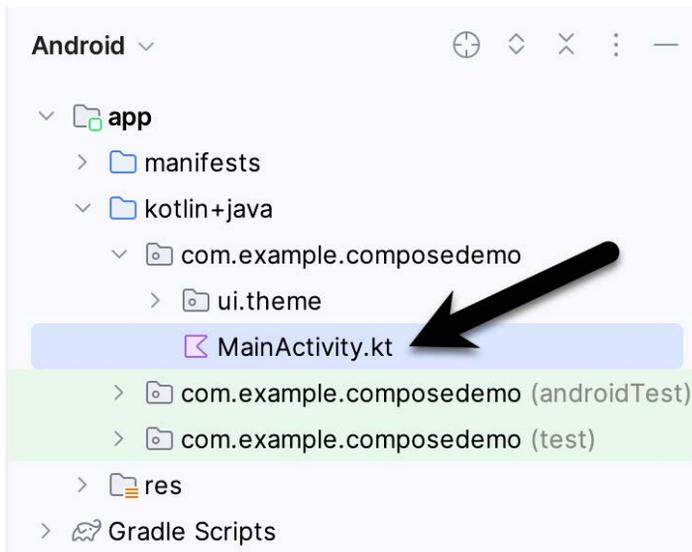


Figure 3-10

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 shown in Figure 3-8 above). Code mode is selected by clicking the button A in the figure below. However, the most helpful option when working with Compose is Split mode. To switch to Split mode, click on the button marked B:

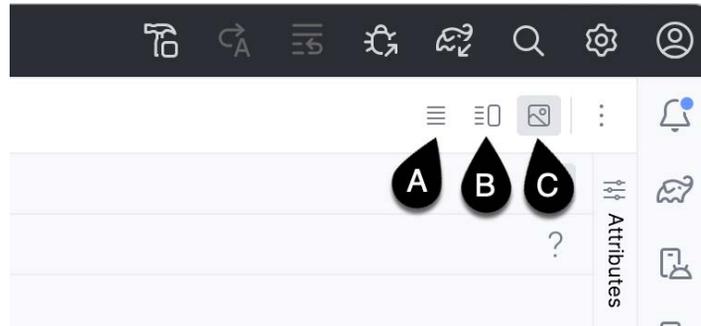


Figure 3-11

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

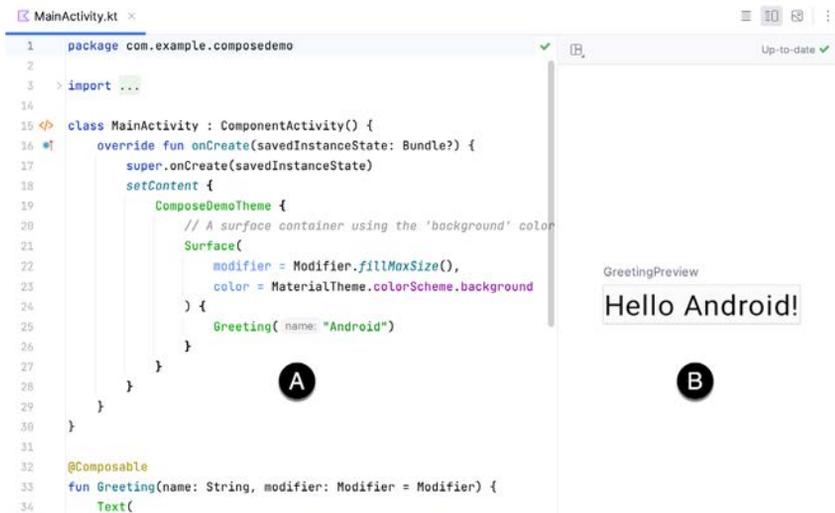


Figure 3-12

Only the Preview panel is displayed when the editor is in Design mode (button C).

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 3-13:

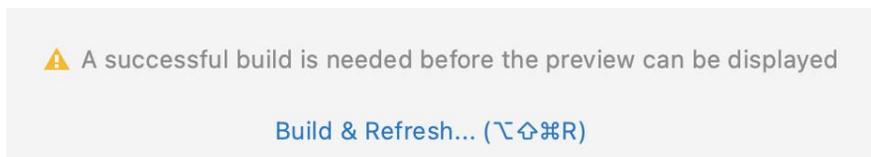


Figure 3-13

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:

GreetingPreview



Figure 3-14

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 3-15 below:

```

1 package com.example.composedemo
2
3 import ...
4
14 class MainActivity : ComponentActivity() {
15

```

Figure 3-15

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(
        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

A Compose Project Overview

(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 3-16:

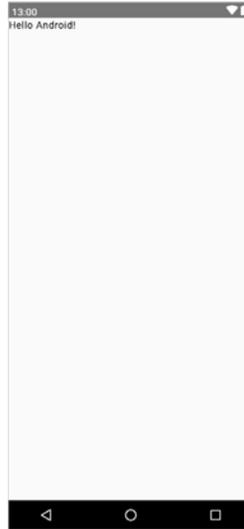


Figure 3-16

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 3-17):



Figure 3-17

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 3-18:



Figure 3-18

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:

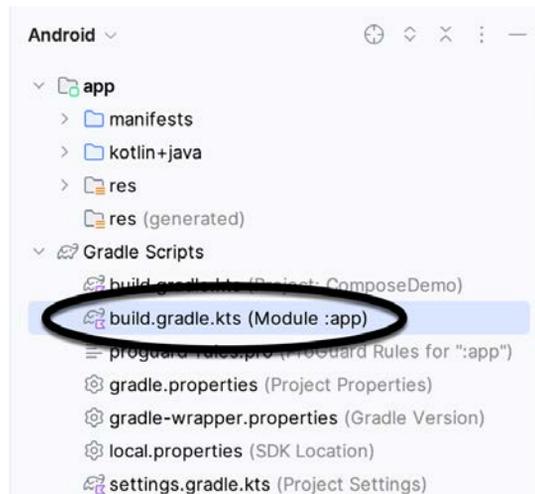


Figure 3-19

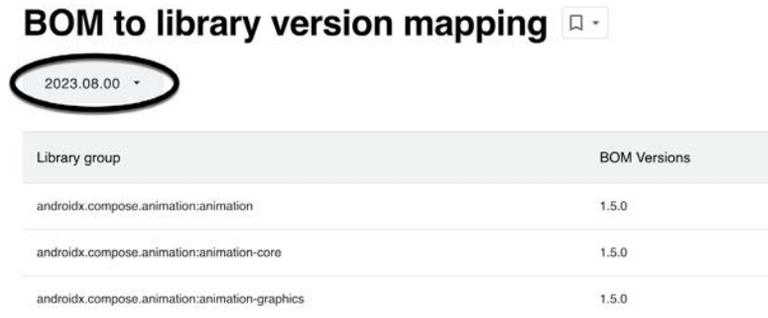
With the file loaded into the editor, locate the *compose-bom* entry in the dependencies section:

```
dependencies {
    .
    .
    implementation(platform("androidx.compose:compose-bom:2023.08.00"))
    .
    .
}
```

In the above example, we can see that the project is using BOM 2023.08.00. 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 3-20 below. For example, the figure shows that BOM 2023.08.00 uses version 1.5.0 of the Compose libraries:



Library group	BOM Versions
androidx.compose.animation:animation	1.5.0
androidx.compose.animation:animation-core	1.5.0
androidx.compose.animation:animation-graphics	1.5.0

Figure 3-20

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:1.12.0'
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.6.2'
    .
    .
}
```

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 the 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:

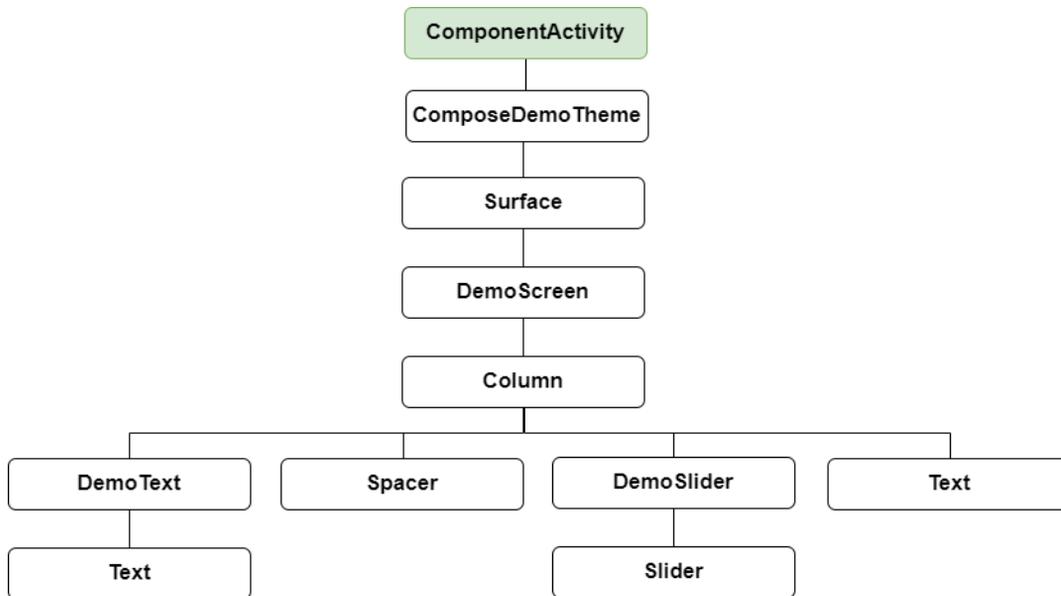


Figure 4-1

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:



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:

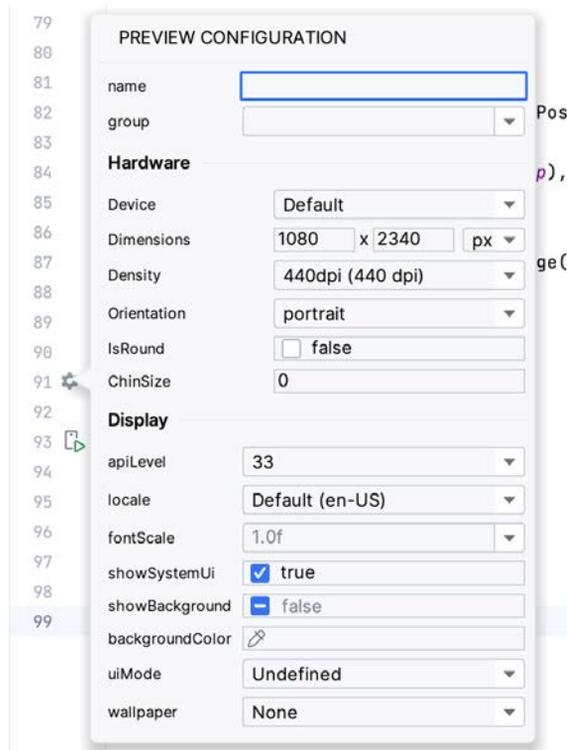


Figure 4-5

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:

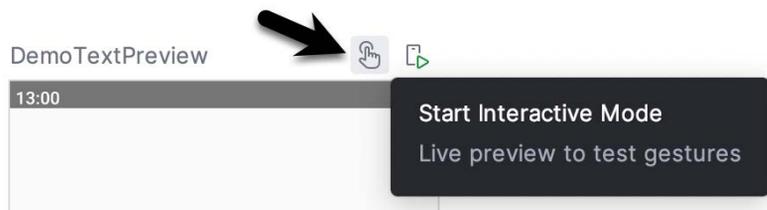


Figure 4-6

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:

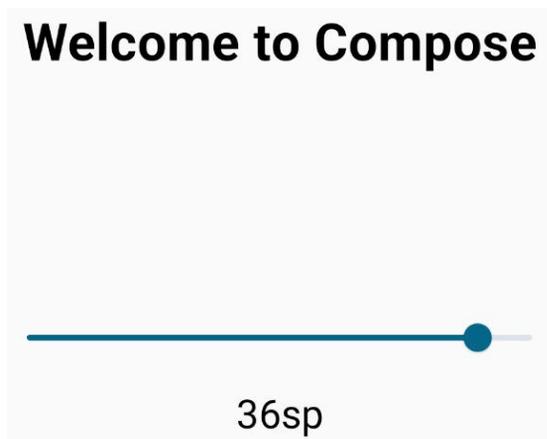


Figure 4-7

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

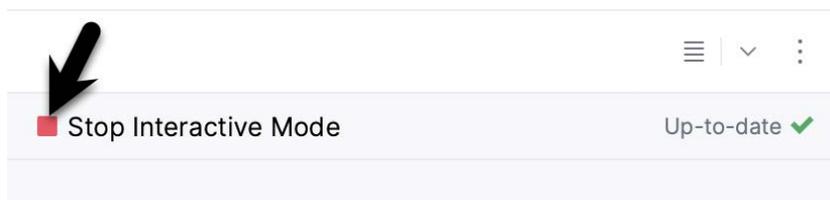


Figure 4-8

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.

5. Creating an Android Virtual Device (AVD) in Android Studio

Although the Android Studio Preview panel allows us to see the layout we are designing, compiling and running an entire app will be necessary to thoroughly test that it works. An Android application may be tested by installing and running it on a physical device or in an Android Virtual Device (AVD) emulator environment. Before an AVD can be used, it must first be created and configured to match the specifications of a particular device model. In this chapter, we will work through creating such a virtual device using the Pixel 4 phone as a reference example.

5.1 About Android Virtual Devices

AVDs are emulators that allow Android applications to be tested without needing to install the application on a physical Android-based device. An AVD may be configured to emulate various hardware features, including screen size, memory capacity, and the presence or otherwise of features such as a camera, GPS navigation support, or an accelerometer. Several emulator templates are installed as part of the standard Android Studio installation, allowing AVDs to be configured for various devices. Custom configurations may be created to match any physical Android device by specifying properties such as processor type, memory capacity, and the size and pixel density of the screen.

An AVD session can appear as a separate window or embedded within the Android Studio window.

New AVDs are created and managed using the Android Virtual Device Manager, which may be used in command-line mode or with a more user-friendly graphical user interface. To create a new AVD, the first step is to launch the AVD Manager. This can be achieved from within the Android Studio environment by clicking the *Device Manager* button in the right-hand tool window bar, as indicated in Figure 5-1:

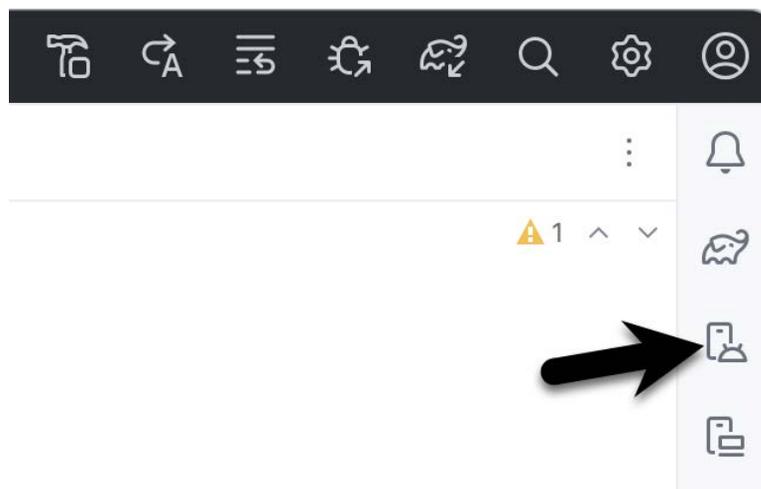


Figure 5-1

Once opened, the manager will appear as a tool window, as shown in Figure 5-2:

Creating an Android Virtual Device (AVD) in Android Studio

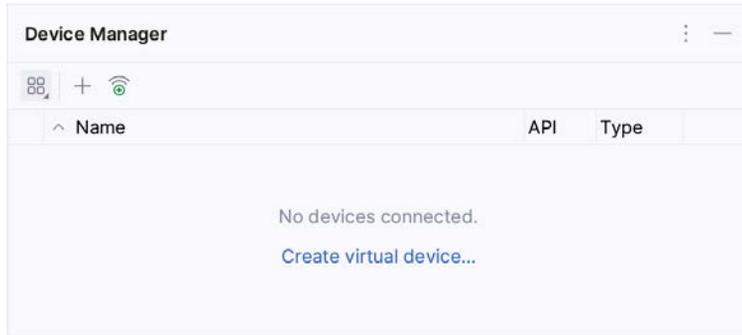


Figure 5-2

If you installed Android Studio for the first time on a computer (as opposed to upgrading an existing Android Studio installation), the installer might have created an initial AVD instance ready for use, as shown in Figure 5-3:



Figure 5-3

If this AVD is present on your system, you can use it to test apps. If no AVD was created, or you would like to create AVDs for different device types, follow the steps in the rest of this chapter.

To add a new AVD, begin by making sure that the Virtual tab is selected before clicking on the *Create device* button to open the *Virtual Device Configuration* dialog:

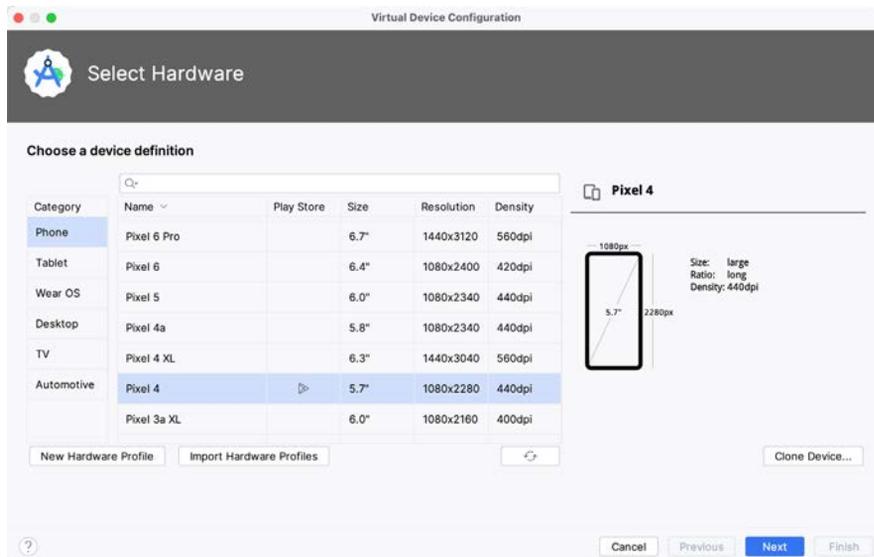


Figure 5-4

Within the dialog, perform the following steps to create a Pixel 4-compatible emulator:

1. Select the Phone option From the Category panel to display the available Android phone AVD templates.
2. Select the *Pixel 4* device option and click *Next*.
3. On the System Image screen, select the latest version of Android. If the system image has not yet been installed, a *Download* link will be provided next to the Release Name. Click this link to download and install the system image before selecting it. If the image you need is not listed, click on the *x86 Images* (or *ARM images* if you are running a Mac with Apple Silicon) and *Other images* tabs to view alternative lists.
4. Click *Next* to proceed and enter a descriptive name (for example, *Pixel 4 API 34*) into the name field or accept the default name.
5. Click *Finish* to create the AVD.
6. If future modifications to the AVD are necessary, re-open the Device Manager, select the AVD from the list, and click on the pencil icon in the Actions column to edit the settings.

5.2 Starting the Emulator

To test the newly created AVD emulator, select the emulator from the Device Manager and click the launch button (the triangle in the Actions column). The emulator will appear embedded into the main Android Studio window and begin the startup process. The amount of time it takes for the emulator to start will depend on the configuration of both the AVD and the system on which it is running:

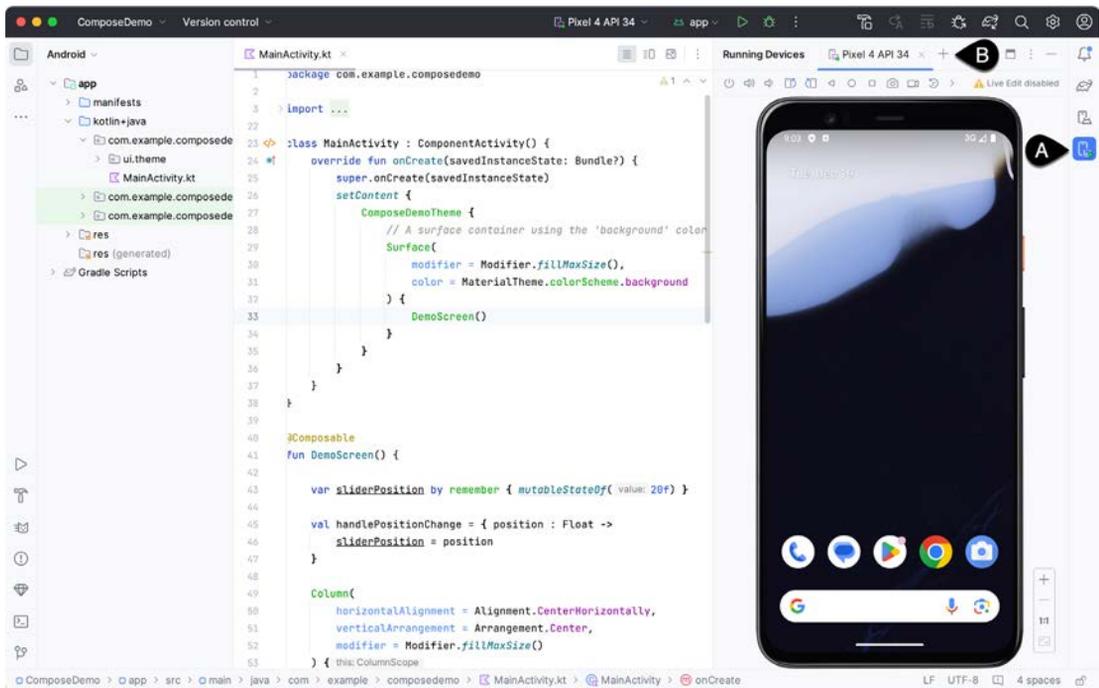


Figure 5-5

To hide and show the emulator tool window, click the Running Devices tool window button (marked A above). Click the “x” close button next to the tab (B) to exit the emulator. The emulator tool window can accommodate multiple emulator sessions, with each session represented by a tab. Figure 5-6, for example, shows a tool window with two emulator sessions:

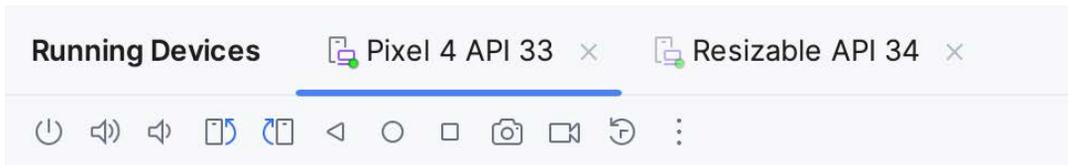


Figure 5-6

To switch between sessions, click on the corresponding tab.

Although the emulator probably defaulted to appearing in portrait orientation, this and other default options can be changed. Within the Device Manager, select the new Pixel 4 entry and click on the pencil icon in the *Actions* column of the device row. In the configuration screen, locate the *Startup orientation* section and change the orientation setting. Exit and restart the emulator session to see this change take effect. More details on the emulator are covered in the next chapter, “*Using and Configuring the Android Studio AVD Emulator*”.

To save time in the next section of this chapter, leave the emulator running before proceeding.

5.3 Running the Application in the AVD

With an AVD emulator configured, the example ComposeDemo application created in the earlier chapter can now be compiled and run. With the ComposeDemo project loaded into Android Studio, make sure that the newly created Pixel 4 AVD is displayed in the device menu (marked A in Figure 5-7 below), then either click the run button represented by a triangle (B), select the *Run -> Run 'app'* menu option or use the Ctrl-R keyboard shortcut:

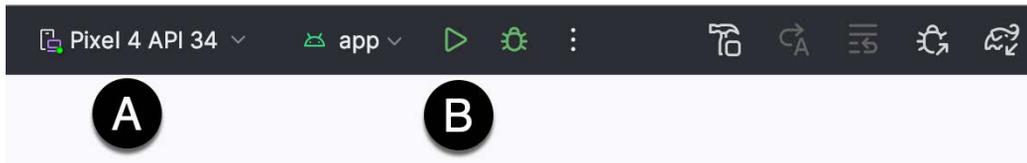


Figure 5-7

The device menu (A) may be used to select a different AVD instance or physical device as the run target and also to run the app on multiple devices. The menu also provides access to the Device Manager as well as device connection configuration and troubleshooting options:

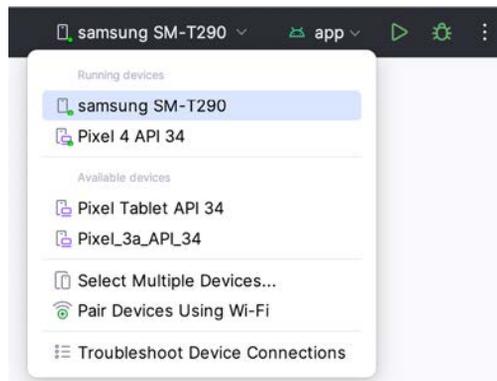


Figure 5-8

Once the application is installed and running, the user interface for the first fragment will appear within the emulator as shown in Figure 5-8:

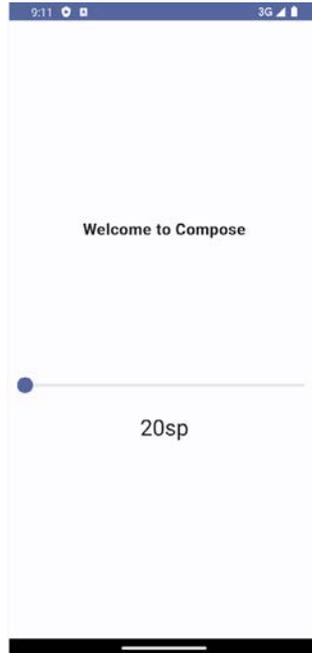


Figure 5-9

Once the run process begins, the Run tool window will appear. The Run tool window will display diagnostic information as the application package is installed and launched. Figure 5-10 shows the Run tool window output from a typical successful application launch:



Figure 5-10

If problems are encountered during the launch process, the Run tool window will provide information to help isolate the problem's cause.

Assuming the application loads into the emulator and runs as expected, we have safely verified that the Android development environment is correctly installed and configured. With the app running, try performing a currency conversion to verify that the app works as intended.

5.4 Real-time updates with Live Edit

With the app running, now is an excellent time to introduce the Live Edit feature. Like interactive mode in the Preview panel, Live Edit updates the appearance and behavior of the app running on the device or emulator as changes are made to the code. This feature allows code changes to be tested in real-time without building and re-running the project. When you launch your first app, the dialog shown in Figure 5-11 may appear providing you the opportunity to enable Live Edit mode:

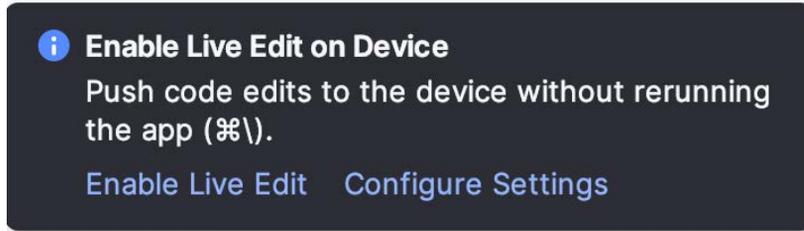


Figure 5-11

You can also enable Live Edit mode by clicking on the *IDE and Project Settings* button highlighted in Figure 5-12, followed by the Settings menu option:

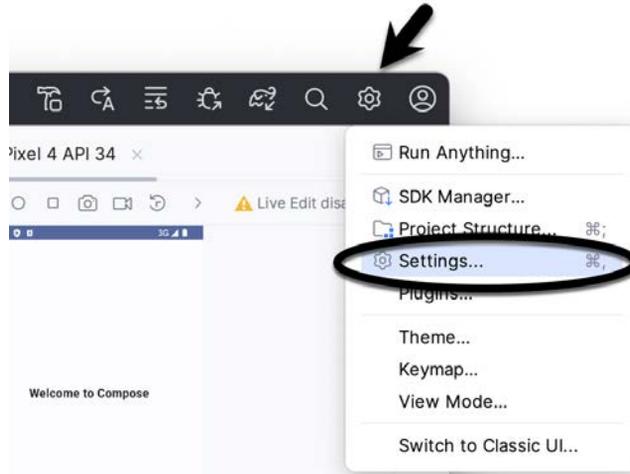


Figure 5-12

Within the side panel of the Settings dialog, navigate to and select the Editor -> Live Edit entry to display the following screen:

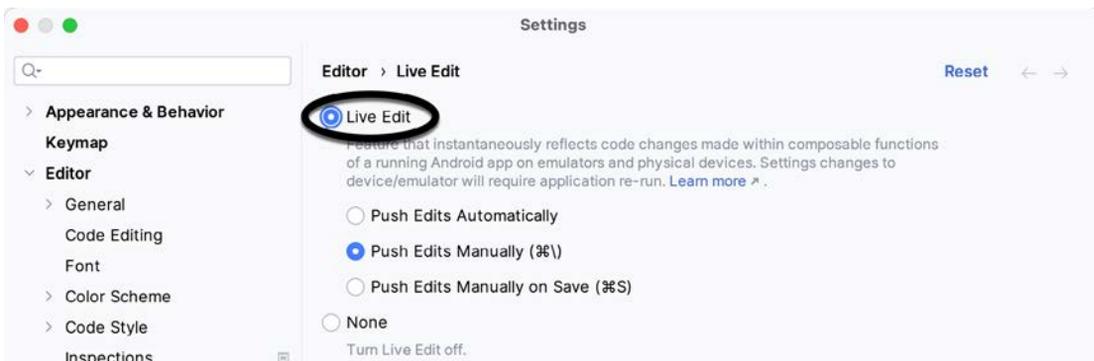


Figure 5-13

Enable the Live Edit option and choose the option to push edits to the running app automatically before clicking on OK. The responsiveness of automatic edit pushes will vary depending on the performance and resources of your computer and Android device. If performance is unacceptably slow, return to the settings screen and switch to pushing edits manually. Use the designated keyboard shortcut whenever you need to update the running app to reflect code changes.

Once you have enabled Live Edit mode, you must restart app before the change takes effect. You can do this from within the Running Devices tool window by clicking on the button marked A in Figure 5-14:

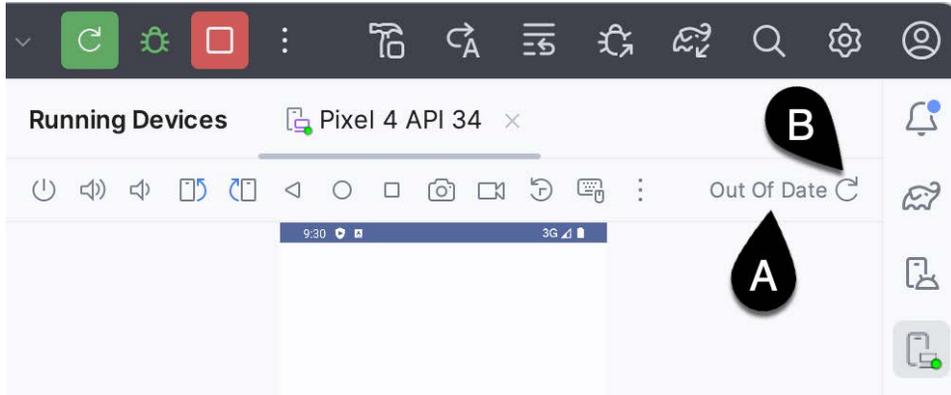


Figure 5-14

Try out Live Edit by changing the text displayed by the DemoText composible as follows:

```
DemoText(message = "This is Compose 1.5", fontSize = sliderPosition)
```

If automatic push edits mode is enabled, the text in the running app will update with each keystroke to reflect the change. For manual mode, changes can be pushed to the running app by clicking the refresh button marked B in Figure 5-14 above.

Live Edit is currently limited to changes made within the body of existing functions. It will not, for example, handle the addition, removal, or renaming of functions.

5.5 Running on Multiple Devices

The run target menu shown in Figure 5-8 above includes an option to run the app on multiple emulators and devices in parallel. When selected, this option displays the dialog in Figure 5-15, providing a list of the AVDs configured on the system and any attached physical devices. Enable the checkboxes next to the emulators or devices to be targeted before clicking on the Run button:

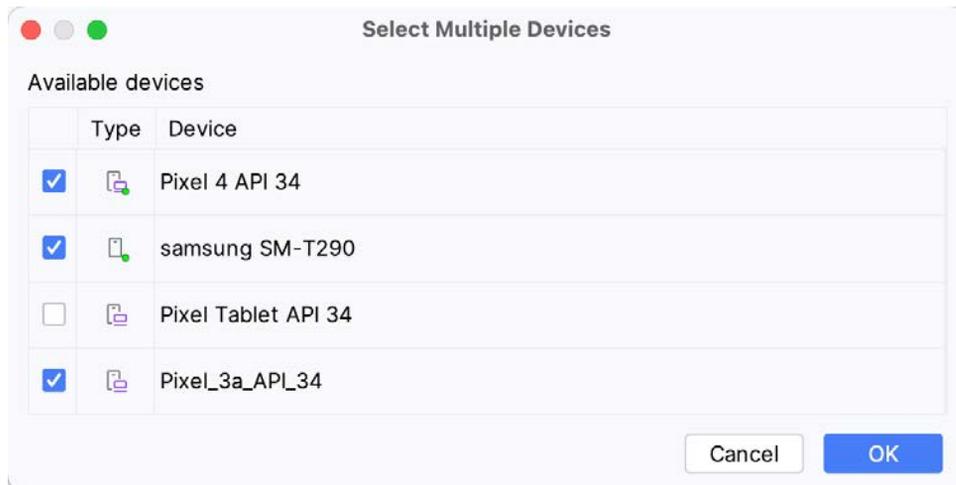


Figure 5-15

Creating an Android Virtual Device (AVD) in Android Studio

After clicking the Run button, Android Studio will launch the app on the selected emulators and devices.

5.6 Stopping a Running Application

To stop a running application, click the stop button located in the main toolbar, as shown in Figure 5-16:

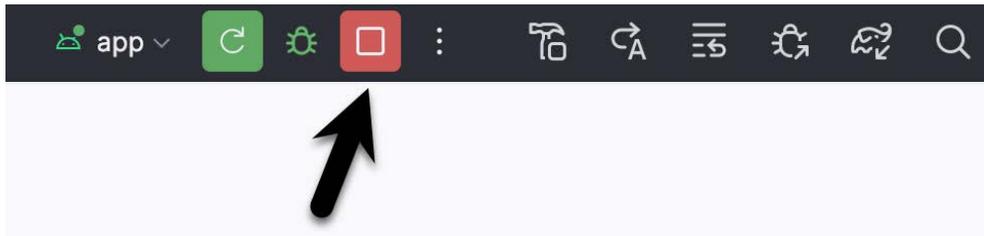


Figure 5-16

An app may also be terminated using the Run tool window. Begin by displaying the *Run* tool window using the window bar button that becomes available when the app is running. Once the Run tool window appears, click the stop button highlighted in Figure 5-17 below:



Figure 5-17

5.7 Supporting Dark Theme

To test how an app behaves when dark theme is enabled, open the Settings app within the running Android instance in the emulator, choose the *Display* category, and enable the *Dark theme* option as shown in Figure 5-18:

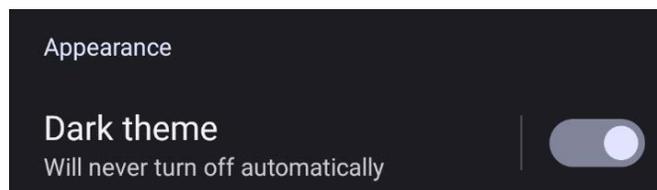


Figure 5-18

With dark theme enabled, run the ComposeDemo app and note that it appears as shown in Figure 5-19:

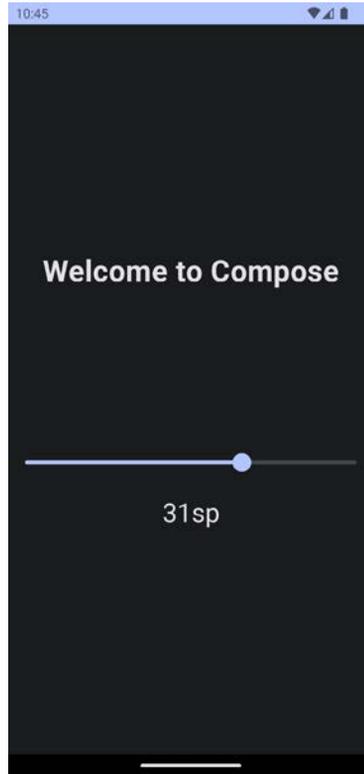


Figure 5-19

Return to the Settings app and turn off Dark theme mode before continuing.

5.8 Running the Emulator in a Separate Window

So far in this chapter, we have only used the emulator as a tool window embedded within the main Android Studio window. The emulator can be configured to appear in a separate window within the Settings dialog, which can be displayed by clicking on the IDE and Project Settings button located in the Android Studio toolbar, as highlighted in Figure 5-20:

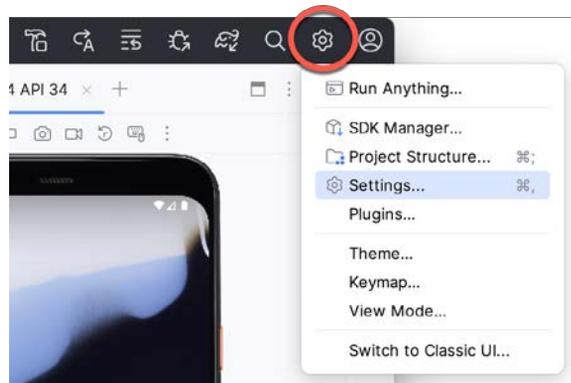


Figure 5-20

Within the Settings dialog, navigate to *Tools -> Emulator* in the side panel, and disable the *Launch in a tool*

Creating an Android Virtual Device (AVD) in Android Studio

window option:

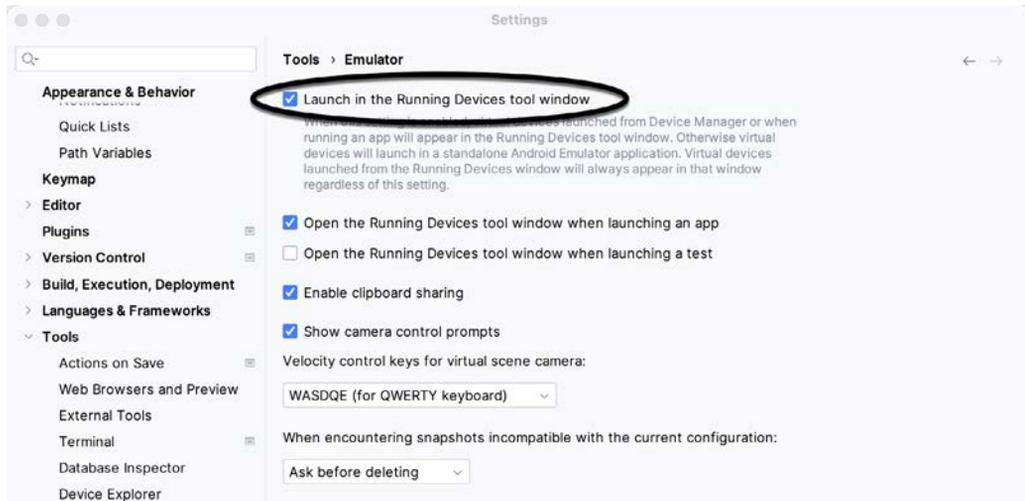


Figure 5-21

With the option disabled, click the Apply button followed by OK to commit the change, then exit the current emulator session by clicking on the close button on the tab marked B in Figure 5-5 above.

Run the sample app once again, at which point the emulator will appear as a separate window, as shown below:

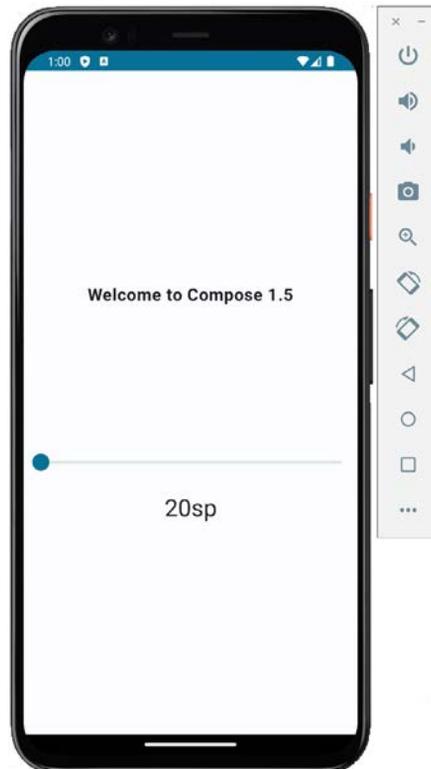


Figure 5-22

Creating an Android Virtual Device (AVD) in Android Studio

The choice of standalone or tool window mode is a matter of personal preference. If you prefer the emulator running in a tool window, return to the settings screen and re-enable the *Launch in a tool window* option. Before committing to standalone mode, however, keep in mind that the Running Devices tool window may also be detached from the main Android Studio window from within the tool window Options menu, which is accessed by clicking the button indicated in Figure 5-23:

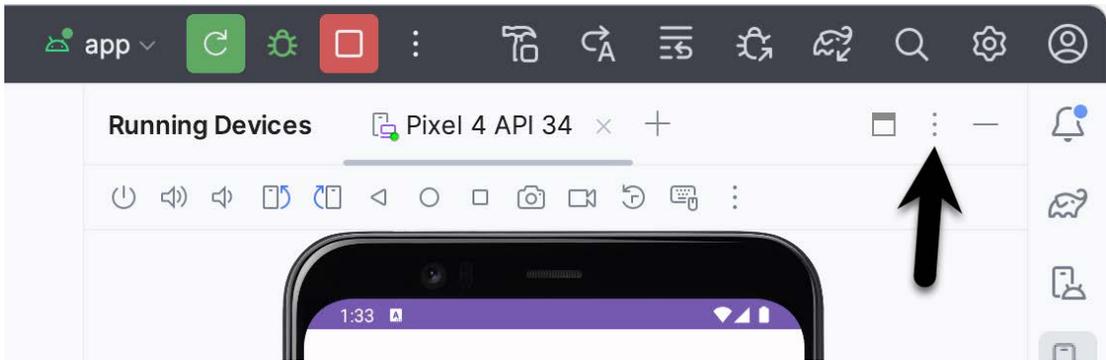


Figure 5-23

From within the Options menu, select *View Mode* -> *Float* to detach the tool window from the Android Studio main window:

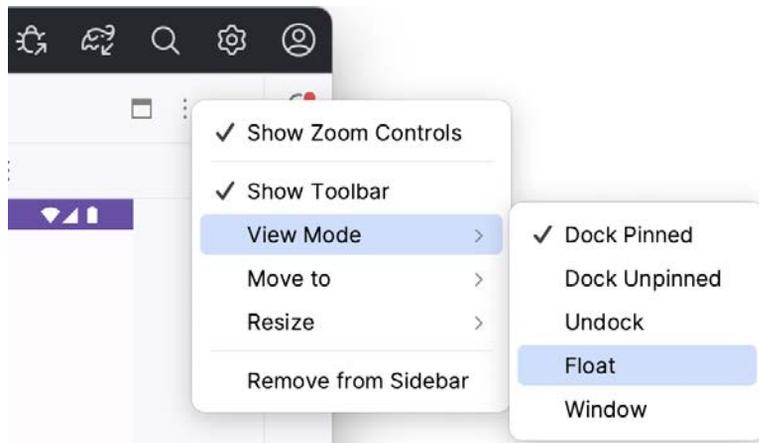


Figure 5-24

To re-dock the Running Devices tool window, click on the Dock button shown in Figure 5-25:

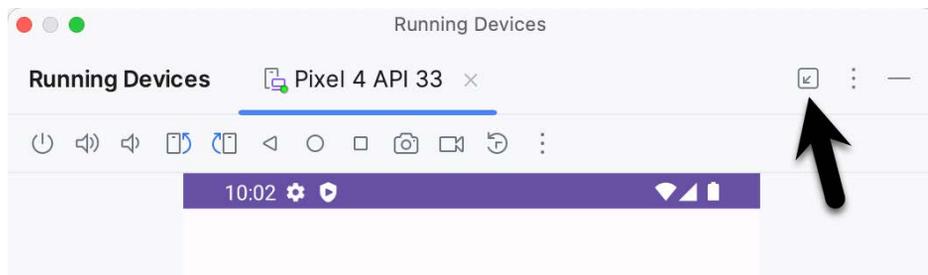


Figure 5-25

5.9 Removing the Device Frame

The emulator can be configured to appear with or without the device frame. To change the setting, exit the emulator, open the Device Manager, select the AVD from the list, and click on the menu button indicated by the arrow in Figure 5-26:

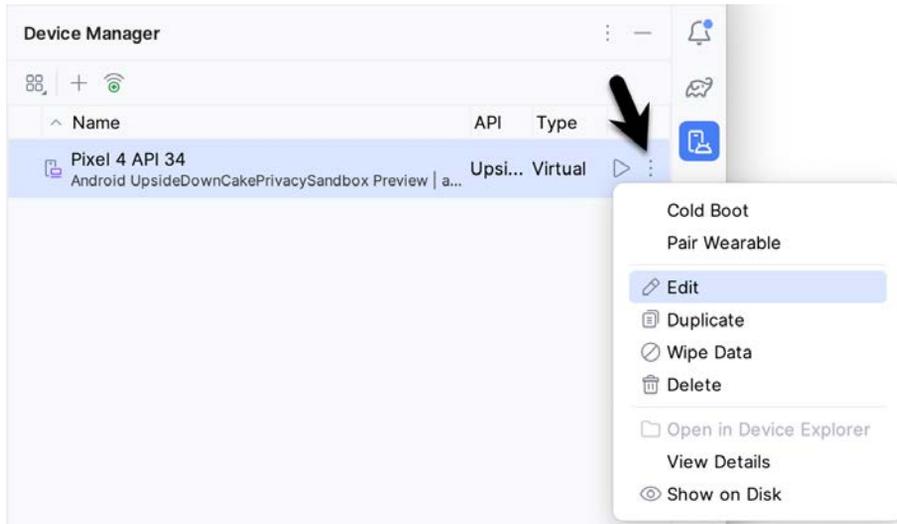


Figure 5-26

Select the Edit option and, in the settings screen, locate and switch off the Enable device frame option before clicking the Finish button:

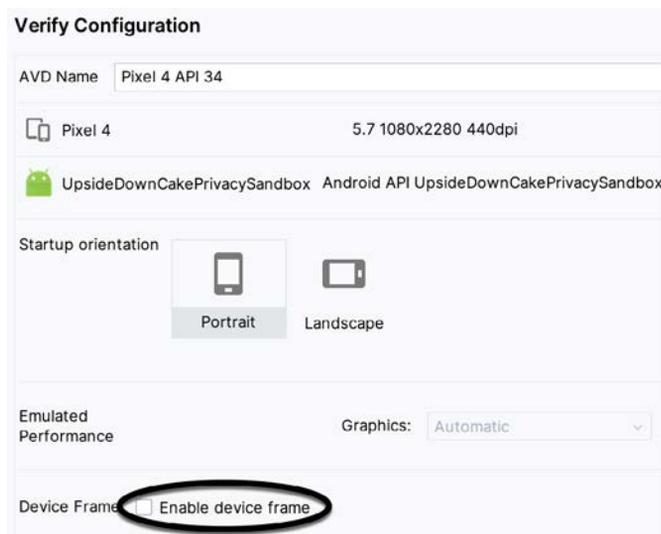


Figure 5-27

Once the device frame has been disabled, the emulator will appear as shown in Figure 5-28 the next time it is launched:

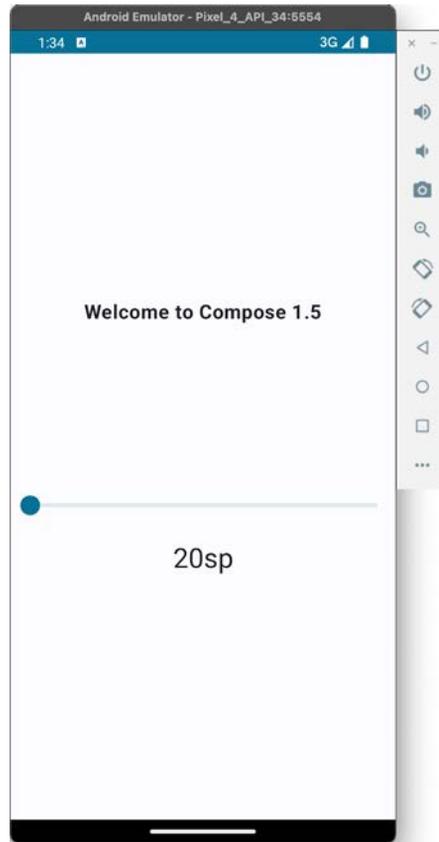


Figure 5-28

5.10 Summary

A typical application development process follows a cycle of coding, compiling, and running in a test environment. Android applications may be tested on a physical Android device or an Android Virtual Device (AVD) emulator. AVDs are created and managed using the Android Studio Device Manager tool, which may be used as a command-line tool or via a graphical user interface. When creating an AVD to simulate a specific Android device model, the virtual device should be configured with a hardware specification matching that of the physical device.

The AVD emulator session may be displayed as a standalone window or embedded into the main Android Studio user interface.

7. A Tour of the Android Studio User Interface

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

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

7.1 The Welcome Screen

The welcome screen (Figure 7-1) is displayed any time that Android Studio is running with no projects currently open (open projects can be closed at any time by selecting the *File* -> *Close Project* menu option). If Android Studio was previously exited while a project was still open, the tool will bypass the welcome screen the next time it is launched, automatically opening the previously active project.

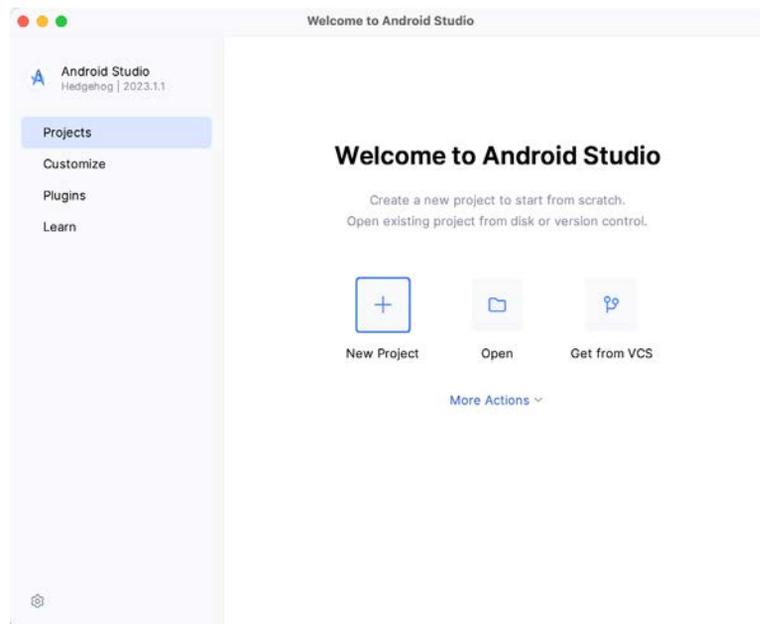


Figure 7-1

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

Additional options are available by selecting the More Actions link or using the menu shown in Figure 7-2 when

A Tour of the Android Studio User Interface

the list of recent projects replaces the More Actions link:

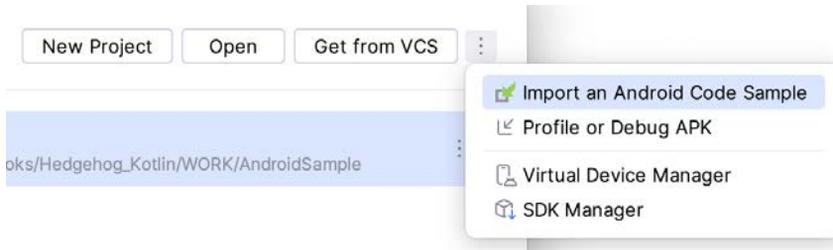


Figure 7-2

7.2 The Menu Bar

The Android Studio main window will appear when a new project is created, or an existing one is opened. When multiple projects are open simultaneously, each will be assigned its own main window. The precise configuration of the window will vary depending on the operating system Android Studio is running on and which tools and panels were displayed the last time the project was open. The appearance, for example, of the main menu bar will differ depending on the host operating system. On macOS, Android Studio follows the standard convention of placing the menu bar along the top edge of the desktop, as illustrated in Figure 7-3:

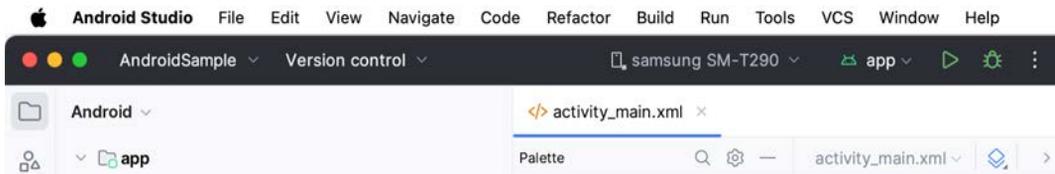


Figure 7-3

When Android Studio is running on Windows or Linux, however, the main menu is accessed via the button highlighted in Figure 7-4:

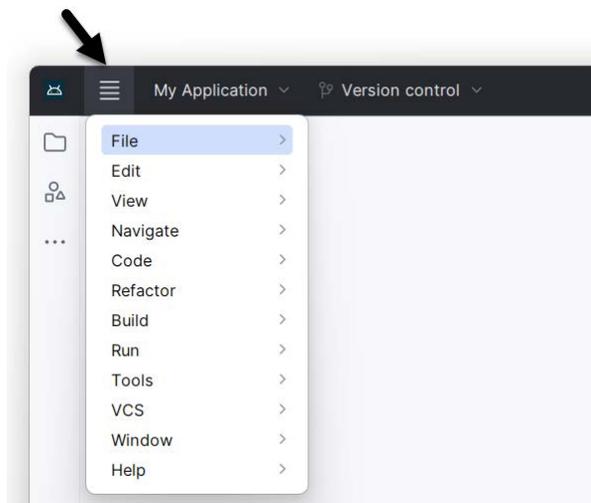


Figure 7-4

7.3 The Main Window

Once a project is open, the Android Studio main window will typically resemble that of Figure 7-5:

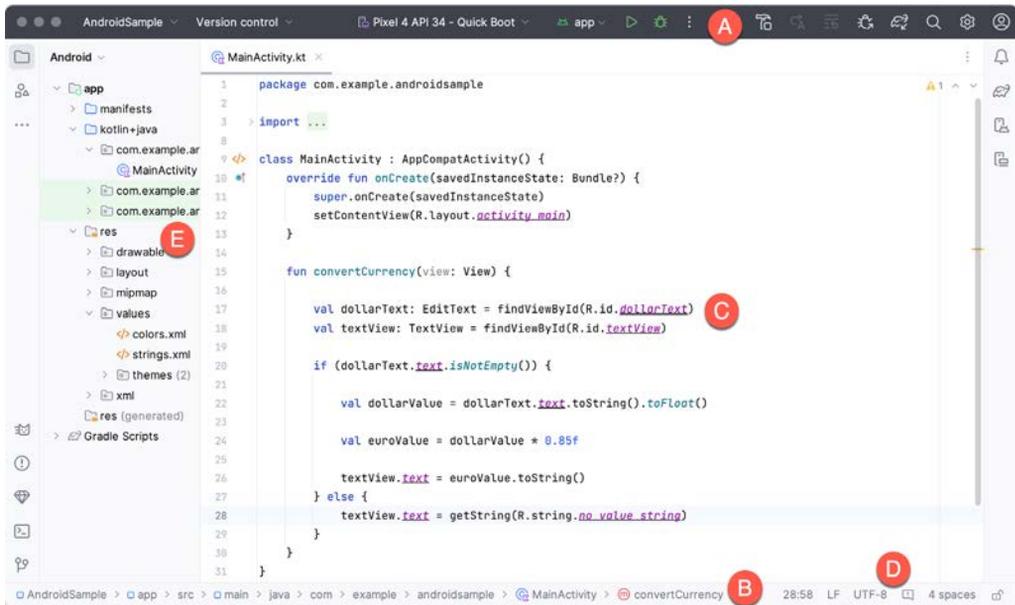


Figure 7-5

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

A – Toolbar – A selection of shortcuts to frequently performed actions. The toolbar buttons provide quick access to a select group of menu bar actions. The toolbar can be customized by right-clicking on the bar and selecting the *Customize Toolbar...* menu option. The toolbar menu shown in Figure 7-6 provides a convenient way to perform tasks such as creating and opening projects and switching between windows when multiple projects are open:

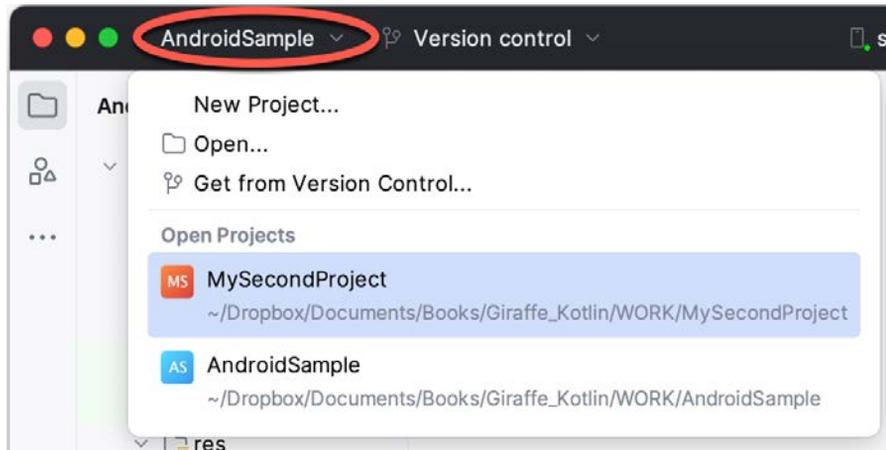


Figure 7-6

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



Figure 7-7

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

C – Editor Window – The editor window displays the content of the file on which the developer is currently working. When multiple files are open, each file is represented by a tab located along the top edge of the editor, as shown in Figure 7-8:

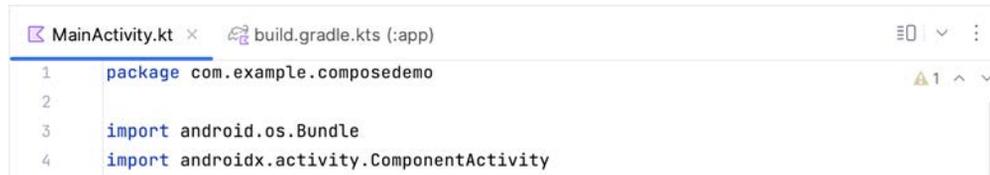


Figure 7-8

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

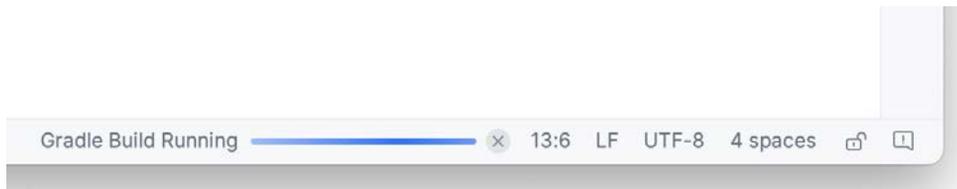


Figure 7-9

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

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

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

7.4 The Tool Windows

In addition to the project view tool window, Android Studio also includes many other windows, which, when enabled, are displayed *tool window bars* that appear along the left and right edges of the main window and contain buttons for showing and hiding each of the tool windows. Figure 7-10 shows typical tool window bar configurations, though the buttons and their positioning may differ for your Android Studio installation.

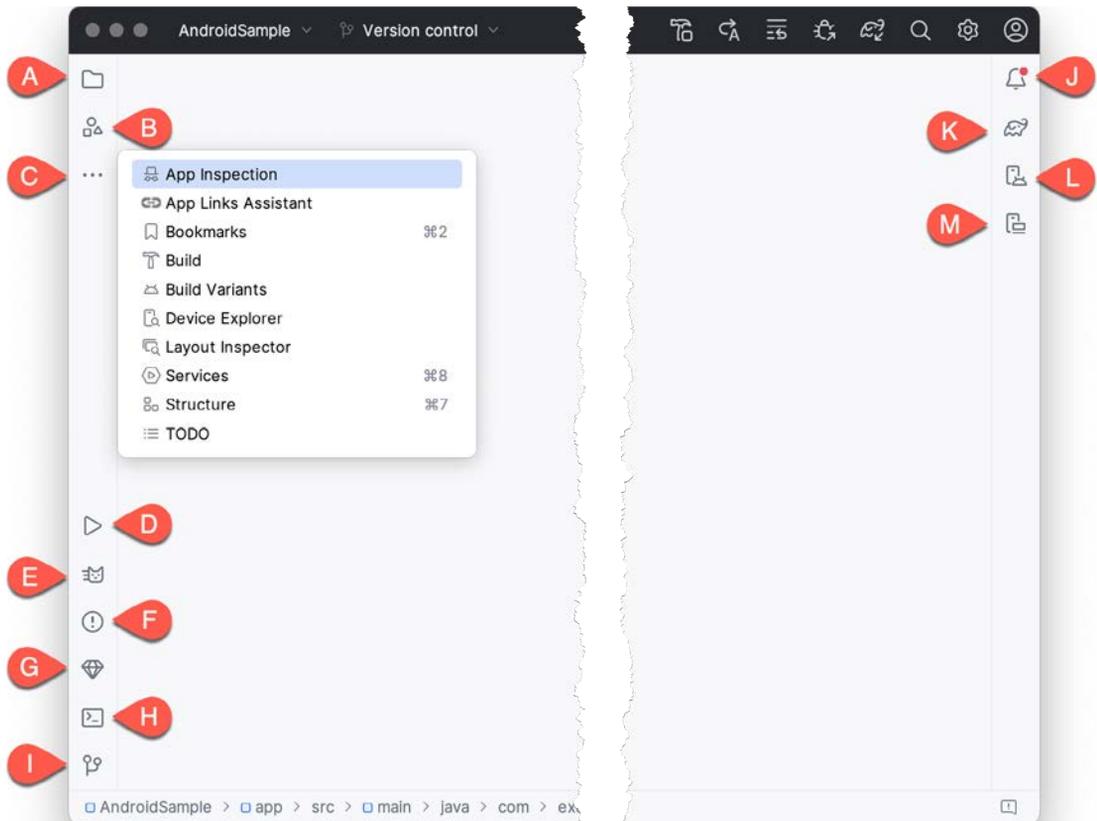


Figure 7-10

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

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

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

and options for taking screenshots and videos of the application and stopping and restarting a process.

- **Problems (F)** - A central location to view all of the current errors or warnings within the project. Double-clicking on an item in the problem list will take you to the problem file and location.
- **App Quality Insights (G)** - Provides access to the cloud-based Firebase app quality and crash analytics platform.
- **Terminal (H)** - Provides access to a terminal window on the system on which Android Studio is running. On Windows systems, this is the Command Prompt interface, while on Linux and macOS systems, this takes the form of a Terminal prompt.
- **Version Control (I)** - This tool window is used when the project files are under source code version control, allowing access to Git repositories and code change history.
- **Notifications (J)** - This tool window is used when the project files are under source code version control, allowing access to Git repositories and code change history.
- **Gradle (K)** - The Gradle tool window provides a view of the Gradle tasks that make up the project build configuration. The window lists the tasks involved in compiling the various elements of the project into an executable application. Right-click on a top-level Gradle task and select the *Open Gradle Config* menu option to load the Gradle build file for the current project into the editor. Gradle will be covered in greater detail later in this book.
- **Device Manager (L)** - Provides access to the Device Manager tool window where physical Android device connections and emulators may be added, removed, and managed.
- **Running Devices (M)** - Contains the AVD emulator if the option has been enabled to run the emulator in a tool window as outlined in the chapter entitled “*Creating an Android Virtual Device (AVD) in Android Studio*”.
- **App Inspection** - Provides access to the Database and Background Task inspectors. The Database Inspector allows you to inspect, query, and modify your app’s databases while running. The Background Task Inspector allows background worker tasks created using WorkManager to be monitored and managed.
- **Bookmarks** - The Bookmarks tool window provides quick access to bookmarked files and code lines. For example, right-clicking on a file in the project view allows access to an Add to Bookmarks menu option. Similarly, you can bookmark a line of code in a source file by moving the cursor to that line and pressing the F11 key (F3 on macOS). All bookmarked items can be accessed through this tool window.
- **Build** - The build tool window displays information about the build process while a project is being compiled and packaged and details of any errors encountered.
- **Build Variants** - The build variants tool window provides a quick way to configure different build targets for the current application project (for example, different builds for debugging and release versions of the application or multiple builds to target different device categories).
- **Device File Explorer** - Available via the *View -> Tool Windows -> Device File Explorer* menu, this tool window provides direct access to the filesystem of the currently connected Android device or emulator, allowing the filesystem to be browsed and files copied to the local filesystem.
- **Layout Inspector** - Provides a visual 3D rendering of the hierarchy of components that make up a user interface layout.
- **Structure** - The structure tool provides a high-level view of the structure of the source file currently displayed in the editor. This information includes a list of items such as classes, methods, and variables in the file.

Selecting an item from the structure list will take you to that location in the source file in the editor window.

- **TODO** – As the name suggests, this tool provides a place to review items that have yet to be completed on the project. Android Studio compiles this list by scanning the source files that make up the project to look for comments that match specified TODO patterns. These patterns can be reviewed and changed by opening the Settings dialog and navigating to the *TODO* entry listed under *Editor*.

7.5 The Tool Window Menus

Each tool window has its own toolbar along the top edge. The menu buttons within these toolbars vary from one tool to the next, though all tool windows contain an Options menu (marked A in Figure 7-11):

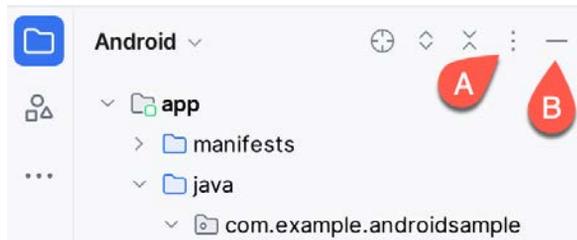


Figure 7-11

The Options menu allows various aspects of the window to be changed. Figure 7-12, for example, shows the Options menu for the Project tool window. Settings are available, for example, to undock a window and to allow it to float outside of the boundaries of the Android Studio main window, and to move and resize the tool panel:

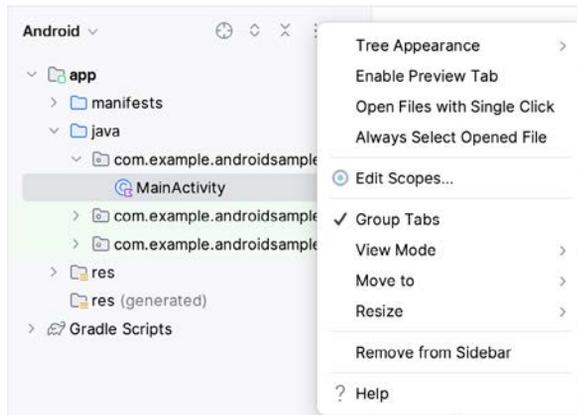


Figure 7-12

All tool windows also include a far-right button on the toolbar (marked B in Figure 7-11 above), providing an additional way to hide the tool window from view. A search of the items within a tool window can be performed by giving that window focus by clicking on it and then typing the search term (for example, the name of a file in the Project tool window). A search box will appear in the window's toolbar, and items matching the search highlighted.

7.6 Android Studio Keyboard Shortcuts

Android Studio includes many keyboard shortcuts to save time when performing common tasks. A complete keyboard shortcut keymap listing can be viewed and printed from within the Android Studio project window by selecting the *Help -> Keyboard Shortcuts PDF* menu option. You may also list and modify the keyboard shortcuts by opening the Settings dialog and clicking on the Keymap entry, as shown in Figure 7-13 below:

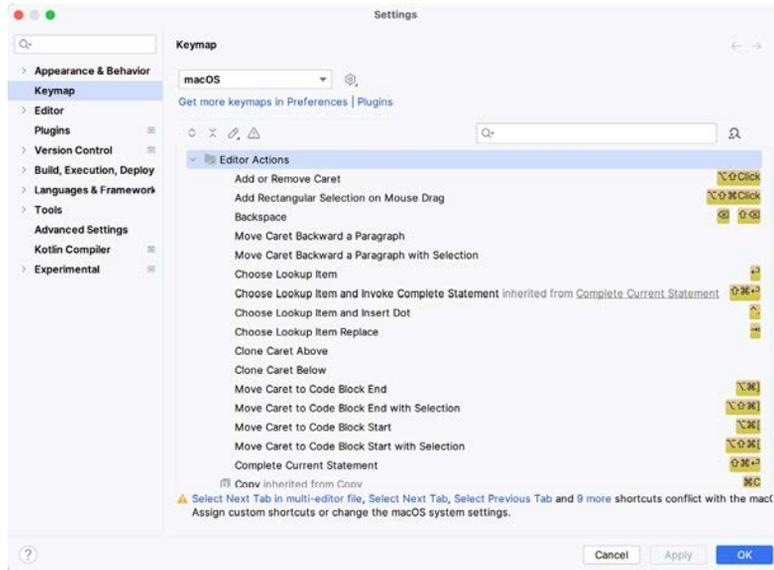


Figure 7-13

7.7 Switcher and Recent Files Navigation

Another useful mechanism for navigating within the Android Studio main window involves using the *Switcher*. Accessed via the Ctrl-Tab keyboard shortcut, the switcher appears as a panel listing both the tool windows and currently open files (Figure 7-14).

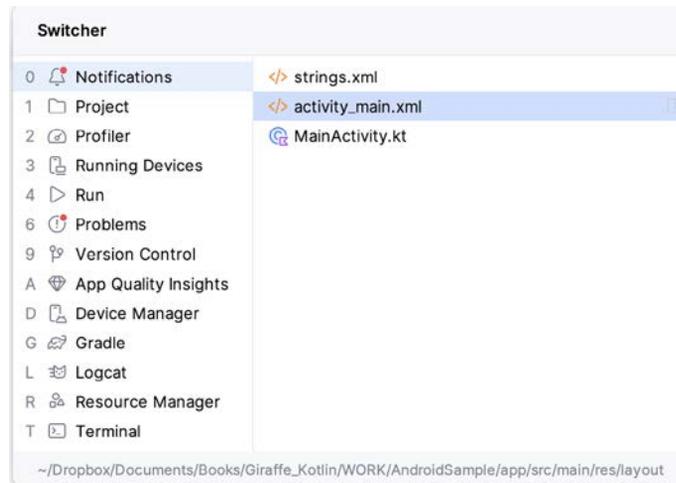


Figure 7-14

Once displayed, the switcher will remain visible as long as the Ctrl key remains depressed. Repeatedly tapping the Tab key while holding down the Ctrl key will cycle through the various selection options while releasing the Ctrl key causes the currently highlighted item to be selected and displayed within the main window.

In addition to the Switcher, the Recent Files panel provides navigation to recently opened files (Figure 7-15). This can be accessed using the Ctrl-E keyboard shortcut (Cmd-E on macOS). Once displayed, either the mouse pointer can be used to select an option, or the keyboard arrow keys can be used to scroll through the file name and tool window options. Pressing the Enter key will select the currently highlighted item:

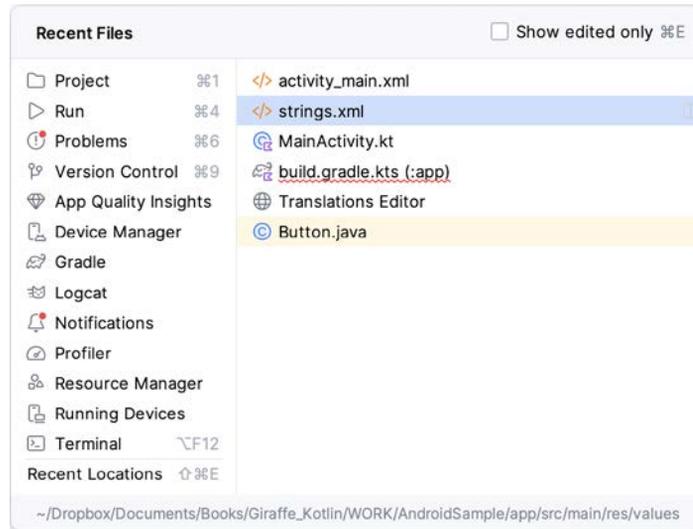


Figure 7-15

7.8 Changing the Android Studio Theme

The overall theme of the Android Studio environment may be changed using the Settings dialog. Once the settings dialog is displayed, select the *Appearance & Behavior* option in the left-hand panel, followed by *Appearance*. Then, change the setting of the *Theme* menu before clicking on the OK button. The themes available will depend on the platform but usually include options such as Light, IntelliJ, Windows, High Contrast, and Darcula. Figure 7-16 shows an example of the main window with the Dark theme selected:

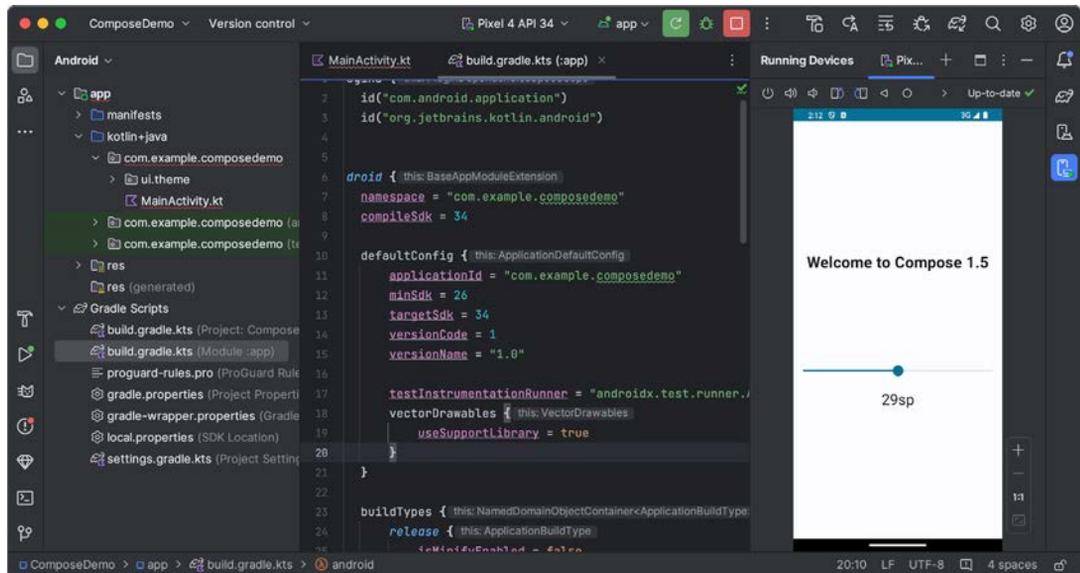


Figure 7-16

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

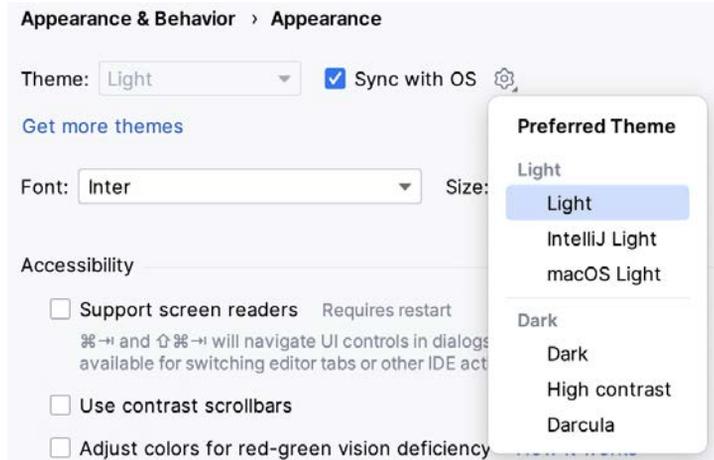


Figure 7-17

7.9 Summary

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

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

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>:

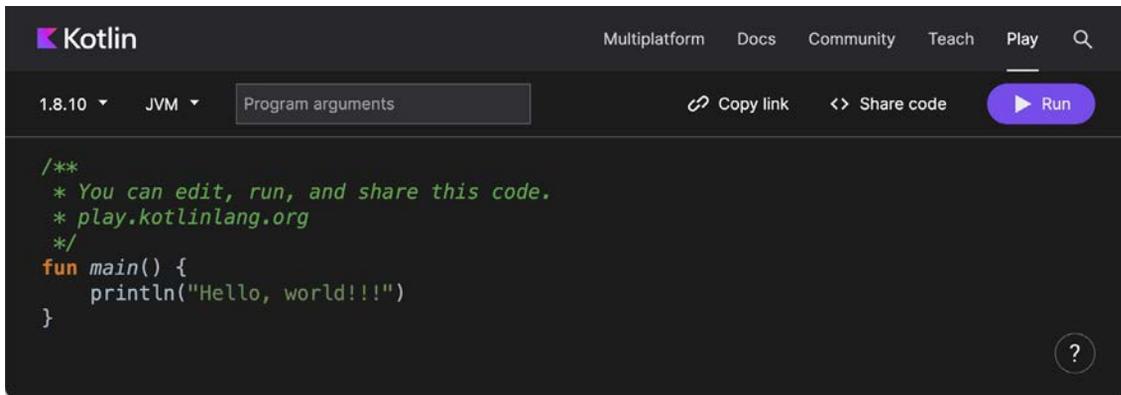


Figure 11-1

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.

12. Kotlin Data Types, Variables and Nullability

Both this and the following few chapters are intended to introduce the basics of the Kotlin programming language. This chapter will focus on the various data types available for use within Kotlin code. This will also include an explanation of constants, variables, type casting and Kotlin's handling of null values.

As outlined in the previous chapter, entitled “*An Introduction to Kotlin*” a useful way to experiment with the language is to use the Kotlin online playground environment. Before starting this chapter, therefore, open a browser window, navigate to <https://play.kotlinlang.org> and use the playground to try out the code in both this and the other Kotlin introductory chapters that follow.

12.1 Kotlin data types

When we look at the different types of software that run on computer systems and mobile devices, from financial applications to graphics-intensive games, it is easy to forget that computers are really just binary machines. Binary systems work in terms of 0 and 1, true or false, set and unset. All the data sitting in RAM, stored on disk drives and flowing through circuit boards and buses are nothing more than sequences of 1s and 0s. Each 1 or 0 is referred to as a bit and bits are grouped together in blocks of 8, each group being referred to as a byte. When people talk about 32-bit and 64-bit computer systems they are talking about the number of bits that can be handled simultaneously by the CPU bus. A 64-bit CPU, for example, is able to handle data in 64-bit blocks, resulting in faster performance than a 32-bit based system.

Humans, of course, don't think in binary. We work with decimal numbers, letters and words. For a human to easily ('easily' being a relative term in this context) program a computer, some middle ground between human and computer thinking is needed. This is where programming languages such as Kotlin come into play. Programming languages allow humans to express instructions to a computer in terms and structures we understand, and then compile that down to a format that can be executed by a CPU.

One of the fundamentals of any program involves data, and programming languages such as Kotlin define a set of *data types* that allow us to work with data in a format we understand when programming. For example, if we want to store a number in a Kotlin program we could do so with syntax similar to the following:

```
val mynumber = 10
```

In the above example, we have created a variable named *mynumber* and then assigned to it the value of 10. When we compile the source code down to the machine code used by the CPU, the number 10 is seen by the computer in binary as:

```
1010
```

Similarly, we can express a letter, the visual representation of a digit ('0' through to '9') or punctuation mark (referred to in computer terminology as *characters*) using the following syntax:

```
val myletter = 'c'
```

Once again, this is understandable by a human programmer, but gets compiled down to a binary sequence for the CPU to understand. In this case, the letter 'c' is represented by the decimal number 99 using the ASCII table (an internationally recognized standard that assigns numeric values to human readable characters). When

converted to binary, it is stored as:

```
10101100011
```

Now that we have a basic understanding of the concept of data types and why they are necessary we can take a closer look at some of the more commonly used data types supported by Kotlin.

12.1.1 Integer data types

Kotlin integer data types are used to store whole numbers (in other words, a number with no decimal places). All integers in Kotlin are signed (in other words capable of storing positive, negative and zero values).

Kotlin provides support for 8, 16, 32 and 64-bit integers (represented by the Byte, Short, Int and Long types respectively).

12.1.2 Floating point data types

The Kotlin floating-point data types are able to store values containing decimal places. For example, 4353.1223 would be stored in a floating-point data type. Kotlin provides two floating-point data types in the form of Float and Double. Which type to use depends on the size of value to be stored and the level of precision required. The Double type can be used to store up to 64-bit floating-point numbers. The Float data type, on the other hand, is limited to 32-bit floating-point numbers.

12.1.3 Boolean data type

Kotlin, like other languages, includes a data type for the purpose of handling true or false (1 or 0) conditions. Two Boolean constant values (*true* and *false*) are provided by Kotlin specifically for working with Boolean data types.

12.1.4 Character data type

The Kotlin Char data type is used to store a single character of rendered text such as a letter, numerical digit, punctuation mark or symbol. Internally characters in Kotlin are stored in the form of 16-bit Unicode grapheme clusters. A grapheme cluster is made of two or more Unicode code points that are combined to represent a single visible character.

The following lines assign a variety of different characters to Character type variables:

```
val myChar1 = 'f'  
val myChar2 = ':'  
val myChar3 = 'X'
```

Characters may also be referenced using Unicode code points. The following example assigns the 'X' character to a variable using Unicode:

```
val myChar4 = '\u0058'
```

Note the use of single quotes when assigning a character to a variable. This indicates to Kotlin that this is a Char data type as opposed to double quotes which indicate a String data type.

12.1.5 String data type

The String data type is a sequence of characters that typically make up a word or sentence. In addition to providing a storage mechanism, the String data type also includes a range of string manipulation features allowing strings to be searched, matched, concatenated and modified. Double quotes are used to surround single line strings during assignment, for example:

```
val message = "You have 10 new messages."
```

Alternatively, a multi-line string may be declared using triple quotes

```
val message = """You have 10 new messages,
                5 old messages
                and 6 spam messages."""
```

The leading spaces on each line of a multi-line string can be removed by making a call to the `trimMargin()` function of the `String` data type:

```
val message = """You have 10 new messages,
                5 old messages
                and 6 spam messages.""".trimMargin()
```

Strings can also be constructed using combinations of strings, variables, constants, expressions, and function calls using a concept referred to as string interpolation. For example, the following code creates a new string from a variety of different sources using string interpolation before outputting it to the console:

```
val username = "John"
val inboxCount = 25
val maxcount = 100
val message = "$username has $inboxCount messages. Message capacity remaining is
${maxcount - inboxCount} messages"

println(message)
```

When executed, the code will output the following message:

```
John has 25 messages. Message capacity remaining is 75 messages.
```

12.1.6 Escape sequences

In addition to the standard set of characters outlined above, there is also a range of special characters (also referred to as escape characters) available for specifying items such as a new line, tab or a specific Unicode value within a string. These special characters are identified by prefixing the character with a backslash (a concept referred to as escaping). For example, the following assigns a new line to the variable named `newline`:

```
var newline = '\n'
```

In essence, any character that is preceded by a backslash is considered to be a special character and is treated accordingly. This raises the question as to what to do if you actually want a backslash character. This is achieved by escaping the backslash itself:

```
var backslash = '\\'
```

The complete list of special characters supported by Kotlin is as follows:

- `\n` - New line
- `\r` - Carriage return
- `\t` - Horizontal tab
- `\\` - Backslash
- `\"` - Double quote (used when placing a double quote into a string declaration)
- `\'` - Single quote (used when placing a single quote into a string declaration)
- `\$` - Used when a character sequence containing a `$` is misinterpreted as a variable in a string template.
- `\unnnn` – Double byte Unicode scalar where `nnnn` is replaced by four hexadecimal digits representing the

Unicode character.

12.2 Mutable variables

Variables are essentially locations in computer memory reserved for storing the data used by an application. Each variable is given a name by the programmer and assigned a value. The name assigned to the variable may then be used in the Kotlin code to access the value assigned to that variable. This access can involve either reading the value of the variable or, in the case of *mutable variables*, changing the value.

12.3 Immutable variables

Often referred to as a *constant*, an immutable variable is similar to a mutable variable in that it provides a named location in memory to store a data value. Immutable variables differ in one significant way in that once a value has been assigned, it cannot subsequently be changed.

Immutable variables are particularly useful if there is a value that is used repeatedly throughout the application code. Rather than use the value each time, it makes the code easier to read if the value is first assigned to a constant which is then referenced in the code. For example, it might not be clear to someone reading your Kotlin code why you used the value 5 in an expression. If, instead of the value 5, you use an immutable variable named *interestRate* the purpose of the value becomes much clearer. Immutable values also have the advantage that if the programmer needs to change a widely used value, it only needs to be changed once in the constant declaration and not each time it is referenced.

12.4 Declaring mutable and immutable variables

Mutable variables are declared using the *var* keyword and may be initialized with a value at creation time. For example:

```
var userCount = 10
```

If the variable is declared without an initial value, the type of the variable must also be declared (a topic that will be covered in more detail in the next section of this chapter). The following, for example, is a typical declaration where the variable is initialized after it has been declared:

```
var userCount: Int
userCount = 42
```

Immutable variables are declared using the *val* keyword.

```
val maxUserCount = 20
```

As with mutable variables, the type must also be specified when declaring the variable without initializing it:

```
val maxUserCount: Int
maxUserCount = 20
```

When writing Kotlin code, immutable variables should always be used in preference to mutable variables whenever possible.

12.5 Data types are objects

All of the above data types are objects, each of which provides a range of functions and properties that may be used to perform a variety of different type specific tasks. These functions and properties are accessed using so-called dot notation. Dot notation involves accessing a function or property of an object by specifying the variable name followed by a dot followed in turn by the name of the property to be accessed or function to be called.

A string variable, for example, can be converted to uppercase via a call to the *toUpperCase()* function of the *String* class:

```
val myString = "The quick brown fox"
val uppercase = myString.toUpperCase()
```

Similarly, the length of a string is available by accessing the length property:

```
val length = myString.length
```

Functions are also available within the String class to perform tasks such as comparisons and checking for the presence of a specific word. The following code, for example, will return a *true* Boolean value since the word “fox” appears within the string assigned to the *myString* variable:

```
val result = myString.contains("fox")
```

All of the number data types include functions for performing tasks such as converting from one data type to another such as converting an Int to a Float:

```
val myInt = 10
val myFloat = myInt.toFloat()
```

A detailed overview of all of the properties and functions provided by the Kotlin data type classes is beyond the scope of this book (there are hundreds). An exhaustive list for all data types can, however, be found within the Kotlin reference documentation available online at:

<https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/>

12.6 Type annotations and type inference

Kotlin is categorized as a statically typed programming language. This essentially means that once the data type of a variable has been identified, that variable cannot subsequently be used to store data of any other type without inducing a compilation error. This contrasts to loosely typed programming languages where a variable, once declared, can subsequently be used to store other data types.

There are two ways in which the type of a variable will be identified. One approach is to use a type annotation at the point the variable is declared in the code. This is achieved by placing a colon after the variable name followed by the type declaration. The following line of code, for example, declares a variable named *userCount* as being of type *Int*:

```
val userCount: Int = 10
```

In the absence of a type annotation in a declaration, the Kotlin compiler uses a technique referred to as *type inference* to identify the type of the variable. When relying on type inference, the compiler looks to see what type of value is being assigned to the variable at the point that it is initialized and uses that as the type. Consider, for example, the following variable declarations:

```
var signalStrength = 2.231
val companyName = "My Company"
```

During compilation of the above lines of code, Kotlin will infer that the *signalStrength* variable is of type *Double* (type inference in Kotlin defaults to *Double* for all floating-point numbers) and that the *companyName* constant is of type *String*.

When a constant is declared without a type annotation it must be assigned a value at the point of declaration:

```
val bookTitle = "Android Studio Development Essentials"
```

If a type annotation is used when the constant is declared, however, the value can be assigned later in the code. For example:

```
val iosBookType = false
val bookTitle: String
```

```
if (iosBookType) {
    bookTitle = "iOS App Development Essentials"
} else {
    bookTitle = "Android Studio Development Essentials"
}
```

12.7 Nullable type

Kotlin nullable types are a concept that does not exist in most other programming languages (with the exception of the *optional* type in Swift). The purpose of nullable types is to provide a safe and consistent approach to handling situations where a variable may have a null value assigned to it. In other words, the objective is to avoid the common problem of code crashing with the null pointer exception errors that occur when code encounters a null value where one was not expected.

By default, a variable in Kotlin cannot have a null value assigned to it. Consider, for example, the following code:

```
val username: String = null
```

An attempt to compile the above code will result in a compilation error similar to the following:

```
Error: Null cannot be a value of a non-null string type String
```

If a variable is required to be able to store a null value, it must be specifically declared as a nullable type by placing a question mark (?) after the type declaration:

```
val username: String? = null
```

The *username* variable can now have a null value assigned to it without triggering a compiler error. Once a variable has been declared as nullable, a range of restrictions are then imposed on that variable by the compiler to prevent it being used in situations where it might cause a null pointer exception to occur. A nullable variable, cannot, for example, be assigned to a variable of non-null type as is the case in the following code:

```
val username: String? = null
val firstname: String = username
```

The above code will elicit the following error when encountered by the compiler:

```
Error: Type mismatch: inferred type is String? but String was expected
```

The only way that the assignment will be permitted is if some code is added to check that the value assigned to the nullable variable is non-null:

```
val username: String? = null

if (username != null) {
    val firstname: String = username
}
```

In the above case, the assignment will only take place if the *username* variable references a non-null value.

12.8 The safe call operator

A nullable variable also cannot be used to call a function or to access a property in the usual way. Earlier in this chapter, the *toUpperCase()* function was called on a *String* object. Given the possibility that this could cause a function to be called on a null reference, the following code will be disallowed by the compiler:

```
val username: String? = null
val uppercase = username.toUpperCase()
```

The exact error message generated by the compiler in this situation reads as follows:

```
Error: (Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?
```

In this instance, the compiler is essentially refusing to allow the function call to be made because no attempt has been made to verify that the variable is non-null. One way around this is to add some code to verify that something other than null value has been assigned to the variable before making the function call:

```
if (username != null) {
    val uppercase = username.toUpperCase()
}
```

A much more efficient way to achieve this same verification, however, is to call the function using the *safe call operator* (represented by `?.`) as follows:

```
val uppercase = username?.toUpperCase()
```

In the above example, if the `username` variable is null, the `toUpperCase()` function will not be called and execution will proceed at the next line of code. If, on the other hand, a non-null value is assigned the `toUpperCase()` function will be called and the result assigned to the `uppercase` variable.

In addition to function calls, the safe call operator may also be used when accessing properties:

```
val uppercase = username?.length
```

12.9 Not-null assertion

The *not-null assertion* removes all of the compiler restrictions from a nullable type, allowing it to be used in the same ways as a non-null type, even if it has been assigned a null value. This assertion is implemented using double exclamation marks after the variable name, for example:

```
val username: String? = null
val length = username!!.length
```

The above code will now compile, but will crash with the following exception at runtime since an attempt is being made to call a function on a non-existent object:

```
Exception in thread "main" kotlin.KotlinNullPointerException
```

Clearly, this causes the very issue that nullable types are designed to avoid. Use of the not-null assertion is generally discouraged and should only be used in situations where you are certain that the value will not be null.

12.10 Nullable types and the let function

Earlier in this chapter, we looked at how the safe call operator can be used when making a call to a function belonging to a nullable type. This technique makes it easier to check if a value is null without having to write an *if* statement every time the variable is accessed. A similar problem occurs when passing a nullable type as an argument to a function which is expecting a non-null parameter. As an example, consider the `times()` function of the `Int` data type. When called on an `Int` object and passed another integer value as an argument, the function multiplies the two values and returns the result. When the following code is executed, for example, the value of 200 will be displayed within the console:

```
val firstNumber = 10
val secondNumber = 20

val result = firstNumber.times(secondNumber)
print(result)
```

The above example works because the `secondNumber` variable is a non-null type. A problem, however, occurs if

Kotlin Data Types, Variables and Nullability

the `secondNumber` variable is declared as being of nullable type:

```
val firstNumber = 10
val secondNumber: Int? = 20

val result = firstNumber.times(secondNumber)
print(result)
```

Now the compilation will fail with the following error message because a nullable type is being passed to a function that is expecting a non-null parameter:

```
Error: Type mismatch: inferred type is Int? but Int was expected
```

A possible solution to this problem is to simply write an *if* statement to verify that the value assigned to the variable is non-null before making the call to the function:

```
val firstNumber = 10
val secondNumber: Int? = 20

if (secondNumber != null) {
    val result = firstNumber.times(secondNumber)
    print(result)
}
```

A more convenient approach to addressing the issue, however, involves use of the *let* function. When called on a nullable type object, the *let* function converts the nullable type to a non-null variable named *it* which may then be referenced within a lambda statement.

```
secondNumber?.let {
    val result = firstNumber.times(it)
    print(result)
}
```

Note the use of the safe call operator when calling the *let* function on `secondVariable` in the above example. This ensures that the function is only called when the variable is assigned a non-null value.

12.11 Late initialization (*lateinit*)

As previously outlined, non-null types need to be initialized when they are declared. This can be inconvenient if the value to be assigned to the non-null variable will not be known until later in the code execution. One way around this is to declare the variable using the *lateinit* modifier. This modifier designates that a value will be initialized with a value later. This has the advantage that a non-null type can be declared before it is initialized, with the disadvantage that the programmer is responsible for ensuring that the initialization has been performed before attempting to access the variable. Consider the following variable declaration:

```
var myName: String
```

Clearly, this is invalid since the variable is a non-null type but has not been assigned a value. Suppose, however, that the value to be assigned to the variable will not be known until later in the program execution. In this case, the *lateinit* modifier can be used as follows:

```
lateinit var myName: String
```

With the variable declared in this way, the value can be assigned later, for example:

```
myName = "John Smith"
print("My Name is " + myName)
```

Of course, if the variable is accessed before it is initialized, the code will fail with an exception:

```
lateinit var myName: String

print("My Name is " + myName)
```

```
Exception in thread "main" kotlin.UninitializedPropertyAccessException: lateinit
property myName has not been initialized
```

To verify whether a `lateinit` variable has been initialized, check the `isInitialized` property on the variable. To do this, we need to access the properties of the variable by prefixing the name with the `::` operator:

```
if (::myName.isInitialized) {
    print("My Name is " + myName)
}
```

12.12 The Elvis operator

The Kotlin Elvis operator can be used in conjunction with nullable types to define a default value that is to be returned if a value or expression result is null. The Elvis operator (`?:`) is used to separate two expressions. If the expression on the left does not resolve to a null value that value is returned, otherwise the result of the rightmost expression is returned. This can be thought of as a quick alternative to writing an if-else statement to check for a null value. Consider the following code:

```
if (myString != null) {
    return myString
} else {
    return "String is null"
}
```

The same result can be achieved with less coding using the Elvis operator as follows:

```
return myString ?: "String is null"
```

12.13 Type casting and type checking

When compiling Kotlin code, the compiler can typically infer the type of an object. Situations will occur, however, where the compiler is unable to identify the specific type. This is often the case when a value type is ambiguous or an unspecified object is returned from a function call. In this situation it may be necessary to let the compiler know the type of object that your code is expecting or to write code that checks whether the object is of a particular type.

Letting the compiler know the type of object that is expected is known as *type casting* and is achieved within Kotlin code using the *as* cast operator. The following code, for example, lets the compiler know that the result returned from the `getSystemService()` method needs to be treated as a `KeyguardManager` object:

```
val keyMgr = getSystemService(Context.KEYGUARD_SERVICE) as KeyguardManager
```

The Kotlin language includes both safe and unsafe cast operators. The above cast is an unsafe cast and will cause the app to throw an exception if the cast cannot be performed. A safe cast, on the other hand, uses the *as?* operator and returns null if the cast cannot be performed:

```
val keyMgr = getSystemService(Context.KEYGUARD_SERVICE) as? KeyguardManager
```

A type check can be performed to verify that an object conforms to a specific type using the *is* operator, for example:

```
if (keyMgr is KeyguardManager) {
```

```
    // It is a KeyguardManager object  
}
```

12.14 Summary

This chapter has begun the introduction to Kotlin by exploring data types together with an overview of how to declare variables. The chapter has also introduced concepts such as nullable types, type casting and type checking and the Elvis operator, each of which is an integral part of Kotlin programming and designed specifically to make code writing less prone to error.

18. An Overview of Compose

Now that Android Studio has been installed and the basics of the Kotlin programming language covered, it is time to start introducing Jetpack Compose.

Jetpack Compose is an entirely new approach to developing apps for all of Google's operating system platforms. The basic goals of Compose are to make app development easier, faster, and less prone to the types of bugs that typically appear when developing software projects. These elements have been combined with Compose-specific additions to Android Studio that allow Compose projects to be tested in near real-time using an interactive preview of the app during the development process.

Many of the advantages of Compose originate from the fact that it is both *declarative* and *data-driven*, topics which will be explained in this chapter.

The discussion in this chapter is intended as a high-level overview of Compose and does not cover the practical aspects of implementation within a project. Implementation and practical examples will be covered in detail in the remainder of the book.

18.1 Development before Compose

To understand the meaning and advantages of the Compose declarative syntax, it helps to understand how user interface layouts were designed before the introduction of Compose. Previously, Android apps were still built entirely using Android Studio together with a collection of associated frameworks that make up the Android Development Kit.

To aid in the design of the user interface layouts that make up the screens of an app, Android Studio includes a tool called the Layout Editor. The Layout Editor is a powerful tool that allows XML files to be created which contain the individual components that make up a screen of an app.

The user interface layout of a screen is designed within the Layout Editor by dragging components (such as buttons, text, text fields, and sliders) from a widget palette to the desired location on the layout canvas. Selecting a component in a scene provides access to a range of property panels where the attributes of the components can be changed.

The layout behavior of the screen (in other words how it reacts to different device screen sizes and changes to device orientation between portrait and landscape) is defined by configuring a range of constraints that dictate how each component is positioned and sized in relation to both the containing window and the other components in the layout.

Finally, any components that need to respond to user events (such as a button tap or slider motion) are connected to methods in the app source code where the event is handled.

At various points during this development process, it is necessary to compile and run the app on a simulator or device to test that everything is working as expected.

18.2 Compose declarative syntax

Compose introduces a declarative syntax that provides an entirely different way of implementing user interface layouts and behavior from the Layout Editor approach. Instead of manually designing the intricate details of the layout and appearance of components that make up a scene, Compose allows the scenes to be described using

An Overview of Compose

a simple and intuitive syntax. In other words, Compose allows layouts to be created by declaring how the user interface should appear without having to worry about the complexity of how the layout is built.

This essentially involves declaring the components to be included in the layout, stating the kind of layout manager in which they are to be contained (column, row, box, list, etc.), and using modifiers to set attributes such as the text on a button, the foreground color of a label, or the handler to be called in the event of a tap gesture. Having made these declarations, all the intricate and complicated details of how to position, constrain and render the layout are handled automatically by Compose. Compose declarations are structured hierarchically, which also makes it easy to create complex views by composing together small, re-usable custom sub-views.

While a layout is being declared and tested, Android Studio provides a preview canvas that changes in real-time to reflect the appearance of the layout. Android Studio also includes an *interactive preview* mode which allows the app to be launched within the preview canvas and fully tested without the need to build and run on a simulator or device.

Coverage of the Compose declaration syntax begins with the chapter entitled “*Composable Functions Overview*”.

18.3 Compose is data-driven

When we say that Compose is data-driven, this is not to say that it is no longer necessary to handle events generated by the user (in other words the interaction between the user and the app user interface). It is still necessary, for example, to know when the user taps a button or moves a slider and to react in some app-specific way. Being data-driven relates more to the relationship between the underlying app data and the user interface and logic of the app.

Before the introduction of Compose, an Android app would contain code responsible for checking the current values of data within the app. If data was likely to change over time, code had to be written to ensure that the user interface always reflected the latest state of the data (perhaps by writing code to frequently check for changes to the data, or by providing a refresh option for the user to request a data update). Similar challenges arise when keeping the user interface state consistent and making sure issues like toggle button settings are stored appropriately. Requirements such as these can become increasingly complex when multiple areas of an app depend on the same data sources.

Compose addresses this complexity by providing a system that is based on *state*. Data that is stored as state ensures that any changes to that data are automatically reflected in the user interface without the need to write any additional code to detect the change. Any user interface component that accesses a state is essentially *subscribed* to that state. When the state is changed anywhere in the app code, any subscriber components to that data will be destroyed and recreated to reflect the data change in a process called *recomposition*. This ensures that when any state on which the user interfaces is dependent changes, all components that rely on that data will automatically update to reflect the latest state. State and recomposition will be covered in the chapter entitled “*An Overview of Compose State and Recomposition*”.

18.4 Summary

Jetpack introduces a different approach to app development than that offered by the Android Studio Layout Editor. Rather than directly implement the way in which a user interface is to be rendered, Compose allows the user interface to be declared in descriptive terms and then does all the work of deciding the best way to perform the rendering when the app runs.

Compose is also data-driven in that data changes drive the behavior and appearance of the app. This is achieved through states and recomposition.

This chapter has provided a very high-level view of Jetpack Compose. The remainder of this book will explore Compose in greater depth.

19. Composable Functions Overview

Composable functions are the building blocks used to create user interfaces for Android apps when developing with Jetpack Compose. In the ComposeDemo project created earlier in the book, we made use of both the built-in compose functions provided with Compose and also created our own functions. In this chapter, we will explore composable functions in more detail, including topics such as stateful and stateless functions, function syntax, and the difference between foundation and material composables.

19.1 What is a composable function?

Composable functions (also referred to as *composables* or *components*) are special Kotlin functions that are used to create user interfaces when working with Compose. A composable function is differentiated from regular Kotlin functions in code using the `@Composable` annotation.

When a composable is called, it is typically passed some data and a set of properties that define how the corresponding section of the user interface is to behave and appear when rendered to the user in the running app. In essence, composable functions transform data into user interface elements. Composables do not return values in the traditional sense of the Kotlin function, but instead, *emit* user interface elements to the Compose runtime system for rendering.

Composable functions can call other composables to create a hierarchy of components as demonstrated in the ComposeDemo project. While a composable function may also call standard Kotlin functions, standard functions may not call composable functions.

A typical Compose-based user interface will be comprised of a combination of built-in and custom-built composables.

19.2 Stateful vs. stateless composables

Composable functions are categorized as being either *stateful* or *stateless*. State, in the context of Compose, is defined as being any value that can change during the execution of an app. For example, a slider position value, the string entered into a text field, or the current setting of a check box are all forms of state.

As we saw in the ComposeDemo project, a composable function can store a state value which defines in some way how the composable function, or those that it calls appear or behave. This is achieved using the *remember* keyword and the *mutableStateOf* function. Our DemoScreen composable, for example, stored the current slider position as state using this technique:

```
@Composable
fun DemoScreen() {

    var sliderPosition by remember { mutableStateOf(20f) }

    .
    .
}
```

Because the DemoScreen contains state, it is considered to be a stateful composable. Now consider the DemoSlider composable which reads as follows:

```
@Composable
```

Composable Functions Overview

```
fun DemoSlider(sliderPosition: Float, onPositionChange : (Float) -> Unit ) {
    Slider(
        modifier = Modifier.padding(10.dp),
        valueRange = 20f..40f,
        value = sliderPosition,
        onValueChange = onPositionChange
    )
}
```

Although this composable is passed and makes use of the state value stored by the DemoScreen, it does not itself store any state value. DemoSlider is, therefore, considered to be a stateless composable function.

The topic of state will be covered in greater detail in the chapter entitled “An Overview of Compose State and Recomposition”.

19.3 Composable function syntax

Composable functions, as we already know, are declared using the @Composable annotation and are written in much the same way as a standard Kotlin function. We can, for example, declare a composable function that does nothing as follows:

```
@Composable
fun MyFunction() {
}
```

We can also call other composables from within the function:

```
@Composable
fun MyFunction() {
    Text("Hello")
}
```

Composables may also be implemented to accept parameters. The following function accepts text, font weight, and color parameters and passes them to the built-in Text composable. The fragment also includes a preview composable to demonstrate how the CustomText function might be called:

```
@Composable
fun CustomText(text: String, fontWeight: FontWeight, color: Color) {
    Text(text = text, fontWeight = fontWeight, color = color)
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    CustomText(text = "Hello Compose", fontWeight = FontWeight.Bold,
               color = Color.Magenta)
}
```

When previewed, magenta-colored bold text reading “Hello Compose” will be rendered in the preview panel.

Just about any Kotlin logic code may be included in the body of a composable function. The following composable, for example, displays different text within a Column depending on the setting of a built-in Switch composable:

```
@Composable
```

```

fun CustomSwitch() {

    val checked = remember { mutableStateOf(true) }

    Column {
        Switch(
            checked = checked.value,
            onCheckedChange = { checked.value = it }
        )
        if (checked.value) {
            Text("Switch is On")
        } else {
            Text("Switch is Off")
        }
    }
}

```

In the above example, we have declared a state value named *checked* initialized to true and then constructed a Column containing a Switch composable. The state of the Switch is based on the value of *checked* and a lambda assigned as the *onCheckedChanged* event handler. This lambda sets the *checked* state to the current Switch setting. Finally, an *if* statement is used to decide which of two Text composables are displayed depending on the current value of the *checked* state. When run, the text displayed will alternate between “Switch is on” and “Switch is off”:



Figure 19-1

Similarly, we could use looping syntax to iterate through the items in a list and display them in a Column separated by instances of the Divider composable:

```

@Composable
fun CustomList(items: List<String>) {
    Column {
        for (item in items) {
            Text(item)
            Divider(color = Color.Black)
        }
    }
}

```

The following composable could be used to preview the above function:

```

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {

```

Composable Functions Overview

```
MyApplicationTheme {  
    CustomList(listOf("One", "Two", "Three", "Four", "Five", "Six"))  
}  
}
```

Once built and refreshed, the composable will appear in the Preview panel as shown in Figure 19-2 below:

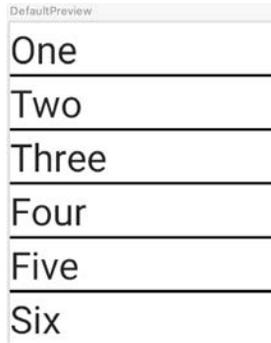


Figure 19-2

19.4 Foundation and Material composables

When developing apps with Compose we do so using a mixture of our own composable functions (for example the `CustomText` and `CustomList` composables created earlier in the chapter) combined with a set of ready to use components provided by the Compose development kit (such as the `Text`, `Button`, `Column` and `Slider` composables).

The composables bundled with Compose fall into three categories, referred to as *Layout*, *Foundation*, and *Material Design* components.

Layout components provide a way to define both how components are positioned on the screen, and how those components behave in relation to each other. The following are all layout composables:

- `Box`
- `BoxWithConstraints`
- `Column`
- `ConstraintLayout`
- `Row`

Foundation components are a set of minimal components that provide basic user interface functionality. While these components do not, by default, impose a specific style or theme, they can be customized to provide any look and behavior you need for your app. The following lists the set of Foundation components:

- `BaseTextField`
- `Canvas`
- `Image`
- `LazyColumn`
- `LazyRow`

- Shape
- Text

The Material Design components, on the other hand, have been designed so that they match Google’s Material theme guidelines and include the following composables:

- AlertDialog
- Button
- Card
- CircularProgressIndicator
- DropdownMenu
- Checkbox
- FloatingActionButton
- LinearProgressIndicator
- ModalDrawer
- RadioButton
- Scaffold
- Slider
- SnackBar
- Switch
- TextField
- TopAppBar
- BottomNavigation

When choosing components, it is important to note that the Foundation and Material Design components are not mutually exclusive. You will inevitably use components from both categories in your design since the Material Design category has components for which there is no Foundation equivalent and vice versa.

19.5 Summary

In this chapter, we have looked at composable functions and explored how they are used to construct Android-based user interfaces. Composable functions are declared using the `@Composable` annotation and use the same syntax as standard Kotlin functions, including the passing and handling of parameters. Unlike standard Kotlin functions, composable functions do not return values. Instead, they *emit* user interface units to be rendered by the Compose runtime. A composable function can be either *stateful* or *stateless* depending on whether the function stores a state value. The built-in composables are categorized as either Layout, Foundation, or Material Design components. The Material Design components conform with the Material style and theme guidelines provided by Google to encourage consistent UI design.

One type of composable we have not yet introduced is the Slot API composable, a topic that will be covered later in the chapter entitled “*An Overview of Compose Slot APIs*”.

20. An Overview of Compose State and Recomposition

State is the cornerstone of how the Compose system is implemented. As such, a clear understanding of state is an essential step in becoming a proficient Compose developer. In this chapter, we will explore and demonstrate the basic concepts of state and explain the meaning of related terms such as *recomposition*, *unidirectional data flow*, and *state hoisting*. The chapter will also cover saving and restoring state through *configuration changes*.

20.1 The basics of state

In declarative languages such as Compose, *state* is generally referred to as “a value that can change over time”. At first glance, this sounds much like any other data in an app. A standard Kotlin variable, for example, is by definition designed to store a value that can change at any time during execution. State, however, differs from a standard variable in two significant ways.

First, the value assigned to a state variable in a composable function needs to be remembered. In other words, each time a composable function containing state (a *stateful function*) is called, it must remember any state values from the last time it was invoked. This is different from a standard variable which would be re-initialized each time a call is made to the function in which it is declared.

The second key difference is that a change to any state variable has far reaching implications for the entire hierarchy tree of composable functions that make up a user interface. To understand why this is the case, we now need to talk about recomposition.

20.2 Introducing recomposition

When developing with Compose, we build apps by creating hierarchies of composable functions. As previously discussed, a composable function can be thought of as taking data and using that data to generate sections of a user interface layout. These elements are then rendered on the screen by the Compose runtime system. In most cases, the data passed from one composable function to another will have been declared as a state variable in a parent function. This means that any change of state value in a parent composable will need to be reflected in any child composables to which the state has been passed. Compose addresses this by performing an operation referred to as *recomposition*.

Recomposition occurs whenever a state value changes within a hierarchy of composable functions. As soon as Compose detects a state change, it works through all of the composable functions in the activity and recomposes any functions affected by the state value change. Recomposing simply means that the function gets called again and passed the new state value.

Recomposing the entire composable tree for a user interface each time a state value changes would be a highly inefficient approach to rendering and updating a user interface. Compose avoids this overhead using a technique called *intelligent recomposition* that involves only recomposing those functions directly affected by the state change. In other words, only functions that read the state value will be recomposed when the value changes.

20.3 Creating the StateExample project

Launch Android Studio and select the New Project option from the welcome screen. Within the resulting new project dialog, choose the *Empty Activity* template before clicking on the Next button.

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

20.4 Declaring state in a composable

The first step in declaring a state value is to wrap it in a `MutableState` object. `MutableState` is a Compose class which is referred to as an *observable type*. Any function that reads a state value is said to have *subscribed* to that observable state. As a result, any changes to the state value will trigger the recomposition of all subscribed functions.

Within Android Studio, open the *MainActivity.kt* file, delete the Greeting composable and modify the class so that it reads as follows:

```
package com.example.stateexample
.
.
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            StateExampleTheme {
                Surface(color = MaterialTheme.colorScheme.background) {
                    DemoScreen()
                }
            }
        }
    }
}

@Composable
fun DemoScreen() {
    MyTextField()
}

@Composable
fun MyTextField() {

}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    StateExampleTheme {
```

```

        DemoScreen()
    }
}

```

The objective here is to implement `MyTextField` as a stateful composable function containing a state variable and an event handler that changes the state based on the user's keyboard input. The result is a text field in which the characters appear as they are typed.

`MutableState` instances are created by making a call to the `mutableStateOf()` runtime function, passing through the initial state value. The following, for example, creates a `MutableState` instance initialized with an empty `String` value:

```
var textState = { mutableStateOf("") }
```

This provides an observable state which will trigger a recomposition of all subscribed functions when the contained value is changed. The above declaration is, however, missing a key element. As previously discussed, state must be remembered through recompositions. As currently implemented, the state will be reinitialized to an empty string each time the function in which it is declared is recomposed. To retain the current state value, we need to use the `remember` keyword:

```
var myState = remember { mutableStateOf("") }
```

Remaining within the `MainActivity.kt` file, add some imports and modify the `MyTextField` composable as follows:

```

.
.
import androidx.compose.material3.*
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.foundation.layout.Column
.
.
@Composable
fun MyTextField() {

    var textState = remember { mutableStateOf("") }

    val onTextChange = { text : String ->
        textState.value = text
    }

    TextField(
        value = textState.value,
        onValueChange = onTextChange
    )
}

```

If the code editor reports that the Material 3 `TextField` is experimental, modify the `MyTextField` composable as follows:

```

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MyTextField() {

```

An Overview of Compose State and Recomposition

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

```
.  
.
```

Test the code using the Preview panel in interactive mode and confirm that keyboard input appears in the TextField as it is typed.

When looking at Compose code examples, you may see MutableState objects declared in different ways. When using the above format, it is necessary to read and set the *value* property of the MutableState instance. For example, the event handler to update the state reads as follows:

```
val onTextChange = { text: String ->  
    textState.value = text  
}
```

Similarly, the current state value is assigned to the TextField as follows:

```
TextField(  
    value = textState.value,  
    onValueChange = onTextChange  
)
```

A more common and concise approach to declaring state is to use Kotlin property delegates via the *by* keyword as follows (note that two additional libraries need to be imported when using property delegates):

```
.  
.  
import androidx.compose.runtime.getValue  
import androidx.compose.runtime.setValue  
.  
.  
@Composable  
fun MyTextField() {  
  
    var textState by remember { mutableStateOf("") }  
.  
.
```

We can now access the state value without needing to directly reference the MutableState *value* property within the event handler:

```
val onTextChange = { text: String ->  
    textState = text  
}
```

This also makes reading the current value more concise:

```
TextField(  
    value = textState,  
    onValueChange = onTextChange  
)
```

A third technique separates the access to a MutableState object into a *value* and a *setter function* as follows:

```
var (textValue, setText) = remember { mutableStateOf("") }
```

When changing the value assigned to the state we now do so by calling the *setText* setter, passing through the new value:

```
val onTextChange = { text: String ->
    setText(text)
}
```

The state value is now accessed by referencing *textValue*:

```
TextField(
    value = textValue,
    onValueChange = onTextChange
)
```

In most cases, the use of the *by* keyword and property delegates is the most commonly used technique because it results in cleaner code. Before continuing with the chapter, revert the example to use the *by* keyword.

20.5 Unidirectional data flow

Unidirectional data flow is an approach to app development whereby state stored in a composable should not be directly changed by any child composable functions. Consider, for example, a composable function named *FunctionA* containing a state value in the form of a Boolean value. This composable calls another composable function named *FunctionB* that contains a *Switch* component. The objective is for the switch to update the state value each time the switch position is changed by the user. In this situation, adherence to unidirectional data flow prohibits *FunctionB* from directly changing the state value.

Instead, *FunctionA* would declare an event handler (typically in the form of a lambda) and pass it as a parameter to the child composable along with the state value. The *Switch* within *FunctionB* would then be configured to call the event handler each time the switch position changes, passing it the current setting value. The event handler in *FunctionA* will then update the state with the new value.

Make the following changes to the *MainActivity.kt* file to implement *FunctionA* and *FunctionB* together with a corresponding modification to the preview composable:

```
@Composable
fun FunctionA() {

    var switchState by remember { mutableStateOf(true) }

    val onSwitchChange = { value : Boolean ->
        switchState = value
    }

    FunctionB(
        switchState = switchState,
        onSwitchChange = onSwitchChange
    )
}

@Composable
fun FunctionB(switchState: Boolean, onSwitchChange : (Boolean) -> Unit ) {
    Switch(
```

An Overview of Compose State and Recomposition

```
        checked = switchState,
        onCheckedChange = onSwitchChange
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    StateExampleTheme {
        Column {
            DemoScreen()
            FunctionA()
        }
    }
}
```

Preview the app using interactive mode and verify that clicking the switch changes the slider position between on and off states.

We can now use this example to break down the state process into the following individual steps which occur when FunctionA is called:

1. The *switchState* state variable is initialized with a true value.
2. The *onSwitchChange* event handler is declared to accept a Boolean parameter which it assigns to *switchState* when called.
3. FunctionB is called and passed both *switchState* and a reference to the *onSwitchChange* event handler.
4. FunctionB calls the built-in Switch component and configures it to display the state assigned to *switchState*. The Switch component is also configured to call the *onSwitchChange* event handler when the user changes the switch setting.
5. Compose renders the Switch component on the screen.

The above sequence explains how the Switch component gets rendered on the screen when the app first launches. We can now explore the sequence of events that occur when the user slides the switch to the “off” position:

1. The switch is moved to the “off” position.
2. The Switch component calls the *onSwitchChange* event handler passing through the current switch position value (in this case *false*).
3. The *onSwitchChange* lambda declared in FunctionA assigns the new value to *switchState*.
4. Compose detects that the *switchState* state value has changed and initiates a recomposition.
5. Compose identifies that FunctionB contains code that reads the value of *switchState* and therefore needs to be recomposed.
6. Compose calls FunctionB with the latest state value and the reference to the event handler.
7. FunctionB calls the Switch composable and configures it with the state and event handler.

8. Compose renders the Switch on the screen, this time with the switch in the “off” position.

The key point to note about this process is that the value assigned to *switchState* is only changed from within FunctionA and never directly updated by FunctionB. The Switch setting is not moved from the “on” position to the “off” position directly by FunctionB. Instead, the state is changed by calling upwards to the event handler located in FunctionA, and allowing recomposition to regenerate the Switch with the new position setting.

As a general rule, data is passed down through a composable hierarchy tree while events are called upwards to handlers in ancestor components as illustrated in Figure 20-1:

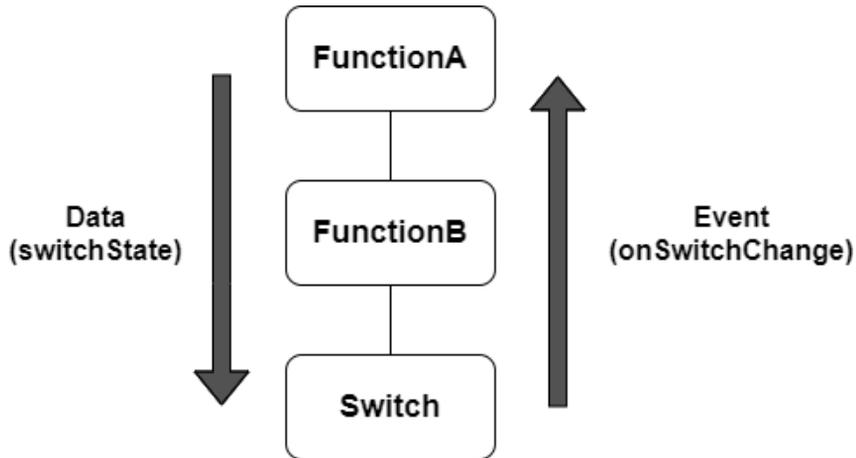


Figure 20-1

20.6 State hoisting

If you look up the word “hoist” in a dictionary it will likely be defined as the act of raising or lifting something. The term *state hoisting* has a similar meaning in that it involves moving state from a child composable up to the calling (parent) composable or a higher ancestor. When the child composable is called by the parent, it is passed the state along with an event handler. When an event occurs in the child composable that requires an update to the state, a call is made to the event handler passing through the new value as outlined earlier in the chapter. This has the advantage of making the child composable stateless and, therefore, easier to reuse. It also allows the state to be passed down to other child composables later in the app development process.

Consider our MyTextField example from earlier in the chapter:

```
@Composable
fun DemoScreen() {
    MyTextField()
}

@Composable
fun MyTextField() {

    var textState by remember { mutableStateOf("") }

    val onTextChange = { text : String ->
        textState = text
    }
}
```

An Overview of Compose State and Recomposition

```
    TextField(  
        value = textState,  
        onValueChange = onTextChange  
    )  
}
```

The self-contained nature of the `MyTextField` composable means that it is not a particularly useful component. One issue is that the text entered by the user is not accessible to the calling function and, therefore, cannot be passed to any sibling functions. It is also not possible to pass a different state and event handler through to the function, thereby limiting its re-usability.

To make the function more useful we need to hoist the state into the parent `DemoScreen` function as follows:

```
@Composable  
fun DemoScreen() {  
  
    var textState by remember { mutableStateOf("") }  
  
    val onTextChange = { text : String ->  
        textState = text  
    }  
  
    MyTextField(text = textState, onTextChange = onTextChange)  
}
```

```
@Composable  
fun MyTextField(text: String, onTextChange : (String) -> Unit) {  
  
    var textState by remember { mutableStateOf("") }  
    —  
    — val onTextChange = { text : String ->  
    — textState = text  
    — }  
  
    TextField(  
        value = text,  
        onValueChange = onTextChange  
    )  
}
```

```
@Preview(showBackground = true)  
@Composable  
fun GreetingPreview() {  
    StateExampleTheme {  
        DemoScreen()  
    }  
}
```

With the state hoisted to the parent function, `MyTextField` is now a stateless, reusable composable which can be called and passed any state and event handler. Also, the text entered by the user is now accessible by the parent function and may be passed down to other composables if necessary.

State hoisting is not limited to moving to the immediate parent of a composable. State can be raised any number of levels upward within the composable hierarchy and subsequently passed down through as many layers of children as needed (within reason). This will often be necessary when multiple children need access to the same state. In such a situation, the state will need to be hoisted up to an ancestor that is common to both children.

In Figure 20-2 below, for example, both `NameField` and `NameText` need access to `textState`. The only way to make the state available to both composables is to hoist it up to the `MainScreen` function since this is the only ancestor both composables have in common:

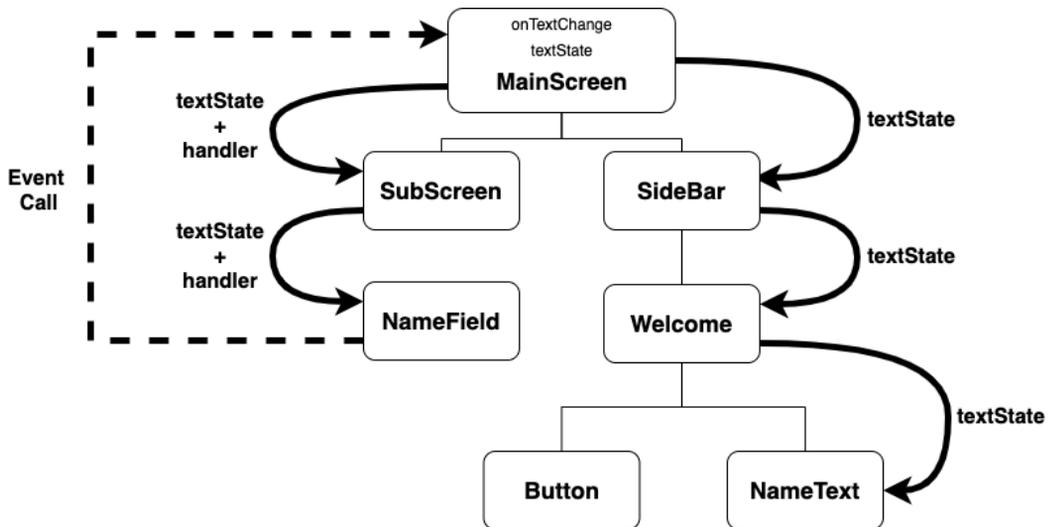


Figure 20-2

The solid arrows indicate the path of `textState` as it is passed down through the hierarchy to the `NameField` and `NameText` functions (in the case of the `NameField`, a reference to the event handler is also passed down), while the dotted line represents the calls from `NameField` function to an event handler declared in `MainScreen` as the text changes.

Note that if you find yourself passing state down through an excessive number of child layers, it may be worth looking at `CompositionLocalProvider`, a topic covered in the chapter entitled “An Introduction to *Composition Local*”.

When adding state to a function, take some time to decide whether hoisting state to the caller (or higher) might make for a more re-usable and flexible composable. While situations will arise where state is only needed to be used locally in a composable, in most cases it probably makes sense to hoist the state up to an ancestor.

20.7 Saving state through configuration changes

We now know that the `remember` keyword can be used to save state values through recompositions. This technique does not, however, retain state between *configuration changes*. A configuration change generally occurs when some aspect of the device changes in a way that alters the appearance of an activity (such as rotating the orientation of the device between portrait and landscape or changing a system-wide font setting).

An Overview of Compose State and Recomposition

Changes such as these will cause the entire activity to be destroyed and recreated. The reasoning behind this is that these changes affect resources such as the layout of the user interface and simply destroying and recreating impacted activities is the quickest way for an activity to respond to the configuration change. The result is a newly initialized activity with no memory of any previous state values.

To experience the effect of a configuration change, run the `StateExample` app on an emulator or device and, once running, enter some text so that it appears in the `TextField` before changing the orientation from portrait to landscape. When using the emulator, device rotation may be simulated using the rotation button located in the emulator toolbar. To complete the rotation on Android 11 or older, it may also be necessary to tap on the rotation button. This appears in the toolbar of the device or emulator screen as shown in Figure 20-3:



Figure 20-3

Before performing the rotation on Android 12 or later, you may need to enter the Settings app, select the Display category and enable the *Auto-rotate screen* option.

Note that after rotation, the `TextField` is now blank and the text entered has been lost. In situations where state needs to be retained through configuration changes, Compose provides the *rememberSaveable* keyword. When *rememberSaveable* is used, the state will be retained not only through recompositions, but also configuration changes. Modify the *textState* declaration to use *rememberSaveable* as follows:

```
.  
.br/>import androidx.compose.runtime.saveable.rememberSaveable  
.br/>@Composable  
fun DemoScreen() {  
  
    var textState by rememberSaveable { mutableStateOf("") }  
.br/>.  
.
```

Build and run the app once again, enter some text and perform another rotation. Note that the text is now preserved following the configuration change.

20.8 Summary

When developing apps with Compose it is vital to have a clear understanding of how state and recomposition work together to make sure that the user interface is always up to date. In this chapter, we have explored state and described how state values are declared, updated, and passed between composable functions. You should also have a better understanding of recomposition and how it is triggered in response to state changes.

We also introduced the concept of unidirectional data flow and explained how data flows down through the

compose hierarchy while data changes are made by making calls upward to event handlers declared within ancestor stateful functions.

An important goal when writing composable functions is to maximize re-usability. This can be achieved, in part, by hoisting state out of a composable up to the calling parent or a higher function in the compose hierarchy.

Finally, the chapter described configuration changes and explained how such changes result in the destruction and recreation of entire activities. Ordinarily, state is not retained through configuration changes unless specifically configured to do so using the *rememberSaveable* keyword.

22. An Overview of Compose Slot APIs

Now that we have a better idea of what composable functions are and how to create them, it is time to explore composables that provide a *slot API*. In this chapter, we will explain what a slot API is, what it is used for and how you can include slots in your own composable functions. We will also explore some of the built-in composables that provide slot API support.

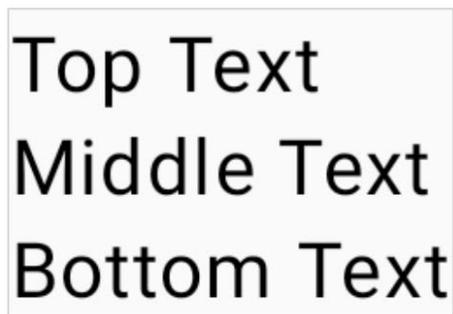
22.1 Understanding slot APIs

As we already know, composable functions can include calls to one or more other composable functions. This usually means that the content of a composable is predefined in terms of which other composables it calls and, therefore, the content it displays. Consider the following function consisting of a Column and three Text components:

```
@Composable
fun SlotDemo() {
    Column {
        Text("Top Text")
        Text("Middle Text")
        Text("Bottom Text")
    }
}
```

The function could be modified to pass in parameters that specify the text to be displayed or even the color and font size of that text. Regardless of the changes we make, however, the function is still restricted to displaying a column containing three Text components:

DefaultPreview



```
Top Text
Middle Text
Bottom Text
```

Figure 22-1

Suppose, however, that we need to display three items in a column, but do not know what composable will take up the middle position until just before the composable is called. In its current form, there is no way to display anything but the declared Text component in the middle position. The solution to this problem is to open up the middle composable as a *slot* into which any other composable may be placed when the function is called. This

is referred to as providing a *slot API* for the composable. API is an abbreviation of Application Programming Interface and, in this context, implies that we are adding a programming interface to our composable that allows the caller to specify the composable to appear within a slot. In fact, a composable function can provide multiple slots to the caller. In the above function, for example, all of the Text components could be declared as slots if required.

22.2 Declaring a slot API

It can be helpful to think of a slot API composable as a user interface template in which one or more elements are left blank. These missing pieces are then passed as parameters when the composable is called and included when the user interface is rendered by the Compose runtime system.

The first step in adding slots to a composable is to specify that it accepts a slot as a parameter. This is essentially a case of declaring that a composable accepts other composables as parameters. In the case of our example SlotDemo composable, we would modify the function signature as follows:

```
@Composable
fun SlotDemo(middleContent: @Composable () -> Unit) {
    .
    .
}
```

When the SlotDemo composable is called, it will now need to be passed a composable function. Note that the function is declared as returning a *Unit* object. Unit is a Kotlin type used to indicate that a function does not return any value. Unit can be considered to be the Kotlin equivalent of *void* in other languages. The parameter has been assigned a label of “middleContent”, though this could be any valid label name that helps to describe the slot and allows us to reference it within the body of the function.

The only remaining change to this composable is to substitute the middleContent component into the Column declaration as follows:

```
@Composable
fun SlotDemo(middleContent: @Composable () -> Unit) {
    Column {
        Text("Top Text")
        middleContent()
        Text("Bottom Text")
    }
}
```

We have now successfully declared a slot API for our SlotDemo composable.

22.3 Calling slot API composables

The next step is to learn how to make use of the slot API configured into our SlotDemo composable. This simply involves passing a composable through as a parameter when making the SlotDemo function call. Suppose, for example, that we need the following composable to appear in the middleContent slot:

```
@Composable
fun ButtonDemo() {
    Button(onClick = { }) {
        Text("Click Me")
    }
}
```

We can now call our SlotDemo composable function as follows:

```
SlotDemo(middleContent = { ButtonDemo() })
```

While this syntax works, it can quickly become cluttered if the composable has more than one slot to be filled. A cleaner syntax reads as follows:

```
SlotDemo {
    ButtonDemo()
}
```

Regardless of the syntax used, the design will be rendered as shown below in Figure 22-2:



Figure 22-2

A slot API is not, of course, limited to a single slot. The SlotDemo example could be composed entirely of slots as follows:

```
@Composable
fun SlotDemo(
    topContent: @Composable () -> Unit,
    middleContent: @Composable () -> Unit,
    bottomContent: @Composable () -> Unit) {
    Column {
        topContent()
        middleContent()
        bottomContent()
    }
}
```

With these changes made, the call to SlotDemo could be structured as follows:

```
SlotDemo(
    topContent = { Text("Top Text") },
    middleContent = { ButtonDemo() },
    bottomContent = { Text("Bottom Text") }
)
```

As with the single slot, this can be abbreviated for clarity:

```
SlotDemo {
```

An Overview of Compose Slot APIs

```
    { Text("Top Text") },  
    { ButtonDemo() },  
    { Text("Bottom Text") }  
)
```

22.4 Summary

In this chapter, we have introduced the concept of slot APIs and demonstrated how they can be added to composable functions. By implementing a slot API, the content of a composable function can be specified dynamically at the point that it is called. This contrasts with the static content of a typical composable where the content is defined at the point the function is written and cannot subsequently be changed. A composable with a slot API is essentially a user interface template containing one or more slots into which other composables can be inserted at runtime.

With the basics of slot APIs covered in this chapter, the next chapter will create a project that puts this theory into practice.

32. A Guide to ConstraintLayout in Compose

As we have seen in the preceding chapters, Compose provides several layout components to design user interfaces in addition to the ability to create custom layouts and modifiers. While these will meet most layout needs, there may still be situations where more detailed control over the positioning and sizing of composables may be required. Before the introduction of Jetpack Compose this capability was provided by the ConstraintLayout manager which is also available from within Compose.

This chapter will outline the basic concepts of ConstraintLayout while the next chapter will provide a detailed overview of how constraint-based layouts can be created using ConstraintLayout within Compose.

32.1 An introduction to ConstraintLayout

Introduced as part of the Android 7 SDK, ConstraintLayout provides a simple, expressive and flexible layout system designed to ease the creation of responsive user interface layouts. ConstraintLayout is of particular use when developing user interface layouts that need to adapt automatically to different screen sizes and changes in device orientation.

32.2 How ConstraintLayout works

In common with all other layouts, ConstraintLayout is responsible for managing the positioning and sizing behavior of its child components. It does this based on the constraint connections that are set on each child.

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

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

32.2.1 Constraints

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

engine in both the horizontal and vertical planes.

32.2.2 Margins

A margin is a form of constraint that specifies a fixed distance. Consider a Button component that needs to be positioned near the top right-hand corner of the device screen. This might be achieved by implementing margin constraints from the top and right-hand edges of the Button connected to the corresponding sides of the parent ConstraintLayout as illustrated in Figure 32-1:

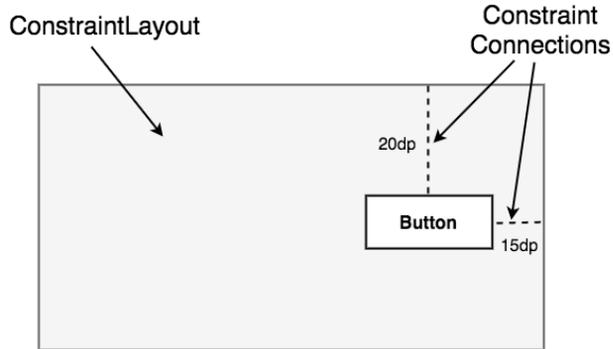


Figure 32-1

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

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

32.2.3 Opposing constraints

Two constraints operating along the same axis on a single composable are referred to as *opposing constraints*. In other words, a component with constraints on both its left and right-hand sides is considered to have horizontally opposing constraints. Figure 32-2, for example, illustrates the addition of both horizontally and vertically opposing constraints to the previous layout:

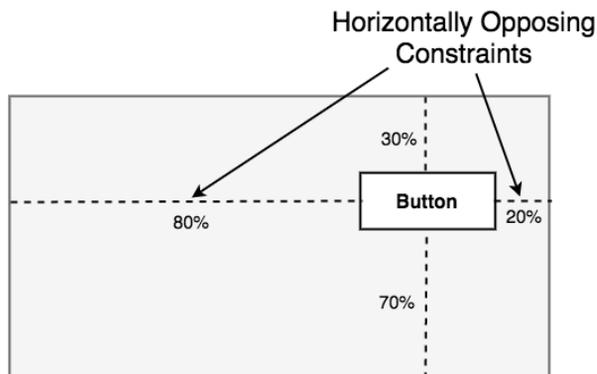


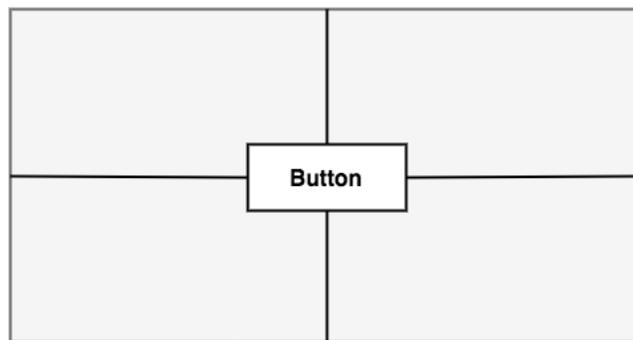
Figure 32-2

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

It is now important to understand that the layout outlined in Figure 32-2 has been implemented using not only opposing constraints but also by applying *constraint bias*.

32.2.4 Constraint bias

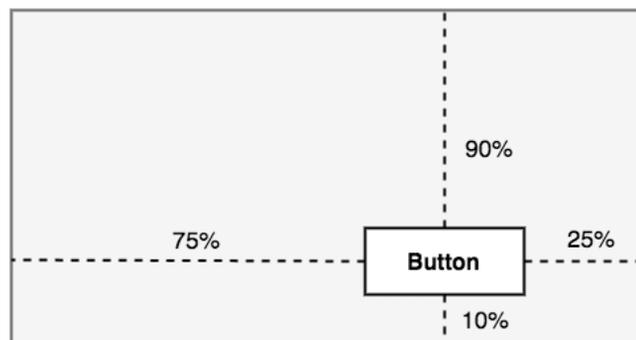
It has now been established that a component in a ConstraintLayout can potentially be subject to opposing constraint connections. By default, opposing constraints are equal, resulting in the corresponding widget being centered along the axis of opposition. Figure 32-3, for example, shows a button centered within the containing ConstraintLayout using opposing horizontal and vertical constraints:



Widget Centered by Opposing Constraints

Figure 32-3

To allow for the adjustment of component position in the case of opposing constraints, the ConstraintLayout implements a feature known as *constraint bias*. Constraint bias allows the positioning of a composable along the axis of opposition to be biased by a specified percentage in favor of one constraint. Figure 32-4, for example, shows the previous constraint layout with a 75% horizontal bias and 10% vertical bias:



Widget Offset using Constraint Bias

Figure 32-4

The next chapter, entitled “*Working with ConstraintLayout in Compose*”, will cover these concepts in greater detail and explain how these features have been integrated into Compose. In the meantime, however, a few more

areas of the ConstraintLayout class need to be covered.

32.2.5 Chains

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

Although Compose provides a helper to ease the creation of chains, it is worth noting that behind the scenes, composables are chained when connected by bi-directional constraints. Figure 32-5, for example, illustrates three Buttons chained in this way:

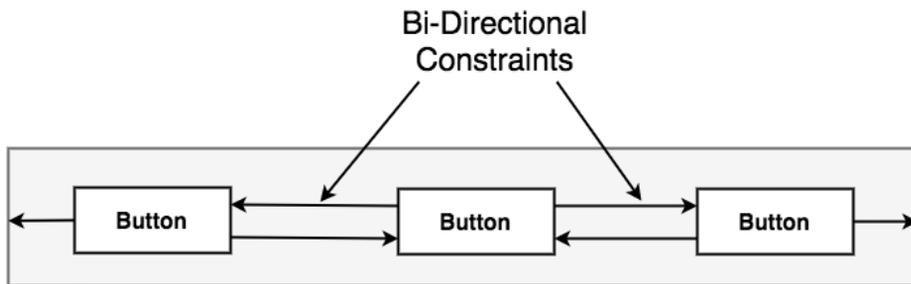


Figure 32-5

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

32.2.6 Chain styles

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

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



Figure 32-6

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



Figure 32-7

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

properties.

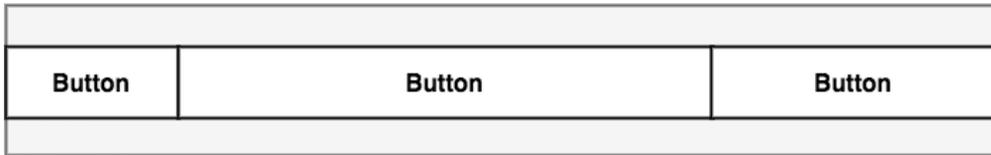


Figure 32-8

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

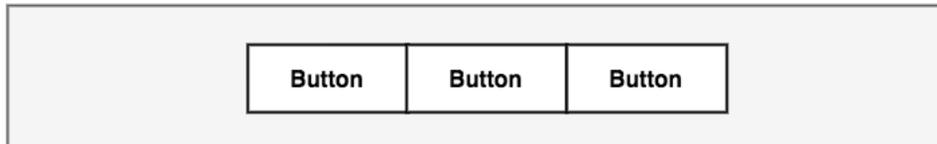


Figure 32-9

32.3 Configuring dimensions

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

- **Dimension.preferredWrapContent** - The size of the composable is dictated by the content it contains (i.e. text or graphics) subject to prevailing constraints.
- **Dimension.wrapContent** - The size of the composable is dictated by the content it contains regardless of prevailing constraints.
- **Dimension.fillToConstraints** - Allows the composable to be sized to fill the space allowed by the prevailing constraints.
- **Dimension.preferredValue** - The composable is fixed to specified dimensions subject to the prevailing constraints.
- **Dimension.value** - The composable is fixed to specified dimensions regardless of the prevailing constraints.

32.4 Guideline helper

Guidelines are special elements available within the `ConstraintLayout` that provide an additional target to which constraints may be connected. Multiple guidelines may be added to a `ConstraintLayout` instance which may, in turn, be configured in horizontal or vertical orientations. Once added, constraint connections may be established from `Composables` in the layout to the guidelines. This is particularly useful when multiple composables need to be aligned along an axis. In Figure 32-10, for example, three `Buttons` contained within a `ConstraintLayout` are constrained along a vertical guideline:

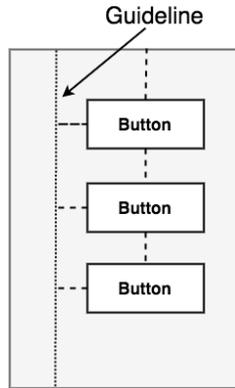


Figure 32-10

32.5 Barrier helper

Rather like guidelines, barriers are virtual views that can be used to constrain composables within a layout. As with guidelines, a barrier can be vertical or horizontal and one or more composables may be constrained to it (to avoid confusion, these will be referred to as *constrained components*). Unlike guidelines where the guideline remains at a fixed position within the layout, however, the position of a barrier is defined by a set of so-called *reference components*. Barriers were introduced to address an issue that occurs with some frequency involving overlapping components. Consider, for example, the layout illustrated in Figure 32-11 below:

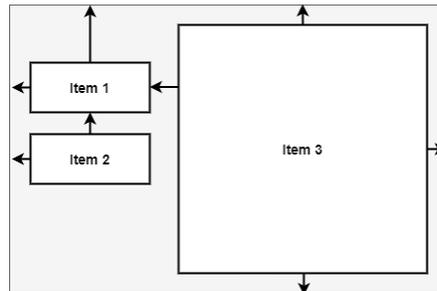


Figure 32-11

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

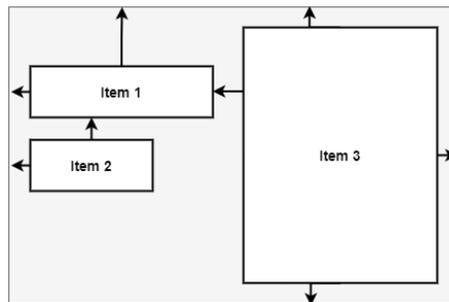


Figure 32-12

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

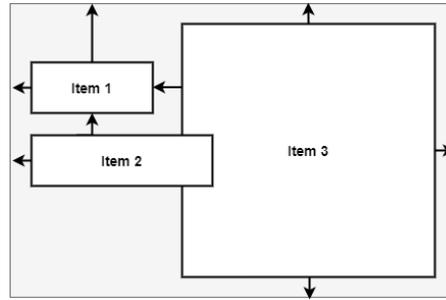


Figure 32-13

Because Item 3 is only constrained by Item 1, it does not resize to accommodate the increase in width of Item 2 causing the components to overlap.

A solution to this problem is to add a vertical barrier and assign Items 1 and 2 as the barrier's *reference components* so that they control the barrier position. The left-hand edge of Item 3 will then be constrained in relation to the barrier, making it a *constrained component*.

Now when either Item 1 or Item 2 increase in width, the barrier will move to accommodate the widest of the two components, causing the width of Item 3 to change in relation to the new barrier position:

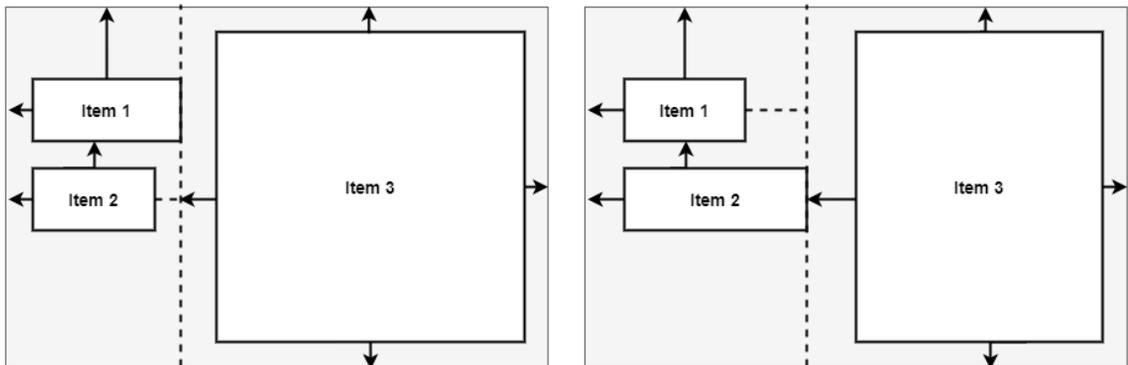


Figure 32-14

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

32.6 Summary

ConstraintLayout is a layout manager introduced with Android 7 and is now available for use within Compose layouts. It is designed to ease the creation of flexible layouts that adapt to the size and orientation of the many Android devices now on the market. ConstraintLayout uses constraints to control the alignment and positioning of components in relation both to each other and to the parent ConstraintLayout instance, guidelines, and barriers. ConstraintLayout provides an alternative when desired layout behavior cannot be achieved using the standard Compose layout techniques.

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()
    }
}
```

```
}
```

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:

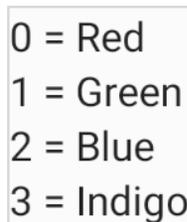
```
LazyColumn {  
    items(1000) { index ->  
        Text("This is item $index");  
    }  
}
```

`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)  
    }  
})  
.br/>{
```

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:

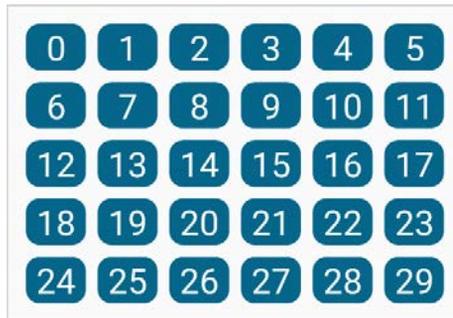


Figure 36-3

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:

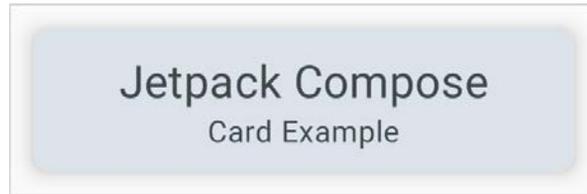


Figure 36-5

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:

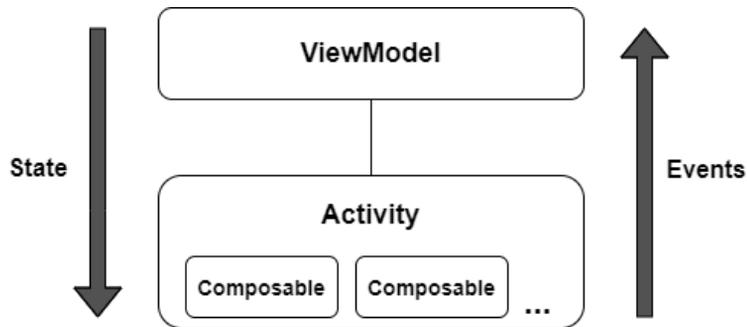


Figure 45-1

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.

48. Room Databases and Compose

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

48.1 Revisiting modern app architecture

The chapter entitled “*Working with ViewModels in Compose*” introduced the concept of modern app architecture and stressed the importance of separating different areas of responsibility within an app. The diagram illustrated in Figure 48-1 outlines the recommended architecture for a typical Android app:

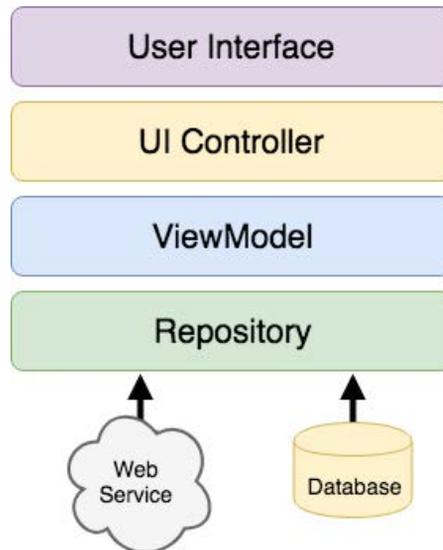


Figure 48-1

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

48.2 Key elements of Room database persistence

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

48.2.1 Repository

The repository module contains all of the code necessary for directly handling all data sources used by the app. This avoids the need for the UI controller and ViewModel to include code directly accessing sources such as

databases or web services.

48.2.2 Room database

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

48.2.3 Data Access Object (DAO)

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

48.2.4 Entities

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

48.2.5 SQLite database

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

The architecture diagram in Figure 48-2 illustrates how these different elements interact to provide Room-based database storage within an Android app:

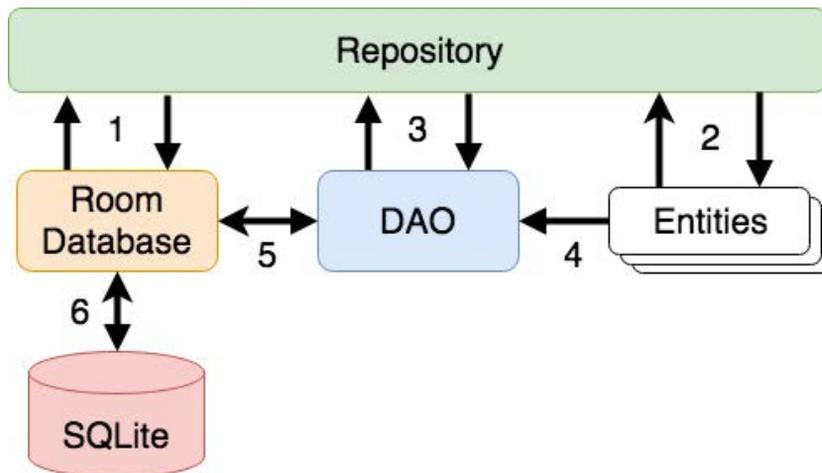


Figure 48-2

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

1. The repository interacts with the Room Database to get a database instance which, in turn, is used to obtain references to DAO instances.

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

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

48.3 Understanding entities

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

```
class Customer {

    var id: Int = 0
    var name: String? = null
    var address: String? = null

    constructor() {}

    constructor(id: Int, name: String, address: String) {
        this.id = id
        this.name = name
        this.address = address
    }

    constructor(name: String, address: String) {
        this.name = name
        this.address = address
    }
}
```

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

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

    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "customerId")
```

Room Databases and Compose

```
var id: Int = 0

@ColumnInfo(name = "customerName")
var name: String? = null
var address: String? = null

constructor() {}

constructor(id: Int, name: String, address: String) {
    this.id = id
    this.name = name
    this.address = address
}

constructor(name: String, address: String) {
    this.name = name
    this.address = address
}
}
```

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

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

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

```
@PrimaryKey(autoGenerate = true)
@NonNull
@ColumnInfo(name = "customerId")
var id: Int = 0
```

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

```
@Ignore
var MyString: String? = null
```

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

```
@Entity(foreignKeys = arrayOf(ForeignKey(entity = Customer::class,
    parentColumns = arrayOf("customerId"),
    childColumns = arrayOf("buyerId"),
    onDelete = ForeignKey.CASCADE,
```

```

        onUpdate = ForeignKey.RESTRICT)))

class Purchase {

    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "purchaseId")
    var purchaseId: Int = 0

    @ColumnInfo(name = "buyerId")
    var buyerId: Int = 0

    .
    .
}

```

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

48.4 Data Access Objects

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

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

```

@Dao
interface CustomerDao {
}

```

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

```

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

```

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

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

```

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

```

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

Room Databases and Compose

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

```
@Insert
fun addCustomer(Customer customer)
```

This is referred to as a convenience annotation because the Room persistence library can infer that the `Customer` entity passed to the `addCustomer()` method is to be inserted into the database without needing the SQL insert statement to be provided. Multiple database records may also be inserted in a single transaction as follows:

```
@Insert
fun insertCustomers(Customer... customers)
```

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

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

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

```
@Delete
fun deleteCustomers(Customer... customers)
```

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

```
@Update
fun updateCustomers(Customer... customers)
```

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

```
@Delete
fun deleteCustomers(Customer... customers): int
```

48.5 The Room database

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

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

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

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [(Customer::class)], version = 1)
abstract class CustomerRoomDatabase: RoomDatabase() {

    abstract fun customerDao(): CustomerDao
```

```

companion object {

    private var INSTANCE: CustomerRoomDatabase? = null

    fun getInstance(context: Context): CustomerRoomDatabase {
        synchronized(this) {
            var instance = INSTANCE

            if (instance == null) {
                instance = Room.databaseBuilder(
                    context.applicationContext,
                    CustomerRoomDatabase::class.java,
                    "customer_database"
                ).fallbackToDestructiveMigration()
                    .build()

                INSTANCE = instance
            }
            return instance
        }
    }
}

```

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

48.6 The Repository

The repository contains the code that makes calls to DAO methods to perform database operations. An example repository might be partially implemented as follows:

```

class CustomerRepository(private val customerDao: CustomerDao) {

    private val coroutineScope = CoroutineScope(Dispatchers.Main)
    .
    .

    fun insertCustomer(customer: Customer) {
        coroutineScope.launch(Dispatchers.IO) {
            customerDao.insertCustomer(customer)
        }
    }

    fun deleteCustomer(name: String) {
        coroutineScope.launch(Dispatchers.IO) {
            customerDao.deleteCustomer(name)
        }
    }
}

```

```

        }
    }
}

```

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

```

val allCustomers: LiveData<List<Customer>>?
customerDao.getAllCustomers()

```

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

```

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

```

Since some database transactions may take a longer time to complete, running the operations on a separate thread avoids the app appearing to lock up. As will be demonstrated in the chapter entitled “*A Compose Room Database and Repository Tutorial*”, we can easily resolve this problem using coroutines.

With all of the classes declared, instances of the database, DAO, and repository need to be created and initialized, the code for which might read as follows:

```

private val repository: CustomerRepository
val customerDb = CustomerRoomDatabase.getInstance(application)
val customerDao = customerDb.customerDao()
repository = CustomerRepository(customerDao)

```

48.7 In-Memory databases

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

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

```

// Create a file storage-based database
instance = Room.databaseBuilder(
    context.applicationContext,
    CustomerRoomDatabase::class.java,
    "customer_database"
).fallbackToDestructiveMigration()
    .build()

// Create an in-memory database
instance = Room.inMemoryDatabaseBuilder(
    context.applicationContext,

```

```

        CustomerRoomDatabase::class.java,
    ).fallbackToDestructiveMigration()
    .build()

```

48.8 Database Inspector

Android Studio includes a Database Inspector tool window which allows the Room databases associated with running apps to be viewed, searched, and modified, as shown in Figure 48-3:

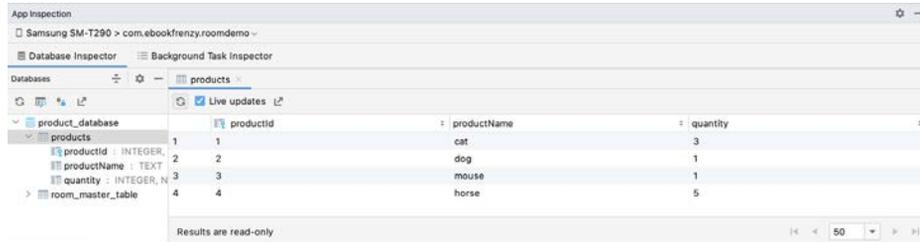


Figure 48-3

Use of the Database Inspector will be covered in the chapter entitled “A *Compose Room Database and Repository Tutorial*”.

48.9 Summary

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

With the basics of SQLite and the Room architecture component covered, the next step is to create an example app that puts this theory into practice.

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:

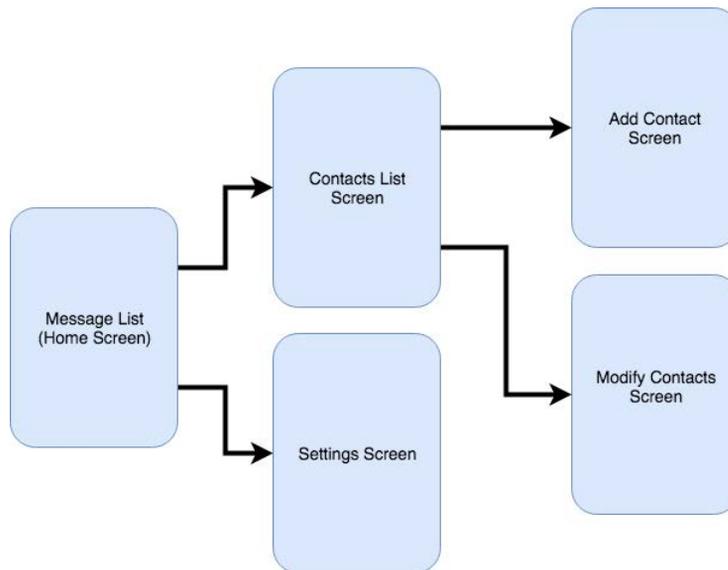


Figure 50-1

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:

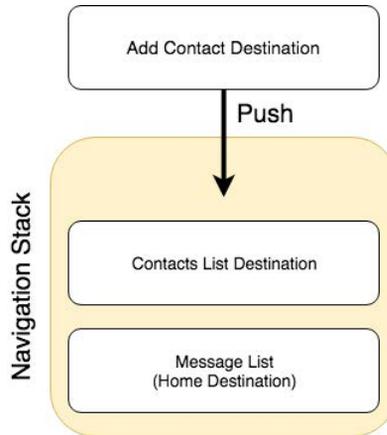


Figure 50-2

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:

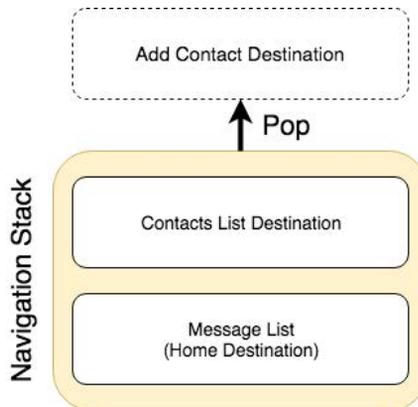


Figure 50-3

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 where 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:

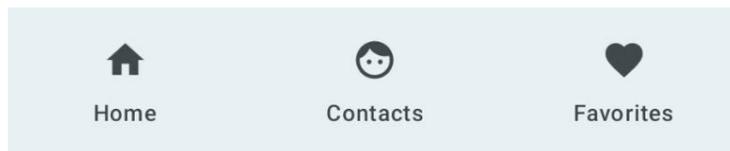


Figure 50-5

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.

59. 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.

59.1 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.

59.2 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 59-1 below:

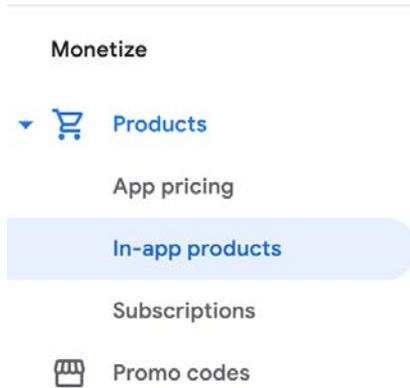


Figure 59-1

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.

59.3 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()

```

59.4 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
    }
})

```

59.5 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.

59.6 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.

59.7 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
    }
}

```

59.8 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
}
```

59.9 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.

63. An Overview of Gradle in Android Studio

Up until this point, it has been taken for granted that Android Studio will take the necessary steps to compile and run the application projects that have been created. Android Studio has been achieving this in the background using a system known as *Gradle*.

It is time to look at how Gradle is used to compile and package an application project's various elements and begin exploring how to configure this system when more advanced requirements are needed for building projects in Android Studio.

63.1 An overview of Gradle

Gradle is an automated build toolkit that allows how projects are built to be configured and managed through a set of build configuration files. This includes defining how a project will be built, what dependencies need to be fulfilled to build successfully, and what the build process's end result (or results) should be.

The strength of Gradle lies in the flexibility that it provides to the developer. The Gradle system is a self-contained, command-line-based environment that can be integrated into other environments using plugins. In the case of Android Studio, Gradle integration is provided through the appropriately named Android Studio Plugin.

Although the Android Studio Plug-in allows Gradle tasks to be initiated and managed from within Android Studio, the Gradle command-line wrapper can still be used to build Android Studio-based projects, including on systems on which Android Studio is not installed.

The configuration rules to build a project are declared in Gradle build files and scripts based on the Groovy programming language.

63.2 Gradle and Android Studio

Gradle brings many powerful features to building Android application projects. Some of the key features are as follows:

63.2.1 Sensible defaults

Gradle implements a concept referred to as *convention over configuration*. This means that Gradle has a predefined set of sensible default configuration settings that will be used unless settings in the build files override them. This means that builds can be performed with the minimum configuration required by the developer. Changes to the build files are only needed when the default configuration does not meet your build needs.

63.2.2 Dependencies

Another key area of Gradle functionality is that of dependencies. Consider, for example, a module within an Android Studio project which triggers an intent to load another module in the project. The first module has, in effect, a dependency on the second module since the application will fail to build if the second module cannot be located and launched at runtime. This dependency can be declared in the Gradle build file for the first module so that the second module is included in the application build, or an error flagged if the second module cannot be found or built. Other examples of dependencies are libraries and JAR files on which the project depends to compile and run.

Gradle dependencies can be categorized as *local* or *remote*. A local dependency references an item that is present on the local file system of the computer system on which the build is being performed. A remote dependency refers to an item that is present on a remote server (typically referred to as a *repository*).

Remote dependencies are handled for Android Studio projects using another project management tool named *Maven*. If a remote dependency is declared in a Gradle build file using Maven syntax, then the dependency will be downloaded automatically from the designated repository and included in the build process. The following dependency declaration, for example, causes the AppCompat library to be added to the project from the Google repository:

```
implementation("androidx.appcompat:appcompat:1.6.1")
```

63.2.3 Build variants

In addition to dependencies, Gradle also provides *build variant* support for Android Studio projects. This allows multiple variations of an application to be built from a single project. Android runs on many different devices encompassing a range of processor types and screen sizes. To target as wide a range of device types and sizes as possible, it will often be necessary to build several variants of an application (for example, one with a user interface for phones and another for tablet-sized screens). Through the use of Gradle, this is now possible in Android Studio.

63.2.4 Manifest entries

Each Android Studio project has associated with it an *AndroidManifest.xml* file containing configuration details about the application. Several manifest entries can be specified in Gradle build files which are then auto-generated into the manifest file when the project is built. This capability complements the build variants feature, allowing elements such as the application version number, application ID, and SDK version information to be configured differently for each build variant.

63.2.5 APK signing

The chapter “*Creating, Testing, and Uploading an Android App Bundle*” covered creating a signed release APK file using the Android Studio environment. It is also possible to include the signing information entered through the Android Studio user interface within a Gradle build file to generate signed APK files from the command line.

63.2.6 ProGuard support

ProGuard is a tool included with Android Studio that optimizes, shrinks, and obfuscates Java byte code to make it more efficient and harder to reverse engineer (the method by which others can identify the logic of an application through analysis of the compiled Java byte code). The Gradle build files allow you to control whether or not ProGuard is run on your application when it is built.

63.3 The Property and Settings Gradle build file

The gradle build configuration consists of configuration, property, and settings files. The *gradle.properties* file, for example, contains mostly esoteric settings relating to the command-line flags used by the Java Virtual Machine (JVM), whether or not the project uses the AndroidX libraries and Kotlin coding style support. As a typical user, it is unlikely that you will need to change any of these settings in this file.

The *settings.gradle.kts* file, on the other hand, defines which online repositories are to be searched when the build system needs to download and install any additional libraries and plugins required to build the project and the project name. A typical *settings.gradle.kts* file will read as follows:

```
pluginManagement {
    repositories {
        google()
        mavenCentral()
    }
}
```

```

        gradlePluginPortal()
    }
}
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
    }
}

```

```

rootProject.name = "ThemeDemo"
include(":app")

```

As with the *gradle.properties* file, it is unlikely that changes will need to be made to this file.

63.4 The top-level Gradle build file

A completed Android Studio project contains everything needed to build an Android application and consists of modules, libraries, manifest files, and Gradle build files.

Each project contains one top-level Gradle build file. This file is listed as *build.gradle.kts* (*Project: <project name>*) and can be found in the project tool window as highlighted in Figure 63-1:

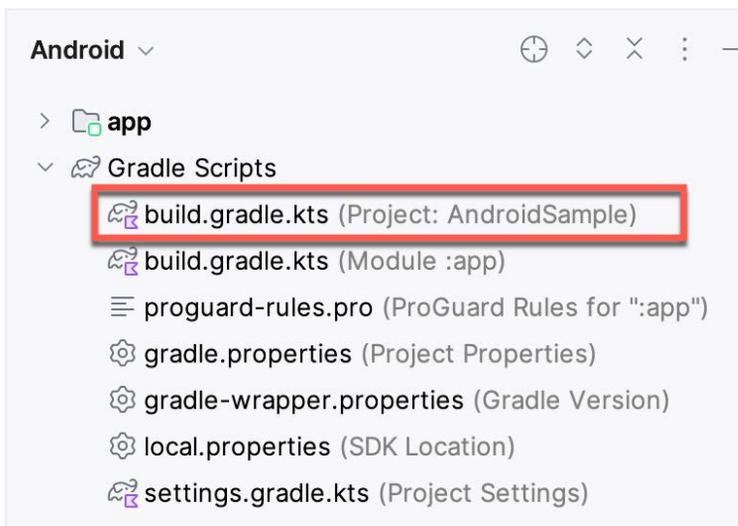


Figure 63-1

By default, the contents of the top-level Gradle build file reads as follows:

```

// Top-level build file where you can add configuration options common to all sub-
// projects/modules.
plugins {
    id("com.android.application") version "8.2.0" apply false
    id("org.jetbrains.kotlin.android") version "1.9.0" apply false
}

```

As it stands, all the file does is declare that remote libraries are to be obtained using the jcenter repository, and that builds depend on the Android plugin for Gradle. In most situations, making any changes to this build file is unnecessary.

63.5 Module level Gradle build files

An Android Studio application project is made up of one or more modules. Take, for example, a hypothetical application project named GradleDemo which contains modules named Module1 and Module2, respectively. In this scenario, each module will require its own Gradle build file. In terms of the project structure, these would be located as follows:

- Module1/build.gradle.kts
- Module2/build.gradle.kts

By default, the Module1 *build.gradle.kts* file would resemble that of the following listing:

```
plugins {
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
}

android {
    namespace = "com.example.gradlesample"
    compileSdk = 34

    defaultConfig {
        applicationId = "com.example.gradlesample"
        minSdk = 26
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }
}

compileOptions {
    sourceCompatibility = JavaVersion.VERSION_1_8
    targetCompatibility = JavaVersion.VERSION_1_8
}
```

```

kotlinOptions {
    jvmTarget = "1.8"
}
kotlinOptions {
    jvmTarget = "1.8"
}
buildFeatures {
    compose = true
}
composeOptions {
    kotlinCompilerExtensionVersion = "1.5.1"
}
packaging {
    resources {
        excludes += "/META-INF/{AL2.0,LGPL2.1}"
    }
}
}

dependencies {

    implementation("androidx.core:core-ktx:1.12.0")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.6.2")
    implementation("androidx.activity:activity-compose:1.8.2")
    implementation(platform("androidx.compose:compose-bom:2023.08.00"))
    implementation("androidx.compose.ui:ui")
    implementation("androidx.compose.ui:ui-graphics")
    implementation("androidx.compose.ui:ui-tooling-preview")
    implementation("androidx.compose.material3:material3")
    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
    androidTestImplementation(platform("androidx.compose:compose-bom:2023.08.00"))
    androidTestImplementation("androidx.compose.ui:ui-test-junit4")
    debugImplementation("androidx.compose.ui:ui-tooling")
    debugImplementation("androidx.compose.ui:ui-test-manifest")
}

```

As is evident from the file content, the build file begins by declaring the use of the Gradle Android application and Kotlin plug-ins:

```

plugins {
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
}

```

The *android* section of the file declares the project namespace and then states the version of the SDK to be used when building Module1.

```
android {
    namespace = "com.example.gradlesample"
    compileSdk = 34
```

The items declared in the *defaultConfig* section define elements to be generated into the module's *AndroidManifest.xml* file during the build. These settings, which may be modified in the build file, are taken from the settings entered within Android Studio when the module was first created:

```
defaultConfig {
    applicationId = "com.example.gradlesample"
    minSdk = 26
    targetSdk = 34
    versionCode = 1
    versionName = "1.0"

    testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
}
```

The *buildTypes* section contains instructions on whether and how to run ProGuard on the APK file when a release version of the application is built:

```
buildTypes {
    release {
        isMinifyEnabled = false
        proguardFiles(
            getDefaultProguardFile("proguard-android-optimize.txt"),
            "proguard-rules.pro"
        )
    }
}
```

As currently configured, ProGuard will not be run when Module1 is built. To enable ProGuard, the *minifyEnabled* entry must be changed from *false* to *true*. The *proguard-rules.pro* file can be found in the module directory of the project. Changes made to this file override the default settings in the *proguard-android.txt* file, which is located in the Android SDK installation directory under *sdk/tools/proguard*.

Since no debug buildType is declared in this file, the defaults will be used (built without ProGuard, signed with a debug key, and debug symbols enabled).

An additional section, entitled *productFlavors*, may also be included in the module build file to enable multiple build variants to be created.

Next, directives are included to specify the versions of the Java compiler and Kotlin Compiler extension to be used when building the project and to enable Jetpack Compose:

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
```

```

kotlinOptions {
    jvmTarget = "1.8"
}

buildFeatures {
    compose = true
}
composeOptions {
    kotlinCompilerExtensionVersion = "1.5.1"
}

```

Finally, the dependencies section lists any local and remote dependencies on which the module depends. The dependency lines in the above example file designate the Android libraries that need to be included from the Android Repository:

```

dependencies {

    implementation("androidx.core:core-ktx:1.12.0")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.6.2")
    implementation("androidx.activity:activity-compose:1.8.2")

    .
    .
}

```

Note that the dependency declarations include version numbers to indicate which library version should be included.

63.6 Configuring signing settings in the build File

The “*Creating, Testing, and Uploading an Android App Bundle*” chapter of this book covered the steps involved in setting up keys and generating a signed release APK file using the Android Studio user interface. These settings may also be declared within a *signingConfigs* section of the *build.gradle.kts* file. For example:

```

.
.
    defaultConfig {
.
.
    }
    signingConfigs {
        release {
            storeFile file("keystore.release")
            storePassword "your keystore password here"
            keyAlias "your key alias here"
            keyPassword "your key password here"
        }
    }
    buildTypes {
.
.

```

```
.  
}
```

The above example embeds the key password information directly into the build file. An alternative to this approach is to extract these values from system environment variables:

```
signingConfigs {  
    release {  
        storeFile file("keystore.release")  
        storePassword System.getenv("KEYSTOREPASSWD")  
        keyAlias "your key alias here"  
        keyPassword System.getenv("KEYPASSWD")  
    }  
}
```

Yet another approach is to configure the build file so that Gradle prompts for the passwords to be entered during the build process:

```
signingConfigs {  
    release {  
        storeFile file("keystore.release")  
        storePassword System.console().readLine  
            ("\\nEnter Keystore password: ")  
        keyAlias "your key alias here"  
        keyPassword System.console().readLine("\\nEnter Key password: ")  
    }  
}
```

63.7 Running Gradle tasks from the command line

Each Android Studio project contains a Gradle wrapper tool to invoke Gradle tasks from the command line. This tool is located in the root directory of each project folder. While this wrapper is executable on Windows systems, it may need to have execute permission enabled on Linux and macOS before it can be used. To enable execute permission, open a terminal window, change directory to the project folder for which the wrapper is needed, and execute the following command:

```
chmod +x gradlew
```

Once the file has execute permissions, the location of the file will either need to be added to your \$PATH environment variable or the name prefixed by ./ to run. For example:

```
./gradlew tasks
```

Gradle views project building in terms of several different tasks. A full listing of tasks that are available for the current project can be obtained by running the following command from within the project directory (remembering to prefix the command with a ./ if running on macOS or Linux):

```
gradlew tasks
```

To build a debug release of the project suitable for device or emulator testing, use the assembleDebug option:

```
gradlew assembleDebug
```

Alternatively, to build a release version of the application:

```
gradlew assembleRelease
```

63.8 Summary

For the most part, Android Studio performs application builds in the background without any intervention from the developer. This build process is handled using the Gradle system, an automated build toolkit designed to allow how projects are built to be configured and managed through a set of build configuration files. While the default behavior of Gradle is adequate for many basic project build requirements, the need to configure the build process is inevitable with more complex projects. This chapter has provided an overview of the Gradle build system and configuration files within the context of an Android Studio project.

Index

Symbols

- ?. 107
- 2D graphics 383
- @Composable 24, 151
- @ExperimentalFoundationApi 328
- :: operator 109
- @Preview 25
 - showSystemUi 25

A

- acknowledgePurchase() method 563
- Activity Manager 95
- adb
 - command-line tool 73
 - connection testing 79
 - device pairing 77
 - enabling on Android devices 73
 - Linux configuration 76
 - list devices 73
 - macOS configuration 74
 - overview 73
 - restart server 74
 - testing connection 79
 - WiFi debugging 77
 - Windows configuration 75
 - Wireless debugging 77
 - Wireless pairing 77
- AlertDialog 155
- align() 229
- alignByBaseline() 221
- Alignment.Bottom 215, 219
- Alignment.BottomCenter 227
- Alignment.BottomEnd 227
- Alignment.BottomStart 227
- Alignment.Center 227
 - Alignment.CenterEnd 227
 - Alignment.CenterHorizontally 215
 - Alignment.CenterStart 227
 - Alignment.CenterVertically 215, 219
 - Alignment.End 215
 - alignment lines 251
 - Alignment.Start 215
 - Alignment.Top 215, 219
 - Alignment.TopCenter 227
 - Alignment.TopEnd 227
 - Alignment.TopStart 227
- Android
 - architecture 93
 - runtime 94
 - SDK Packages 6
- Android Architecture Components 403
- Android Debug Bridge. *See* ADB
- Android Development
 - System Requirements 3
- android.hardware.camera 521
- Android Jetpack 403
- Android Libraries 94
- Android Monitor tool window 46
- Android Native Development Kit 95
- 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 93
- Android Studio
 - Animation Inspector 381
 - Asset Studio 186

Index

- changing theme 71
- Database Inspector 145
- downloading 3
- Editor Window 66
- installation 4
- Layout Editor 149
- Linux installation 5
- macOS installation 4
- Navigation Bar 65
- Project tool window 66
- setup wizard 5
- Status Bar 66
- Toolbar 65
- Tool window bars 66
- tool windows 66
- updating 12
- Welcome Screen 63
- Windows installation 4
- Android Support Library, 403
- Android Virtual Device. *See* AVD
 - overview 39
- Android Virtual Device Manager 39
- AndroidX libraries 592
- animate as state functions 367
- animateColorAsState() function 367, 371, 373
- animateDpAsState() function 373, 378
- AnimatedVisibility 355
 - animation specs 359
 - enter and exit animations 358
 - expandHorizontally() 358
 - expandIn() 358
 - expandVertically() 358
 - fadeIn() 358
 - fadeOut() 359
 - MutableTransitionState 363
 - scaleIn() 359
 - scaleOut() 359
 - shrinkHorizontally() 359
 - shrinkOut() 359
 - shrinkVertically() 359
 - slideIn() 359
 - slideInHorizontally() 359
 - slideInVertically() 359
 - slideOut() 359
 - slideOutHorizontally() 359
 - slideOutVertically() 359
- animateEnterExit() modifier 362
- animateFloatAsState() function 368
- animateScrollTo() function 304, 315
- animateScrollToItem(index: Int) 304
- animateScrollTo(value: Int) 303
- Animation
 - auto-starting 362
 - combining animations 378
 - inspector 381
 - keyframes 377
 - KeyframesSpec 377
 - motion 373
 - spring effects 376
 - state-based 367
 - visibility 355
- Animation damping
 - DampingRatioHighBouncy 376
 - DampingRatioLowBouncy 376
 - DampingRatioMediumBouncy 376
 - DampingRatioNoBouncy 376
- Animation Inspector 381
- AnimationSpec 359
 - tween() function 360
- Animation specs 359
- Animation stiffness
 - StiffnessHigh 377
 - StiffnessLow 377
 - StiffnessMedium 377
 - StiffnessMediumLow 377
 - StiffnessVeryLow 377
- annotated strings 203, 399
 - append function 203
 - buildAnnotatedString function 203
 - ParagraphStyle 204
 - SpanStyle 203
- API Key 531
- APK analyzer 556
- APK file 549

- APK File
 - analyzing 556
- APK Signing 592
- APK Wizard dialog 548
- App Bundles 545
 - creating 549
 - overview 545
 - revisions 555
 - uploading 552
- append function 203
- App Inspector 67
- Application
 - stopping 46
- Application Framework 95
- Arrangement.Bottom 217
- Arrangement.Center 216, 217
- Arrangement.End 216
- Arrangement.SpaceAround 218
- Arrangement.SpaceBetween 218
- Arrangement.SpaceEvenly 218
- Arrangement.Start 216
- Arrangement.Top 217
- ART 94
- as 109
- as? 109
- asFlow() builder 497
- Asset Studio 186
- asSharedFlow() 508
- asStateFlow() 507
- async 295
- AVD
 - Change posture 61
 - cold boot 58
 - command-line creation 39
 - creation 39
 - device frame 50
 - Display mode 60
 - launch in tool window 50
 - overview 39
 - quickboot 58
 - Resizable 60
 - running an application 42
 - Snapshots 57
 - standalone 47
 - starting 41
 - Startup size and orientation 42
- B**
 - background modifier 200
 - barriers 281
 - Barriers 266
 - constrained views 266
 - baseline
 - alignment 219
 - baselines 253
 - BaseTextField 154
 - BillingClient 564
 - acknowledgePurchase() method 563
 - consumeAsync() method 563
 - getPurchaseState() method 562
 - initialization 560, 569
 - launchBillingFlow() method 562
 - queryProductDetailsAsync() method 561
 - queryPurchasesAsync() method 564
 - startConnection() method 561
 - BillingResult 577
 - getDebugMessage() 577
 - Bill of Materials. *See* BOM
 - Biometric Authentication 519
 - callbacks 523
 - overview 519
 - tutorial 519
 - Biometric library 520
 - BiometricManager 519
 - BiometricPrompt 519
 - Bitwise AND 115
 - Bitwise Inversion 114
 - Bitwise Left Shift 116
 - Bitwise OR 115
 - Bitwise Right Shift 116
 - Bitwise XOR 115
 - BOM 26
 - build.gradle.kts 26
 - compose-bom 26

Index

- library version mapping 27
 - override library version 27
 - Boolean 102
 - BottomNavigation 155, 459
 - BottomNavigationBarItem 459
 - Box 154
 - align() 229
 - alignment 227
 - Alignment.BottomCenter 227
 - Alignment.BottomEnd 227
 - Alignment.BottomStart 227
 - Alignment.Center 227
 - Alignment.CenterEnd 227
 - Alignment.CenterStart 227
 - Alignment.TopCenter 227
 - Alignment.TopEnd 227
 - Alignment.TopStart 227
 - BoxScope 229
 - contentAlignment 227
 - matchParentSize() 229
 - overview 225
 - tutorial 225
 - BoxScope
 - align() 229
 - matchParentSize() 229
 - modifiers 229
 - BoxWithConstraints 154
 - Brush Text Styling 204
 - buffer() operator 502
 - buildAnnotatedString function 203
 - Build tool window 68
 - Build Variants , 68
 - tool window 68
 - Button 155
 - by keyword 160
- ## C
- CAMERA permission 521
 - CameraUpdateFactory class
 - methods 541
 - cancelAndJoin() 296
 - cancelChildren() 296
 - Canvas 154
 - DrawScope 383
 - inset() function 387
 - overview 383
 - size 383
 - Card 155
 - example 308
 - C/C++ Libraries 94
 - centerAround() function 270
 - chain head 264
 - chaining modifiers 195
 - chains 264
 - chain styles 264
 - Char 102
 - Checkbox 155, 184
 - Circle class 527
 - CircleShape 229
 - CircularProgressIndicator 155
 - clickable 200
 - clip 200
 - Clip Art 187
 - clip() modifier 229
 - CircleShape 229
 - CutCornerShape 229
 - RectangleShape 229
 - RoundedCornerShape 229
 - close() function 395
 - Code completion 84
 - Code Editor
 - basics 81
 - Code completion 84
 - Code Generation 86
 - Code Reformatting 89
 - Document Tabs 82
 - Editing area 82
 - Gutter Area 82
 - Live Templates 90
 - Splitting 84
 - Statement Completion 86
 - Status Bar 83
 - Code Generation 86
 - Code Reformatting 89

- code samples
 - download 1
- Coil
 - library 320
 - rememberImagePainter() function 321
- cold boot 58
- Cold flow 507
 - convert to hot 510
- collectLatest() operator 502
- collect() operator 498
- ColorFilter 398
- color filtering 398
- Column 154
 - Alignment.CenterHorizontally 215
 - Alignment.End 215
 - Alignment.Start 215
 - Arrangement.Bottom 217
 - Arrangement.Center 217
 - Arrangement.SpaceAround 218
 - Arrangement.SpaceBetween 218
 - Arrangement.SpaceEvenly 218
 - Arrangement.Top 217
 - Layout alignment 214
 - list 301
 - list tutorial 311
 - overview 212
 - scope 219
 - scope modifiers 219
 - spacing 218
 - tutorial 211
 - verticalArrangement 216
- Column lists 301
- ColumnScope 219
 - Modifier.align() 219
 - Modifier.alignBy() 219
 - Modifier.weight() 219
- combine() operator 506
- combining modifiers 200
- Communicating Sequential Processes 293
- Companion Objects 139
- components 151
- Composable
 - adding a 30
 - previewing 32
- Composable function
 - syntax 152
- composable functions 151
- composables
 - add modifier support 196
- Composables
 - Foundation 154
 - Material 154
- Compose
 - before 149
 - components 151
 - data-driven 150
 - declarative syntax 149
 - functions 151
 - layout overview 247
 - modifiers 193
 - overview 149
 - state 150
- compose-bom 26
- compose() method 455
- CompositionLocal
 - example 171
 - overview 169
 - state 172, 173
 - syntax 170
- compositionLocalOf() function 170
- conflate() operator 502
- constrainAs() modifier function 269
- constrain() function 285
- Constraint bias 274
- Constraint Bias 263
- ConstraintLayout 154
 - adding constraints 270
 - barriers 281
 - Barriers 266
 - basic constraints 272
 - centerAround() function 270
 - chain head 264
 - chains 264
 - chain styles 264

Index

- constrainAs() function 269
- constrain() function 285
- Constraint bias 274
- Constraint Bias 263
- Constraint margins 275
- Constraints 261
- constraint sets 284
- createEndBarrier() 281
- createHorizontalChain() 279
- createRefFor() function 285
- createRef() function 269
- createRefs() function 269
- createStartBarrier() 281
- createTopBarrier() 281
- createVerticalChain() 279
- creating chains 279
- generating references 269
- guidelines 280
- Guidelines 265
- how to call 269
- layout() modifier 286
- library 271
- linkTo() function 270
- Margins 262
- Opposing constraints 273
- Opposing Constraints 262, 276
- overview of 261
- Packed chain 265
- reference assignment 269
- Spread chain 264
- Spread inside chain 264
- Weighted chain 264
- Widget Dimensions 265
- Constraint margins 275
- constraints 256
- constraint sets 284
- consumeAsync() method 563
- ConsumeParams 572
- contentAlignment 227
- Content Provider 95
- Coroutine Builders 295
 - async 295
 - coroutineScope 295
 - launch 295
 - runBlocking 295
 - supervisorScope 295
 - withContext 295
- Coroutine Dispatchers 294
- Coroutines 304, 495
 - channel communication 297
 - coroutine scope 304
 - CoroutineScope 304
 - GlobalScope 294
 - LaunchedEffect 298
 - rememberCoroutineScope() 304
 - rememberCoroutineScope() function 294
 - SideEffect 298
 - Side Effects 298
 - Suspend Functions 294
 - suspending 296
 - ViewModelScope 294
 - vs Threads 293
 - vs. Threads 293
- coroutineScope 295
- CoroutineScope 294, 304
 - rememberCoroutineScope() 304
- createEndBarrier() 281
- createHorizontalChain() 279
- createRefFor() function 285
- createRef() function 269
- createRefs() 269
- createStartBarrier() 281
- createTopBarrier() 281
- createVerticalChain() 279
- cross axis arrangement 245
- Crossfading 363
- currentBackStackEntryAsState() method 460, 476
- Custom Accessors 137
- Custom layout 255
 - building 255
 - constraints 256
 - Layout() composable 256
 - measurables 256
 - overview 255

- Placeable 256
- syntax 255
- custom layout modifiers 247
 - alignment lines 251
 - baselines 253
 - creating 249
 - default position 249
- Custom layouts
 - overview 247
 - tutorial 247
- Custom Theme
 - building 583
- CutCornerShape 229

- D**
- DampingRatioHighBouncy 376
- DampingRatioLowBouncy 376
- DampingRatioMediumBouncy 376
- DampingRatioNoBouncy 376
- Dark Theme 46
 - enable on device 46
- dashPathEffect() method 385
- Data Access Object (DAO) 426, 438
- Data Access Objects 429
- Database Inspector 433, 451
 - live updates 451
 - SQL query 451
- Database Rows 420
- Database Schema 419
- Database Tables 419
- data-driven 150
- DDMS 46
- Debugging
 - enabling on device 73
- declarative syntax 149
- Default Function Parameters 129
- default position 249
- derivedStateOf 331
- Device File Explorer 68
- device frame 50
- Device Mirroring 79
 - enabling 79
- device pairing 77
- Dispatchers.Default 295
- Dispatchers.IO 295
- Dispatchers.Main 294
- drag gestures 484
- drawable
 - folder 186
- drawArc() function 394
- drawCircle() function 390
- drawImage() function 397
- Drawing
 - arcs 394
 - circle 390
 - close() 395
 - dashed lines 385
 - dashPathEffect() 385
 - drawArc() 394
 - drawImage() 397
 - drawPath() 395
 - drawPoints() 396
 - drawRect() 385
 - drawRoundRect() 388
 - gradients 391
 - images 397
 - line 383
 - oval 390
 - points 396
 - rectangle 385
 - rotate() 389
 - rotation 389
- Drawing text 399
- drawLine() function 384
- drawPath() function 395
- drawPoints() function 396
- drawRect() function 385
- drawRoundRect() function 388
- DrawScope 383
- drawText() function 399, 400
- DropdownMenu 155
- DROP_LATEST 509
- DROP_OLDEST 509
- DurationBasedAnimationSpec 359

Index

Dynamic colors

enabling in Android 589

E

Elvis Operator 109

emit 151

Empty Compose Activity

template 16

Emulator

battery 56

cellular configuration 56

configuring fingerprints 58

directional pad 56

extended control options 55

Extended controls 55

fingerprint 56

location configuration 56

phone settings 56

Resizable 60

resize 55

rotate 54

Screen Record 57

Snapshots 57

starting 41

take screenshot 54

toolbar 53

toolbar options 53

tool window mode 59

Virtual Sensors 57

zoom 54

enablePendingPurchases() method 563

enabling ADB support 73

enter animations 358

EnterTransition.None 362

Errata 2

Escape Sequences 103

ettings.gradle file 592

exit animations 358

ExitTransition.None 362

expandHorizontally() 358

expandIn() 358

expandVertically() 358

Extended Control

options 55

F

fadeIn() 358

fadeOut() 359

Files

switching between 82

fillMaxHeight 200

fillMaxSize 200

fillMaxWidth 200

filter() operator 500

findStartDestination() method 460

Fingerprint

emulation 58

Fingerprint authentication

device configuration 520

overview 519

steps to implement 519

tutorial 519

firstVisibleItemIndex 306

flatMapConcat() operator 505

flatMapMerge() operator 505

Float 102

FloatingActionButton 155

Flow 495

asFlow() builder 497

asSharedFlow() 508

asStateFlow() 507

backgroundn handling 515

buffering 502

buffer() operator 502

builder 497

cold 507

collect() 501

collecting data 501

collectLatest() operator 502

combine() operator 506

conflate() operator 502

emit() 497

emitting data 497

filter() operator 500

- flatMapConcat() operator 505
 - flatMapMerge() operator 505
 - flattening 504
 - flowOf() builder 497
 - flow of flows 504
 - fold() operator 504
 - hot 507
 - MutableSharedFlow 508
 - MutableStateFlow 507
 - onEach() operator 506
 - reduce() operator 503, 504
 - repeatOnLifecycle 516
 - SharedFlow 508
 - shareIn() function 510
 - single() operator 502
 - StateFlow 507
 - transform() operator 500
 - try/finally 501
 - zip() operator 506
 - flow builder 497
 - FlowColumn 233, 239, 244
 - cross axis arrangement 245
 - maxItemsInEachColumn 234
 - tutorial 239
 - Flow layout
 - arrangement 242
 - Flow layouts
 - cross axis arrangement 235
 - fillMaxHeight() 237
 - fillMaxWidth() 237
 - Fractional sizing 237
 - horizontalArrangement 245
 - Item alignment 236
 - item weights 245
 - main axis arrangement 234
 - verticalArrangement 245
 - weight 236
 - flowOf() builder 497
 - flow of flows 504
 - FlowRow 233, 239, 241
 - cross axis arrangement 245
 - horizontalArrangement 242
 - item alignment 242
 - maxItemsInEachRow 234
 - tutorial 239
 - Flows
 - combining 506
 - Introduction to 495
 - FontWeight 31
 - forEachIndexed 245
 - forEach loop 258
 - Forward-geocoding 533
 - Foundation components 154
 - Foundation Composables 154
 - Foundation libraries 346
 - Foundation library 239
 - FragmentActivity 521
 - Function Parameters
 - variable number of 129
 - Functions 127
- ## G
- Geocoder object 534
 - Geocoding 533
 - Gestures 481
 - click 481
 - drag 484
 - horizontalScroll() 488
 - overview 481
 - pinch gestures 490
 - PointerInputScope 483
 - rememberScrollableState() function 487
 - rememberScrollState() 488
 - rememberTransformableState() 490
 - rotation gestures 491
 - scrollable() modifier 487
 - scroll modifiers 488
 - taps 483
 - translation gestures 492
 - tutorial 481
 - verticalScroll() 488
 - getDebugMessage() 577
 - getFromLocation() method 534
 - getPurchaseState() method 562

Index

getStringArray() method 319

GlobalScope 294

GNU/Linux 94

Google Cloud

 billing account 528

 new project 529

GoogleMap 527

Google Maps Android API 527

 Controlling the Map Camera 541

 displaying controls 537

 Map Markers 540

 overview 527

Google Maps SDK 527

 API Key 531

 Credentials 530

 enabling 530

 Maps SDK for Android 530

Google Play App Signing 548

Google Play Billing Library 559

Google Play Console 566

 Creating an in-app product 566

 License Testers 567

Google Play Developer Console 546

Google Play store 17

Gradient drawing 391

Gradle

 APK signing settings 597

 Build Variants 592

 command line tasks 598

 dependencies 591

 Manifest Entries 592

 overview 591

 sensible defaults 591

Gradle Build File

 top level 593

Gradle Build Files

 module level 594

gradle.properties file 592

Graphics

 drawing 383

Grid

 overview 301

groupBy() function 305

guidelines 280

H

Higher-order Functions 131

horizontalArrangement 216, 218, 245

HorizontalPager 343

 animateScrollToPage() 345

 scrollToPage() 345

 state 344

 syntax 343

horizontalScroll() 488

Hot flows 507

I

Image 154

 add drawable resource 186

 painterResource method 188

Immutable Variables 104

INAPP 564

In-App Products 559

In-App Purchasing 565

 acknowledgePurchase() method 563

 BillingClient 560

 BillingResult 577

 consumeAsync() method 563

 ConsumeParams 572

 Consuming purchases 572

 enablePendingPurchases() method 563

 getPurchaseState() method 562

 Google Play Billing Library 559

 launchBillingFlow() method 562

 Libraries 565

 newBuilder() method 560

 onBillingServiceDisconnected() callback 570

 onBillingServiceDisconnected() method 561

 onBillingSetupFinished() listener 570

 onProductDetailsResponse() callback 570

 Overview 559

 ProductDetail 562

 ProductDetails 571

 products 559

- ProductType 564
- Purchase Flow 571
- PurchaseResponseListener 564
- PurchasesUpdatedListener 562
- PurchaseUpdatedListener 571
- purchase updates 571
- queryProductDetailsAsync() 570
- queryProductDetailsAsync() method 561
- queryPurchasesAsync() 572
- queryPurchasesAsync() method 564
- startConnection() method 561
- subscriptions 559
- tutorial 565
- Initializer Blocks 137
- In-Memory Database 432
- Inner Classes 138
- inset() function 387
- IntrinsicSize.Max 291
- IntrinsicSize.Min 291, 292
- intelligent recomposition 157
- IntelliJ IDEA 97
- Interactive mode 36
- Intrinsic measurements 287
- IntrinsicSize 287
 - intrinsic measurements 287
 - Max 287
 - Min 287
 - tutorial 289
- is 109
- isInitialized property 109
- isSystemInDarkTheme() function 172
- item() function 302
- items() function 302
- itemsIndexed() function 302

J

- Java
 - convert to Kotlin 97
- Java Native Interface 95
- JetBrains 97
- Jetpack Compose
 - see Compose 149

- join() 296

K

- keyboardOptions 415
- Keyboard Shortcuts 69
- keyframe 360
- keyframes 377
 - KeyframesSpec 377
- keyframes() function 377
- KeyframesSpec 377
- Keystore File
 - creation 548
- Kotlin
 - accessing class properties 137
 - and Java 97
 - arithmetic operators 111
 - assignment operator 111
 - augmented assignment operators 112
 - bitwise operators 114
 - Boolean 102
 - break 122
 - breaking from loops 121
 - calling class methods 137
 - Char 102
 - class declaration 133
 - class initialization 134
 - class properties 134
 - Companion Objects 139
 - conditional control flow 123
 - continue labels 122
 - continue statement 122
 - control flow 119
 - convert from Java 97
 - Custom Accessors 137
 - data types 101
 - decrement operator 112
 - Default Function Parameters 129
 - defining class methods 134
 - do ... while loop 121
 - Elvis Operator 109
 - equality operators 113
 - Escape Sequences 103

Index

- expression syntax 111
- Float 102
- Flow 495
- for-in statement 119
- function calling 128
- Functions 127
- groupBy() function 305
- Higher-order Functions 131
- if ... else ... expressions 124
- if expressions 123
- Immutable Variables 104
- increment operator 112
- inheritance 143
- Initializer Blocks 137
- Inner Classes 138
- introduction 97
- Lambda Expressions 130
- let Function 107
- Local Functions 128
- logical operators 113
- looping 119
- Mutable Variables 104
- Not-Null Assertion 107
- Nullable Type 106
- Overriding inherited methods 146
- playground 98
- Primary Constructor 134
- properties 137
- range operator 114
- Safe Call Operator 106
- Secondary Constructors 134
- Single Expression Functions 128
- String 102
- subclassing 143
- subStringBefore() method 321
- Type Annotations 105
- Type Casting 109
- Type Checking 109
- Type Inference 105
- variable parameters 129
- when statement 124
- while loop 120

L

- Lambda Expressions 130
- lateinit 108
- Late Initialization 108
- launch 295
- launchBillingFlow() method 562
- LaunchedEffect 298
- launchSingleTop 457
- Layout alignment 214
- Layout arrangement 216
- Layout arrangement spacing 218
- Layout components 154
- Layout() composable 256
- Layout Editor 149
- Layout Inspector 68
- layout modifier 200
- layout() modifier 286
- LazyColumn 154, 301
 - creation 302
 - scroll position detection 306
- LazyHorizontalStaggeredGrid 335, 340
 - syntax 336
- LazyList
 - tutorial 317
- Lazy lists 301
 - Scrolling 303
- LazyListScope 302
 - item() function 302
 - items() function 302
 - itemsIndexed() function 302
 - stickyHeader() function 304
- LazyListState 306
 - firstVisibleItemIndex 306
- LazyRow 154, 301
 - creation 302
 - scroll position detection 306
- LazyVerticalGrid 301
 - adaptive mode 306
 - fixed mode 306
- LazyVerticalStaggeredGrid 335, 338
 - syntax 335
- let Function 107

- libc 94
 - License Testers 567
 - Lifecycle.State.CREATED 516
 - Lifecycle.State.DESTROYED 516
 - Lifecycle.State.INITIALIZED 516
 - Lifecycle.State.RESUMED 516
 - Lifecycle.State.STARTED 516
 - LinearProgressIndicator 155
 - lineTo() 395
 - lineTo() function 395
 - linkTo() function 270
 - Linux Kernel 94
 - list devices 73
 - Lists
 - clickable items 324
 - enabling scrolling 303
 - overview 301
 - literals
 - live editing 32
 - LiveData 406
 - observeAsState() 407
 - Live Edit 43
 - disabling 32
 - enabling 32
 - of literals 32
 - Live Templates 90
 - Local Functions 128
 - Location Manager 95
 - Logcat
 - tool window 67
- M**
- MainActivity.kt file 20
 - template code 29
 - map method 256
 - Maps 527
 - MAP_TYPE_HYBRID 536
 - MAP_TYPE_NONE 536
 - MAP_TYPE_NORMAL 536
 - MAP_TYPE_SATELLITE 536
 - MAP_TYPE_TERRAIN 536
 - Marker class 527
 - matchParentSize() 229
 - Material Composables 154
 - Material Design 2 579
 - Material Design 2 Theming 579
 - Material Design 3 579
 - Material Design components 155
 - Material Theme Builder 583
 - Material You 579
 - maxValue property 315
 - measurables 256
 - measure() function 400
 - measureTimeMillis() function 502
 - Memory Indicator 83
 - Minimum SDK
 - setting 17
 - ModalDrawer 155
 - Modern Android architecture 403
 - modifier
 - adding to composable 196
 - chaining 195
 - combining 200
 - creating a 194
 - ordering 196
 - tutorial 193
 - Modifier.align() 219
 - Modifier.alignBy() 219
 - modifiers
 - build-in 200
 - overview 193
 - Modifier.weight() 219
 - move() method 541
 - multiple devices
 - testing app on 45
 - MutableLiveData 406
 - MutableSharedFlow 508
 - MutableState 158
 - MutableStateFlow 507
 - mutableStateOf function 151
 - mutableStateOf() function 159
 - MutableTransitionState 363
 - Mutable Variables 104
 - My Location Layer 527

Index

N

NavHost 455, 467, 475
NavHostController 453, 467, 475
navigate() method 457
Navigation 453

- BottomNavigation 459
- BottomNavigationItem 459
- compose() method 455
- currentBackStackEntryAsState() method 460
- declaring routes 464
- findStartDestination() method 460
- graph 455
- launchSingleTop 457
- NavHost 455, 467
- NavHostController 453, 467
- navigate() method 457
- navigation graph 453
- NavType 458
- overview 453
- passing arguments
- popUpTo() method 457
- route 455
- stack 453, 454
- start destination 455
- tutorial 463

Navigation Architecture Component 453
NavigationBar 476
NavigationBarItem 476
Navigation bars 459
navigation graph 453, 455
Navigation Host 455
NavType 458
newBuilder() method 560
Notifications Manager 95
Not-Null Assertion 107
Nullable Type 106

O

observeAsState() 407
Offset() function 384
offset modifier 200
onBillingServiceDisconnected() callback 570

onBillingServiceDisconnected() method 561
onBillingSetupFinished() listener 570
onCreate() method 24
onEach() operator 506
onProductDetailsResponse() callback 570
OpenJDK 3
Opposing constraints 273
OutlinedButton 331
OutlinedTextField 409

P

Package Manager 95
Package name 17
Packed chain 265
padding 200
Pager 343

- animateScrollToPage() 345
- scrollToPage() 345
- state 344
- syntax , 234

Pager state 344
painterResource method 188
ParagraphStyle 204
PathEffect 385
pinch gestures 490
Placeable 256
PointerInputScope 483

- drag gestures 486
- tap gestures 483

popUpTo() method 457
Preview configuration picker 35
Preview panel 25

- build and refresh 25
- Interactive mode 36
- settings 35

Primary Constructor 134
Problems

- tool window 68

ProductDetail 562
ProductDetails 571
ProductType 564
Profiler

- tool window 68
 - proguard-rules.pro file 596
 - ProGuard Support 592
 - project
 - create new 16
 - package name 17
 - Project tool window 19, 67
 - Android mode 19
 - PurchaseResponseListener 564
 - PurchasesUpdatedListener 562, 571
- ## Q
- queryProductDetailsAsync() 570
 - queryProductDetailsAsync() method 561
 - queryPurchaseHistoryAsync() method 564
 - queryPurchasesAsync() 572
 - queryPurchasesAsync() method 564
 - quickboot snapshot 58
 - Quick Documentation 89
- ## R
- RadioButton 155
 - Random
 - nextInt() 240
 - Random.nextInt() method 337, 240
 - Range Operator 114
 - Recent Files Navigation 70
 - recomposition 150
 - intelligent recomposition 157
 - overview 157
 - RectangleShape 229
 - reduce() operator 503, 504
 - relativeLineTo() function 395
 - Release Preparation 545
 - rememberCoroutineScope() function 294, 304, 313
 - rememberDraggableState() function 484
 - rememberImagePainter() function 321
 - remember keyword 159
 - rememberPagerState 344
 - rememberSaveable keyword 166
 - rememberScrollableState() function 487
 - rememberScrollState() 488
 - rememberScrollState() function 303, 313
 - rememberTextMeasurer() function 399
 - rememberTransformableState() 490
 - rememberTransformationState() function 490
 - repeatable() function 361
 - RepeatableSpec
 - repeatable() 361
 - RepeatMode.Reverse 361
 - repeatOnLifecycle 516
 - Repository
 - tutorial 435
 - Resizable Emulator 60
 - Resource Manager 95, 67
 - Reverse-geocoding 533
 - Reverse Geocoding 533
 - Room
 - Data Access Object (DAO) 426
 - entities 426, 427
 - In-Memory Database 432
 - Repository 425
 - Room Database 426
 - tutorial 435
 - Room Database Persistence 425
 - Room persistence library 436
 - Room Persistence Library 423
 - rotate modifier 200
 - rotation gestures 491
 - RoundedCornerShape 229
 - Row 154
 - Alignment.Bottom 215
 - Alignment.CenterVertically 215
 - Alignment.Top 215
 - Arrangement.Center 216
 - Arrangement.End 216
 - Arrangement.SpaceAround 218
 - Arrangement.SpaceBetween 218
 - Arrangement.SpaceEvenly 218
 - Arrangement.Start 216
 - horizontalArrangement 216
 - Layout alignment 214
 - Layout arrangement 216
 - list 301

Index

- list example 316
- overview 212
- scope 219
- scope modifiers 219
- spacing 218
- tutorial 211
- Row lists 301
- RowScope 219
 - Modifier.align() 219
 - Modifier.alignBy() 219
 - Modifier.alignByBaseline() 219
 - Modifier.paddingFrom() 220
 - Modifier.weight() 220
- Run
 - tool window 67
- runBlocking 295
- Running Devices
 - tool window 79
- S**
- Safe Call Operator 106
- Scaffold 155, 477
 - bottomBar 478
 - TopAppBar 478
- scaleIn() 359
- scale modifier 200
- scaleOut() 359
- Scope modifiers
 - weights 223
- scrollable modifier 200
- scrollable() modifier 487, 488
- Scroll detection
 - example 327
- scroll modifiers 488
- ScrollState
 - maxValue property 315
 - rememberScrollState() function 303
- scrollToItem(index: Int) 304
- scrollToPage() 345
- scrollTo(value: Int) 303
- SDK Packages 6
- SDK settings 17
- Secondary Constructors 134
- Secure Sockets Layer (SSL) 94
- settings.gradle.kts file 592
- Shape 155
- Shapes
 - CircleShape 229
 - CutCornerShape 229
 - RectangleShape 229
 - RoundedCornerShape 229
- SharedFlow 508, 511
 - backgroundn handling 515
 - DROP_LATEST 509
 - DROP_OLDEST 509
 - in ViewModel 512
 - repeatOnLifecycle 516
 - SUSPEND 509
 - tutorial 511
- shareIn() function 510
- SharingStarted.Eagerly() 510
- SharingStarted.Lazily() 510
- SharingStarted.WhileSubscribed() 510
- showSystemUi 25, 312
- shrinkHorizontally() 359
- shrinkOut() 359
- shrinkVertically() 359
- SideEffect 298
- Side Effects 298
- single() operator 502
- size modifier 200
- slideIn() 359
- slideInHorizontally() 359
- slideInVertically() 359
- slideOut() 359
- slideOutHorizontally() 359
- slideOutVertically() 359
- Slider 155
- Slider component 33
- Slot APIs
 - calling 178
 - declaring 178
 - overview 177
 - tutorial 181

- Snackbar 155
 - Snapshots
 - emulator 57
 - SpanStyle 203
 - Spread chain 264
 - Spread inside chain 264
 - Spring effects 376
 - spring() function 376
 - SQL 420
 - SQLite 419
 - AVD command-line use 421
 - Columns and Data Types 419
 - overview 420
 - Primary keys 420
 - Staggered Grids 335
 - startConnection() method 561
 - start destination 455
 - state 150
 - basics of 157
 - by keyword 160
 - configuration changes 165
 - declaring 158
 - hoisting 163
 - MutableState 158
 - mutableStateOf() function 159
 - overview 157
 - remember keyword 159
 - rememberSaveable 166
 - Unidirectional data flow 161
 - StateFlow 507
 - stateful 157
 - stateful composables 151
 - State hoisting 163
 - stateless composables 151
 - Statement Completion 86
 - staticCompositionLocalOf() function 170, 172
 - Status Bar Widgets 83
 - Memory Indicator 83
 - stickyHeader 328
 - stickyHeader() function 304
 - Sticky headers
 - adding 328
 - example 327
 - stickyHeader() function 304
 - StiffnessHigh 377
 - StiffnessLow 377
 - StiffnessMedium 377
 - StiffnessMediumLow 377
 - StiffnessVeryLow 377
 - String 102
 - Structure
 - tool window 68
 - Structured Query Language 420
 - Structure tool window 68
 - SUBS 564
 - subscriptions 559
 - substringBefore() method 321
 - supervisorScope 295
 - Surface component 23, 227
 - SUSPEND 509
 - Suspend Functions 294
 - Switch 155
 - Switcher 70
 - system requirements 3
- ## T
- Telephony Manager 95
 - Terminal
 - tool window 68
 - Text 155
 - Text component 152
 - TextField 155
 - TextMeasurer 399
 - measure() function 400
 - TextStyle 416
 - Theme
 - building a custom 583
 - Theming 579
 - tutorial 585
 - TODO
 - tool window 69
 - Tool window bars 66
 - Tool windows 66
 - TopAppBar 155, 478

Index

trailingIcon 416
TransformableState 490
transform() operator 500
translation gestures 492
try/finally 501
tween() function 360
Type Annotations 105
Type Casting 109
Type Checking 109
Type Inference 105
Type.kt file 582

U

UI Controllers 404
UI_NIGHT_MODE_YES 173
UiSettings class 527
Unidirectional data flow 161
updateTransition() function 368, 373, 378
upload key 548
USB connection issues
 resolving 76
USE_BIOMETRIC permission 521

V

Vector Asset
 add to project 186
verticalArrangement 216, 218
VerticalPager
 animateScrollToPage() 345
 scrollToPage() 345
 state 344
 syntax , 234
verticalScroll() 488
verticalScroll() modifier 313
ViewModel
 example 410
 lifecycle library 406, 410, 496, 511
 LiveData 406
 observeAsState() 407
 overview 403
 tutorial 409
 using state 404

viewModel() 406, 412, 446
 ViewModelProvider Factory 445
 ViewModelStoreOwner 446
viewModel() function 406, 412, 446
ViewModelProvider Factory 445
ViewModelScope 294
ViewModelStoreOwner 446
View System 95
Virtual Device Configuration dialog 40
Virtual Sensors 57
Visibility animation 355

W

Weighted chain 264
Welcome screen 63
while Loop 120
Widget Dimensions 265
WiFi debugging 77
Wireless debugging 77
Wireless pairing 77
withContext 295

X

XML resource
 reading an 317

Z

zip() operator 506