

Jetpack Compose 1.3 Essentials

Jetpack Compose 1.3 Essentials

ISBN-13: 978-1-951442-63-7

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

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

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

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

Rev: 1.0

Table of Contents

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	10
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	12
3. A Compose Project Overview	13
3.1 About the project.....	13
3.2 Creating the project	14
3.3 Creating an activity	14
3.4 Defining the project and SDK settings	15
3.5 Previewing the example project	16
3.6 Reviewing the main activity.....	18
3.7 Preview updates.....	22
3.8 Bill of Materials and the Compose version	22
3.9 Summary	24
4. An Example Compose Project	25
4.1 Getting started	25
4.2 Removing the template Code	25
4.3 The Composable hierarchy	26
4.4 Adding the DemoText composable	26
4.5 Previewing the DemoText composable.....	28
4.6 Adding the DemoSlider composable.....	28

Table of Contents

4.7 Adding the DemoScreen composible	29
4.8 Previewing the DemoScreen composible.....	31
4.9 Adjusting preview settings	31
4.10 Testing in interactive mode.....	32
4.11 Completing the project.....	33
4.12 Summary	34
5. Creating an Android Virtual Device (AVD) in Android Studio	35
5.1 About Android Virtual Devices	35
5.2 Starting the emulator	36
5.3 Running the application in the AVD	37
5.4 Real-time updates with Live Edit	39
5.5 Running on multiple devices	40
5.6 Stopping a running application	40
5.7 Supporting dark theme.....	41
5.8 Running the emulator in a separate window.....	41
5.9 Enabling the device frame.....	43
5.10 Summary	43
6. Using and Configuring the Android Studio AVD Emulator	45
6.1 The Emulator Environment	45
6.2 Emulator Toolbar Options	45
6.3 Working in Zoom Mode	47
6.4 Resizing the Emulator Window.....	47
6.5 Extended Control Options.....	47
6.5.1 Location.....	48
6.5.2 Displays.....	48
6.5.3 Cellular	48
6.5.4 Battery.....	48
6.5.5 Camera.....	48
6.5.6 Phone	48
6.5.7 Directional Pad.....	48
6.5.8 Microphone.....	48
6.5.9 Fingerprint	48
6.5.10 Virtual Sensors	49
6.5.11 Snapshots.....	49
6.5.12 Record and Playback	49
6.5.13 Google Play	49
6.5.14 Settings	49
6.5.15 Help.....	49
6.6 Working with Snapshots.....	49
6.7 Configuring Fingerprint Emulation	50
6.8 The Emulator in Tool Window Mode.....	51
6.9 Creating a Resizable Emulator.....	52
6.10 Summary	53
7. A Tour of the Android Studio User Interface	55
7.1 The Welcome Screen	55
7.2 The Main Window	56
7.3 The Tool Windows	57
7.4 Android Studio Keyboard Shortcuts	60

7.5 Switcher and Recent Files Navigation	61
7.6 Changing the Android Studio Theme	62
7.7 Summary	63
8. Testing Android Studio Apps on a Physical Android Device.....	65
8.1 An overview of the Android Debug Bridge (ADB)	65
8.2 Enabling USB debugging ADB on Android devices	65
8.2.1 macOS ADB configuration	66
8.2.2 Windows ADB configuration	67
8.2.3 Linux adb configuration.....	68
8.3 Resolving USB connection issues.....	68
8.4 Enabling wireless debugging on Android devices	69
8.5 Testing the adb connection	71
8.6 Summary	71
9. The Basics of the Android Studio Code Editor.....	73
9.1 The Android Studio editor	73
9.2 Code mode.....	75
9.3 Splitting the editor window.....	76
9.4 Code completion	76
9.5 Statement completion	78
9.6 Parameter information	78
9.7 Parameter name hints.....	78
9.8 Code generation	78
9.9 Code folding.....	79
9.10 Quick documentation lookup	81
9.11 Code reformatting.....	81
9.12 Finding sample code.....	82
9.13 Live templates	82
9.14 Summary	83
10. An Overview of the Android Architecture	85
10.1 The Android software stack	85
10.2 The Linux kernel.....	86
10.3 Android runtime – ART	86
10.4 Android libraries	86
10.4.1 C/C++ libraries.....	86
10.5 Application framework.....	87
10.6 Applications	87
10.7 Summary	87
11. An Introduction to Kotlin.....	89
11.1 What is Kotlin?	89
11.2 Kotlin and Java.....	89
11.3 Converting from Java to Kotlin	89
11.4 Kotlin and Android Studio	90
11.5 Experimenting with Kotlin	90
11.6 Semi-colons in Kotlin	91
11.7 Summary	91
12. Kotlin Data Types, Variables and Nullability	93

Table of Contents

12.1 Kotlin data types.....	93
12.1.1 Integer data types	94
12.1.2 Floating point data types.....	94
12.1.3 Boolean data type.....	94
12.1.4 Character data type.....	94
12.1.5 String data type.....	94
12.1.6 Escape sequences.....	95
12.2 Mutable variables	96
12.3 Immutable variables.....	96
12.4 Declaring mutable and immutable variables	96
12.5 Data types are objects	96
12.6 Type annotations and type inference.....	97
12.7 Nullable type.....	98
12.8 The safe call operator	98
12.9 Not-null assertion	99
12.10 Nullable types and the let function.....	99
12.11 Late initialization (lateinit)	100
12.12 The Elvis operator	101
12.13 Type casting and type checking.....	101
12.14 Summary.....	102
13. Kotlin Operators and Expressions	103
13.1 Expression syntax in Kotlin	103
13.2 The Basic assignment operator.....	103
13.3 Kotlin arithmetic operators.....	103
13.4 Augmented assignment operators	104
13.5 Increment and decrement operators	104
13.6 Equality operators	105
13.7 Boolean logical operators.....	105
13.8 Range operator	106
13.9 Bitwise operators	106
13.9.1 Bitwise inversion	106
13.9.2 Bitwise AND	107
13.9.3 Bitwise OR.....	107
13.9.4 Bitwise XOR.....	107
13.9.5 Bitwise left shift	108
13.9.6 Bitwise right shift	108
13.10 Summary.....	109
14. Kotlin Control Flow	111
14.1 Looping control flow.....	111
14.1.1 The Kotlin <i>for-in</i> Statement.....	111
14.1.2 The <i>while</i> loop	112
14.1.3 The <i>do ... while</i> loop	113
14.1.4 Breaking from Loops	113
14.1.5 The <i>continue</i> statement	114
14.1.6 Break and continue labels	114
14.2 Conditional control flow	115
14.2.1 Using the <i>if</i> expressions	115
14.2.2 Using <i>if ... else ...</i> expressions	116

14.2.3 Using <i>if ... else if ...</i> Expressions	116
14.2.4 Using the <i>when</i> statement	116
14.3 Summary	117
15. An Overview of Kotlin Functions and Lambdas	119
15.1 What is a function?	119
15.2 How to declare a Kotlin function	119
15.3 Calling a Kotlin function	120
15.4 Single expression functions	120
15.5 Local functions	120
15.6 Handling return values	121
15.7 Declaring default function parameters	121
15.8 Variable number of function parameters	121
15.9 Lambda expressions	122
15.10 Higher-order functions	123
15.11 Summary	124
16. The Basics of Object-Oriented Programming in Kotlin	125
16.1 What is an object?	125
16.2 What is a class?	125
16.3 Declaring a Kotlin class	125
16.4 Adding properties to a class	126
16.5 Defining methods	126
16.6 Declaring and initializing a class instance	126
16.7 Primary and secondary constructors	126
16.8 Initializer blocks	129
16.9 Calling methods and accessing properties	129
16.10 Custom accessors	129
16.11 Nested and inner classes	130
16.12 Companion objects	131
16.13 Summary	133
17. An Introduction to Kotlin Inheritance and Subclassing	135
17.1 Inheritance, classes, and subclasses	135
17.2 Subclassing syntax	135
17.3 A Kotlin inheritance example	136
17.4 Extending the functionality of a subclass	137
17.5 Overriding inherited methods	138
17.6 Adding a custom secondary constructor	139
17.7 Using the SavingsAccount class	139
17.8 Summary	139
18. An Overview of Compose	141
18.1 Development before Compose	141
18.2 Compose declarative syntax	141
18.3 Compose is data-driven	142
18.4 Summary	142
19. Composable Functions Overview	143
19.1 What is a composable function?	143
19.2 Stateful vs. stateless composables	143

Table of Contents

19.3 Composable function syntax	144
19.4 Foundation and Material composables	146
19.5 Summary	147
20. An Overview of Compose State and Recomposition.....	149
20.1 The basics of state	149
20.2 Introducing recomposition	149
20.3 Creating the StateExample project.....	150
20.4 Declaring state in a composable.....	150
20.5 Unidirectional data flow.....	153
20.6 State hoisting.....	155
20.7 Saving state through configuration changes.....	157
20.8 Summary	158
21. An Introduction to Composition Local.....	161
21.1 Understanding CompositionLocal	161
21.2 Using CompositionLocal	162
21.3 Creating the CompLocalDemo project.....	163
21.4 Designing the layout.....	163
21.5 Adding the CompositionLocal state	164
21.6 Accessing the CompositionLocal state.....	165
21.7 Testing the design.....	165
21.8 Summary	168
22. An Overview of Compose Slot APIs	169
22.1 Understanding slot APIs	169
22.2 Declaring a slot API.....	170
22.3 Calling slot API composables.....	170
22.4 Summary	172
23. A Compose Slot API Tutorial.....	173
23.1 About the project.....	173
23.2 Creating the SlotApiDemo project	173
23.3 Preparing the MainActivity class file.....	173
23.4 Creating the MainScreen composable.....	174
23.5 Adding the ScreenContent composable	175
23.6 Creating the Checkbox composable	176
23.7 Implementing the ScreenContent slot API.....	177
23.8 Adding an Image drawable resource	178
23.9 Writing the TitleImage composable	179
23.10 Completing the MainScreen composable.....	180
23.11 Previewing the project.....	182
23.12 Summary.....	183
24. Using Modifiers in Compose.....	185
24.1 An overview of modifiers.....	185
24.2 Creating the ModifierDemo project.....	185
24.3 Creating a modifier	186
24.4 Modifier ordering.....	188
24.5 Adding modifier support to a composable	188
24.6 Common built-in modifiers	192

24.7 Combining modifiers.....	192
24.8 Summary	193
25. Annotated Strings and Brush Styles.....	195
25.1 What are annotated strings?	195
25.2 Using annotated strings.....	195
25.3 Brush Text Styling	196
25.4 Creating the example project.....	197
25.5 An example SpanStyle annotated string.....	197
25.6 An example ParagraphStyle annotated string	198
25.7 A Brush style example	201
25.8 Summary	202
26. Composing Layouts with Row and Column	203
26.1 Creating the RowColDemo project	203
26.2 Row composable.....	204
26.3 Column composable.....	204
26.4 Combining Row and Column composables	205
26.5 Layout alignment	206
26.6 Layout arrangement positioning.....	208
26.7 Layout arrangement spacing.....	210
26.8 Row and Column scope modifiers.....	211
26.9 Scope modifier weights	215
26.10 Summary	216
27. Box Layouts in Compose.....	217
27.1 An introduction to the Box composable.....	217
27.2 Creating the BoxLayout project	217
27.3 Adding the TextCell composable	217
27.4 Adding a Box layout.....	218
27.5 Box alignment.....	219
27.6 BoxScope modifiers	221
27.7 Using the clip() modifier	221
27.8 Summary	223
28. Custom Layout Modifiers.....	225
28.1 Compose layout basics	225
28.2 Custom layouts	225
28.3 Creating the LayoutModifier project.....	225
28.4 Adding the ColorBox composable.....	226
28.5 Creating a custom layout modifier	227
28.6 Understanding default position.....	227
28.7 Completing the layout modifier	227
28.8 Using a custom modifier	228
28.9 Working with alignment lines	229
28.10 Working with baselines	231
28.11 Summary	231
29. Building Custom Layouts.....	233
29.1 An overview of custom layouts	233
29.2 Custom layout syntax	233

Table of Contents

29.3 Using a custom layout.....	234
29.4 Creating the CustomLayout project	235
29.5 Creating the CascadeLayout composable	235
29.6 Using the CascadeLayout composable	237
29.7 Summary	238
30. A Guide to ConstraintLayout in Compose.....	239
30.1 An introduction to ConstraintLayout	239
30.2 How ConstraintLayout works.....	239
30.2.1 Constraints.....	239
30.2.2 Margins.....	240
30.2.3 Opposing constraints.....	240
30.2.4 Constraint bias.....	241
30.2.5 Chains.....	242
30.2.6 Chain styles	242
30.3 Configuring dimensions.....	243
30.4 Guideline helper	243
30.5 Barrier helper.....	244
30.6 Summary	245
31. Working with ConstraintLayout in Compose	247
31.1 Calling ConstraintLayout.....	247
31.2 Generating references.....	247
31.3 Assigning a reference to a composable.....	247
31.4 Adding constraints.....	248
31.5 Creating the ConstraintLayout project	248
31.6 Adding the ConstraintLayout library	249
31.7 Adding a custom button composable.....	249
31.8 Basic constraints.....	250
31.9 Opposing constraints.....	251
31.10 Constraint bias.....	252
31.11 Constraint margins	253
31.12 The importance of opposing constraints and bias	254
31.13 Creating chains.....	257
31.14 Working with guidelines	258
31.15 Working with barriers	259
31.16 Decoupling constraints with constraint sets.....	262
31.17 Summary.....	264
32. Working with IntrinsicSize in Compose.....	265
32.1 Intrinsic measurements.....	265
32.2 Max. vs Min. Intrinsic Size measurements	265
32.3 About the example project.....	266
32.4 Creating the IntrinsicSizeDemo project.....	267
32.5 Creating the custom text field.....	267
32.6 Adding the Text and Box components.....	268
32.7 Adding the top-level Column.....	268
32.8 Testing the project.....	269
32.9 Applying IntrinsicSize.Max measurements	269
32.10 Applying IntrinsicSize.Min measurements	270
32.11 Summary.....	270

33. Coroutines and LaunchedEffects in Jetpack Compose.....	271
33.1 What are coroutines?	271
33.2 Threads vs. coroutines	271
33.3 Coroutine Scope	272
33.4 Suspend functions	272
33.5 Coroutine dispatchers.....	272
33.6 Coroutine builders	273
33.7 Jobs.....	273
33.8 Coroutines – suspending and resuming	274
33.9 Coroutine channel communication.....	275
33.10 Understanding side effects	276
33.11 Summary	277
34. An Overview of Lists and Grids in Compose	279
34.1 Standard vs. lazy lists	279
34.2 Working with Column and Row lists	279
34.3 Creating lazy lists	280
34.4 Enabling scrolling with ScrollState	281
34.5 Programmatic scrolling.....	281
34.6 Sticky headers	282
34.7 Responding to scroll position	284
34.8 Creating a lazy grid	284
34.9 Summary	287
35. A Compose Row and Column List Tutorial	289
35.1 Creating the ListDemo project	289
35.2 Creating a Column-based list.....	289
35.3 Enabling list scrolling	291
35.4 Manual scrolling.....	291
35.5 A Row list example.....	294
35.6 Summary	294
36. A Compose Lazy List Tutorial	295
36.1 Creating the LazyListDemo project.....	295
36.2 Adding list data to the project	295
36.3 Reading the XML data.....	297
36.4 Handling image loading	298
36.5 Designing the list item composable.....	300
36.6 Building the lazy list.....	301
36.7 Testing the project.....	302
36.8 Making list items clickable.....	302
36.9 Summary	304
37. Lazy List Sticky Headers and Scroll Detection	305
37.1 Grouping the list item data	305
37.2 Displaying the headers and items	305
37.3 Adding sticky headers.....	306
37.4 Reacting to scroll position	307
37.5 Adding the scroll button	309
37.6 Testing the finished app.....	310
37.7 Summary	311

38. A Compose Lazy Staggered Grid Tutorial	313
38.1 Lazy Staggered Grids	313
38.2 Creating the StaggeredGridDemo project	314
38.3 Adding the Box composable	315
38.4 Generating random height and color values	315
38.5 Creating the Staggered List	316
38.6 Testing the project.....	317
38.7 Switching to a horizontal staggered grid.....	318
38.8 Summary	319
39. Compose Visibility Animation	321
39.1 Creating the AnimateVisibility project	321
39.2 Animating visibility	321
39.3 Defining enter and exit animations	324
39.4 Animation specs and animation easing	325
39.5 Repeating an animation	327
39.6 Different animations for different children	327
39.7 Auto-starting an animation	328
39.8 Implementing crossfading	329
39.9 Summary	331
40. Compose State-Driven Animation.....	333
40.1 Understanding state-driven animation	333
40.2 Introducing animate as state functions	333
40.3 Creating the AnimateState project.....	334
40.4 Animating rotation with animateFloatAsState.....	334
40.5 Animating color changes with animateColorAsState.....	337
40.6 Animating motion with animateDpAsState	339
40.7 Adding spring effects	342
40.8 Working with keyframes	343
40.9 Combining multiple animations	344
40.10 Using the Animation Inspector.....	347
40.11 Summary.....	348
41. Canvas Graphics Drawing in Compose	349
41.1 Introducing the Canvas component	349
41.2 Creating the CanvasDemo project.....	349
41.3 Drawing a line and getting the canvas size	349
41.4 Drawing dashed lines.....	351
41.5 Drawing a rectangle	351
41.6 Applying rotation	355
41.7 Drawing circles and ovals.....	356
41.8 Drawing gradients.....	357
41.9 Drawing arcs	360
41.10 Drawing paths	361
41.11 Drawing points	362
41.12 Drawing an image	363
41.13 Drawing text	365
41.14 Summary.....	367
42. Working with ViewModels in Compose	369

42.1 What is Android Jetpack?	369
42.2 The “old” architecture	369
42.3 Modern Android architecture	369
42.4 The ViewModel component.....	369
42.5 ViewModel implementation using state.....	370
42.6 Connecting a ViewModel state to an activity.....	371
42.7 ViewModel implementation using LiveData.....	372
42.8 Observing ViewModel LiveData within an activity	373
42.9 Summary	373
43. A Compose ViewModel Tutorial.....	375
43.1 About the project.....	375
43.2 Creating the ViewModelDemo project	376
43.3 Adding the ViewModel	376
43.4 Accessing DemoViewModel from MainActivity	377
43.5 Designing the temperature input composable	378
43.6 Designing the temperature input composable	380
43.7 Completing the user interface design.....	382
43.8 Testing the app.....	384
43.9 Summary	384
44. An Overview of Android SQLite Databases	385
44.1 Understanding database tables.....	385
44.2 Introducing database schema	385
44.3 Columns and data types	385
44.4 Database rows	386
44.5 Introducing primary keys	386
44.6 What is SQLite?	386
44.7 Structured Query Language (SQL).....	386
44.8 Trying SQLite on an Android Virtual Device (AVD)	387
44.9 The Android Room persistence library.....	389
44.10 Summary	389
45. Room Databases and Compose	391
45.1 Revisiting modern app architecture	391
45.2 Key elements of Room database persistence	391
45.2.1 Repository	391
45.2.2 Room database	392
45.2.3 Data Access Object (DAO)	392
45.2.4 Entities.....	392
45.2.5 SQLite database	392
45.3 Understanding entities	393
45.4 Data Access Objects.....	395
45.5 The Room database.....	396
45.6 The Repository.....	397
45.7 In-Memory databases	398
45.8 Database Inspector.....	399
45.9 Summary	399
46. A Compose Room Database and Repository Tutorial	401
46.1 About the RoomDemo project.....	401

Table of Contents

46.2	Creating the RoomDemo project.....	402
46.3	Modifying the build configuration	402
46.4	Building the entity.....	403
46.5	Creating the Data Access Object.....	404
46.6	Adding the Room database.....	405
46.7	Adding the repository.....	406
46.8	Adding the ViewModel	408
46.9	Designing the user interface	410
46.10	Writing a ViewModelProvider Factory class.....	412
46.11	Completing the MainScreen function.....	414
46.12	Testing the RoomDemo app.....	417
46.13	Using the Database Inspector.....	417
46.14	Summary.....	418
47.	An Overview of Navigation in Compose	419
47.1	Understanding navigation.....	419
47.2	Declaring a navigation controller.....	421
47.3	Declaring a navigation host	421
47.4	Adding destinations to the navigation graph	421
47.5	Navigating to destinations.....	422
47.6	Passing arguments to a destination.....	424
47.7	Working with bottom navigation bars	425
47.8	Summary	427
48.	A Compose Navigation Tutorial	429
48.1	Creating the NavigationDemo project	429
48.2	About the NavigationDemo project	429
48.3	Declaring the navigation routes	429
48.4	Adding the home screen	430
48.5	Adding the welcome screen.....	431
48.6	Adding the profile screen	432
48.7	Creating the navigation controller and host.....	433
48.8	Implementing the screen navigation	433
48.9	Passing the user name argument.....	434
48.10	Testing the project.....	435
48.11	Summary.....	437
49.	A Compose Navigation Bar Tutorial.....	439
49.1	Creating the BottomBarDemo project	439
49.2	Declaring the navigation routes	439
49.3	Designing bar items.....	440
49.4	Creating the bar item list.....	440
49.5	Adding the destination screens	441
49.6	Creating the navigation controller and host.....	443
49.7	Designing the navigation bar.....	443
49.8	Working with the Scaffold component.....	445
49.9	Testing the project.....	446
49.10	Summary.....	446
50.	Detecting Gestures in Compose.....	447
50.1	Compose gesture detection.....	447

50.2 Creating the GestureDemo project.....	447
50.3 Detecting click gestures.....	447
50.4 Detecting taps using PointerInputScope.....	449
50.5 Detecting drag gestures.....	450
50.6 Detecting drag gestures using PointerInputScope.....	452
50.7 Scrolling using the scrollable modifier.....	453
50.8 Scrolling using the scroll modifiers.....	454
50.9 Detecting pinch gestures.....	456
50.10 Detecting rotation gestures.....	457
50.11 Detecting translation gestures.....	458
50.12 Summary.....	459
51. An Introduction to Kotlin Flow.....	461
51.1 Understanding Flows.....	461
51.2 Creating the sample project.....	461
51.3 Adding a view model to the project.....	462
51.4 Declaring the flow.....	463
51.5 Emitting flow data.....	463
51.6 Collecting flow data as state.....	464
51.7 Transforming data with intermediaries.....	465
51.8 Collecting flow data.....	467
51.9 Adding a flow buffer.....	468
51.10 More terminal flow operators.....	469
51.11 Flow flattening.....	470
51.12 Combining multiple flows.....	472
51.13 Hot and cold flows.....	473
51.14 StateFlow.....	473
51.15 SharedFlow.....	474
51.16 Converting a flow from cold to hot.....	476
51.17 Summary.....	476
52. A Jetpack Compose SharedFlow Tutorial.....	477
52.1 About the project.....	477
52.2 Creating the SharedFlowDemo project.....	477
52.3 Adding a view model to the project.....	478
52.4 Declaring the SharedFlow.....	478
52.5 Collecting the flow values.....	479
52.6 Testing the SharedFlowDemo app.....	481
52.7 Handling flows in the background.....	481
52.8 Summary.....	483
53. Creating, Testing, and Uploading an Android App Bundle.....	485
53.1 The release preparation process.....	485
53.2 Android app bundles.....	485
53.3 Register for a Google Play Developer Console account.....	486
53.4 Configuring the app in the console.....	487
53.5 Enabling Google Play app signing.....	488
53.6 Creating a keystore file.....	488
53.7 Creating the Android app bundle.....	490
53.8 Generating test APK files.....	491
53.9 Uploading the app bundle to the Google Play Developer Console.....	492

Table of Contents

53.10 Exploring the app bundle.....	493
53.11 Managing testers	494
53.12 Rolling the app out for testing.....	494
53.13 Uploading new app bundle revisions	495
53.14 Analyzing the app bundle file.....	496
53.15 Summary.....	496
54. An Overview of Android In-App Billing	499
54.1 Preparing a project for In-App purchasing.....	499
54.2 Creating In-App products and subscriptions.....	499
54.3 Billing client initialization.....	500
54.4 Connecting to the Google Play Billing library	501
54.5 Querying available products.....	501
54.6 Starting the purchase process	502
54.7 Completing the purchase	502
54.8 Querying previous purchases	503
54.9 Summary	504
55. An Android In-App Purchasing Tutorial	505
55.1 About the In-App purchasing example project.....	505
55.2 Creating the InAppPurchase project	505
55.3 Adding libraries to the project.....	505
55.4 Adding the App to the Google Play Store	506
55.5 Creating an In-App product	506
55.6 Enabling license testers.....	507
55.7 Creating a purchase helper class	508
55.8 Adding the StateFlow streams	509
55.9 Initializing the billing client.....	509
55.10 Querying the product.....	510
55.11 Handling purchase updates	511
55.12 Launching the purchase flow.....	511
55.13 Consuming the product	512
55.14 Restoring a previous purchase.....	512
55.15 Completing the MainActivity.....	513
55.16 Testing the app.....	515
55.17 Troubleshooting	517
55.18 Summary.....	518
56. Working with Compose Theming	519
56.1 Material Design 2 vs. Material Design 3	519
56.2 Material Design 3 theming	519
56.3 Building a custom theme	523
56.4 Summary	524
57. A Material Design 3 Theming Tutorial	525
57.1 Creating the ThemeDemo project	525
57.2 Designing the user interface	525
57.3 Building a new theme	527
57.4 Adding the theme to the project	528
57.5 Enabling dynamic colors.....	529
57.6 Summary	530

58. An Overview of Gradle in Android Studio	531
58.1 An overview of Gradle.....	531
58.2 Gradle and Android Studio	531
58.2.1 Sensible defaults	531
58.2.2 Dependencies.....	531
58.2.3 Build variants.....	532
58.2.4 Manifest entries	532
58.2.5 APK signing.....	532
58.2.6 ProGuard support	532
58.3 The Properties and Settings Gradle build files	532
58.4 The top-level gradle build file	533
58.5 Module level Gradle build files.....	534
58.6 Configuring signing settings in the build file.....	537
58.7 Running Gradle tasks from the command-line	538
58.8 Summary	538
Index	539

1. Start Here

This book aims to teach you how to build Android applications using Jetpack Compose 1.3, Android Studio Flamingo (2022.2.1), Material Design 3, and the Kotlin programming language.

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

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

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

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

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

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

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

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

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

1.1 For Kotlin programmers

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

1.2 For new Kotlin programmers

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

1.3 Downloading the code samples

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

<https://www.ebookfrenzy.com/retail/compose13/index.php>

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

Start Here

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

1.4 Feedback

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

1.5 Errata

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

<https://www.ebookfrenzy.com/errata/compose13.html>

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

2. Setting up an Android Studio Development Environment

Before any work can begin on the development of 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) which also includes the Android Software Development Kit (SDK), the Kotlin plug-in and 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 Flamingo 2022.2.1 using the Android API 33 SDK (Tiramisu) which, at the time of writing, are the latest versions.

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 it is important to note that there may be some minor differences between this book and the software. A web search for “Android Studio Flamingo” 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 Flamingo 2022.2.1 in the archives:

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

2.3 Installing Android Studio

Once downloaded, the exact steps to install Android Studio differ depending on the operating system on which the installation is being 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 users of the system. When prompted to select the components to install, make sure 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 on 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 in the form of 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:

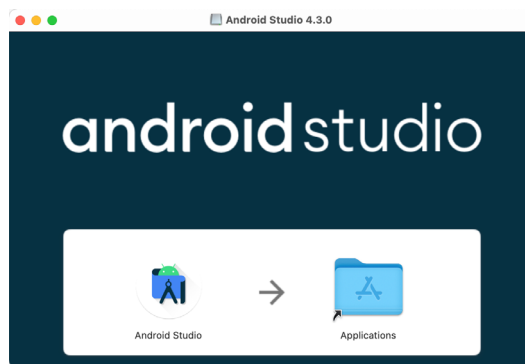


Figure 2-1

To install the package, simply 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:

```
unzip /<path to package>/android-studio-ide-<version>-linux.zip
```

Note that the Android Studio bundle will be installed into a subdirectory named *android-studio*. Assuming, therefore, 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
```

When running on a 64-bit Linux system, it may be necessary to install some 32-bit support libraries before Android Studio will run. On Ubuntu these libraries can be installed using the following command:

```
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386 lib32z1 libbz2-1.0:i386
```

On Red Hat and Fedora-based 64-bit systems, use the following command:

```
sudo yum install zlib.i686 ncurses-libs.i686 bzip2-libs.i686
```

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 on 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:

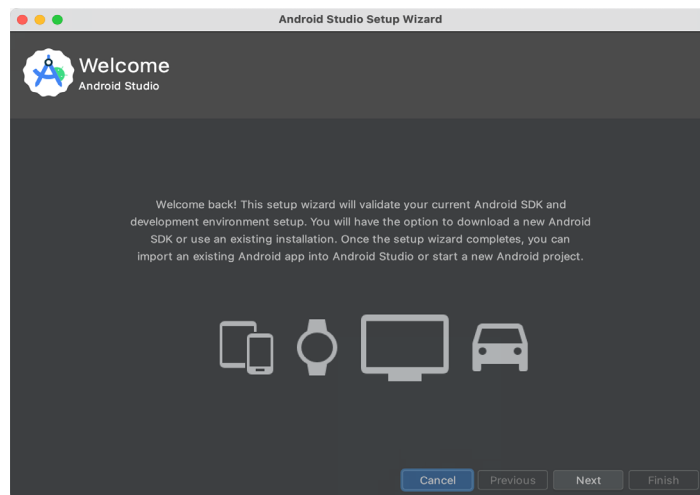


Figure 2-2

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.

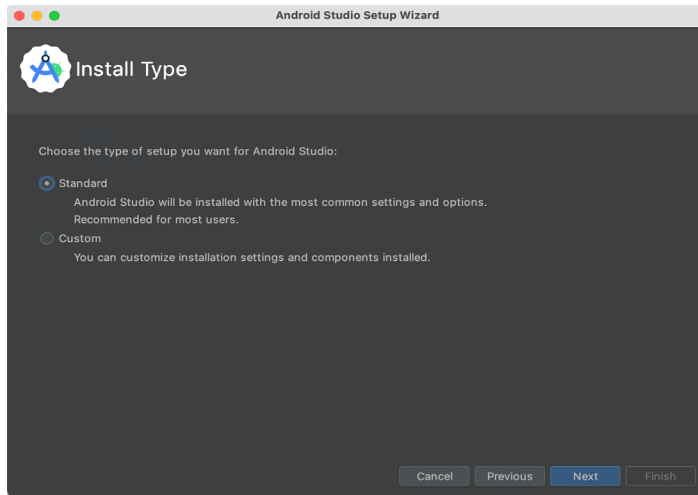


Figure 2-3

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 on 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:

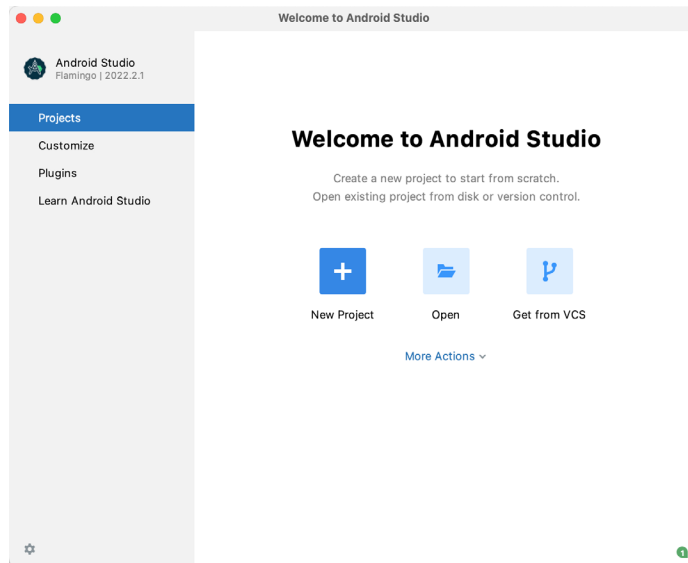


Figure 2-4

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 Preferences dialog will appear as shown in Figure 2-5:

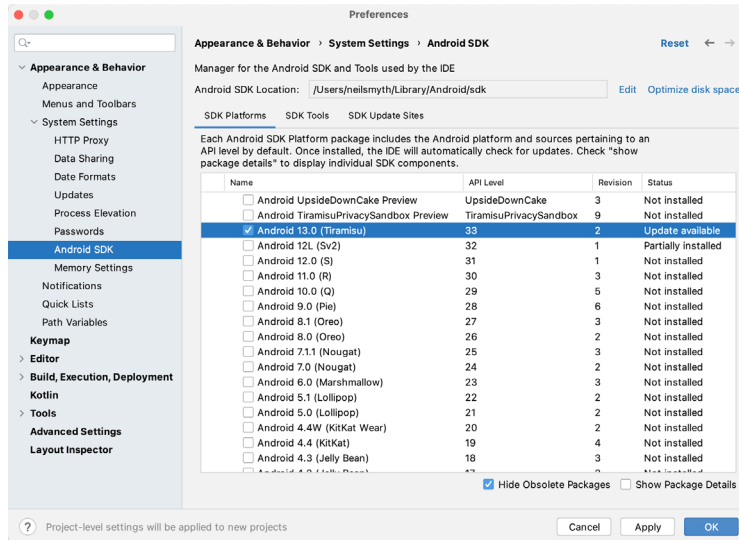


Figure 2-5

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

Most of the examples in this book will support older versions of Android as far back as Android 8.0 (Oreo). This is to ensure 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 on the *Apply* button. In the resulting confirmation dialog click on the *OK* button to install the SDK. 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:

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 on the *Apply* button.

In addition to the Android SDK packages, several tools are also installed for building Android applications. To

Setting up an Android Studio Development Environment

view the currently installed packages and check for updates, remain within the SDK settings screen and select the SDK Tools tab as shown in Figure 2-7:

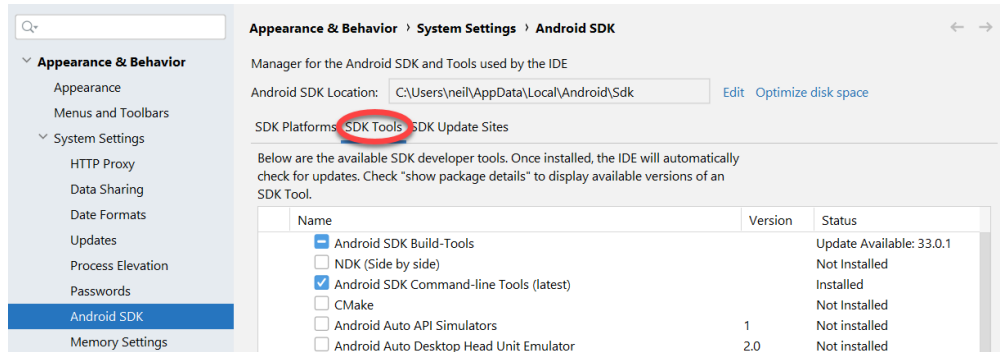


Figure 2-7

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

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

*Note 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, simply select the checkboxes next to those packages and click on the *Apply* button to initiate the installation process. If the HAXM emulator settings dialog appears, select the recommended memory allocation:

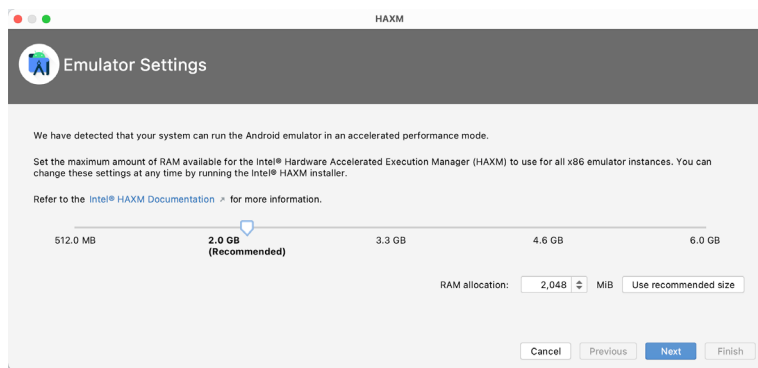


Figure 2-8

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

Apply button again.

2.6 Installing the Android SDK Command-line Tools

Android Studio includes a set of 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 enable the *Show Package Details* option in the bottom left-hand corner of the window. Next, scroll down the list of packages and, when the *Android SDK Command-line Tools (latest)* package comes into view, enable it as shown in Figure 2-9:

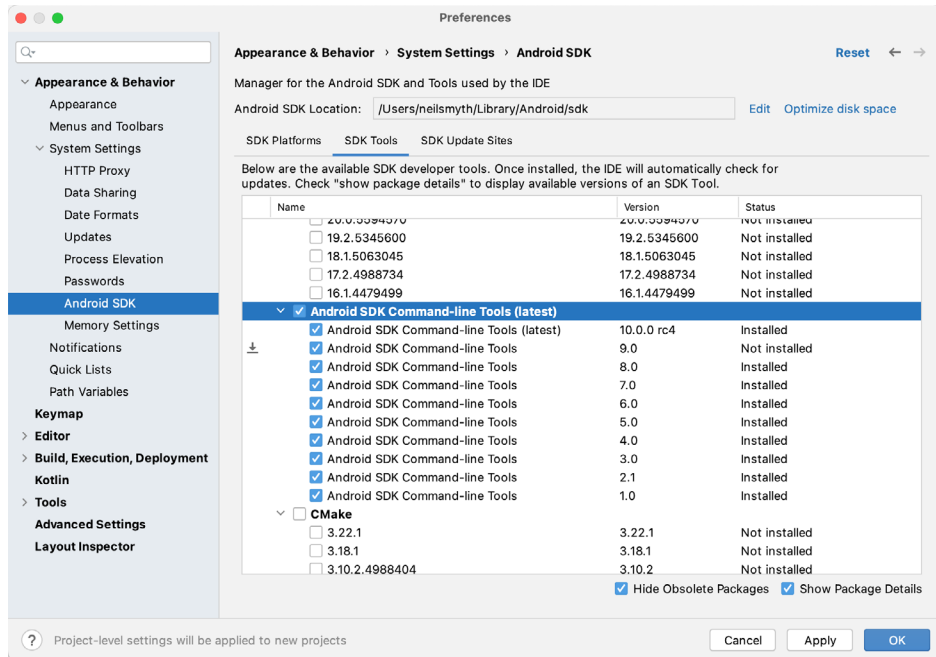


Figure 2-9

After you have selected the command-line tools package, click on *Apply* followed by *OK* to complete the installation. When the installation completes, click *Finish* and close the SDK Manager dialog.

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

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

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

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

Setting up an Android Studio Development Environment

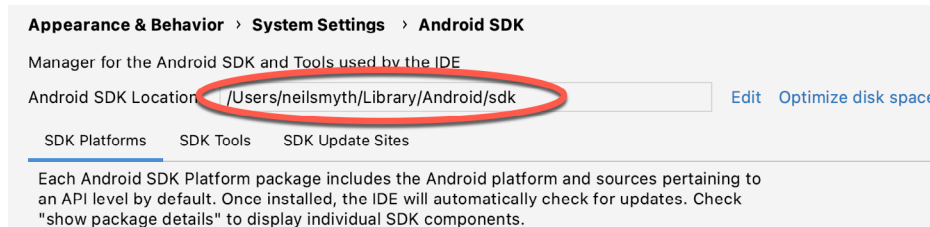


Figure 2-10

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 on the *Edit...* button. Using the *New* button in the edit dialog, add three 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 on 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 on the *Environment Variables...* button. Follow the steps outlined for Windows 8.1 starting from step 3.

2.6.4 Linux

On Linux, this configuration can typically be achieved 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 that consists of 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:

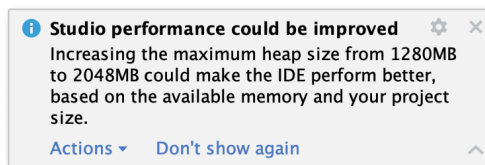


Figure 2-11

To view and modify the current memory configuration, select the *File -> Settings...* (*Android Studio ->*

Setting up an Android Studio Development Environment

Preferences... on macOS) menu option 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.

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.

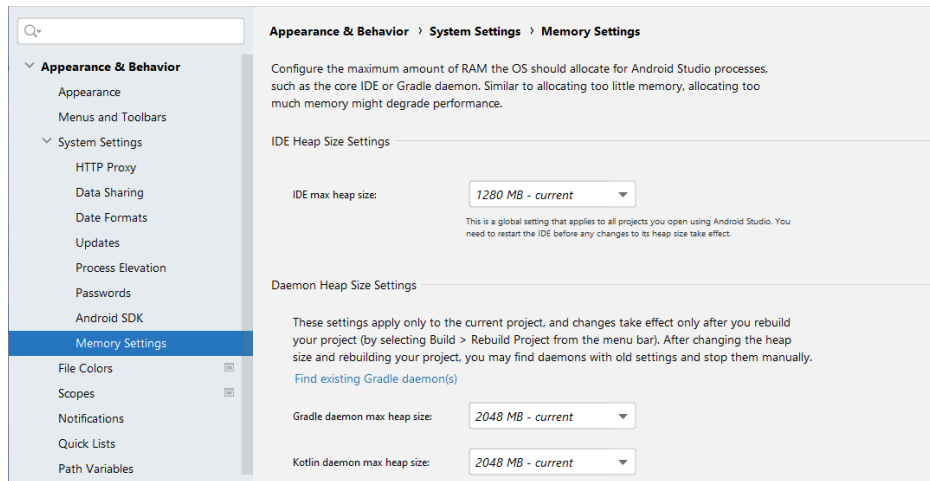


Figure 2-12

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, a number of 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.

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). In this chapter, we have covered 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 make use of 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 serve to 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. In the next chapter, we 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 of 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:

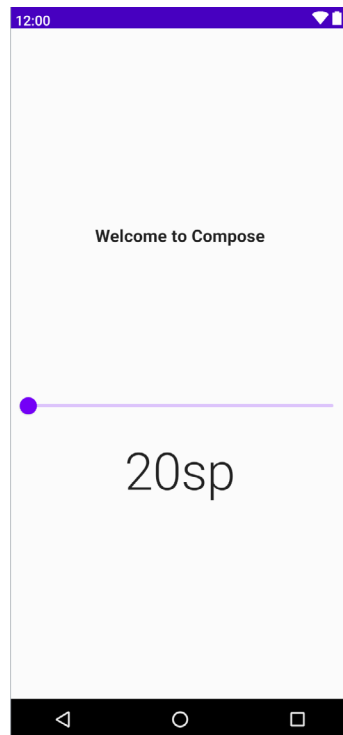


Figure 3-1

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:

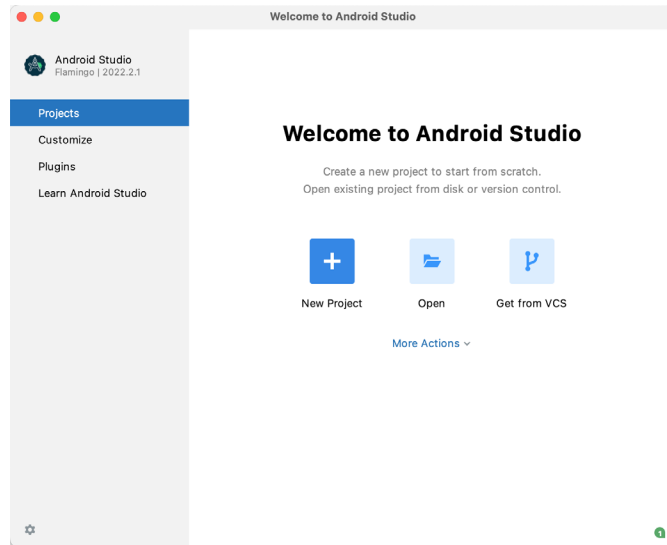


Figure 3-2

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 a range of different activity types is 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:

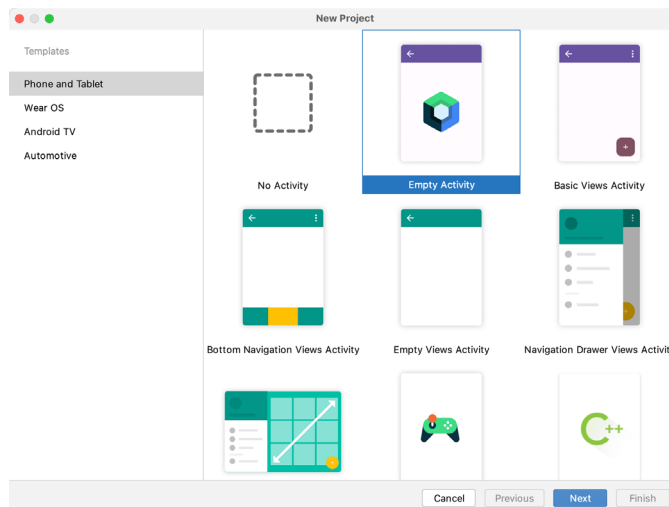


Figure 3-3

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:

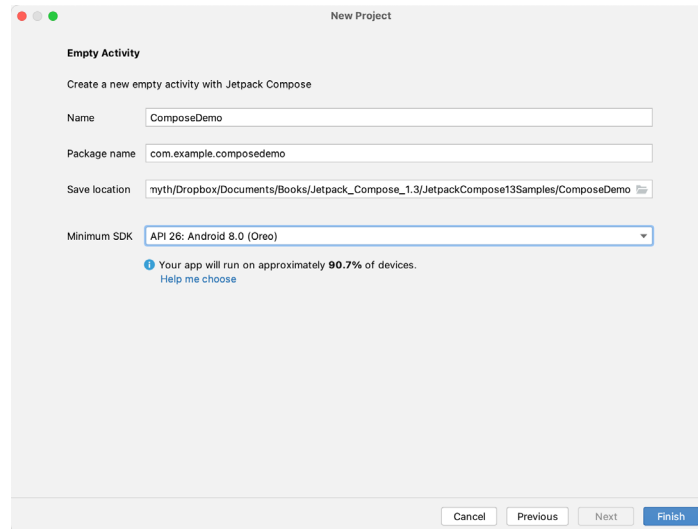


Figure 3-4

The *Package name* is used to uniquely identify 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 name of the application. 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 of the 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:

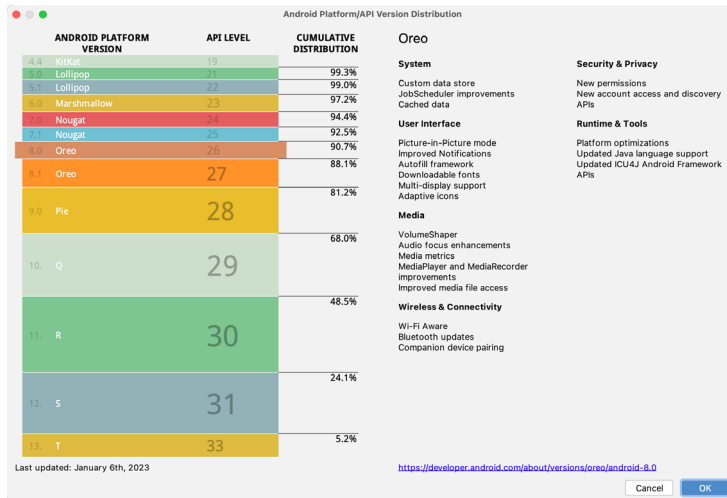


Figure 3-5

Click on the *Finish* button to create the project.

3.5 Previewing the example project

At this point, Android Studio should have created a minimal example application and opened the main window.

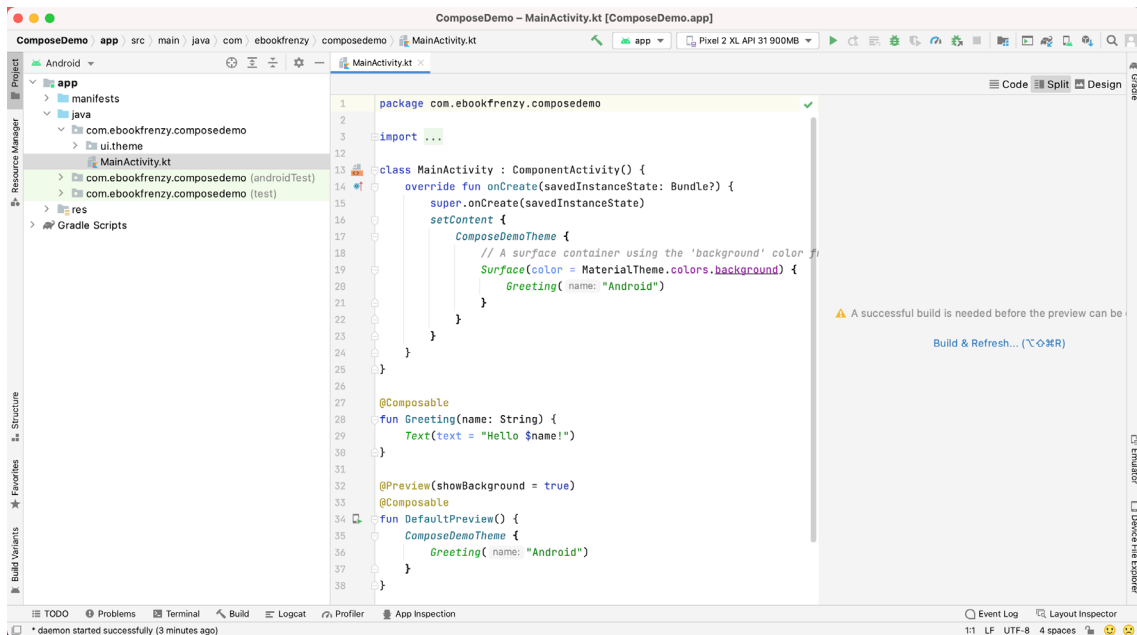


Figure 3-6

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 3-7. If the panel is not currently in Android mode, use the menu to switch mode:

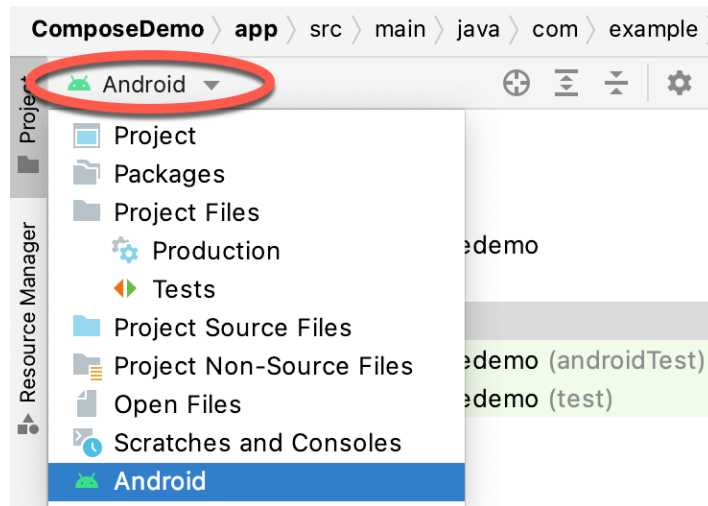


Figure 3-7

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 3-8:

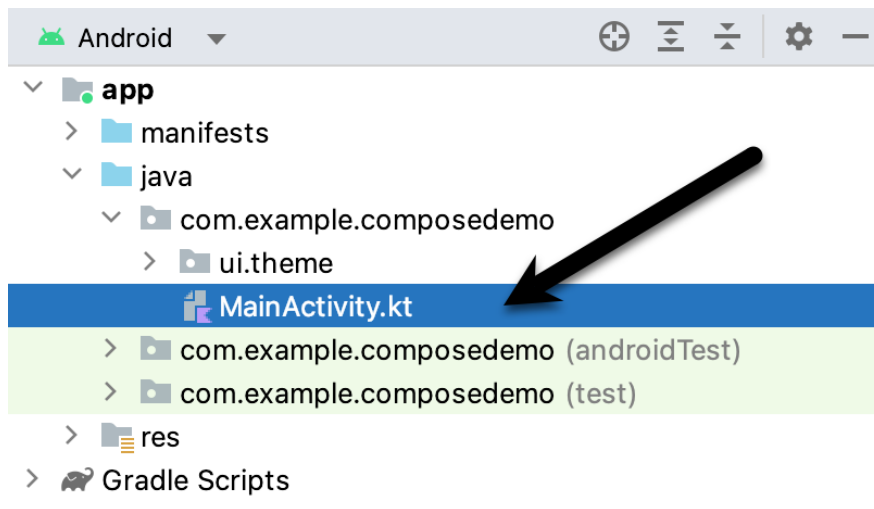


Figure 3-8

Double-click on this file to load it into the main code editor panel. The editor can be used in different modes when writing code, the most useful of which when working with Compose is Split mode. The current mode can be changed using the buttons marked A in Figure 3-9. Split mode displays the code editor (B) alongside the Preview panel (C) in which the current user interface design will appear:

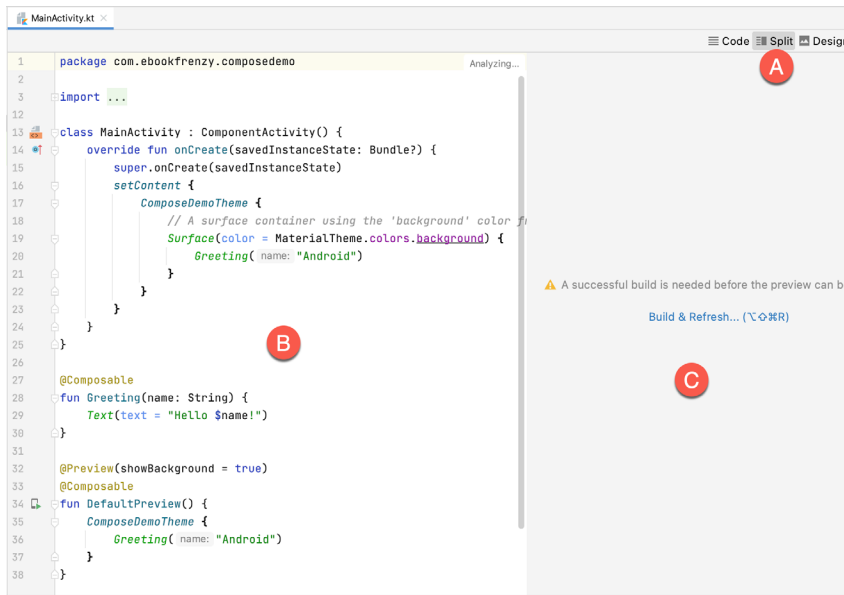


Figure 3-9

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 3-10:

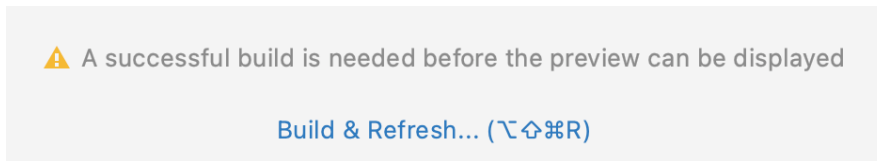


Figure 3-10

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

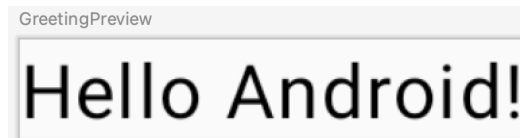


Figure 3-11

3.6 Reviewing the main activity

Android applications are created by bringing together 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 is comprised of 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 “+” button indicated by the arrow in Figure 3-12 below:

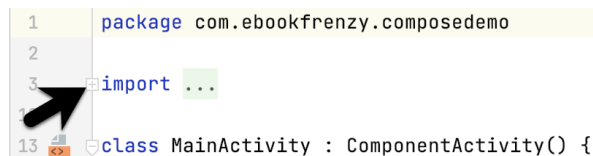


Figure 3-12

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 {
.
.
```

A Compose Project Overview

```
.  
    }  
}  
}
```

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 (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 3-13:

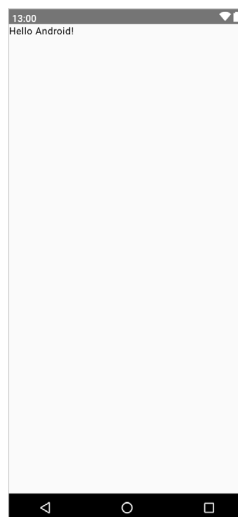


Figure 3-13

3.7 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 3-14):

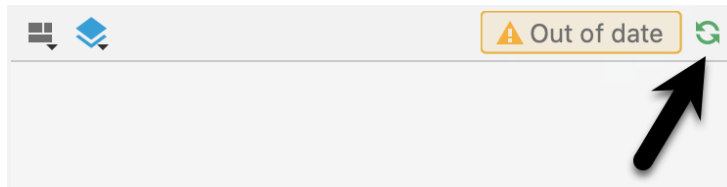


Figure 3-14

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 3-15:



Figure 3-15

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.

3.8 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 (Module: app)` file (highlighted in the figure below) and double-click on it to load it into the editor:

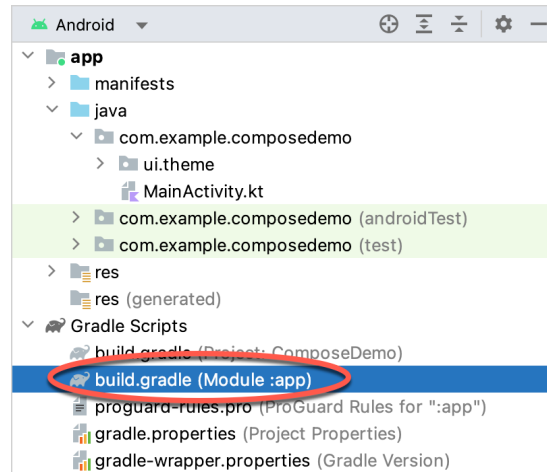


Figure 3-16

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

```
dependencies {
    .
    .
    implementation platform('androidx.compose:compose-bom:2022.10.00')
    .
    .
}
```

In the above example, we can see that the project is using BOM 2022.10.00. 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 3-17 below. For example, the figure shows that BOM 2022.10.00 uses version 1.3.2 of the Compose libraries:

BOM to library version mapping 🔍

2022.12.00 ▾

Library group	BOM Versions
androidx.compose.animation:animation	1.3.2
androidx.compose.animation:animation-core	1.3.2
androidx.compose.animation:animation-graphics	1.3.2
androidx.compose.foundation:foundation	1.3.2

Figure 3-17

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* file that include a specific version number, as is the case with the *core-ktx* and *lifecycle-runtime-ktx* libraries:

A Compose Project Overview

```
dependencies {  
  
    implementation 'androidx.core:core-ktx:1.8.0'  
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.3.1'  
  
    .  
    .  
}
```

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

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

3.9 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.

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 and test basic functionality using interactive mode, it will be necessary to compile and run an entire app to fully test that it works. An Android application may be tested by installing and running it either 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 the creation of such a virtual device using the Pixel 4a phone as a reference example.

5.1 About Android Virtual Devices

AVDs are essentially emulators that allow Android applications to be tested without the necessity to install the application on a physical Android-based device. An AVD may be configured to emulate a variety of hardware features including options such as screen size, memory capacity, and the presence or otherwise of features such as a camera, GPS navigation support, or an accelerometer. As part of the standard Android Studio installation, several emulator templates are installed allowing AVDs to be configured for a range of different 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 either 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 either 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 selecting the *Tools -> Device Manager* menu option from within the main window.

Once launched, the manager will appear as a tool window as shown in Figure 5-1:

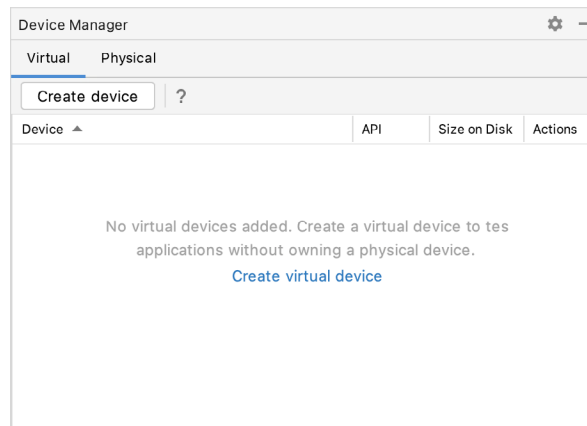


Figure 5-1

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:

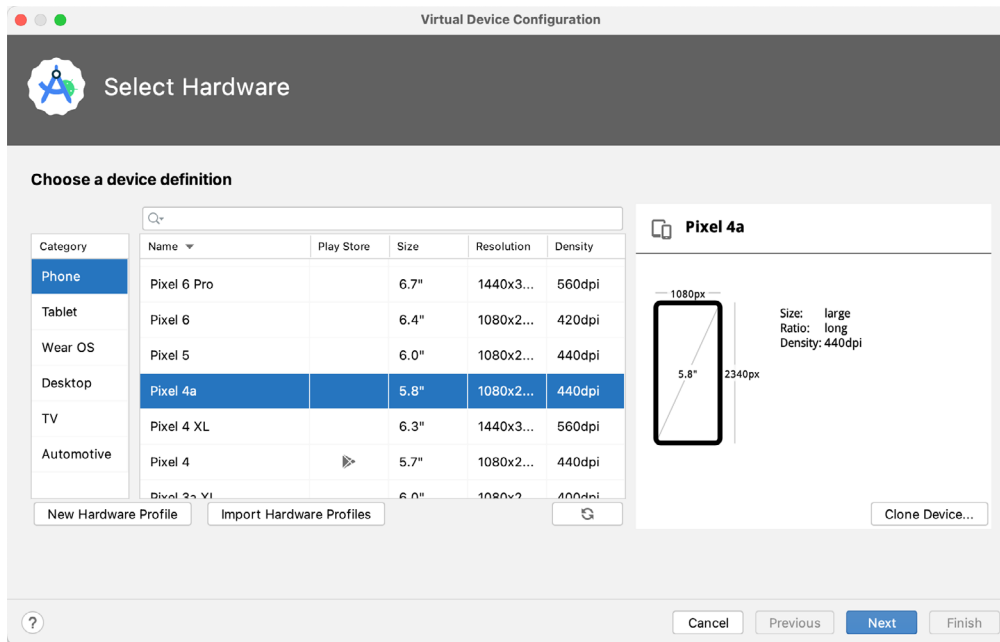


Figure 5-2

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

1. From the *Category* panel, select the *Phone* option to display the list of available Android phone AVD templates.
2. Select the *Pixel 4a* device option and click *Next*.
3. On the System Image screen, select the latest version of Android. Note that 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 4a API 33*) into the name field or simply accept the default name.
5. Click *Finish* to create the AVD.
6. With the AVD created, the Device Manager may now be closed. If future modifications to the AVD are necessary, simply 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 perform a test run of the newly created AVD emulator, simply select the emulator from the Device Manager and click on 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:

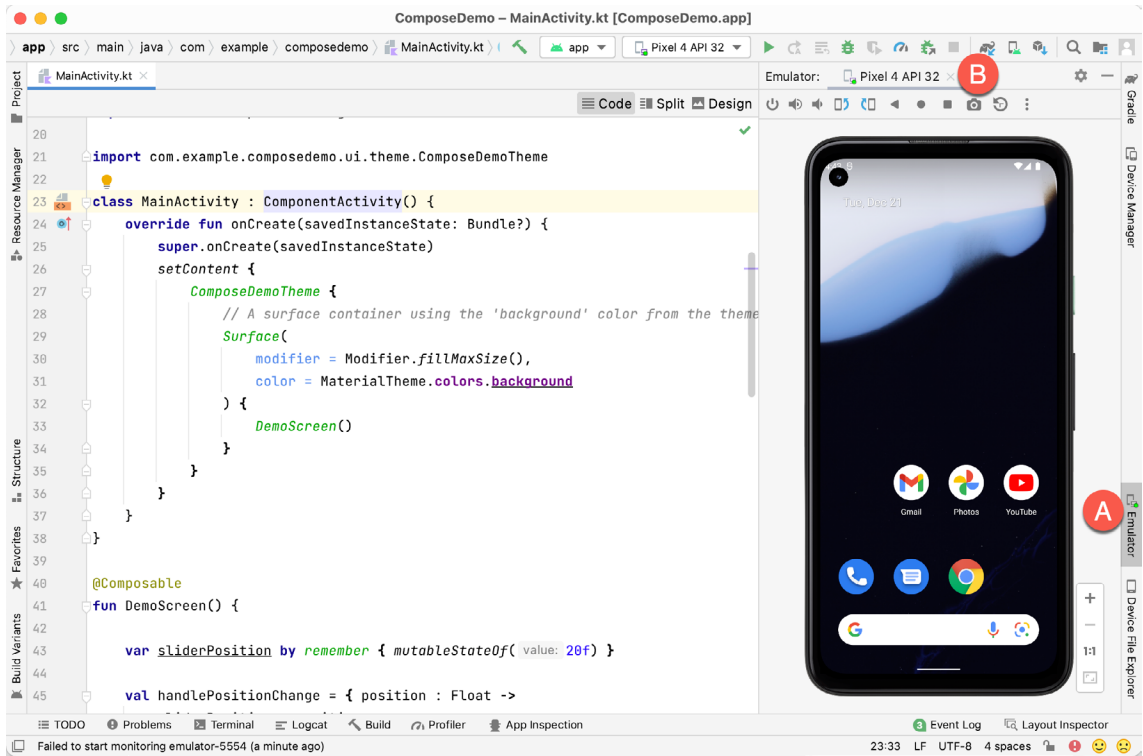


Figure 5-3

To hide and show the emulator tool window, click on the Emulator tool window button (marked A above). Click on 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 5-4, for example, shows a tool window with two emulator sessions:

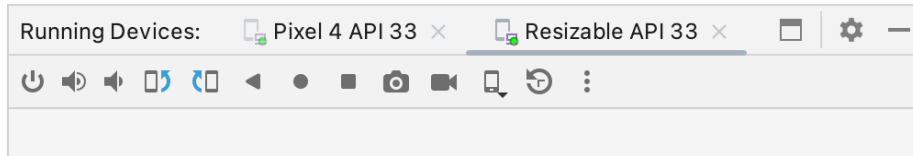


Figure 5-4

To switch between sessions, simply 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 4a 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 now can be compiled and run. With the ComposeDemo project loaded into Android Studio, make sure that the

Creating an Android Virtual Device (AVD) in Android Studio

newly created Pixel 4a AVD is displayed in the device menu (marked A in Figure 5-5 below), then either click on the run button represented by a green triangle (B), select the *Run -> Run 'app'* menu option or use the Ctrl-R keyboard shortcut:

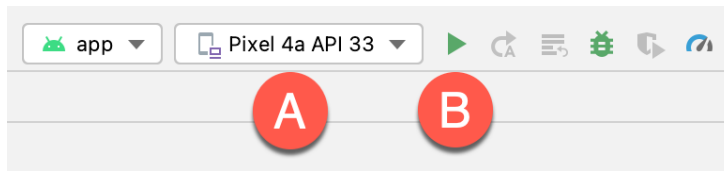


Figure 5-5

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:

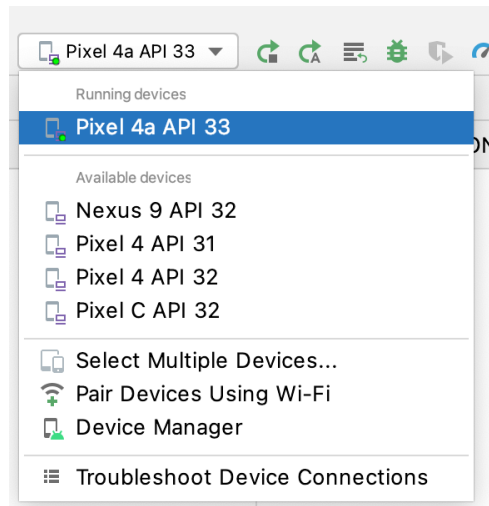


Figure 5-6

The app can also be run on the currently selected target by clicking on the icon in the editor gutter next to the preview composable declaration as indicated by the arrow in Figure 5-7:

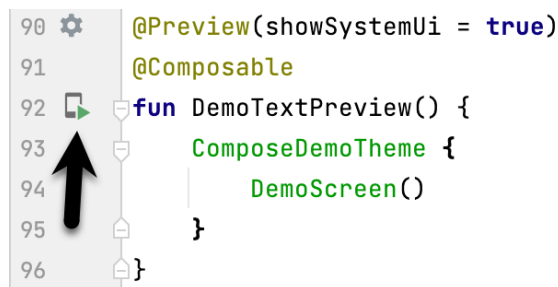


Figure 5-7

Once the application is installed and running, the user interface layout defined by the *MainScreen* function will appear within the emulator:

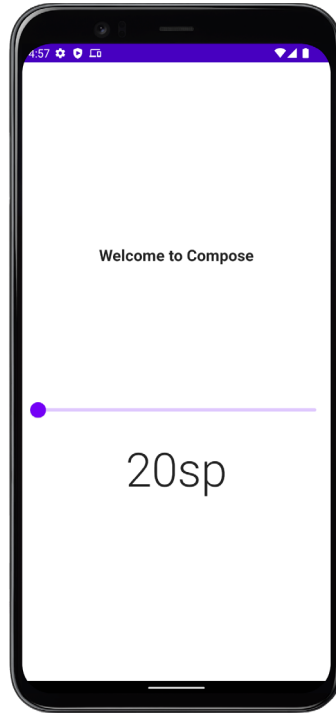


Figure 5-8

If the activity does not automatically launch, check to see if the launch icon has appeared among the apps on the emulator. If it has, simply click on it to launch the application. Once the run process begins, the Run tool window will become available. The Run tool window will display diagnostic information as the application package is installed and launched. Figure 5-9 shows the Run tool window output from a successful application launch:

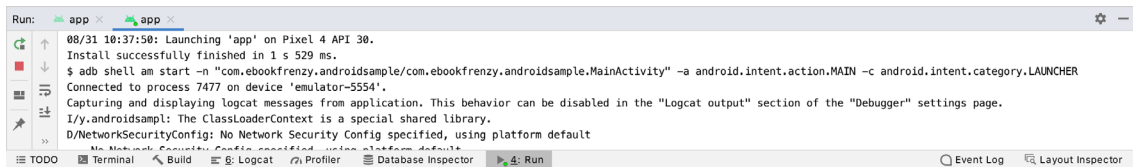


Figure 5-9

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

Assuming that the application loads into the emulator and runs as expected, we have safely verified that the Android development environment is correctly installed and configured.

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 having to build and re-run the project. Try out Live Edit by changing the text displayed by the DemoText composable as follows:

```
DemoText(message = "This is Compose 1.3", fontSize = sliderPosition)
```

With each keystroke, the text in the running app will update to reflect the change. Live Edit is currently limited

Creating an Android Virtual Device (AVD) in Android Studio

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 menu shown in Figure 5-6 above includes an option to run the app on multiple emulators and devices in parallel. When selected, this option displays the dialog shown in Figure 5-10 providing a list of both 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:

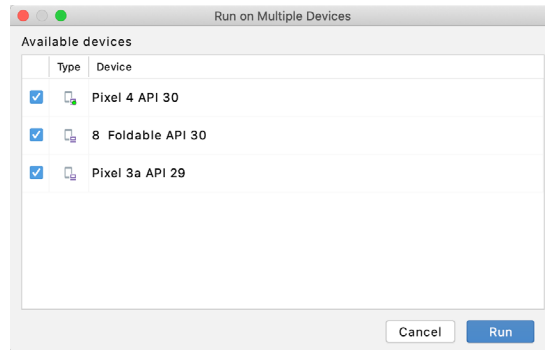


Figure 5-10

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

5.6 Stopping a running application

To stop a running application, simply click on the stop button located in the main toolbar as shown in Figure 5-11:

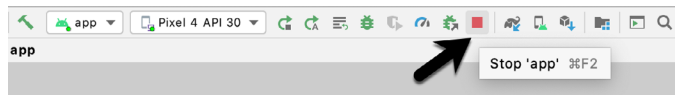


Figure 5-11

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 as illustrated in Figure 5-12:



Figure 5-12

Once the Run tool window appears, click the stop button highlighted in Figure 5-13 below:

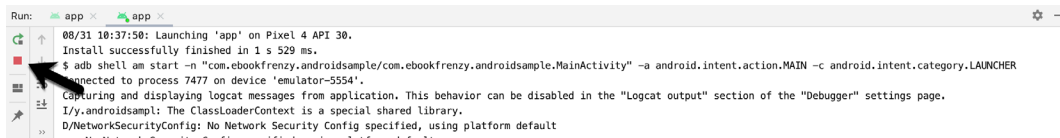


Figure 5-13

5.7 Supporting dark theme

Android 10 introduced the much-awaited dark theme, support for which is enabled by default in Android Studio Compose-based app projects. To test dark theme in the AVD emulator, open the Settings app within the running Android instance in the emulator. Within the Settings app, choose the *Display* category and enable the *Dark theme* option as shown in Figure 5-14 so that the screen background turns black:

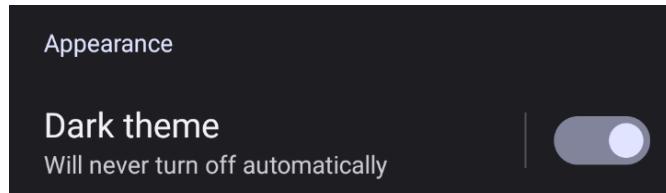


Figure 5-14

With dark theme enabled, run the ComposeDemo app and note that it appears using a dark theme including a black background and a purple background color on the button as shown in Figure 5-15:

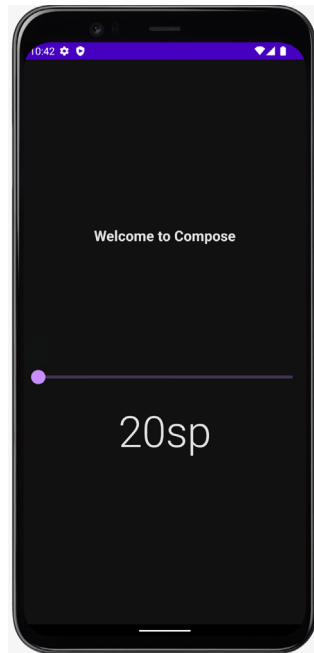


Figure 5-15

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. To run the emulator in a separate window, select the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS), navigate to *Tools -> Emulator* in the left-hand navigation panel of the preferences dialog, and disable the *Launch in a tool window* option:

Creating an Android Virtual Device (AVD) in Android Studio

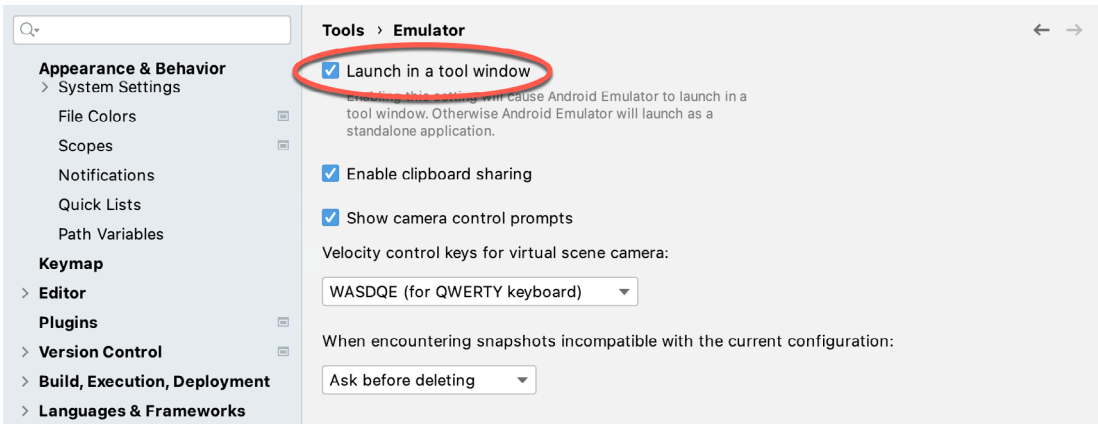


Figure 5-16

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 5-3 above.

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

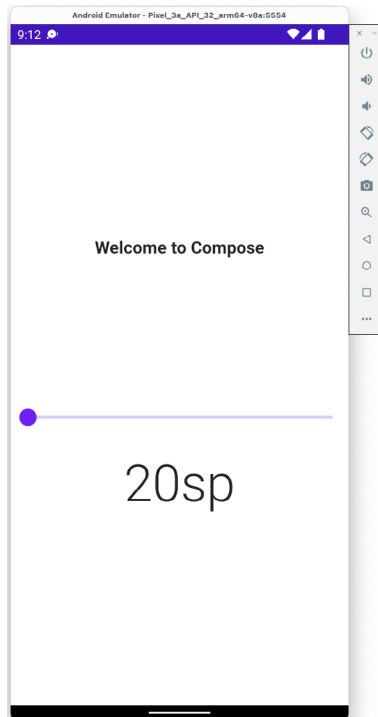


Figure 5-17

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 emulator tool window may also be detached from the main Android Studio window by clicking on the settings button (represented by the gear icon) in the tool emulator toolbar and selecting the *View Mode -> Float* menu option:

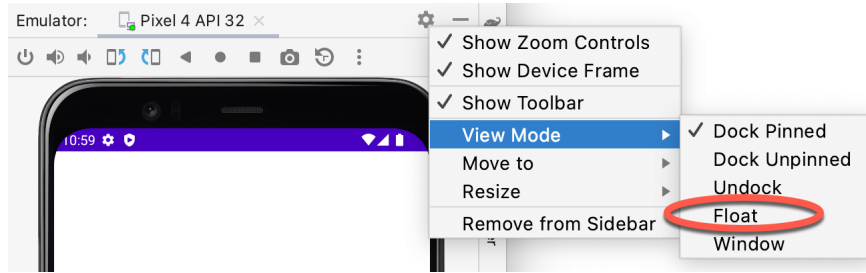


Figure 5-18

5.9 Enabling the device frame

The emulator can be configured to appear with (Figure 5-15) or without the device frame (Figure 5-17). To change the setting, open the Device Manager, select the AVD from the list, and click on the pencil icon in the Actions column to edit the settings. In the settings screen, locate and change the Enable Device Frame option:

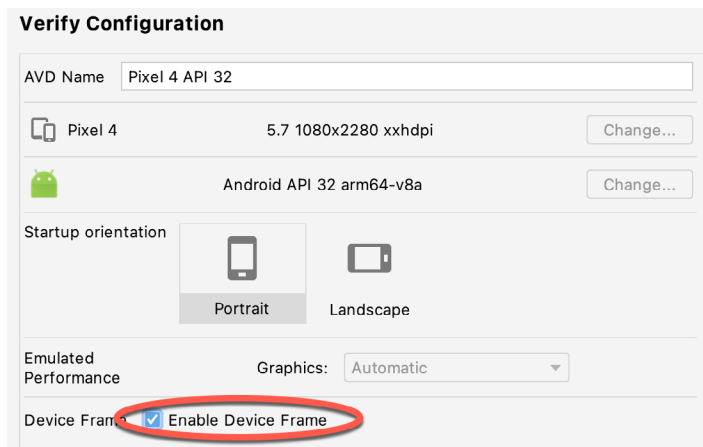


Figure 5-19

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 either a physical Android device or using an Android Virtual Device (AVD) emulator. AVDs are created and managed using the Android Studio Device Manager tool. 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, doing so 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 considerably shorten the learning curve in later chapters of the book. With this in mind, this chapter will provide an initial overview of the various areas and components that make up 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 next time it is launched, automatically opening the previously active project.

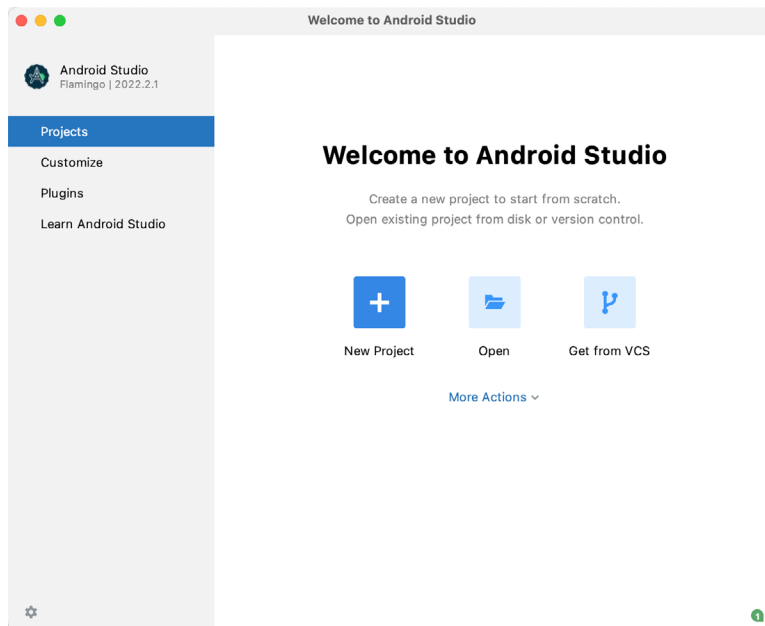


Figure 7-1

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 clicking on the menu button as shown in Figure 7-2:

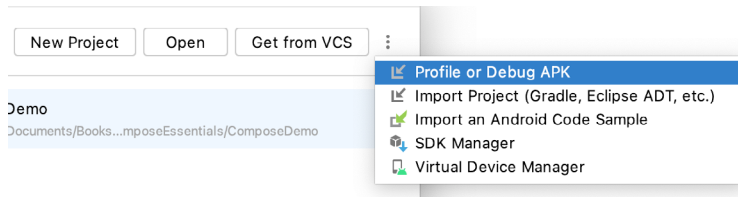


Figure 7-2

7.2 The Main Window

When a new project is created, or an existing one opened, the Android Studio *main window* will appear. When multiple projects are open simultaneously, each will be assigned its own main window. The precise configuration of the window will vary depending on which tools and panels were displayed the last time the project was open, but will typically resemble that of Figure 7-3.

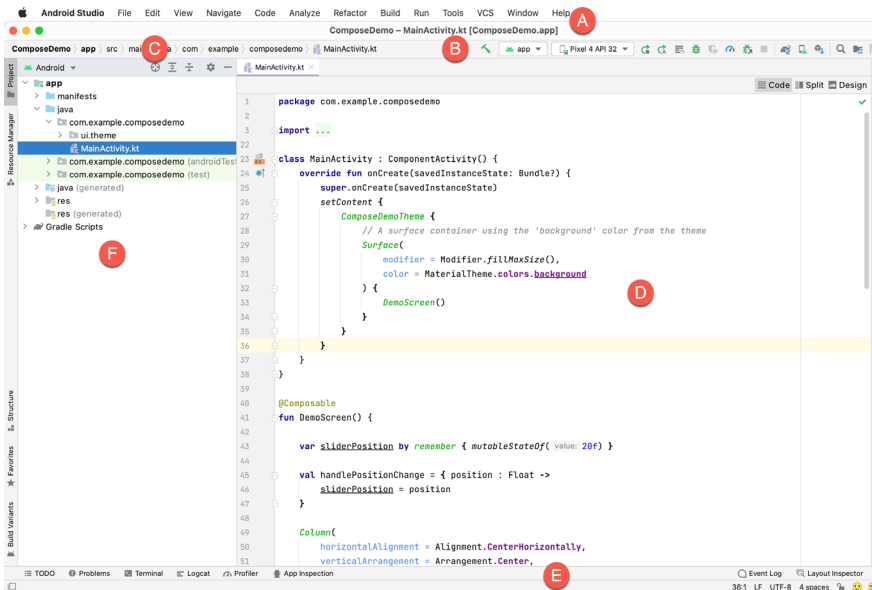


Figure 7-3

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

A – Menu Bar – Contains a range of menus for performing tasks within the Android Studio environment.

B – 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 Menus and Toolbars...* menu option. If the toolbar is not visible, it can be displayed using the *View -> Appearance -> Toolbar* menu option.

C – 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. Select a method from the list to be taken to the corresponding location within the code editor. Hide and display this bar using the *View -> Appearance -> Navigation Bar* menu option.

D – 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 7-4:

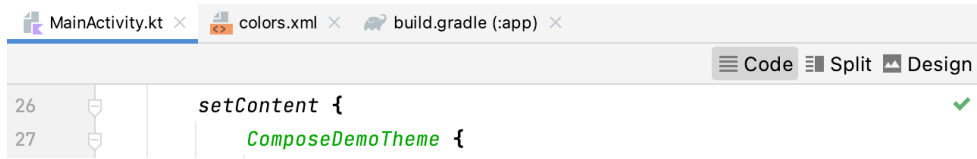


Figure 7-4

E – Status Bar – The status bar displays informational messages about the project and the activities of Android Studio together with the tools menu button located in the far left corner. Hovering over items in the status bar will display a description of that field. Many fields are interactive, allowing the user to click to perform tasks or obtain more detailed status information.

F – 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 tool windows available within the Android Studio environment.

7.3 The Tool Windows

In addition to the project view tool window, Android Studio also includes many other windows which, when enabled, are displayed along the bottom and sides of the main window. The tool window quick access menu can be displayed by hovering the mouse pointer over the button located in the far left-hand corner of the status bar (Figure 7-5) without clicking the mouse button.

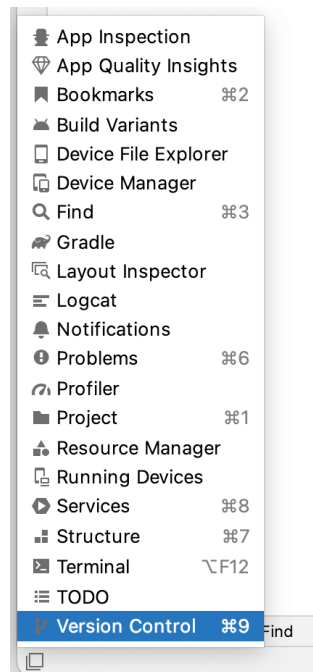


Figure 7-5

A Tour of the Android Studio User Interface

Selecting an item from the quick access menu will cause the corresponding tool window to appear within the main window.

Alternatively, a set of *tool window bars* can be displayed by clicking on the quick access menu icon in the status bar. These bars appear along the left, right, and bottom edges of the main window (as indicated by the arrows in Figure 7-6) and contain buttons for showing and hiding each of the tool windows. When the tool window bars are displayed, a second click on the button in the status bar will hide them.

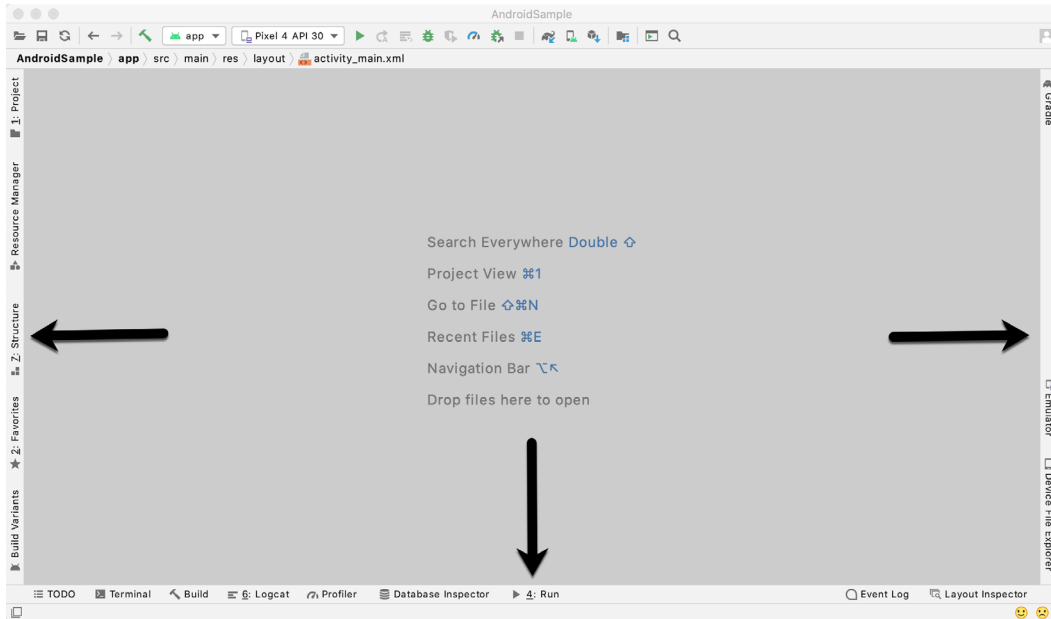


Figure 7-6

Clicking on a button will display the corresponding tool window while a second click will hide the window. Buttons prefixed with a number (for example 1: Project) indicate that the tool window may also be displayed by pressing the Alt key on the keyboard (or the Command key for macOS) together with the corresponding number.

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.

Each tool window has its own toolbar along the top edge. The buttons within these toolbars vary from one tool to the next, though all tool windows contain a settings option, represented by the cog icon, which allows various aspects of the window to be changed. Figure 7-7 shows the settings menu for the Project tool window. Options 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.

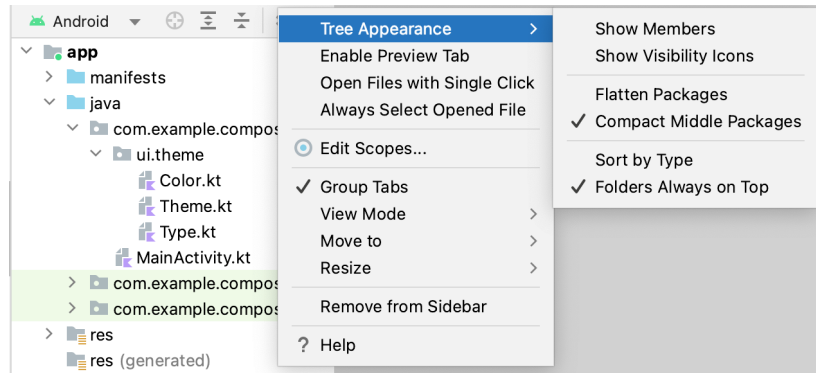


Figure 7-7

All of the windows also include a far-right button on the toolbar providing an additional way to hide the tool window from view. A search of the items within a tool window can be performed simply by giving that window focus by clicking in 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.

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

- **App Inspector** - Provides access to the Database and Background Task inspectors. The Database Inspector allows you to inspect, query, and modify your app's databases while the app is running. The Background Task Inspector allows background worker tasks created using WorkManager to be monitored and managed.
- **Build** - The build tool window displays information about the build process while a project is being compiled and packaged and displays 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.
- **Device Manager** - Provides access to the Device Manager tool window where physical Android device connections and emulators may be added, removed, and managed.
- **Emulator** - 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*”.
- **Event Log** – The event log window displays messages relating to events and activities performed within Android Studio. The successful build of a project, for example, or the fact that an application is now running will be reported within this tool window.
- **Favorites** – A variety of project items can be added to the favorites list. Right-clicking on a file in the project view, for example, provides access to an *Add to Favorites* menu option. Similarly, a method in a source file can be added as a favorite by right-clicking on it in the Structure tool window. Anything added to a Favorites list can be accessed through this Favorites tool window.
- **Find** - Search for code and text within your project files.
- **Gradle** – The Gradle tool window provides a view of the Gradle tasks that make up the project build

A Tour of the Android Studio User Interface

configuration. The window lists the tasks that are 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.

- **Layout Inspector** - Provides a visual 3D rendering of the hierarchy of components that make up a user interface layout.
- **Logcat** – The Logcat tool window provides access to the monitoring log output from a running application in addition to options for taking screenshots and videos of the application and stopping and restarting a process.
- **Problems** - A central location in which 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.
- **Profiler** – The Android Profiler tool window provides real-time monitoring and analysis tools for identifying performance issues within running apps, including CPU, memory and network usage. This option becomes available when an app is currently running.
- **Project** – 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** - A tool for adding and managing resources and assets such as images, colors, and layout files contained with the project.
- **Run** – 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 is failing to install and run on a device or emulator, this window will typically provide diagnostic information relating to the problem.
- **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.
- **Running Devices** - Displays any AVD instances running within the current Android Studio session.
- **Terminal** – 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.
- **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 selecting the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS) and navigating to the *TODO* page listed under *Editor*.
- **Version Control** - This tool window is used when the project files are under source code version control, allowing access to Git repositories and code change history.

7.4 Android Studio Keyboard Shortcuts

Android Studio includes an abundance of keyboard shortcuts designed to save time when performing common tasks. A full keyboard shortcut keymap listing can be viewed and printed from within the Android Studio project window by selecting the *Help -> Keyboard Shortcuts* menu option. You may also list and modify the keyboard shortcuts by selecting the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS)

and clicking on the Keymap entry as shown in Figure 7-8 below:

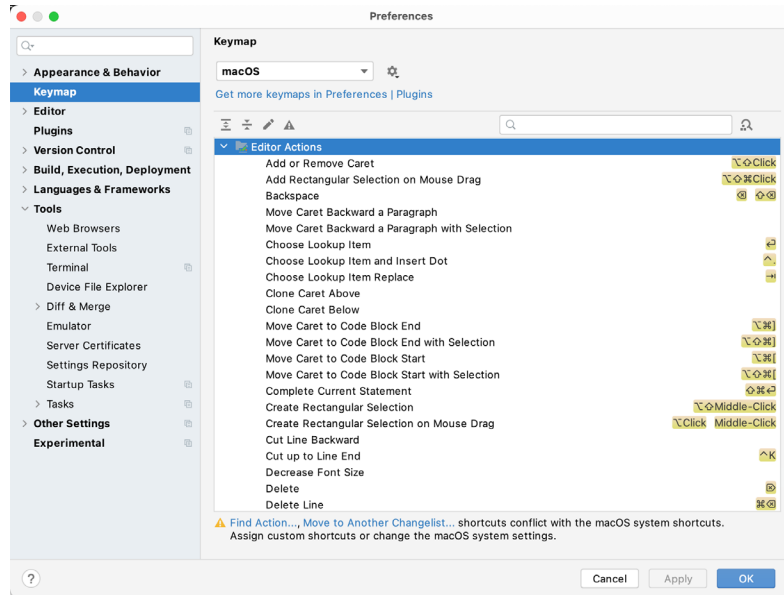


Figure 7-8

7.5 Switcher and Recent Files Navigation

Another useful mechanism for navigating within the Android Studio main window involves the use of 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 7-9).

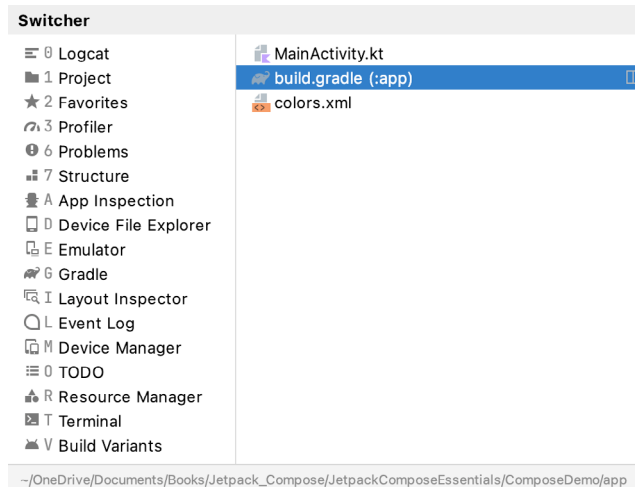


Figure 7-9

Once displayed, the switcher will remain visible for 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, navigation to recently opened files is provided by the Recent Files panel (Figure 7-10). 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 used to scroll through the file name and tool window options. Pressing the Enter key will select the currently highlighted item.

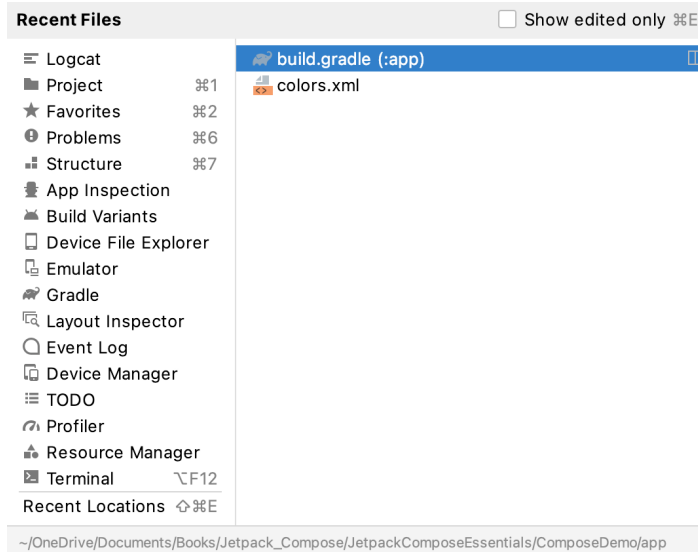


Figure 7-10

7.6 Changing the Android Studio Theme

The overall theme of the Android Studio environment may be changed either from the welcome screen using the *Customize* option or via the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS) of the main window.

Once the settings dialog is displayed, select the *Appearance & Behavior* option followed by *Appearance* in the left-hand panel and then change the setting of the *Theme* menu before clicking on the *Apply* button. The themes available will depend on the platform but usually include options such as Light, IntelliJ, Windows, High Contrast, and Darcula. Figure 7-11 shows an example of the main window with the Darcula theme selected:

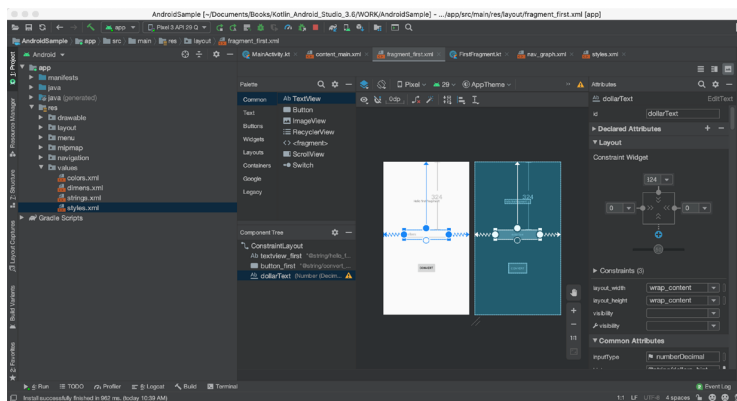


Figure 7-11

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:

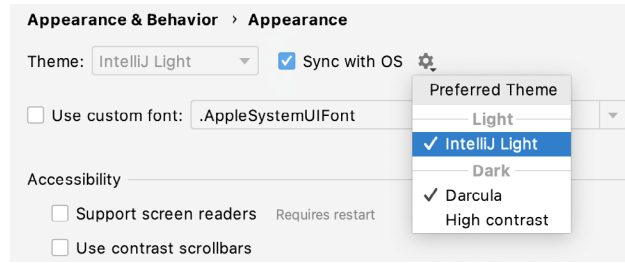


Figure 7-12

7.7 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 and bottom edges of the main window and can be accessed either using the quick access menu located in the status bar or via the optional tool window bars.

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.

14. Kotlin Control Flow

Regardless of the programming language used, application development is largely an exercise in applying logic, and much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed, how many times it is executed, and, conversely, which code gets bypassed when the program is running. This is often referred to as *control flow* since it controls the *flow* of program execution. Control flow typically falls into the categories of *looping control* (how often code is executed) and *conditional control flow* (whether or not code is executed). This chapter is intended to provide an introductory overview of both types of control flow in Kotlin.

14.1 Looping control flow

This chapter will begin by looking at control flow in the form of loops. Loops are essentially sequences of Kotlin statements that are to be executed repeatedly until a specified condition is met. The first looping statement we will explore is the *for* loop.

14.1.1 The Kotlin *for-in* Statement

The for-in loop is used to iterate over a sequence of items contained in a collection or number range.

The syntax of the for-in loop is as follows:

```
for variable name in collection or range {  
    // code to be executed  
}
```

In this syntax, *variable name* is the name to be used for a variable that will contain the current item from the collection or range through which the loop is iterating. The code in the body of the loop will typically use this name as a reference to the current item in the loop cycle. The *collection* or *range* references the item through which the loop is iterating. This could, for example, be an array of string values, a range operator, or even a string of characters.

Consider, for example, the following for-in loop construct:

```
for (index in 1..5) {  
    println("Value of index is $index")  
}
```

The loop begins by stating that the current item is to be assigned to a constant named *index*. The statement then declares a closed range operator to indicate that the for loop is to iterate through a range of numbers, starting at 1 and ending at 5. The body of the loop simply prints out a message to the console indicating the current value assigned to the *index* constant, resulting in the following output:

```
Value of index is 1  
Value of index is 2  
Value of index is 3  
Value of index is 4  
Value of index is 5
```

The for-in loop is of particular benefit when working with collections such as arrays. In fact, the for-in loop can be used to iterate through any object that contains more than one item. The following loop, for example, outputs

Kotlin Control Flow

each of the characters in the specified string:

```
for (index in "Hello") {
    println("Value of index is $index")
}
```

The operation of a for-in loop may be configured using the *downTo* and *until* functions. The *downTo* function causes the for loop to work backward through the specified collection until the specified number is reached. The following for loop counts backward from 100 until the number 90 is reached:

```
for (index in 100 downTo 90) {
    print("$index.. ")
}
```

When executed, the above loop will generate the following output:

```
100.. 99.. 98.. 97.. 96.. 95.. 94.. 93.. 92.. 91.. 90..
```

The *until* function operates in much the same way with the exception that counting starts from the bottom of the collection range and works up until (but not including) the specified endpoint (a concept referred to as a half-closed range):

```
for (index in 1 until 10) {
    print("$index.. ")
}
```

The output from the above code will range from the start value of 1 through to 9:

```
1.. 2.. 3.. 4.. 5.. 6.. 7.. 8.. 9..
```

The increment used on each iteration through the loop may also be defined using the *step* function as follows:

```
for (index in 0 until 100 step 10) {
    print("$index.. ")
}
```

The above code will result in the following console output:

```
0.. 10.. 20.. 30.. 40.. 50.. 60.. 70.. 80.. 90..
```

14.1.2 The *while* loop

The Kotlin *for* loop described previously works well when it is known in advance how many times a particular task needs to be repeated in a program. There will, however, be instances where code needs to be repeated until a certain condition is met, with no way of knowing in advance how many repetitions are going to be needed to meet that criterion. To address this need, Kotlin includes the *while* loop.

Essentially, the *while* loop repeats a set of tasks while a specified condition is met. The *while* loop syntax is defined as follows:

```
while condition {
    // Kotlin statements go here
}
```

In the above syntax, *condition* is an expression that will return either *true* or *false* and the *// Kotlin statements go here* comment represents the code to be executed while the condition expression is true. For example:

```
var myCount = 0
```

```
while (myCount < 100) {
```

```

    myCount++
    println(myCount)
}

```

In the above example, the *while* expression will evaluate whether the *myCount* variable is less than 100. If it is already greater than 100, the code in the braces is skipped and the loop exits without performing any tasks.

If, on the other hand, *myCount* is not greater than 100 the code in the braces is executed and the loop returns to the *while* statement and repeats the evaluation of *myCount*. This process repeats until the value of *myCount* is greater than 100, at which point the loop exits.

14.1.3 The *do ... while* loop

It is often helpful to think of the *do ... while* loop as an inverted *while* loop. The *while* loop evaluates an expression before executing the code contained in the body of the loop. If the expression evaluates to *false* on the first check then the code is not executed. The *do ... while* loop, on the other hand, is provided for situations where you know that the code contained in the body of the loop will *always* need to be executed at least once. For example, you may want to keep stepping through the items in an array until a specific item is found. You know that you have to at least check the first item in the array to have any hope of finding the entry you need. The syntax for the *do ... while* loop is as follows:

```

do {
    // Kotlin statements here
} while conditional expression

```

In the *do ... while* example below the loop will continue until the value of a variable named *i* equals 0:

```

var i = 10

do {
    i--
    println(i)
} while (i > 0)

```

14.1.4 Breaking from Loops

Having created a loop, it is possible that under certain conditions you might want to break out of the loop before the completion criteria have been met (particularly if you have created an infinite loop). One such example might involve continually checking for activity on a network socket. Once activity has been detected it will most likely be necessary to break out of the monitoring loop and perform some other task.

To break out of a loop, Kotlin provides the *break* statement which breaks out of the current loop and resumes execution at the code directly after the loop. For example:

```

var j = 10

for (i in 0..100)
{
    j += j

    if (j > 100) {
        break
    }
}

```

Kotlin Control Flow

```
    println("j = $j")
}
```

In the above example, the loop will continue to execute until the value of *j* exceeds 100 at which point the loop will exit and execution will continue with the next line of code after the loop.

14.1.5 The *continue* statement

The *continue* statement causes all remaining code statements in a loop to be skipped, and execution to be returned to the top of the loop. In the following example, the *println* function is only called when the value of variable *i* is an even number:

```
var i = 1

while (i < 20)
{
    i += 1

    if (i % 2 != 0) {
        continue
    }

    println("i = $i")
}
```

The *continue* statement in the above example will cause the *println* call to be skipped unless the value of *i* can be divided by 2 with no remainder. If the *continue* statement is triggered, execution will skip to the top of the while loop and the statements in the body of the loop will be repeated (until the value of *i* exceeds 19).

14.1.6 Break and continue labels

Kotlin expressions may be assigned a label by preceding the expression with a label name followed by the @ sign. This label may then be referenced when using break and continue statements to designate where execution is to resume. This is particularly useful when breaking out of nested loops. The following code contains a for loop nested within another for loop. The inner loop contains a break statement which is executed when the value of *j* reaches 10:

```
for (i in 1..100) {

    println("Outer loop i = $i")

    for (j in 1..100) {
        println("Inner loop j = $j")
        if (j == 10) break
    }
}
```

As currently implemented, the break statement will exit the inner for loop but execution will resume at the top of the outer for loop. Suppose, however, that the break statement is required to also exit the outer loop. This can be achieved by assigning a label to the outer loop and referencing that label with the break statement as follows:

```
outerloop@ for (i in 1..100) {
```

```
println("Outer loop i = $i")

for (j in 1..100) {

    println("Inner loop j = $j")

    if (j == 10) break@outerloop
}
}
```

Now when the value assigned to variable `j` reaches 10 the `break` statement will break out of both loops and resume execution at the line of code immediately following the outer loop.

14.2 Conditional control flow

In the previous chapter, we looked at how to use logical expressions in Kotlin to determine whether something is *true* or *false*. Since programming is largely an exercise in applying logic, much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed and, conversely, which code gets bypassed when the program is executing.

14.2.1 Using the *if* expressions

The *if* expression is perhaps the most basic of control flow options available to the Kotlin programmer. Programmers who are familiar with C, Swift, C++, or Java will immediately be comfortable using Kotlin *if* statements, although there are some subtle differences.

The basic syntax of the Kotlin *if* expression is as follows:

```
if (boolean expression) {
    // Kotlin code to be performed when expression evaluates to true
}
```

Unlike some other programming languages, it is important to note that the braces are optional in Kotlin if only one line of code is associated with the *if* expression. In fact, in this scenario, the statement is often placed on the same line as the *if* expression.

Essentially if the *Boolean expression* evaluates to *true* then the code in the body of the statement is executed. If, on the other hand, the expression evaluates to *false* the code in the body of the statement is skipped.

For example, if a decision needs to be made depending on whether one value is greater than another, we would write code similar to the following:

```
val x = 10

if (x > 9) println("x is greater than 9!")
```

Clearly, `x` is indeed greater than 9 causing the message to appear in the console panel.

At this point, it is important to notice that we have been referring to the *if* expression instead of the *if* statement. The reason for this is that unlike the *if* statement in other programming languages, the Kotlin *if* returns a result. This allows *if* constructs to be used within expressions. As an example, a typical *if* expression to identify the largest of two numbers and assign the result to a variable might read as follows:

```
if (x > y)
    largest = x
else
```

Kotlin Control Flow

```
largest = y
```

The same result can be achieved using the *if* statement within an expression using the following syntax:

```
variable = if (condition) return_val_1 else return_val_2
```

The original example can, therefore be re-written as follows:

```
val largest = if (x > y) x else y
```

The technique is not limited to returning the values contained within the condition. The following example is also a valid use of *if* in an expression, in this case assigning a string value to the variable:

```
val largest = if (x > y) "x is greatest" else "y is greatest"  
println(largest)
```

For those familiar with programming languages such as Java, this feature allows code constructs similar to ternary statements to be implemented in Kotlin.

14.2.2 Using *if... else ...* expressions

The next variation of the *if* expression allows us to also specify some code to perform if the expression in the *if* expression evaluates to *false*. The syntax for this construct is as follows:

```
if (boolean expression) {  
    // Code to be executed if expression is true  
} else {  
    // Code to be executed if expression is false  
}
```

The braces are, once again, optional if only one line of code is to be executed.

Using the above syntax, we can now extend our previous example to display a different message if the comparison expression evaluates to be *false*:

```
val x = 10  
  
if (x > 9) println("x is greater than 9!")  
    else println("x is less than 9!")
```

In this case, the second `println` statement will execute if the value of `x` was less than 9.

14.2.3 Using *if... else if... Expressions*

So far we have looked at *if* statements that make decisions based on the result of a single logical expression. Sometimes it becomes necessary to make decisions based on several different criteria. For this purpose, we can use the *if... else if... construct*, an example of which is as follows:

```
var x = 9  
  
if (x == 10) println("x is 10")  
    else if (x == 9) println("x is 9")  
        else if (x == 8) println("x is 8")  
            else println("x is less than 8")  
}
```

14.2.4 Using the *when* statement

The Kotlin *when* statement provides a cleaner alternative to the *if... else if... construct* and uses the following syntax:


```
when (value) {  
    match1 -> // code to be executed on match  
    match2 -> // code to be executed on match  
    .  
    .  
    else -> // default code to executed if no match  
}
```

Using this syntax, the previous *if... else if...* construct can be rewritten to use the *when* statement:

```
when (x) {  
    10 -> println ("x is 10")  
    9 -> println("x is 9")  
    8 -> println("x is 8")  
    else -> println("x is less than 8")  
}
```

The *when* statement is similar to the *switch* statement found in many other programming languages.

14.3 Summary

The term *control flow* is used to describe the logic that dictates the execution path that is taken through the source code of an application as it runs. This chapter has looked at the two types of control flow provided by Kotlin (looping and conditional) and explored the various Kotlin constructs that are available to implement both forms of control flow logic.

15. An Overview of Kotlin Functions and Lambdas

Kotlin functions and lambdas are a vital part of writing well-structured and efficient code and provide a way to organize programs while avoiding code repetition. In this chapter, we will look at how functions and lambdas are declared and used within Kotlin.

15.1 What is a function?

A function is a named block of code that can be called upon to perform a specific task. It can be provided data on which to perform the task and is capable of returning results to the code that called it. For example, if a particular arithmetic calculation needs to be performed in a Kotlin program, the code to perform the arithmetic can be placed in a function. The function can be programmed to accept the values on which the arithmetic is to be performed (referred to as parameters) and to return the result of the calculation. At any point in the program code where the calculation is required, the function is simply called, parameter values passed through as arguments and the result returned.

The terms parameter and argument are often used interchangeably when discussing functions. There is, however, a subtle difference. The values that a function can accept when it is called are referred to as parameters. At the point that the function is called and passed those values, however, they are referred to as arguments.

15.2 How to declare a Kotlin function

A Kotlin function is declared using the following syntax:

```
fun <function name> (<para name>: <para type>, <para name>: <para type>, ... ):  
<return type> {  
    // Function code  
}
```

This combination of function name, parameters, and return type is referred to as the function *signature* or *type*. Explanations of the various fields of the function declaration are as follows:

- `fun` – The prefix keyword used to notify the Kotlin compiler that this is a function.
- `<function name>` - The name assigned to the function. This is the name by which the function will be referenced when it is called from within the application code.
- `<para name>` - The name by which the parameter is to be referenced in the function code.
- `<para type>` - The type of the corresponding parameter.
- `<return type>` - The data type of the result returned by the function. If the function does not return a result then no return type is specified.
- `Function code` - The code of the function that does the work.

As an example, the following function takes no parameters, returns no result, and simply displays a message:

```
fun sayHello() {
```

An Overview of Kotlin Functions and Lambdas

```
println("Hello")
}
```

The following sample function, on the other hand, takes an integer and a string as parameters and returns a string result:

```
fun buildMessageFor(name: String, count: Int): String {
    return("$name, you are customer number $count")
}
```

15.3 Calling a Kotlin function

Once declared, functions are called using the following syntax:

```
<function name> (<arg1>, <arg2>, ... )
```

Each argument passed through to a function must match the parameters the function is configured to accept. For example, to call a function named `sayHello` that takes no parameters and returns no value, we would write the following code:

```
sayHello()
```

In the case of a message that accepts parameters, the function could be called as follows:

```
buildMessageFor("John", 10)
```

15.4 Single expression functions

When a function contains a single expression, it is not necessary to include the braces around the expression. All that is required is an equals sign (=) after the function declaration followed by the expression. The following function contains a single expression declared in the usual way:

```
fun multiply(x: Int, y: Int): Int {
    return x * y
}
```

Below is the same function expressed as a single line expression:

```
fun multiply(x: Int, y: Int): Int = x * y
```

When using single-line expressions, the return type may be omitted in situations where the compiler can infer the type returned by the expression making for even more compact code:

```
fun multiply(x: Int, y: Int) = x * y
```

15.5 Local functions

A local function is a function that is embedded within another function. In addition, a local function has access to all of the variables contained within the enclosing function:

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

    val name = "John"
    val count = 5

    fun displayString() {
        for (index in 0..count) {
            println(name)
        }
    }
}
```

```
        displayString()
    }
}
```

15.6 Handling return values

To call a function named `buildMessage` that takes two parameters and returns a result, on the other hand, we might write the following code:

```
val message = buildMessageFor("John", 10)
```

To improve code readability, the parameter names may also be specified when making the function call:

```
val message = buildMessageFor(name = "John", count = 10)
```

In the above examples, we have created a new variable called `message` and then used the assignment operator (`=`) to store the result returned by the function.

15.7 Declaring default function parameters

Kotlin provides the ability to designate a default parameter value to be used if the value is not provided as an argument when the function is called. This simply involves assigning the default value to the parameter when the function is declared.

To see default parameters in action the `buildMessageFor` function will be modified so that the string “Customer” is used as a default if a customer name is not passed through as an argument. Similarly, the `count` parameter is declared with a default value of 0:

```
fun buildMessageFor(name: String = "Customer", count: Int = 0): String {
    return("$name, you are customer number $count")
}
```

When parameter names are used when making the function call, any parameters for which defaults have been specified may be omitted. The following function call, for example, omits the customer name argument but still compiles because the parameter name has been specified for the second argument:

```
val message = buildMessageFor(count = 10)
```

If parameter names are not used within the function call, however, only the trailing arguments may be omitted:

```
val message = buildMessageFor("John") // Valid
val message = buildMessageFor(10) // Invalid
```

15.8 Variable number of function parameters

It is not always possible to know in advance the number of parameters a function will need to accept when it is called within the application code. Kotlin handles this possibility through the use of the `vararg` keyword to indicate that the function accepts an arbitrary number of parameters of a specified data type. Within the body of the function, the parameters are made available in the form of an array object. The following function, for example, takes as parameters a variable number of String values and then outputs them to the console panel:

```
fun displayStrings(vararg strings: String)
{
    for (string in strings) {
        println(string)
    }
}
```

```
displayStrings("one", "two", "three", "four")
```

An Overview of Kotlin Functions and Lambdas

Kotlin does not permit multiple vararg parameters within a function and any single parameters supported by the function must be declared before the vararg declaration:

```
fun displayStrings(name: String, vararg strings: String)
{
    for (string in strings) {
        println(string)
    }
}
```

15.9 Lambda expressions

Having covered the basics of functions in Kotlin it is now time to look at the concept of lambda expressions. Essentially, lambdas are self-contained blocks of code. The following code, for example, declares a lambda, assigns it to a variable named `sayHello`, and then calls the function via the lambda reference:

```
val sayHello = { println("Hello") }
sayHello()
```

Lambda expressions may also be configured to accept parameters and return results. The syntax for this is as follows:

```
{<para name>: <para type>, <para name> <para type>, ... ->
    // Lambda expression here
}
```

The following lambda expression, for example, accepts two integer parameters and returns an integer result:

```
val multiply = { val1: Int, val2: Int -> val1 * val2 }
val result = multiply(10, 20)
```

Note that the above lambda examples have assigned the lambda code block to a variable. This is also possible when working with functions. Of course, the following syntax will execute the function and assign the result of that execution to a variable, instead of assigning the function itself to the variable:

```
val myvar = myfunction()
```

To assign a function reference to a variable, simply remove the parentheses and prefix the function name with double colons (`::`) as follows. The function may then be called simply by referencing the variable name:

```
val myvar = ::myfunction
myvar()
```

A lambda block may be executed directly by placing parentheses at the end of the expression including any arguments. The following lambda directly executes the multiplication lambda expression multiplying 10 by 20.

```
val result = { val1: Int, val2: Int -> val1 * val2 }(10, 20)
```

The last expression within a lambda serves as the expression's return value (hence the value of 200 being assigned to the result variable in the above multiplication examples). In fact, unlike functions, lambdas do not support the `return` statement. In the absence of an expression that returns a result (such as an arithmetic or comparison expression), simply declaring the value as the last item in the lambda will cause that value to be returned. The following lambda returns the Boolean `true` value after printing a message:

```
val result = { println("Hello"); true }()
```

Similarly, the following lambda simply returns a string literal:

```
val nextmessage = { println("Hello"); "Goodbye" }()
```

A particularly useful feature of lambdas and the ability to create function references is that they can be both passed to functions as arguments and returned as results. This concept, however, requires an understanding of function types and higher-order functions.

15.10 Higher-order functions

On the surface, lambdas and function references do not seem to be particularly compelling features. The possibilities that these features offer become more apparent, however, when we consider that lambdas and function references have the same capabilities as many other data types. In particular, these may be passed through as arguments to another function, or even returned as a result from a function.

A function that is capable of receiving a function or lambda as an argument, or returning one as a result is referred to as a *higher-order function*.

Before we look at what is, essentially, the ability to plug one function into another, it is first necessary to explore the concept of *function types*. The type of a function is dictated by a combination of the parameters it accepts and the type of result it returns. A function that accepts an Int and a Double as parameters and returns a String result for example is considered to have the following function type:

```
(Int, Double) -> String
```

To accept a function as a parameter, the receiving function simply declares the type of function it can accept.

As an example, we will begin by declaring two unit conversion functions:

```
fun inchesToFeet (inches: Double): Double {
    return inches * 0.0833333
}

fun inchesToYards (inches: Double): Double {
    return inches * 0.0277778
}
```

The example now needs an additional function, the purpose of which is to perform a unit conversion and print the result in the console panel. This function needs to be as general-purpose as possible, capable of performing a variety of different measurement unit conversions. To demonstrate functions as parameters, this new function will take as a parameter a function type that matches both the inchesToFeet and inchesToYards functions together with a value to be converted. Since the type of these functions is equivalent to (Double) -> Double, our general-purpose function can be written as follows:

```
fun outputConversion(converterFunc: (Double) -> Double, value: Double) {
    val result = converterFunc(value)
    println("Result of conversion is $result")
}
```

When the outputConversion function is called, it will need to be passed a function matching the declared type. That function will be called to perform the conversion and the result displayed in the console panel. This means that the same function can be called to convert inches to both feet and yards, simply by “plugging in” the appropriate converter function as a parameter, keeping in mind that it is the function reference that is being passed as an argument:

```
outputConversion(::inchesToFeet, 22.45)
outputConversion(::inchesToYards, 22.45)
```

Functions can also be returned as a data type simply by declaring the type of the function as the return type. The following function is configured to return either our inchesToFeet or inchesToYards function type (in other

An Overview of Kotlin Functions and Lambdas

words a function that accepts and returns a Double value) based on the value of a Boolean parameter:

```
fun decideFunction(feet: Boolean): (Double) -> Double
{
    if (feet) {
        return ::inchesToFeet
    } else {
        return ::inchesToYards
    }
}
```

When called, the function will return a function reference which can then be used to perform the conversion:

```
val converter = decideFunction(true)
val result = converter(22.4)
println(result)
```

15.11 Summary

Functions and lambda expressions are self-contained blocks of code that can be called upon to perform a specific task and provide a mechanism for structuring code and promoting reuse. This chapter has introduced the basic concepts of function and lambda declaration and implementation in addition to the use of higher-order functions that allow lambdas and functions to be passed as arguments and returned as results.

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:

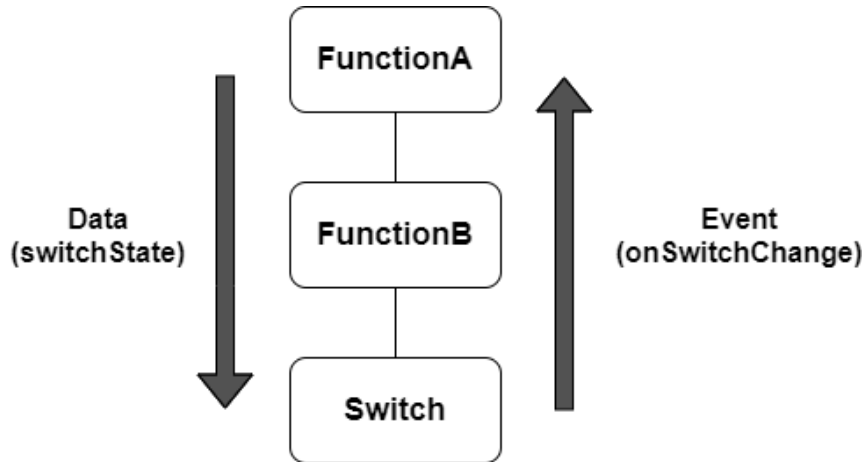


Figure 20-1

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:

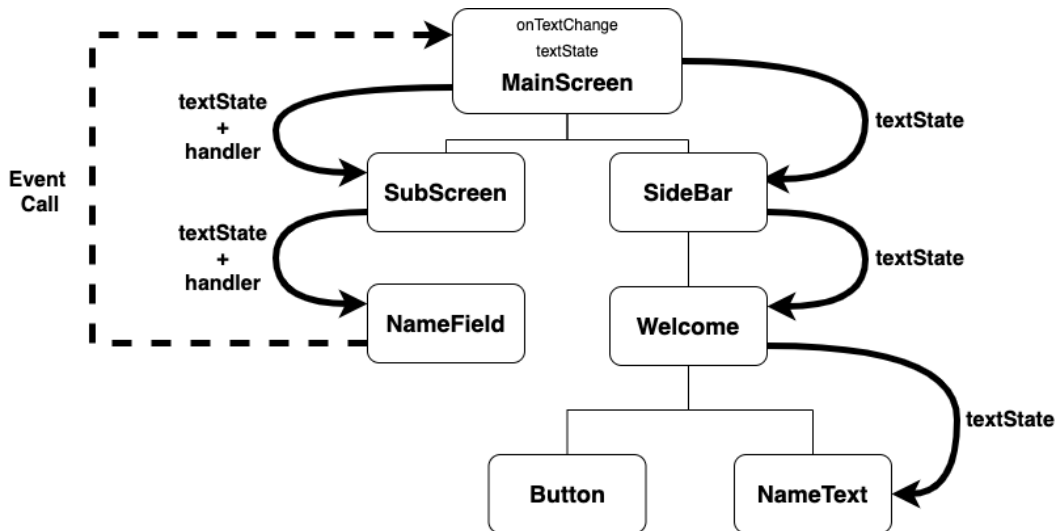


Figure 20-2

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/>.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.

34. 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.

34.1 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.

34.2 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.

34.3 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:

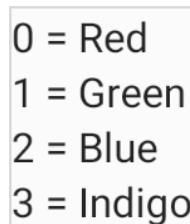
```
LazyColumn {  
    items(1000) { index ->  
        Text("This is item $index");  
    }  
}
```

`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 34-1 below:



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

Figure 34-1

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.

34.4 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()
    }
}
```

34.5 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)  
    }  
    .  
    .  
})
```

34.6 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 34-2, 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 34-2

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 34-2 above.

34.7 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
}
```

34.8 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:

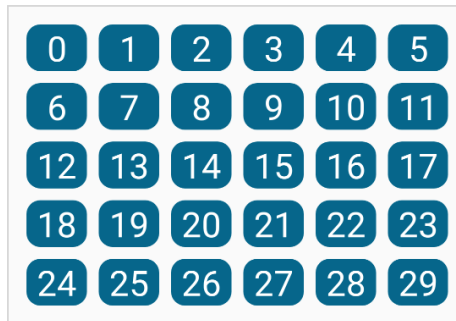


Figure 34-3

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 34-4 below:

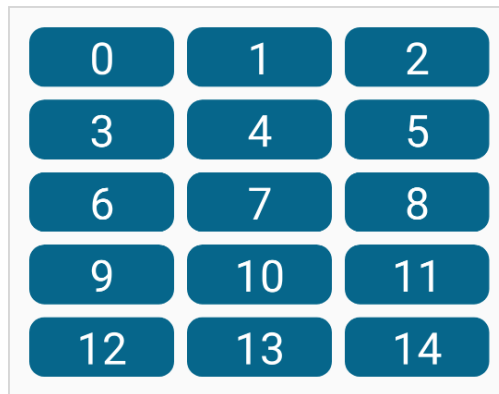


Figure 34-4

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 34-5:

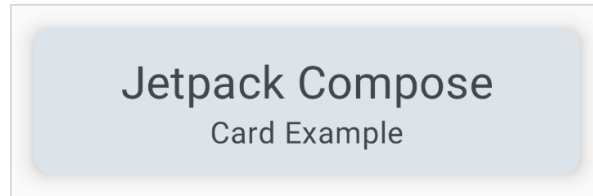


Figure 34-5

34.9 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.

42. 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.

42.1 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.

42.2 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.

42.3 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.

42.4 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 42-1 illustrates this concept as it relates to activities and ViewModels:

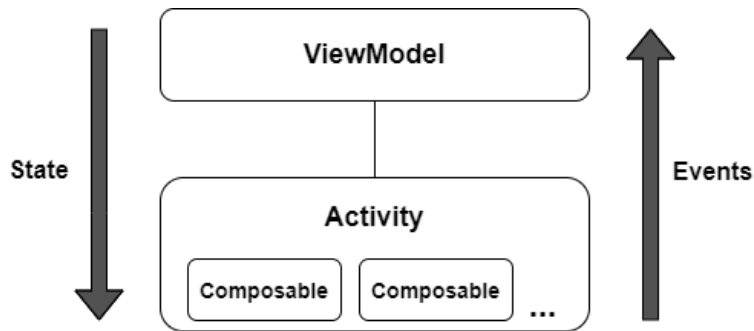


Figure 42-1

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*”.

42.5 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.

42.6 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:2.4.1'  
    .  
    .  
}
```

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”.

42.7 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.

42.8 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*”.

42.9 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.

51. An Introduction to Kotlin Flow

The earlier chapter, “*Coroutines and LaunchedEffects in Jetpack Compose*” taught us about Kotlin Coroutines. It explained how we can use them to perform multiple tasks concurrently without blocking the main thread. However, a shortcoming of suspend functions is that they are typically only useful for performing tasks that either do not return a result or only return a single value. In this chapter, we will introduce Kotlin Flows and explore how these can be used to return sequential streams of results from coroutine-based tasks.

By the end of the chapter, you should understand the Flow, StateFlow, and SharedFlow Kotlin types and appreciate the difference between hot and cold flow streams. In the next chapter (“*A Jetpack Compose SharedFlow Tutorial*”), we will look more closely at using SharedFlow within the context of an example Android app project.

51.1 Understanding Flows

Flows are a part of the Kotlin programming language and are designed to allow multiple values to be returned sequentially from coroutine-based asynchronous tasks. A stream of data arriving over time via a network connection would, for example, be an ideal situation for using a Kotlin flow.

Flows are comprised of *producers*, *intermediaries*, and *consumers*. Producers are responsible for providing the data that makes up the flow. The code that retrieves the stream of data from our hypothetical network connection, for example, would be considered a producer. As each data value becomes available, the producer *emits* that value to the flow. The consumer sits at the opposite end of the flow stream and collects the values as the producer emits them.

Intermediaries may be placed between the producer and consumer to perform additional operations on the data, such as filtering the stream, performing further processing, or transforming the data in other ways before it reaches the consumer. Figure 51-1 illustrates the typical structure of a Kotlin flow:

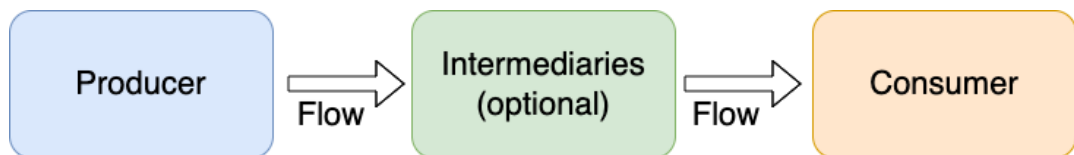


Figure 51-1

The flow shown in the above diagram consists of a single producer and consumer. However, in practice, multiple consumers can collect emissions from a single producer, and for a single consumer to collect data from multiple producers.

The remainder of this chapter will demonstrate many key features of Kotlin flows within the context of Jetpack Compose-based development.

51.2 Creating the sample project

Launch Android Studio and create a new *Empty Activity* project named FlowDemo, specifying *com.example.flowdemo* as the package name and selecting a minimum API level of API 26: Android 8.0 (Oreo).

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

ScreenSetup which, in turn, calls a function named mainScreen:

```
@Composable
fun ScreenSetup() {
    MainScreen()
}

@Composable
fun MainScreen() {

}
```

Edit the *onCreate()* method function to call ScreenSetup instead of Greeting (we will modify the GreetingPreview composable later).

Next, modify the *build.gradle (Module: app)* file to add the Compose view model and Kotlin runtime extensions libraries to the dependencies section:

```
dependencies {
    .
    .
    implementation 'androidx.lifecycle:lifecycle-viewmodel-compose:2.5.1'
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.5.1'
    .
    .
}
```

When prompted, click on the Sync Now button at the top of the editor panel to commit the change.

51.3 Adding a view model to the project

For this project, the flow will reside in a view model class. Add this model to the project by locating and right-clicking on the *app -> java -> com.example.flowdemo* entry in the project tool window and selecting the *New -> Kotlin Class/File* menu option. In the resulting dialog, name the class *DemoViewModel* before tapping the keyboard Enter key. Once created, modify the file so that it reads as follows:

```
package com.example.flowdemo

import androidx.lifecycle.ViewModel

class DemoViewModel : ViewModel() {
}
```

Return to the *MainActivity.kt* file and make changes to access an instance of the view model:

```
.
.
import androidx.lifecycle.viewmodel.compose.viewModel
.
.
@Composable
fun ScreenSetup(viewModel: DemoViewModel = viewModel()) {
    MainScreen()
}
```

```
}
```

51.4 Declaring the flow

The Kotlin Flow type represents the most basic form of flow. Each flow can only emit data of a single type which must be specified when the flow is declared. The following declaration, for example, declares a Flow instance designed to stream String-based data:

```
Flow<String>
```

When declaring a flow, we need to assign the code to generate the data stream. This code is referred to as the *producer block*. This can be achieved using the *flow()* builder, which takes as a parameter a coroutine suspend block containing the producer block code. For example, add the following code to the *DemoViewModel.kt* file to declare a flow named *myFlow* designed to emit a stream of integer values:

```
package com.example.flowdemo

import androidx.lifecycle.ViewModel
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

class DemoViewModel : ViewModel() {

    val myFlow: Flow<Int> = flow {
        // Producer block
    }
}
```

As an alternative to the flow builder, the *flowOf()* builder can be used to convert a fixed set of values into a flow:

```
val myFlow2 = flowOf(2, 4, 6, 8)
```

Also, many Kotlin collection types now include an *asFlow()* extension function that can be called to convert the contained data to a flow. The following code, for example, converts an array of string values to a flow:

```
val myArrayFlow = arrayOf<String>("Red", "Green", "Blue").asFlow()
```

51.5 Emitting flow data

Once a flow has been built, the next step is to ensure the data is emitted so that it reaches any consumers observing it. Of the three flow builders we looked at in the previous section, only the *flowOf()* and *asFlow()* builders create flows that automatically emit the data as soon as a consumer starts collecting. In the case of the *flow* builder, however, we need to write code to manually emit each value as it becomes available. We achieve this by making calls to the *emit()* function and passing through as an argument the current value to be streamed. The following changes to our *myFlow* declaration implement a loop that emits the value of an incrementing counter. In addition, a 2-second delay is performed on each loop iteration to demonstrate the asynchronous nature of flow streams:

```
val myFlow: Flow<Int> = flow {
    for (i in 0..9) {
        emit(i)
        delay(2000)
    }
}
```

51.6 Collecting flow data as state

As we will see later in the chapter, one way to collect data from a flow within a consumer is to call the `collect()` method on the flow instance. When working with Compose, however, a less flexible, but more convenient option is to convert the flow to state by calling the `collectAsState()` function on the flow instance. This allows us to treat the data just as we would any other state within our code. To see this in action, edit the `MainActivity.kt` file and make the following changes:

```
.
.
import androidx.compose.runtime.*
import kotlinx.coroutines.flow.*
.
.
@Composable
fun ScreenSetup(viewModel: DemoViewModel = viewModel()) {
    MainScreen(viewModel.myFlow)
}

@Composable
fun MainScreen(flow: Flow<Int>) {
    val count by flow.collectAsState(initial = 0)
}
.
.
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    FlowDemoTheme {
        ScreenSetup(viewModel())
    }
}
```

The changes pass a `myFlow` reference to the `MainScreen` composable where it is converted to a `State` with an initial value of 0. Next, we need to design a simple user interface to display the count values as they are emitted to the flow:

```
.
.
import androidx.compose.foundation.layout.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.unit.sp
.
.
@Composable
fun MainScreen(myFlow: Flow<Int>) {
    val count by myFlow.collectAsState(initial = 0)
```



```

Column(
    modifier = Modifier.fillMaxSize(),
    verticalArrangement = Arrangement.Center,
    horizontalAlignment = Alignment.CenterHorizontally
) {
    Text(text = "$count", style = TextStyle(fontSize = 40.sp))
}
}

```

Try out the app either using the preview panel in interactive mode, or by running it on a device or emulator. Once the app starts, the count value displayed on the Text component should increment as the flow emits each new value.

51.7 Transforming data with intermediaries

In the previous example, we passed the data values to the consumer without any modifications. However, we can change the data between the producer and consumer by applying one or more *intermediate flow operators*. In this section, we will look at some of these operators.

We can use the `map()` operator to convert the value to another value. For example, we can use `map()` to convert our integer value to a string and add some additional text. Edit the `DemoViewModel.kt` file and create a modified version of our flow as follows:

```

.
.
class DemoViewModel : ViewModel() {

    val myFlow: Flow<Int> = flow {
        for (i in 0..9) {
            emit(i)
            delay(2000)
        }
    }

    val newFlow = myFlow.map {
        "Current value = $it"
    }
}

```

Before we can test this operator, some changes are needed within the `MainActivity.kt` file to use this new flow:

```

Composable
fun ScreenSetup(viewModel: DemoViewModel = viewModel()) {
    MainScreen(viewModel.newFlow)
}

@Composable
fun MainScreen(flow: Flow<String>) {
    val count by flow.collectAsState(initial = "Current value =")
.
.

```

When the code is executed, the text will display the text string updated with the count:

```
Current value = 1
Current value = 2
.
.
```

The *map()* operator will perform the conversion on every collected value. We can use the *filter()* operator to control which values get collected. The filter code block must contain an expression that returns a Boolean value. Only if the expression evaluates to true does the value pass through to the collection. For example, the following code filters odd numbers out of the data flow (note that we've left the *map()* operator in place to demonstrate the chaining of operators):

```
val newFlow = myFlow
    .filter {
        it % 2 == 0
    }
    .map {
        "Current value = $it"
    }
```

The above changes will display count updates only for even numbers.

The *transform()* operator serves a similar purpose to *map()* but provides more flexibility. The *transform()* operator also needs to manually emit the modified result. A particular advantage of *transform()* is that it can emit multiple values, for example:

```
val newFlow = myFlow
    .transform {
        emit("Value = $it")
        delay(1000)
        val doubled = it * 2
        emit("Value doubled = $doubled")
    }
```

```
// Output
Value = 0
Value doubled = 0
Value = 1
Value doubled = 2
Value = 2
Value doubled = 4
Value = 3
.
.
```

Before moving to the next step, revert the *newFlow* declaration to its original form:

```
val newFlow = myFlow.map {
    "Current value = $it"
}
```

51.8 Collecting flow data

So far in this chapter, we have used the `collectAsState()` function to convert a flow to a State instance. Behind the scenes, this method uses the `collect()` function to initiate the data collection. Although `collectAsState()` works well most of the time, there will be situations where you may need to call `collect()`. In fact, `collect()` is just one of several so-called *terminal flow operators* that can be called directly to achieve results that aren't possible using `collectAsState()`.

These operators are *suspend* functions so can only be called from within a coroutine scope. In the chapter entitled “*Coroutines and LaunchedEffects in Jetpack Compose*”, we looked at coroutines and explained how to use `LaunchedEffect` to execute asynchronous code safely from within a composable function. Once we have implemented the `LaunchedEffect` call, we still need the streamed values to be stored as state, so we also need a mutable state into which to store the latest value. Bringing these requirements together, modify the `MainScreen` function so that it reads as follows:

```
@Composable
fun MainScreen(flow: Flow<String>) {

    var count by remember { mutableStateOf<String>("Current value =")}

    LaunchedEffect(Unit) {
        flow.collect {
            count = it
        }
    }

    Column(
        modifier = Modifier.fillMaxSize(),
        .
        .
    )
}
```

Test the app and verify that the text component updates as expected. Now that we are using the `collect()` function we can begin to explore some options that were not available to us when we were using `collectAsState()`.

For example, to add code to be executed when the stream ends, the collection can be performed in a *try/finally* construct, for example:

```
LaunchedEffect(Unit) {
    try {
        flow.collect {
            count = it
        }
    } finally {
        count = "Flow stream ended."
    }
}
```

The `collect()` operator will collect every value emitted by the producer, even if new values are emitted while the last value is still being processed in the consumer. For example, our producer is configured to emit a new value every two seconds. Suppose, however, that we simulate our consumer taking 2.5 seconds to process each collected value. When executed, we will still see all of the values listed in the output because `collect()` does

not discard any uncollected values regardless of whether more recent values have been emitted since the last collection. This type of behavior is essential to avoid data loss within the flow. In some situations, however, the consumer may be uninterested in any intermediate values emitted between the most recently processed value and the latest emitted value. In this case, the `collectLatest()` operator can be called on the flow instance. This operator works by canceling the current collection if a new value arrives before processing completes on the previous value and restarts the process on the latest value.

The `conflate()` operator is similar to the `collectLatest()` operator except that instead of canceling the current collection operation when a new value arrives, `conflate()` allows the current operation to complete, but discards intermediate values that arrive during this process. When the current operation completes, the most recent value is then collected.

Another collection operator is the `single()` operator. This operator collects a single value from the flow and throws an exception if it finds another value in the stream. This operator is useful where the appearance of a second stream value indicates that something else has gone wrong somewhere in the app or data source.

51.9 Adding a flow buffer

When a consumer takes time to process the values emitted by a producer, there is the potential for execution time inefficiencies to occur. Suppose, for example, that in addition to the two-second delay between each emission from our `newFlow` producer, the collection process in our consumer takes an additional second to complete. We can simulate this behavior as follows:

```
.
.
import kotlin.system.measureTimeMillis
import kotlinx.coroutines.delay
.
.
LaunchedEffect(Unit) {

    val elapsedTime = measureTimeMillis {
        flow.collect {
            count = it
            delay(1000)
        }
    }
    count = "Duration = $elapsedTime"
}
```

To allow us to measure the total time to fully process the flow, the consumer code has been placed in the closure of a call to the Kotlin `measureTimeMillis()` function. Run the app and, after execution completes, a duration similar to the following will be reported:

```
Duration = 30044
```

This accounts for approximately 20 seconds to process the 10 values within `newFlow` and an additional 10 seconds for those values to be collected. There is an inefficiency here because the producer is waiting for the consumer to process each value before starting on the next value. This would be much more efficient if the producer did not have to wait for the consumer. We could, of course, use the `collectLatest()` or `conflate()` operators, but only if the loss of intermediate values is not a concern. To speed up the processing while also collecting every emitted value we can make use of the `buffer()` operator. This operator buffers values as they are emitted and passes them

to the consumer when it is ready to receive them. This allows the producer to continue emitting values while the consumer processes preceding values while ensuring that every emitted value is collected. The *buffer()* operator may be applied to a flow as follows:

```
LaunchedEffect("Unit") {

    val elapsedTime = measureTimeMillis {
        flow
            .buffer()
            .collect {
                count = it
                delay(1000)
            }
    }
    count = "Duration = $elapsedTime"
}
```

Execution of the above code indicates that we have now reclaimed the 10 seconds previously lost in the collection code:

```
Duration = 20052
```

51.10 More terminal flow operators

The *reduce()* operator is one of several other terminal flow operators that can be used in place of a collection operator to make changes to the flow data. The *reduce()* operator takes two parameters in the form of an *accumulator* and a *value*. The first flow value is placed in the accumulator and a specified operation is performed between the accumulator and the current value (with the result stored in the accumulator). To try this out we need to revert to using *myFlow* instead of *newFlow* in addition to adding the *reduce()* operator call:

```
@Composable
fun ScreenSetup(viewModel: DemoViewModel = viewModel()) {
    MainScreen(viewModel.myFlow)
}

@Composable
fun MainScreen(flow: Flow<Int>) {

    var count by remember { mutableStateOf<Int>(0) }

    LaunchedEffect(Unit) {

        flow
            .reduce { accumulator, value ->
                count = accumulator
                accumulator + value
            }
    }
}
```

The *fold()* operator works similarly to the *reduce()* operator, with the exception that it is passed an initial accumulator value:

```
LaunchedEffect(Unit) {  
  
    flow  
        .fold(10) { accumulator, value ->  
            count = accumulator  
            accumulator + value  
        }  
}
```

51.11 Flow flattening

As we have seen in earlier examples, we can use operators to perform tasks on values collected from a flow. An interesting situation occurs, however, when that task itself creates one or more flows resulting in a “flow of flows”. In situations where this occurs, these streams can be *flattened* into a single stream.

Consider the following example code which declares two flows:

```
val myFlow: Flow<Int> = flow {  
    for (i in 1..5) {  
        delay(1000)  
        emit(i)  
    }  
}  
  
fun doubleIt(value: Int) = flow {  
    emit(value)  
    delay(1000)  
    emit(value + value)  
}
```

If we were to call *doubleIt()* for each value in the *myFlow* stream we would end up with a separate flow for each value. This problem can be solved by concatenating the *doubleIt()* streams into a single flow using the *flatMapConcat()* operator as follows:

```
@Composable  
fun ScreenSetup(viewModel: DemoViewModel = viewModel()) {  
    MainScreen(viewModel)  
}  
  
@Composable  
fun MainScreen(viewModel: DemoViewModel) {
```

```

var count by remember { mutableStateOf<Int>(0) }

LaunchedEffect(Unit) {

    viewModel.myFlow
        .flatMapConcat { viewModel.doubleIt(it) }
        .collect { count = it }
}

```

When this modified code executes we will see the following output from the `collect()` operator:

```

1
2
2
4
3
6
4
8
5
10

```

As we can see from the output, the `doubleIt()` flow has emitted the value provided by `myFlow` followed by the doubled value. When using the `flatMapConcat()` operator, the `doubleIt()` calls are being performed synchronously, causing execution to wait until `doubleIt()` has emitted both values before processing the next flow value. The emitted values can instead be collected asynchronously using the `flatMapMerge()` operator as follows:

```

viewModel.myFlow
    .flatMapMerge { viewModel.doubleIt(it) }
    .collect {
        count = it
        println("Count = $it")
    }
}

```

Because the collection is being performed asynchronously the displayed value change too quickly to see all of the count values. Display the Logcat tool window to see the full list of collected values generated by the `println()` call:

```

I/System.out: Count = 1
I/System.out: Count = 2
I/System.out: Count = 2
I/System.out: Count = 4
I/System.out: Count = 3
I/System.out: Count = 6
I/System.out: Count = 4
I/System.out: Count = 8
I/System.out: Count = 5
I/System.out: Count = 10

```

51.12 Combining multiple flows

Multiple flows can be combined into a single flow using the `zip()` and `combine()` operators. The following code demonstrates the `zip()` operator being used to convert two flows into a single flow:

```
var count by remember { mutableStateOf<String>("") }

LaunchedEffect(Unit) {

    val flow1 = (1..5).asFlow()
        .onEach { delay(1000) }
    val flow2 = flowOf("one", "two", "three", "four")
        .onEach { delay(1500) }
    flow1.zip(flow2) { value, string -> "$value, $string" }
        .collect { count = it }
}

// Output
1, one
2, two
3, three
4, four
```

Note that we have applied the `onEach()` operator to both flows in the above code. This is a useful operator for performing a task on receipt of each stream value.

The `zip()` operator will wait until both flows have emitted a new value before performing the collection. The `combine()` operator works slightly differently in that it proceeds as soon as either flow emits a new value, using the last value emitted by the other flow in the absence of a new value:

```
.
.
val flow1 = (1..5).asFlow()
    .onEach { delay(1000) }
val flow2 = flowOf("one", "two", "three", "four")
    .onEach { delay(1500) }
flow1.combine(flow2) { value, string -> "$value, $string" }
    .collect { count = it }
.
.
// Output
1, one
2, one
3, one
3, two
4, two
4, three
5, three
5, four
```

As we can see from the output, multiple instances have occurred where the last value has been reused on a flow

because a new value was emitted on the other.

51.13 Hot and cold flows

So far in this chapter, we have looked exclusively at the Kotlin Flow type. Kotlin also provides additional types in the form of `StateFlow` and `SharedFlow`. Before exploring these, however, it is important to understand the concept of *hot* and *cold* flows.

A stream declared using the `Flow` type is referred to as a *cold flow* because the code within the producer does not begin executing until a consumer begins collecting values. `StateFlow` and `SharedFlow`, on the other hand, are referred to as *hot flows* because they begin emitting values immediately, regardless of whether any consumers are collecting the values.

Once a consumer begins collecting from a hot flow, it will receive the latest value emitted by the producer followed by any subsequent values. Unless steps are taken to implement caching, any previous values emitted before the collection starts will be lost.

Another important difference between `Flow`, `StateFlow`, and `SharedFlow` is that a Flow-based stream cannot have multiple collectors. Each Flow collector launches a new flow with its own independent data stream. With `StateFlow` and `SharedFlow`, on the other hand, multiple collectors share access to the same flow.

51.14 StateFlow

`StateFlow`, as the name suggests, is primarily used as a way to observe a change in state within an app such as the current setting of a counter, toggle button, or slider. Each `StateFlow` instance is used to store a single value that is likely to change over time and to notify all consumers when those changes occur. This enables you to write code that *reacts* to changes in state instead of code that has to continually check whether or not a state value has changed. `StateFlow` behaves the same way as `LiveData` with the exception that `LiveData` has lifecycle awareness and does not require an initial value (`LiveData` was covered previously in the chapter titled “*Working with ViewModels in Compose*”).

To create a `StateFlow` stream, begin by creating an instance of `MutableStateFlow`, passing through a mandatory initial value. This is the variable that will be used to change the current state value from within the app code:

```
private val _stateFlow = MutableStateFlow(0)
```

Next, call `asStateFlow()` on the `MutableStateFlow` instance to convert it into a `StateFlow` from which changes in state can be collected:

```
val stateFlow = _stateFlow.asStateFlow()
```

Once created, any changes to the state are made via the *value* property of the mutable state instance. The following code, for example, increments the state value:

```
_stateFlow.value += 1
```

Once the flow is active, the state can be consumed using `collectAsState()` or directly using a collection function, though it is generally recommended to collect from `StateFlow` using the `collectLatest()` operator. To try out an example, begin by making the following modifications to the `DemoViewModel.kt` file:

```
.
.
class DemoViewModel : ViewModel() {

    private val _stateFlow = MutableStateFlow(0)
    val stateFlow = _stateFlow.asStateFlow()
```

```
fun increaseValue() {  
    _stateFlow.value += 1  
}
```

Next, edit the *MainActivity.kt* file and change *MainScreen* so that it collects from the new state flow and to add a button configured to call the view model *increaseValue()* function:

```
.  
.  
import androidx.compose.material3.Button  
.  
.  
@Composable  
fun MainScreen(viewModel: DemoViewModel) {  
  
    val count by viewModel.stateFlow.collectAsState()  
  
    Column(  
        modifier = Modifier.fillMaxSize(),  
        verticalArrangement = Arrangement.Center,  
        horizontalAlignment = Alignment.CenterHorizontally  
    ) {  
        Text(text = "$count", style = TextStyle(fontSize = 40.sp))  
        Button(onClick = { viewModel.increaseValue() }) {  
            Text("Click Me")  
        }  
    }  
}
```

Run the app and verify that the button updates the count Text component with the incremented count value each time it is clicked.

51.15 SharedFlow

SharedFlow provides a more general-purpose streaming option than that offered by StateFlow. Some of the key differences between StateFlow and SharedFlow are as follows:

- Consumers are generally referred to as *subscribers*.
- An initial value is not provided when creating a SharedFlow instance.
- SharedFlow allows values that were emitted prior to collection starting to be “replayed” to the collector.
- SharedFlow *emits* values instead of using a *value* property.

SharedFlow instances are created using *MutableSharedFlow* as the backing property on which we call the *asSharedFlow()* function to obtain a SharedFlow reference. For example, make the following changes to the *DemoViewModel* class to declare a shared flow:

```
.  
.
```

```
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.channels.BufferOverflow
.
.
class DemoViewModel : ViewModel() {

    private val _sharedFlow = MutableSharedFlow<Int>(
        replay = 10,
        onBufferOverflow = BufferOverflow.DROP_OLDEST
    )

    val sharedFlow = _sharedFlow.asSharedFlow()
.
.
```

As configured above, new flow subscribers will receive the last 10 values before receiving any new values. The above flow is also configured to discard the oldest value when more than 10 values are buffered. The full set of options for handling buffer overflows are as follows:

- **DROP_LATEST** - The latest value is dropped when the buffer is full leaving the buffer unchanged as new values are processed.
- **DROP_OLDEST** - Treats the buffer as a “first-in, first-out” stack where the oldest value is dropped to make room for a new value when the buffer is full.
- **SUSPEND** - The flow is suspended when the buffer is full.

Values are emitted on a SharedFlow stream by calling the *emit()* method of the MutableSharedFlow instance from within a coroutine. Remaining in the *DemoViewModel.kt* file, add a new method that can be called from the main activity to start the shared flow:

```
fun startSharedFlow() {

    viewModelScope.launch {
        for (i in 1..5) {
            _sharedFlow.emit(i)
            delay(2000)
        }
    }
}
```

Finally, make the following changes to the MainScreen composable:

```
@Composable
fun MainScreen(viewModel: DemoViewModel) {

    val count by viewModel.sharedFlow.collectAsState(initial = 0)

    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
```

```
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(text = "$count", style = TextStyle(fontSize = 40.sp))
        Button(onClick = { viewModel.startSharedFlow() }) {
            Text("Click Me")
        }
    }
}
```

Run the app on a device or emulator (shared flow code does not always work in the interactive preview) and verify that clicking the button causes the count to begin updating. Note that since new values are being emitted from within a coroutine you can click on the button repeatedly and collect values from multiple flows.

One final point to note about shared flows is that the current number of subscribers to a SharedFlow stream can be obtained via the *subscriptionCount* property of the mutable instance:

```
val subCount = _sharedFlow.subscriptionCount
```

51.16 Converting a flow from cold to hot

A cold flow can be made hot by calling the *shareIn()* function on the flow. This call requires a coroutine scope in which to execute the flow, a replay value, and a start policy setting indicating the conditions under which the flow is to start and stop. The available start policy options are as follows:

- **SharingStarted.WhileSubscribed()** - The flow is kept alive as long as it has active subscribers.
- **SharingStarted.Eagerly()** - The flow begins immediately and remains active even in the absence of active subscribers.
- **SharingStarted.Lazily()** - The flow begins only after the first consumer subscribes and remains active even in the absence of active subscribers.

We could, for example, make one of our earlier cold flows hot using the following code:

```
val hotFlow = myFlow.shareIn(
    viewModelScope,
    replay = 1,
    started = SharingStarted.WhileSubscribed()
)
```

51.17 Summary

Kotlin flows allow sequential data or state changes to be returned over time from asynchronous tasks. A flow consists of a producer that emits a sequence of values and consumers that collect and process those values. The flow stream can be manipulated between the producer and consumer by applying one or more intermediary operators including transformations and filtering. Flows are created based on the Flow, StateFlow, and SharedFlow types. A Flow-based stream can only have a single collector while StateFlow and SharedFlow can have multiple collectors.

Flows are categorized as being hot or cold. A cold flow does not begin emitting values until a consumer begins collection. Hot flows, on the other hand, begin emitting values as soon as they are created, regardless of whether or not the values are being collected. In the case of SharedFlow, a predefined number of values may be buffered and subsequently replayed to new subscribers when they begin collecting values. A cold flow can be made hot via a call to the flow's *shareIn()* function.

54. 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.

54.1 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* 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.

54.2 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 54-1 below:

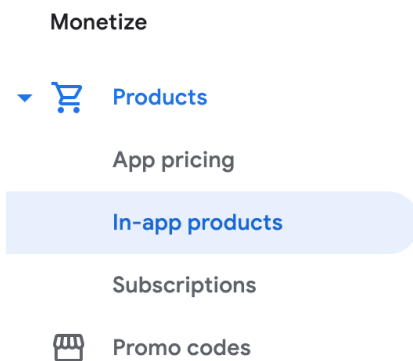


Figure 54-1

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.

54.3 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()

```

54.4 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
    }
})

```

54.5 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.

54.6 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.

54.7 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
    }
}

```

54.8 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
}
```

54.9 Summary

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

Index

Symbols

?. 99
 2D graphics 349
 @Composable 20, 143
 @ExperimentalFoundationApi 306
 :: operator 101
 @Preview 21
 showSystemUi 21

A

acknowledgePurchase() method 503
 Activity Manager 87
 adb
 command-line tool 65
 connection testing 71
 device pairing 69
 enabling on Android devices 65
 Linux configuration 68
 list devices 65
 macOS configuration 66
 overview 65
 restart server 66
 testing connection 71
 WiFi debugging 69
 Windows configuration 67
 Wireless debugging 69
 Wireless pairing 69
 AlertDialog 147
 align() 221
 alignByBaseline() 213
 Alignment.Bottom 207, 211
 Alignment.BottomCenter 219
 Alignment.BottomEnd 219
 Alignment.BottomStart 219
 Alignment.Center 219

Alignment.CenterEnd 219
 Alignment.CenterHorizontally 207
 Alignment.CenterStart 219
 Alignment.CenterVertically 207, 211
 Alignment.End 207
 alignment lines 229
 Alignment.Start 207
 Alignment.Top 207, 211
 Alignment.TopCenter 219
 Alignment.TopEnd 219
 Alignment.TopStart 219
 Android
 architecture 85
 runtime 86
 SDK Packages 6
 Android Architecture Components 369
 Android Debug Bridge. *See* ADB
 Android Development
 System Requirements 3
 Android Jetpack 369
 Android Libraries 86
 Android Monitor tool window 40
 Android Native Development Kit 87
 Android SDK Location
 identifying 9
 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 85
 Android Studio
 Animation Inspector 347
 Asset Studio 178
 changing theme 62

Index

- Database Inspector 417
- downloading 3
- Editor Window 57
- installation 4
- Layout Editor 141
- Linux installation 5
- macOS installation 4
- Main Window 56
- Menu Bar 56
- Navigation Bar 56
- Project tool window 57
- setup wizard 5
- Status Bar 57
- Toolbar 56
- Tool window bars 58
- tool windows 57
- updating 12
- Welcome Screen 55
- Windows installation 4
- Android Support Library, 369
- Android Virtual Device. *See* AVD
 - overview 35
- Android Virtual Device Manager 35
- AndroidX libraries 532
- animate as state functions 333
- animateColorAsState() function 333, 337, 338
- animateDpAsState() function 339, 343
- AnimatedVisibility 321
 - animation specs 325
 - enter and exit animations 324
 - expandHorizontally() 324
 - expandIn() 324
 - expandVertically() 324
 - fadeIn() 324
 - fadeOut() 325
 - MutableTransitionState 329
 - scaleIn() 325
 - scaleOut() 325
 - shrinkHorizontally() 325
 - shrinkOut() 325
 - shrinkVertically() 325
 - slideIn() 325
 - slideInHorizontally() 325
 - slideInVertically() 325
 - slideOut() 325
 - slideOutHorizontally() 325
 - slideOutVertically() 325
- animateEnterExit() modifier 328
- animateFloatAsState() function 334
- animateScrollTo() function 282, 293
- animateScrollToItem(index: Int) 282
- animateScrollTo(value: Int) 281
- Animation
 - auto-starting 328
 - combining animations 344
 - inspector 347
 - keyframes 343
 - KeyframesSpec 343
 - motion 339
 - spring effects 342
 - state-based 333
 - visibility 321
- Animation damping
 - DampingRatioHighBouncy 342
 - DampingRatioLowBouncy 342
 - DampingRatioMediumBouncy 342
 - DampingRatioNoBouncy 342
- Animation Inspector 347
- AnimationSpec 325
 - tween() function 326
- Animation specs 325
- Animation stiffness
 - StiffnessHigh 343
 - StiffnessLow 343
 - StiffnessMedium 343
 - StiffnessMediumLow 343
 - StiffnessVeryLow 343
- annotated strings 195, 365
 - append function 195
 - buildAnnotatedString function 195
 - ParagraphStyle 196
 - SpanStyle 195
- APK analyzer 496
- APK file 490

- APK File
 - analyzing 496
- APK Signing 532
- APK Wizard dialog 488
- App Bundles 485
 - creating 490
 - overview 485
 - revisions 495
 - uploading 492
- append function 195
- App Inspector 59
- Application
 - stopping 40
- Application Framework 87
- Arrangement.Bottom 209
- Arrangement.Center 208, 209
- Arrangement.End 208
- Arrangement.SpaceAround 210
- Arrangement.SpaceBetween 210
- Arrangement.SpaceEvenly 210
- Arrangement.Start 208
- Arrangement.Top 209
- ART 86
- as 101
- as? 101
- asFlow() builder 463
- Asset Studio 178
- asSharedFlow() 474
- asStateFlow() 473
- async 273
- AVD
 - cold boot 50
 - command-line creation 35
 - creation 35
 - device frame 43
 - Display mode 52
 - launch in tool window 43
 - overview 35
 - quickboot 50
 - Resizable 52
 - running an application 37
 - Snapshots 49
 - standalone 41
 - starting 36
 - Startup size and orientation 37
- B**
- background modifier 192
- barriers 259
- Barriers 244
 - constrained views 244
- baseline
 - alignment 211
- baselines 231
- BaseTextField 146
- BillingClient 504
 - acknowledgePurchase() method 503
 - consumeAsync() method 503
 - getPurchaseState() method 502
 - initialization 500, 509
 - launchBillingFlow() method 502
 - queryProductDetailsAsync() method 501
 - queryPurchasesAsync() method 504
 - startConnection() method 501
- BillingResult 517
 - getDebugMessage() 517
- Bill of Materials. *See* BOM
- Bitwise AND 107
- Bitwise Inversion 106
- Bitwise Left Shift 108
- Bitwise OR 107
- Bitwise Right Shift 108
- Bitwise XOR 107
- BOM 22
 - build.gradle 22
 - compose-bom 23
 - library version mapping 23
 - override library version 24
- Boolean 94
- BottomNavigation 147, 425
- BottomNavigationItem 425
- Box 146
 - align() 221
 - alignment 219

Index

- Alignment.BottomCenter 219
- Alignment.BottomEnd 219
- Alignment.BottomStart 219
- Alignment.Center 219
- Alignment.CenterEnd 219
- Alignment.CenterStart 219
- Alignment.TopCenter 219
- Alignment.TopEnd 219
- Alignment.TopStart 219
- BoxScope 221
- contentAlignment 219
- matchParentSize() 221
- overview 217
- tutorial 217
- BoxScope
 - align() 221
 - matchParentSize() 221
 - modifiers 221
- BoxWithConstraints 146
- Brush Text Styling 196
- buffer() operator 468
- buildAnnotatedString function 195
- Build tool window 59
- Build Variants 59, 532
 - tool window 59
- Button 147
- by keyword 152
- C**
- cancelAndJoin() 274
- cancelChildren() 274
- Canvas 146
 - DrawScope 349
 - inset() function 353
 - overview 349
 - size 349
- Card 147
 - example 286
- C/C++ Libraries 86
- centerAround() function 248
- chain head 242
- chaining modifiers 187
- chains 242
- chain styles 242
- Char 94
- Checkbox 147, 176
- CircleShape 221
- CircularProgressIndicator 147
- clickable 192
- clip 192
- Clip Art 179
- clip() modifier 221
 - CircleShape 221
 - CutCornerShape 221
 - RectangleShape 221
 - RoundedCornerShape 221
- close() function 361
- Code completion 76
- Code editor 17
 - Split mode 17
- Code Editor
 - basics 73
 - Code completion 76
 - Code folding 79
 - Code Generation 78
 - Code mode 75
 - Code Reformatting 81
 - Document Tabs 74
 - Editing area 74
 - Gutter Area 74
 - Live Templates 82
 - Parameter information 78
 - Parameter name hints 78
 - sample code 82
 - Splitting 76
 - Statement Completion 78
 - Status Bar 75
- Code folding 79
- Code Generation 78
- Code mode 75
- Code reformatting 81
- code samples
 - download 1
- Coil

- library 298
- rememberImagePainter() function 299
- cold boot 50
- Cold flow 473
 - convert to hot 476
- collectLatest() operator 468
- collect() operator 464
- ColorFilter 365
- color filtering 365
- Column 146
 - Alignment.CenterHorizontally 207
 - Alignment.End 207
 - Alignment.Start 207
 - Arrangement.Bottom 209
 - Arrangement.Center 209
 - Arrangement.SpaceAround 210
 - Arrangement.SpaceBetween 210
 - Arrangement.SpaceEvenly 210
 - Arrangement.Top 209
 - Layout alignment 206
 - list 279
 - list tutorial 289
 - overview 204
 - scope 211
 - scope modifiers 211
 - spacing 210
 - tutorial 203
 - verticalArrangement 208
- Column lists 279
- ColumnScope 211
 - Modifier.align() 211
 - Modifier.alignBy() 211
 - Modifier.weight() 211
- combine() operator 472
- combining modifiers 192
- Communicating Sequential Processes 271
- Companion Objects 131
- components 143
- Composable
 - adding a 26
 - previewing 28
- Composable function
 - syntax 144
- composable functions 143
- composables
 - add modifier support 188
- Composables
 - Foundation 146
 - Material 146
- Compose
 - before 141
 - components 143
 - data-driven 142
 - declarative syntax 141
 - functions 143
 - layout overview 225
 - modifiers 185
 - overview 141
 - state 142
- compose-bom 23
- compose() method 421
- CompositionLocal
 - example 163
 - overview 161
 - state 164, 165
 - syntax 162
- compositionLocalOf() function 162
- conflate() operator 468
- constrainAs() modifier function 247
- constrain() function 263
- Constraint bias 252
- Constraint Bias 241
- ConstraintLayout 146
 - adding constraints 248
 - barriers 259
 - Barriers 244
 - basic constraints 250
 - centerAround() function 248
 - chain head 242
 - chains 242
 - chain styles 242
 - constrainAs() function 247
 - constrain() function 263
 - Constraint bias 252

Index

- Constraint Bias 241
- Constraint margins 253
- Constraints 239
- constraint sets 262
- createEndBarrier() 259
- createHorizontalChain() 257
- createRefFor() function 263
- createRef() function 247
- createRefs() function 247
- createStartBarrier() 259
- createTopBarrier() 259
- createVerticalChain() 257
- creating chains 257
- generating references 247
- guidelines 258
- Guidelines 243
- how to call 247
- layout() modifier 264
- library 249
- linkTo() function 248
- Margins 240
- Opposing constraints 251
- Opposing Constraints 240, 254
- overview of 239
- Packed chain 243
- reference assignment 247
- Spread chain 242
- Spread inside chain 242
- Weighted chain 242
- Widget Dimensions 243
- Constraint margins 253
- constraints 234
- constraint sets 262
- consumeAsync() method 503
- ConsumeParams 512
- contentAlignment 219
- Content Provider 87
- Coroutine Builders 273
 - async 273
 - coroutineScope 273
 - launch 273
 - runBlocking 273
 - supervisorScope 273
 - withContext 273
- Coroutine Dispatchers 272
- Coroutines 282, 461
 - channel communication 275
 - coroutine scope 282
 - CoroutineScope 282
 - GlobalScope 272
 - LaunchedEffect 276
 - rememberCoroutineScope() 282
 - rememberCoroutineScope() function 272
 - SideEffect 276
 - Side Effects 276
 - Suspend Functions 272
 - suspending 274
 - ViewModelScope 272
 - vs Threads 271
 - vs. Threads 271
- coroutineScope 273
- CoroutineScope 272, 282
 - rememberCoroutineScope() 282
- createEndBarrier() 259
- createHorizontalChain() 257
- createRefFor() function 263
- createRef() function 247
- createRefs() 247
- createStartBarrier() 259
- createTopBarrier() 259
- createVerticalChain() 257
- Crossfading 329
- currentBackStackEntryAsState() method 426, 444
- Custom Accessors 129
- Custom layout 233
 - building 233
 - constraints 234
 - Layout() composable 234
 - measurables 234
 - overview 233
 - Placeable 234
 - syntax 233
- custom layout modifiers 225
 - alignment lines 229

- baselines 231
 - creating 227
 - default position 227
 - Custom layouts
 - overview 225
 - tutorial 225
 - Custom Theme
 - building 523
 - CutCornerShape 221
- D**
- DampingRatioHighBouncy 342
 - DampingRatioLowBouncy 342
 - DampingRatioMediumBouncy 342
 - DampingRatioNoBouncy 342
 - Dark Theme 41
 - enable on device 41
 - dashPathEffect() method 351
 - Data Access Object (DAO) 392, 404
 - Data Access Objects 395
 - Database Inspector 399, 417
 - live updates 417
 - SQL query 417
 - Database Rows 386
 - Database Schema 385
 - Database Tables 385
 - data-driven 142
 - DDMS 40
 - Debugging
 - enabling on device 65
 - declarative syntax 141
 - Default Function Parameters 121
 - default position 227
 - Device File Explorer 59
 - device frame 43
 - Device Manager 59
 - device pairing 69
 - Dispatchers.Default 273
 - Dispatchers.IO 273
 - Dispatchers.Main 272
 - drag gestures 450
 - drawable
 - folder 178
 - drawArc() function 360
 - drawCircle() function 356
 - drawImage() function 363
 - Drawing
 - arcs 360
 - circle 356
 - close() 361
 - dashed lines 351
 - dashPathEffect() 351
 - drawArc() 360
 - drawImage() 363
 - drawPath() 361
 - drawPoints() 362
 - drawRect() 351
 - drawRoundRect() 354
 - gradients 357
 - images 363
 - line 349
 - oval 356
 - points 362
 - rectangle 351
 - rotate() 355
 - rotation 355
 - Drawing text 365
 - drawLine() function 350
 - drawPath() function 361
 - drawPoints() function 362
 - drawRect() function 351
 - drawRoundRect() function 354
 - DrawScope 349
 - drawText() function 365, 366
 - DropDownMenu 147
 - DROP_LATEST 475
 - DROP_OLDEST 475
 - DurationBasedAnimationSpec 325
 - Dynamic colors
 - enabling in Android 529
- E**
- Elvis Operator 101
 - emit 143

Index

Empty Compose Activity

template 14

Emulator 59

battery 48

cellular configuration 48

configuring fingerprints 50

directional pad 48

extended control options 47

Extended controls 47

fingerprint 48

location configuration 48

phone settings 48

Resizable 52

resize 47

rotate 46

Screen Record 49

Snapshots 49

starting 36

take screenshot 46

toolbar 45

toolbar options 45

tool window mode 51

Virtual Sensors 49

zoom 46

enablePendingPurchases() method 503

enabling ADB support 65

enter animations 324

EnterTransition.None 328

Errata 2

Escape Sequences 95

Event Log 59

exit animations 324

ExitTransition.None 328

expandHorizontally() 324

expandIn() 324

expandVertically() 324

Extended Control

options 47

F

fadeIn() 324

fadeOut() 325

Favorites

tool window 59

Files

switching between 74

fillMaxHeight 192

fillMaxSize 192

fillMaxWidth 192

filter() operator 466

findStartDestination() method 426

Fingerprint

emulation 50

firstVisibleItemIndex 284

flatMapConcat() operator 471

flatMapMerge() operator 471

Float 94

FloatingActionButton 147

Flow 461

asFlow() builder 463

asSharedFlow() 474

asStateFlow() 473

backgroundn handling 481

buffering 468

buffer() operator 468

builder 463

cold 473

collect() 467

collecting data 467

collectLatest() operator 468

combine() operator 472

conflate() operator 468

emit() 463

emitting data 463

filter() operator 466

flatMapConcat() operator 471

flatMapMerge() operator 471

flattening 470

flowOf() builder 463

flow of flows 470

fold() operator 470

hot 473

MutableSharedFlow 474

MutableStateFlow 473

- onEach() operator 472
 - reduce() operator 469, 470
 - repeatOnLifecycle 482
 - SharedFlow 474
 - shareIn() function 476
 - single() operator 468
 - StateFlow 473
 - transform() operator 466
 - try/finally 467
 - zip() operator 472
 - flow builder 463
 - flowOf() builder 463
 - flow of flows 470
 - Flows
 - combining 472
 - Introduction to 461
 - FontWeight 27
 - forEach loop 236
 - Foundation components 146
 - Foundation Composables 146
 - Function Parameters
 - variable number of 121
 - Functions 119
- G**
- Gestures 447
 - click 447
 - drag 450
 - horizontalScroll() 454
 - overview 447
 - pinch gestures 456
 - PointerInputScope 449
 - rememberScrollableState() function 453
 - rememberScrollState() 454
 - rememberTransformableState() 456
 - rotation gestures 457
 - scrollable() modifier 453
 - scroll modifiers 454
 - taps 449
 - translation gestures 458
 - tutorial 447
 - verticalScroll() 454
 - getDebugMessage() 517
 - getPurchaseState() method 502
 - getStringArray() method 297
 - GlobalScope 272
 - GNU/Linux 86
 - Google Play Billing Library 499
 - Google Play Console 506
 - Creating an in-app product 506
 - License Testers 507
 - Google Play Developer Console 486
 - Google Play store 15
 - Gradient drawing 357
 - Gradle
 - APK signing settings 537
 - Build Variants 532
 - command line tasks 538
 - dependencies 531
 - Manifest Entries 532
 - overview 531
 - tool window 59
 - Gradle Build File
 - top level 533
 - Gradle Build Files
 - module level 534
 - gradle.properties file 532
 - Graphics
 - drawing 349
 - Grid
 - overview 279
 - groupBy() function 283
 - guidelines 258
- H**
- Higher-order Functions 123
 - horizontalArrangement 208, 210
 - horizontalScroll() 454
 - Hot flows 473
- I**
- Image 146
 - add drawable resource 178
 - painterResource method 180

Index

- Immutable Variables 96
 - INAPP 504
 - In-App Products 499
 - In-App Purchasing 505
 - acknowledgePurchase() method 503
 - BillingClient 500
 - BillingResult 517
 - consumeAsync() method 503
 - ConsumeParams 512
 - Consuming purchases 512
 - enablePendingPurchases() method 503
 - getPurchaseState() method 502
 - Google Play Billing Library 499
 - launchBillingFlow() method 502
 - Libraries 505
 - newBuilder() method 500
 - onBillingServiceDisconnected() callback 510
 - onBillingServiceDisconnected() method 501
 - onBillingSetupFinished() listener 510
 - onProductDetailsResponse() callback 510
 - Overview 499
 - ProductDetail 502
 - ProductDetails 511
 - products 499
 - ProductType 504
 - Purchase Flow 511
 - PurchaseResponseListener 504
 - PurchasesUpdatedListener 502
 - PurchaseUpdatedListener 511
 - purchase updates 511
 - queryProductDetailsAsync() 510
 - queryProductDetailsAsync() method 501
 - queryPurchasesAsync() 512
 - queryPurchasesAsync() method 504
 - startConnection() method 501
 - subscriptions 499
 - tutorial 505
 - Initializer Blocks 129
 - In-Memory Database 398
 - Inner Classes 130
 - inset() function 353
 - IntrinsicSize.Max 269
 - IntrinsicSize.Min 269, 270
 - intelligent recomposition 149
 - IntelliJ IDEA 89
 - Interactive mode 32
 - Intrinsic measurements 265
 - IntrinsicSize 265
 - intrinsic measurements 265
 - Max 265
 - Min 265
 - tutorial 267
 - is 101
 - isInitialized property 101
 - isSystemInDarkTheme() function 164
 - item() function 280
 - items() function 280
 - itemsIndexed() function 280
- ## J
- Java
 - convert to Kotlin 89
 - Java Native Interface 87
 - JetBrains 89
 - Jetpack Compose
 - see Compose 141
 - join() 274
- ## K
- keyboardOptions 381
 - Keyboard Shortcuts 60
 - keyframe 326
 - keyframes 343
 - KeyframesSpec 343
 - keyframes() function 343
 - KeyframesSpec 343
 - Keystore File
 - creation 488
 - Kotlin
 - accessing class properties 129
 - and Java 89
 - arithmetic operators 103
 - assignment operator 103
 - augmented assignment operators 104

- bitwise operators 106
- Boolean 94
- break 114
- breaking from loops 113
- calling class methods 129
- Char 94
- class declaration 125
- class initialization 126
- class properties 126
- Companion Objects 131
- conditional control flow 115
- continue labels 114
- continue statement 114
- control flow 111
- convert from Java 89
- Custom Accessors 129
- data types 93
- decrement operator 104
- Default Function Parameters 121
- defining class methods 126
- do ... while loop 113
- Elvis Operator 101
- equality operators 105
- Escape Sequences 95
- expression syntax 103
- Float 94
- Flow 461
- for-in statement 111
- function calling 120
- Functions 119
- groupBy() function 283
- Higher-order Functions 123
- if ... else ... expressions 116
- if expressions 115
- Immutable Variables 96
- increment operator 104
- inheritance 135
- Initializer Blocks 129
- Inner Classes 130
- introduction 89
- Lambda Expressions 122
- let Function 99

- Local Functions 120
- logical operators 105
- looping 111
- Mutable Variables 96
- Not-Null Assertion 99
- Nullable Type 98
- Overriding inherited methods 138
- playground 90
- Primary Constructor 126
- properties 129
- range operator 106
- Safe Call Operator 98
- Secondary Constructors 126
- Single Expression Functions 120
- String 94
- subclassing 135
- substringBefore() method 299
- Type Annotations 97
- Type Casting 101
- Type Checking 101
- Type Inference 97
- variable parameters 121
- when statement 116
- while loop 112

L

- Lambda Expressions 122
- lateinit 100
- Late Initialization 100
- launch 273
- launchBillingFlow() method 502
- LaunchedEffect 276
- launchSingleTop 423
- Layout alignment 206
- Layout arrangement 208
- Layout arrangement spacing 210
- Layout components 146
- Layout() composable 234
- Layout Editor 141
- Layout Inspector 60
- layout modifier 192
- layout() modifier 264

Index

- LazyColumn 146, 279
 - creation 280
 - scroll position detection 284
- LazyHorizontalStaggeredGrid 313, 318
 - syntax 314
- LazyList
 - tutorial 295
- Lazy lists 279
 - Scrolling 281
- LazyListScope 280
 - item() function 280
 - items() function 280
 - itemsIndexed() function 280
 - stickyHeader() function 282
- LazyListState 284
 - firstVisibleItemIndex 284
- LazyRow 146, 279
 - creation 280
 - scroll position detection 284
- LazyVerticalGrid 279
 - adaptive mode 284
 - fixed mode 284
- LazyVerticalStaggeredGrid 313, 316
 - syntax 313
- let Function 99
- libc 86
- License Testers 507
- Lifecycle.State.CREATED 482
- Lifecycle.State.DESTROYED 482
- Lifecycle.State.INITIALIZED 482
- Lifecycle.State.RESUMED 482
- Lifecycle.State.STARTED 482
- LinearProgressIndicator 147
- lineTo() 361
- lineTo() function 361
- linkTo() function 248
- Linux Kernel 86
- list devices 65
- Lists
 - clickable items 302
 - enabling scrolling 281
 - overview 279

- literals
 - live editing 28
- LiveData 372
 - observeAsState() 373
- Live Edit 39
 - disabling 28
 - enabling 28
 - of literals 28
- Live Templates 82
- Local Functions 120
- Location Manager 87
- Logcat
 - tool window 60

M

- MainActivity.kt file 17
 - template code 25
- map method 234
- matchParentSize() 221
- Material Composables 146
- Material Design 2 519
- Material Design 2 Theming 519
- Material Design 3 519
- Material Design components 147
- Material Theme Builder 523
- Material You 519
- maxValue property 293
- measurables 234
- measure() function 367
- measureTimeMillis() function 468
- Minimum SDK
 - setting 15
- ModalDrawer 147
- Modern Android architecture 369
- modifier
 - adding to composable 188
 - chaining 187
 - combining 192
 - creating a 186
 - ordering 188
 - tutorial 185
- Modifier.align() 211

Modifier.alignBy() 211

modifiers

 build-in 192

 overview 185

Modifier.weight() 211

multiple devices

 testing app on 40

MutableLiveData 372

MutableSharedFlow 474

MutableState 150

MutableStateFlow 473

mutableStateOf function 143

mutableStateOf() function 151

MutableTransitionState 329

Mutable Variables 96

N

NavHost 421, 433, 443

NavHostController 419, 433, 443

navigate() method 423

Navigation 419

 BottomNavigation 425

 BottomNavigationItem 425

 compose() method 421

 currentBackStackEntryAsState() method 426

 declaring routes 429

 findStartDestination() method 426

 graph 421

 launchSingleTop 423

 library 439

 NavHost 421, 433

 NavHostController 419, 433

 navigate() method 423

 navigation graph 419

 NavType 424

 overview 419

 passing arguments

 popUpTo() method 423

 route 421

 stack 419, 420

 start destination 421

 tutorial 429

Navigation Architecture Component 419

NavigationBar 444

NavigationBarItem 444

Navigation bars 425

 navigation graph 419, 421

 Navigation Host 421

 navigation library 439

 NavType 424

 newBuilder() method 500

 Notifications Manager 87

 Not-Null Assertion 99

 Nullable Type 98

O

observeAsState() 373

Offset() function 350

offset modifier 192

onBillingServiceDisconnected() callback 510

onBillingServiceDisconnected() method 501

onBillingSetupFinished() listener 510

onCreate() method 21

onEach() operator 472

onProductDetailsResponse() callback 510

OpenJDK 3

Opposing constraints 251

OutlinedButton 309

OutlinedTextField 375

P

Package Manager 87

Package name 15

Packed chain 243

padding 192

painterResource method 180

ParagraphStyle 196

Parameter name hints 78

PathEffect 351

pinch gestures 456

Placeable 234

PointerInputScope 449

 drag gestures 452

 tap gestures 449

Index

popUpTo() method 423

Preview panel 22

 build and refresh 22

 Interactive mode 32

 settings 31

Primary Constructor 126

Problems

 tool window 60

ProductDetail 502

ProductDetails 511

ProductType 504

Profiler

 tool window 60

proguard-rules.pro file 536

ProGuard Support 532

project

 create new 14

 package name 15

Project

 tool window 60

Project tool window 16, 60

 Android mode 16

PurchaseResponseListener 504

PurchasesUpdatedListener 502, 511

Q

queryProductDetailsAsync() 510

queryProductDetailsAsync() method 501

queryPurchaseHistoryAsync() method 504

queryPurchasesAsync() 512

queryPurchasesAsync() method 504

quickboot snapshot 50

Quick Documentation 81

R

RadioButton 147

Random.nextInt() method 316

Range Operator 106

Recent Files Navigation 61

recomposition 142

 intelligent recomposition 149

 overview 149

RectangleShape 221

reduce() operator 469, 470

relativeLineTo() function 361

release mode 485

Release Preparation 485

rememberCoroutineScope() function 272, 282, 291

rememberDraggableState() function 450

rememberImagePainter() function 299

remember keyword 151

rememberSaveable keyword 158

rememberScrollableState() function 453

rememberScrollState() 454

rememberScrollState() function 281, 291

rememberTextMeasurer() function 365

rememberTransformableState() 456

rememberTransformationState() function 456

repeatable() function 327

RepeatableSpec

 repeatable() 327

RepeatMode.Reverse 327

repeatOnLifecycle 482

Repository

 tutorial 401

Resizable Emulator 52

Resource Manager 60, 87

Room

 Data Access Object (DAO) 392

 entities 392, 393

 In-Memory Database 398

 Repository 391

Room Database 392

 tutorial 401

Room Database Persistence 391

Room persistence library 402

Room Persistence Library 389

rotate modifier 192

rotation gestures 457

RoundedCornerShape 221

Row 146

 Alignment.Bottom 207

 Alignment.CenterVertically 207

 Alignment.Top 207

- Arrangement.Center 208
- Arrangement.End 208
- Arrangement.SpaceAround 210
- Arrangement.SpaceBetween 210
- Arrangement.SpaceEvenly 210
- Arrangement.Start 208
- horizontalArrangement 208
- Layout alignment 206
- Layout arrangement 208
- list 279
- list example 294
- overview 204
- scope 211
- scope modifiers 211
- spacing 210
- tutorial 203
- Row lists 279
- RowScope 211
 - Modifier.align() 211
 - Modifier.alignBy() 211
 - Modifier.alignByBaseline() 211
 - Modifier.paddingFrom() 212
 - Modifier.weight() 212
- Run
 - tool window 60
- runBlocking 273
- S**
- Safe Call Operator 98
- Scaffold 147, 445
 - bottomBar 445
 - TopAppBar 446
- scaleIn() 325
- scale modifier 192
- scaleOut() 325
- Scope modifiers
 - weights 215
- scrollable modifier 192
- scrollable() modifier 453, 454
- Scroll detection
 - example 305
- scroll modifiers 454
- ScrollState
 - maxValue property 293
 - rememberScrollState() function 281
- scrollToItem(index: Int) 282
- scrollTo(value: Int) 281
- SDK Packages 6
- SDK settings 15
- Secondary Constructors 126
- Secure Sockets Layer (SSL) 86
- settings.gradle file 532
- Shape 147
- Shapes
 - CircleShape 221
 - CutCornerShape 221
 - RectangleShape 221
 - RoundedCornerShape 221
- SharedFlow 474, 477
 - backgroundn handling 481
 - DROP_LATEST 475
 - DROP_OLDEST 475
 - in ViewModel() 478
 - repeatOnLifecycle 482
 - SUSPEND 475
 - tutorial 477
- shareIn() function 476
- SharingStarted.Eagerly() 476
- SharingStarted.Lazily() 476
- SharingStarted.WhileSubscribed() 476
- showSystemUi 21, 290
- shrinkHorizontally() 325
- shrinkOut() 325
- shrinkVertically() 325
- SideEffect 276
- Side Effects 276
- single() operator 468
- size modifier 192
- slideIn() 325
- slideInHorizontally() 325
- slideInVertically() 325
- slideOut() 325
- slideOutHorizontally() 325
- slideOutVertically() 325

Index

- Slider 147
 - Slider component 29
 - Slot APIs
 - calling 170
 - declaring 170
 - overview 169
 - tutorial 173
 - Snackbar 147
 - Snapshots
 - emulator 49
 - SpanStyle 195
 - Spread chain 242
 - Spread inside chain 242
 - Spring effects 342
 - spring() function 342
 - SQL 386
 - SQLite 385
 - AVD command-line use 387
 - Columns and Data Types 385
 - overview 386
 - Primary keys 386
 - Staggered Grids 313
 - startConnection() method 501
 - start destination 421
 - state 142
 - basics of 149
 - by keyword 152
 - configuration changes 157
 - declaring 150
 - hoisting 155
 - MutableState 150
 - mutableStateOf() function 151
 - overview 149
 - remember keyword 151
 - rememberSaveable 158
 - Unidirectional data flow 153
 - StateFlow 473
 - stateful 149
 - stateful composables 143
 - State hoisting 155
 - stateless composables 143
 - Statement Completion 78
 - staticCompositionLocalOf() function 162, 164
 - stickyHeader 306
 - stickyHeader() function 282
 - Sticky headers
 - adding 306
 - example 305
 - stickyHeader() function 282
 - StiffnessHigh 343
 - StiffnessLow 343
 - StiffnessMedium 343
 - StiffnessMediumLow 343
 - StiffnessVeryLow 343
 - String 94
 - Structure
 - tool window 60
 - Structured Query Language 386
 - Structure tool window 60
 - SUBS 504
 - subscriptions 499
 - substringBefore() method 299
 - supervisorScope 273
 - Surface component 20, 219
 - SUSPEND 475
 - Suspend Functions 272
 - Switch 147
 - Switcher 61
 - system requirements 3
- ## T
- Telephony Manager 87
 - Terminal
 - tool window 60
 - Text 147
 - Text component 144
 - TextField 147
 - TextMeasurer 365
 - measure() function 367
 - TextStyle 382
 - Theme
 - building a custom 523
 - Theming 519
 - tutorial 525

- TODO
 - tool window 60
 - Tool window bars 58
 - Tool windows 57
 - TopAppBar 147, 446
 - trailingIcon 382
 - TransformableState 456
 - transform() operator 466
 - translation gestures 458
 - try/finally 467
 - tween() function 326
 - Type Annotations 97
 - Type Casting 101
 - Type Checking 101
 - Type Inference 97
 - Type.kt file 522
- U**
- UI Controllers 370
 - UI_NIGHT_MODE_YES 165
 - Unidirectional data flow 153
 - updateTransition() function 334, 339, 344
 - USB connection issues
 - resolving 68
- V**
- Vector Asset
 - add to project 178
 - verticalArrangement 208, 210
 - verticalScroll() 454
 - verticalScroll() modifier 291
 - ViewModel
 - example 376
 - lifecycle library 372, 376, 462, 477
 - LiveData 372
 - observeAsState() 373
 - overview 369
 - tutorial 375
 - using state 370
 - viewModel() 372, 378, 412
 - ViewModelProvider Factory 412
 - ViewModelStoreOwner 412
 - viewModel() function 372, 378, 412
 - ViewModelProvider Factory 412
 - ViewModelScope 272
 - ViewModelStoreOwner 412
 - View System 87
 - Virtual Device Configuration dialog 36
 - Virtual Sensors 49
 - Visibility animation 321
- W**
- Weighted chain 242
 - Welcome screen 55
 - while Loop 112
 - Widget Dimensions 243
 - WiFi debugging 69
 - Wireless debugging 69
 - Wireless pairing 69
 - withContext 273
- X**
- XML resource
 - reading an 295
- Z**
- zip() operator 472

