

Jetpack Compose 1.7 Essentials



Jetpack Compose 1.7 Essentials

Jetpack Compose 1.7 Essentials

ISBN-13: 978-1-965764-03-9

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

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

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

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

Rev: 1.0



<https://www.payloadbooks.com>

Contents

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.....	2
1.4 Feedback.....	2
1.5 Errata.....	2
1.6 Take the knowledge tests	2
2. Setting up an Android Studio Development Environment.....	3
2.1 System requirements.....	3
2.2 Downloading the Android Studio package	3
2.3 Installing Android Studio.....	4
2.3.1 Installation on Windows	4
2.3.2 Installation on macOS	4
2.3.3 Installation on Linux.....	5
2.4 The Android Studio setup wizard	5
2.5 Installing additional Android SDK packages	6
2.6 Installing the Android SDK Command-line Tools.....	9
2.6.1 Windows 8.1	10
2.6.2 Windows 10	11
2.6.3 Windows 11	11
2.6.4 Linux	11
2.6.5 macOS.....	11
2.7 Android Studio memory management	11
2.8 Updating Android Studio and the SDK	12
2.9 Summary	13
3. A Compose Project Overview.....	15
3.1 About the project.....	15
3.2 Creating the project	16
3.3 Creating an activity	16
3.4 Defining the project and SDK settings.....	17
3.5 Previewing the example project	18
3.6 Reviewing the main activity.....	21
3.7 Preview updates.....	24
3.8 Bill of Materials and the Compose version.....	25
3.9 Take the knowledge test.....	26
3.10 Summary	26
4. An Example Compose Project	27
4.1 Getting started	27
4.2 Removing the template Code	27
4.3 The Composable hierarchy	28
4.4 Adding the DemoText composable	28

Table of Contents

4.5 Previewing the DemoText composable	30
4.6 Adding the DemoSlider composable	30
4.7 Adding the DemoScreen composable	31
4.8 Previewing the DemoScreen composable	33
4.9 Adjusting preview settings	33
4.10 Testing in interactive mode	34
4.11 Completing the project	35
4.12 Summary	36
5. Creating an Android Virtual Device (AVD) in Android Studio	37
5.1 About Android Virtual Devices	37
5.2 Starting the Emulator	39
5.3 Running the Application in the AVD	40
5.4 Real-time updates with Live Edit	41
5.5 Running on Multiple Devices	43
5.6 Stopping a Running Application	44
5.7 Supporting Dark Theme	44
5.8 Running the Emulator in a Separate Window	45
5.9 Removing the Device Frame	48
5.10 Take the knowledge test	49
5.11 Summary	49
6. Using and Configuring the Android Studio AVD Emulator	51
6.1 The Emulator Environment	51
6.2 Emulator Toolbar Options	51
6.3 Working in Zoom Mode	53
6.4 Resizing the Emulator Window	53
6.5 Extended Control Options	53
6.5.1 Location	54
6.5.2 Displays	54
6.5.3 Cellular	54
6.5.4 Battery	54
6.5.5 Camera	54
6.5.6 Phone	54
6.5.7 Directional Pad	54
6.5.8 Microphone	54
6.5.9 Fingerprint	54
6.5.10 Virtual Sensors	55
6.5.11 Snapshots	55
6.5.12 Record and Playback	55
6.5.13 Google Play	55
6.5.14 Settings	55
6.5.15 Help	55
6.6 Working with Snapshots	55
6.7 Configuring Fingerprint Emulation	56
6.8 The Emulator in Tool Window Mode	58
6.9 Common Android Settings	58
6.10 Creating a Resizable Emulator	59
6.11 Take the knowledge test	61
6.12 Summary	61

7. A Tour of the Android Studio User Interface	63
7.1 The Welcome Screen	63
7.2 The Menu Bar	64
7.3 The Main Window	64
7.4 The Tool Windows	66
7.5 The Tool Window Menus.....	69
7.6 Android Studio Keyboard Shortcuts	70
7.7 Switcher and Recent Files Navigation	70
7.8 Changing the Android Studio Theme	71
7.9 Take the knowledge test.....	72
7.10 Summary	72
8. Testing Android Studio Apps on a Physical Android Device.....	73
8.1 An Overview of the Android Debug Bridge (ADB).....	73
8.2 Enabling USB Debugging ADB on Android Devices.....	73
8.2.1 macOS ADB Configuration	74
8.2.2 Windows ADB Configuration	75
8.2.3 Linux adb Configuration	76
8.3 Resolving USB Connection Issues	76
8.4 Enabling Wireless Debugging on Android Devices	77
8.5 Testing the adb Connection	79
8.6 Device Mirroring.....	79
8.7 Take the knowledge test.....	79
8.8 Summary	79
9. The Basics of the Android Studio Code Editor.....	81
9.1 The Android Studio Editor.....	81
9.2 Splitting the Editor Window.....	84
9.3 Code Completion.....	84
9.4 Statement Completion.....	86
9.5 Parameter Information	86
9.6 Parameter Name Hints	86
9.7 Code Generation	86
9.8 Code Folding.....	88
9.9 Quick Documentation Lookup	89
9.10 Code Reformatting.....	89
9.11 Finding Sample Code	90
9.12 Live Templates	90
9.13 Take the knowledge test	91
9.14 Summary	91
10. An Overview of the Android Architecture.....	93
10.1 The Android Software Stack	93
10.2 The Linux Kernel.....	94
10.3 Hardware Abstraction Layer.....	94
10.4 Android Runtime – ART.....	94
10.4.1 Dalvik and DEX	95
10.4.2 The ART and AOT	95
10.4.3 ART and the Linux kernel.....	95
10.5 Android Libraries.....	96

Table of Contents

10.6 C/C++ Libraries	97
10.7 Native Development Kit.....	98
10.8 Application Framework.....	98
10.9 Applications	99
10.10 Take the knowledge test	99
10.11 Summary	99
11. An Introduction to Kotlin.....	101
11.1 What is Kotlin?	101
11.2 Kotlin and Java.....	101
11.3 Converting from Java to Kotlin	101
11.4 Kotlin and Android Studio	102
11.5 Experimenting with Kotlin	102
11.6 Semi-colons in Kotlin	103
11.7 Summary	103
12. Kotlin Data Types, Variables and Nullability	105
12.1 Kotlin data types.....	105
12.1.1 Integer data types	106
12.1.2 Floating point data types.....	106
12.1.3 Boolean data type.....	106
12.1.4 Character data type.....	106
12.1.5 String data type.....	106
12.1.6 Escape sequences.....	107
12.2 Mutable variables	108
12.3 Immutable variables.....	108
12.4 Declaring mutable and immutable variables.....	108
12.5 Data types are objects	108
12.6 Type annotations and type inference.....	109
12.7 Nullable type	110
12.8 The safe call operator	110
12.9 Not-null assertion	111
12.10 Nullable types and the let function.....	111
12.11 Late initialization (lateinit)	112
12.12 The Elvis operator	113
12.13 Type casting and type checking.....	113
12.14 Take the knowledge test	114
12.15 Summary	114
13. Kotlin Operators and Expressions	115
13.1 Expression syntax in Kotlin	115
13.2 The Basic assignment operator.....	115
13.3 Kotlin arithmetic operators.....	115
13.4 Augmented assignment operators	116
13.5 Increment and decrement operators	116
13.6 Equality operators	117
13.7 Boolean logical operators.....	117
13.8 Range operator	118
13.9 Bitwise operators	118
13.9.1 Bitwise inversion	118
13.9.2 Bitwise AND	119

13.9.3 Bitwise OR.....	119
13.9.4 Bitwise XOR.....	119
13.9.5 Bitwise left shift	120
13.9.6 Bitwise right shift	120
13.10 Take the knowledge test	121
13.11 Summary	121
14. Kotlin Control Flow	123
14.1 Looping control flow.....	123
14.1.1 The Kotlin for-in Statement	123
14.1.2 The while loop	124
14.1.3 The do ... while loop	125
14.1.4 Breaking from Loops	125
14.1.5 The continue statement	126
14.1.6 Break and continue labels	126
14.2 Conditional control flow	127
14.2.1 Using the if expressions	127
14.2.2 Using if ... else ... expressions	128
14.2.3 Using if ... else if ... Expressions	128
14.2.4 Using the when statement.....	128
14.3 Take the knowledge test	129
14.4 Summary	129
15. An Overview of Kotlin Functions and Lambdas	131
15.1 What is a function?	131
15.2 How to declare a Kotlin function.....	131
15.3 Calling a Kotlin function.....	132
15.4 Single expression functions.....	132
15.5 Local functions	132
15.6 Handling return values.....	133
15.7 Declaring default function parameters	133
15.8 Variable number of function parameters	133
15.9 Lambda expressions.....	134
15.10 Higher-order functions	135
15.11 Take the knowledge test	136
15.12 Summary	136
16. The Basics of Object-Oriented Programming in Kotlin.....	137
16.1 What is an object?	137
16.2 What is a class?	137
16.3 Declaring a Kotlin class	137
16.4 Adding properties to a class	138
16.5 Defining methods.....	138
16.6 Declaring and initializing a class instance	138
16.7 Primary and secondary constructors	138
16.8 Initializer blocks	141
16.9 Calling methods and accessing properties.....	141
16.10 Custom accessors	141
16.11 Nested and inner classes.....	142
16.12 Companion objects.....	143
16.13 Take the knowledge test	145

Table of Contents

16.14 Summary	145
17. An Introduction to Kotlin Inheritance and Subclassing.....	147
17.1 Inheritance, classes, and subclasses	147
17.2 Subclassing syntax.....	147
17.3 A Kotlin inheritance example.....	148
17.4 Extending the functionality of a subclass.....	149
17.5 Overriding inherited methods	150
17.6 Adding a custom secondary constructor	151
17.7 Using the SavingsAccount class.....	151
17.8 Take the knowledge test	152
17.9 Summary	152
18. Introducing Gemini in Android Studio.....	153
18.1 Introducing Gemini AI	153
18.2 Enabling Gemini in Android Studio	153
18.3 Gemini configuration	155
18.4 Asking Gemini questions.....	156
18.5 Question contexts.....	157
18.6 Inline code completion.....	157
18.7 Transforming and documenting code.....	158
18.8 Take the knowledge test	160
18.9 Summary	160
19. An Overview of Compose	161
19.1 Development before Compose.....	161
19.2 Compose declarative syntax	161
19.3 Compose is data-driven	162
19.4 Take the knowledge test	162
19.5 Summary	163
20. A Guide to Gradle Version Catalogs.....	165
20.1 Library and Plugin Dependencies.....	165
20.2 Project Gradle Build File	165
20.3 Module Gradle Build Files	165
20.4 Version Catalog File.....	166
20.5 Adding Dependencies	167
20.6 Library Updates.....	168
20.7 Take the knowledge test	168
20.8 Summary	168
21. Composable Functions Overview	171
21.1 What is a composable function?	171
21.2 Stateful vs. stateless composables.....	171
21.3 Composable function syntax	172
21.4 Composable hierarchy.....	174
21.5 Foundation and Material composables	175
21.6 Take the knowledge test	176
21.7 Summary	176
22. An Overview of Compose State and Recomposition.....	179
22.1 The basics of state.....	179

22.2 Introducing recomposition.....	179
22.3 Creating the StateExample project.....	180
22.4 Declaring state in a composable.....	180
22.5 Unidirectional data flow.....	183
22.6 State hoisting.....	185
22.7 How high to hoist?	187
22.8 Saving state through configuration changes.....	188
22.9 Take the knowledge test	190
22.10 Summary	190
23. An Introduction to Composition Local.....	191
23.1 Understanding CompositionLocal	191
23.2 Using CompositionLocal	192
23.3 Creating the CompLocalDemo project.....	193
23.4 Designing the layout	193
23.5 Adding the CompositionLocal state.....	195
23.6 Accessing the CompositionLocal state.....	195
23.7 Testing the design.....	195
23.8 Take the knowledge test	197
23.9 Summary	198
24. An Overview of Compose Slot APIs	199
24.1 Understanding slot APIs	199
24.2 Declaring a slot API	200
24.3 Calling slot API composables.....	200
24.4 Take the knowledge test	202
24.5 Summary	202
25. A Compose Slot API Tutorial.....	203
25.1 About the project.....	203
25.2 Creating the SlotApiDemo project	203
25.3 Preparing the MainActivity class file.....	203
25.4 Creating the MainScreen composable.....	204
25.5 Adding the ScreenContent composable	205
25.6 Creating the Checkbox composable	206
25.7 Implementing the ScreenContent slot API.....	207
25.8 Adding an Image drawable resource	208
25.9 Coding the TitleImage composable.....	209
25.10 Completing the MainScreen composable.....	210
25.11 Previewing the project.....	212
25.12 Summary	213
26. Using Modifiers in Compose.....	215
26.1 An overview of modifiers.....	215
26.2 Creating the ModifierDemo project	215
26.3 Creating a modifier	216
26.4 Modifier ordering.....	218
26.5 Adding modifier support to a composable	218
26.6 Common built-in modifiers	222
26.7 Combining modifiers.....	223
26.8 Take the knowledge test	223

Table of Contents

26.9 Summary	223
27. Annotated Strings and Brush Styles.....	225
27.1 What are annotated strings?	225
27.2 Using annotated strings.....	225
27.3 Brush Text Styling	226
27.4 Creating the example project.....	227
27.5 An example SpanStyle annotated string.....	227
27.6 An example ParagraphStyle annotated string	229
27.7 A Brush style example	231
27.8 Take the knowledge test	233
27.9 Summary	233
28. Composing Layouts with Row and Column	235
28.1 Creating the RowColDemo project	235
28.2 Row composable.....	236
28.3 Column composable.....	237
28.4 Combining Row and Column composables.....	237
28.5 Layout alignment	238
28.6 Layout arrangement positioning.....	241
28.7 Layout arrangement spacing.....	242
28.8 Row and Column scope modifiers.....	243
28.9 Scope modifier weights	247
28.10 Take the knowledge test	248
28.11 Summary	248
29. Box Layouts in Compose.....	249
29.1 An introduction to the Box composable.....	249
29.2 Creating the BoxLayout project	249
29.3 Adding the TextCell composable	249
29.4 Adding a Box layout.....	250
29.5 Box alignment.....	251
29.6 BoxScope modifiers	253
29.7 Using the clip() modifier	254
29.8 Take the knowledge test	255
29.9 Summary	256
30. An Introduction to FlowRow and FlowColumn.....	257
30.1 FlowColumn and FlowRow	257
30.2 Maximum number of items	258
30.3 Working with main axis arrangement	258
30.4 Understanding cross-axis arrangement	259
30.5 Item alignment	260
30.6 Controlling item size.....	260
30.7 Take the knowledge test	262
30.8 Summary	262
31. A FlowRow and FlowColumn Tutorial	263
31.1 Creating the FlowLayoutDemo project.....	263
31.2 Generating random height and color values	263
31.3 Adding the Box Composable.....	264

31.4 Modifying the Flow arrangement	265
31.5 Modifying item alignment	266
31.6 Switching to FlowColumn	267
31.7 Using cross-axis arrangement.....	268
31.8 Adding item weights.....	269
31.9 Summary	270
32. Custom Layout Modifiers.....	271
32.1 Compose layout basics	271
32.2 Custom layouts	271
32.3 Creating the LayoutModifier project	271
32.4 Adding the ColorBox composable.....	272
32.5 Creating a custom layout modifier	273
32.6 Understanding default position.....	273
32.7 Completing the layout modifier	273
32.8 Using a custom modifier	274
32.9 Working with alignment lines	275
32.10 Working with baselines	277
32.11 Take the knowledge test	278
32.12 Summary	278
33. Building Custom Layouts.....	279
33.1 An overview of custom layouts	279
33.2 Custom layout syntax	279
33.3 Using a custom layout.....	280
33.4 Creating the CustomLayout project	281
33.5 Creating the CascadeLayout composable	281
33.6 Using the CascadeLayout composable	283
33.7 Take the knowledge test	284
33.8 Summary	284
34. A Guide to ConstraintLayout in Compose.....	285
34.1 An introduction to ConstraintLayout	285
34.2 How ConstraintLayout works.....	285
34.2.1 Constraints.....	285
34.2.2 Margins.....	286
34.2.3 Opposing constraints.....	286
34.2.4 Constraint bias.....	287
34.2.5 Chains	288
34.2.6 Chain styles.....	288
34.3 Configuring dimensions.....	289
34.4 Guideline helper	289
34.5 Barrier helper.....	290
34.6 Take the knowledge test	291
34.7 Summary	292
35. Working with ConstraintLayout in Compose	293
35.1 Calling ConstraintLayout.....	293
35.2 Generating references.....	293
35.3 Assigning a reference to a composable.....	293
35.4 Adding constraints.....	294

Table of Contents

35.5 Creating the ConstraintLayout project	294
35.6 Adding the ConstraintLayout library	295
35.7 Adding a custom button composable.....	295
35.8 Basic constraints.....	296
35.9 Opposing constraints.....	297
35.10 Constraint bias.....	299
35.11 Constraint margins	299
35.12 The importance of opposing constraints and bias	301
35.13 Creating chains.....	303
35.14 Working with guidelines	304
35.15 Working with barriers	305
35.16 Decoupling constraints with constraint sets.....	308
35.17 Take the knowledge test	310
35.18 Summary	310
36. Working with IntrinsicSize in Compose.....	313
36.1 Intrinsic measurements.....	313
36.2 Max. vs Min. Intrinsic Size measurements	313
36.3 About the example project	314
36.4 Creating the IntrinsicSizeDemo project.....	315
36.5 Creating the custom text field.....	315
36.6 Adding the Text and Box components	316
36.7 Adding the top-level Column.....	316
36.8 Testing the project.....	317
36.9 Applying IntrinsicSize.Max measurements	317
36.10 Applying IntrinsicSize.Min measurements	318
36.11 Take the knowledge test	319
36.12 Summary	319
37. Coroutines and LaunchedEffects in Jetpack Compose.....	321
37.1 What are coroutines?	321
37.2 Threads vs. coroutines	321
37.3 Coroutine Scope	322
37.4 Suspend functions	322
37.5 Coroutine dispatchers.....	322
37.6 Coroutine builders	323
37.7 Jobs	323
37.8 Coroutines – suspending and resuming	324
37.9 Coroutine channel communication.....	325
37.10 Understanding side effects	326
37.11 Take the knowledge test	327
37.12 Summary	327
38. An Overview of Lists and Grids in Compose	329
38.1 Standard vs. lazy lists	329
38.2 Working with Column and Row lists	329
38.3 Creating lazy lists	330
38.4 Enabling scrolling with ScrollState	331
38.5 Programmatic scrolling.....	331
38.6 Sticky headers	332
38.7 Responding to scroll position	334

38.8 Creating a lazy grid	334
38.9 Take the knowledge test	337
38.10 Summary	337
39. A Compose Row and Column List Tutorial	339
39.1 Creating the ListDemo project.....	339
39.2 Creating a Column-based list.....	339
39.3 Enabling list scrolling	341
39.4 Manual scrolling.....	341
39.5 A Row list example.....	344
39.6 Summary	344
40. A Compose Lazy List Tutorial	345
40.1 Creating the LazyListDemo project.....	345
40.2 Adding list data to the project	345
40.3 Reading the XML data.....	347
40.4 Handling image loading	348
40.5 Designing the list item composable.....	350
40.6 Building the lazy list.....	352
40.7 Testing the project.....	352
40.8 Making list items clickable	353
40.9 Take the knowledge test	354
40.10 Summary	355
41. Lazy List Sticky Headers and Scroll Detection	357
41.1 Grouping the list item data	357
41.2 Displaying the headers and items	357
41.3 Adding sticky headers.....	358
41.4 Reacting to scroll position	359
41.5 Adding the scroll button	361
41.6 Testing the finished app.....	363
41.7 Take the knowledge test	363
41.8 Summary	363
42. A Compose Lazy Staggered Grid Tutorial	365
42.1 Lazy Staggered Grids	365
42.2 Creating the StaggeredGridDemo project	366
42.3 Adding the Box composable.....	367
42.4 Generating random height and color values	367
42.5 Creating the Staggered List	368
42.6 Testing the project.....	369
42.7 Switching to a horizontal staggered grid.....	369
42.8 Take the knowledge test	371
42.9 Summary	371
43. VerticalPager and HorizontalPager in Compose	373
43.1 The Pager composables.....	373
43.2 Working with pager state	375
43.3 About the PagerDemo project.....	375
43.4 Creating the PagerDemo project.....	375
43.5 Adding the book cover images	376

Table of Contents

43.6 Adding the HorizontalPager	376
43.7 Creating the page content	378
43.8 Testing the pager	379
43.9 Adding the arrow buttons	379
43.10 Take the knowledge test	382
43.11 Summary	382
44. Compose Visibility Animation	385
44.1 Creating the AnimateVisibility project	385
44.2 Animating visibility	385
44.3 Defining enter and exit animations	388
44.4 Animation specs and animation easing	389
44.5 Repeating an animation	391
44.6 Different animations for different children	391
44.7 Auto-starting an animation	392
44.8 Implementing crossfading	393
44.9 Take the knowledge test	395
44.10 Summary	395
45. Compose State-Driven Animation.....	397
45.1 Understanding state-driven animation	397
45.2 Introducing animate as state functions	397
45.3 Creating the AnimateState project.....	398
45.4 Animating rotation with animateFloatAsState.....	398
45.5 Animating color changes with animateColorAsState.....	401
45.6 Animating motion with animateDpAsState	403
45.7 Adding spring effects	406
45.8 Working with keyframes	407
45.9 Combining multiple animations	408
45.10 Using the Animation Inspector.....	411
45.11 Take the knowledge test	412
45.12 Summary	412
46. Canvas Graphics Drawing in Compose	413
46.1 Introducing the Canvas component	413
46.2 Creating the CanvasDemo project.....	413
46.3 Drawing a line and getting the canvas size	413
46.4 Drawing dashed lines.....	415
46.5 Drawing a rectangle	415
46.6 Applying rotation	419
46.7 Drawing circles and ovals.....	420
46.8 Drawing gradients.....	421
46.9 Drawing arcs	424
46.10 Drawing paths	425
46.11 Drawing points	426
46.12 Drawing an image	427
46.13 Drawing text	429
46.14 Take the knowledge test	431
46.15 Summary	431
47. Working with ViewModels in Compose	433

47.1 What is Android Jetpack?	433
47.2 The “old” architecture	433
47.3 Modern Android architecture	433
47.4 The ViewModel component	433
47.5 ViewModel implementation using state.....	434
47.6 Connecting a ViewModel state to an activity.....	435
47.7 ViewModel implementation using LiveData.....	437
47.8 Observing ViewModel LiveData within an activity	437
47.9 Take the knowledge test	438
47.10 Summary	438
48. A Compose ViewModel Tutorial.....	439
48.1 About the project.....	439
48.2 Creating the ViewModelDemo project	440
48.3 Adding the ViewModel	441
48.4 Accessing DemoViewModel from MainActivity	442
48.5 Designing the temperature input composable	443
48.6 Designing the temperature input composable	444
48.7 Completing the user interface design.....	447
48.8 Testing the app.....	448
48.9 Summary	448
49. An Overview of Android SQLite Databases	449
49.1 Understanding database tables.....	449
49.2 Introducing database schema	449
49.3 Columns and data types	449
49.4 Database rows	450
49.5 Introducing primary keys	450
49.6 What is SQLite?	450
49.7 Structured Query Language (SQL)	450
49.8 Trying SQLite on an Android Virtual Device (AVD)	451
49.9 The Android Room persistence library.....	453
49.10 Take the knowledge test	453
49.11 Summary	453
50. Room Databases and Compose	455
50.1 Revisiting modern app architecture	455
50.2 Key elements of Room database persistence	455
50.2.1 Repository	455
50.2.2 Room database	456
50.2.3 Data Access Object (DAO)	456
50.2.4 Entities.....	456
50.2.5 SQLite database	456
50.3 Understanding entities	457
50.4 Data Access Objects.....	459
50.5 The Room database	460
50.6 The Repository.....	461
50.7 In-Memory databases	462
50.8 Database Inspector.....	463
50.9 Take the knowledge test	463
50.10 Summary	463

Table of Contents

51. A Compose Room Database and Repository Tutorial	465
51.1 About the RoomDemo project.....	465
51.2 Creating the RoomDemo project.....	466
51.3 Modifying the build configuration	466
51.4 Building the entity.....	468
51.5 Creating the Data Access Object.....	470
51.6 Adding the Room database.....	471
51.7 Adding the repository.....	472
51.8 Adding the ViewModel	473
51.9 Designing the user interface	475
51.10 Writing a ViewModelProvider Factory class.....	476
51.11 Completing the MainScreen function.....	479
51.12 Testing the RoomDemo app	482
51.13 Using the Database Inspector	482
51.14 Take the knowledge test	483
51.15 Summary	483
52. An Overview of Navigation in Compose	485
52.1 Understanding navigation.....	485
52.2 Declaring a navigation controller.....	487
52.3 Declaring a navigation host	487
52.4 Adding destinations to the navigation graph	487
52.5 Navigating to destinations.....	488
52.6 Passing arguments to a destination.....	490
52.7 Working with bottom navigation bars	491
52.8 Summary	493
53. A Compose Navigation Tutorial	495
53.1 Creating the NavigationDemo project	495
53.2 About the NavigationDemo project	495
53.3 Declaring the navigation routes	496
53.4 Adding the home screen	496
53.5 Adding the welcome screen	498
53.6 Adding the profile screen	498
53.7 Creating the navigation controller and host.....	499
53.8 Implementing the screen navigation	500
53.9 Passing the user name argument.....	500
53.10 Testing the project.....	501
53.11 Take the knowledge test	503
53.12 Summary	503
54. A Compose Navigation Bar Tutorial.....	505
54.1 Creating the BottomBarDemo project	505
54.2 Declaring the navigation routes	506
54.3 Designing bar items	506
54.4 Creating the bar item list.....	506
54.5 Adding the destination screens	507
54.6 Creating the navigation controller and host.....	509
54.7 Designing the navigation bar.....	510
54.8 Working with the Scaffold component.....	511

54.9 Testing the project.....	512
54.10 Take the knowledge test	513
54.11 Summary	513
55. Detecting Gestures in Compose.....	515
55.1 Compose gesture detection.....	515
55.2 Creating the GestureDemo project.....	515
55.3 Detecting click gestures.....	515
55.4 Detecting taps using PointerInputScope.....	517
55.5 Detecting drag gestures	518
55.6 Detecting drag gestures using PointerInputScope.....	520
55.7 Scrolling using the scrollable modifier.....	521
55.8 Scrolling using the scroll modifiers	522
55.9 Detecting pinch gestures.....	524
55.10 Detecting rotation gestures.....	525
55.11 Detecting translation gestures	526
55.12 Take the knowledge test	527
55.13 Summary	527
56. An Introduction to Kotlin Flow.....	529
56.1 Understanding Flows.....	529
56.2 Creating the sample project.....	529
56.3 Adding a view model to the project.....	531
56.4 Declaring the flow	531
56.5 Emitting flow data.....	532
56.6 Collecting flow data as state.....	532
56.7 Transforming data with intermediaries	534
56.8 Collecting flow data	535
56.9 Adding a flow buffer	537
56.10 More terminal flow operators.....	538
56.11 Flow flattening.....	539
56.12 Combining multiple flows	541
56.13 Hot and cold flows	542
56.14 StateFlow	542
56.15 SharedFlow.....	543
56.16 Converting a flow from cold to hot	545
56.17 Take the knowledge test	546
56.18 Summary	546
57. A Jetpack Compose SharedFlow Tutorial	547
57.1 About the project.....	547
57.2 Creating the SharedFlowDemo project.....	547
57.3 Adding a view model to the project.....	548
57.4 Declaring the SharedFlow	548
57.5 Collecting the flow values	549
57.6 Testing the SharedFlowDemo app	551
57.7 Handling flows in the background.....	551
57.8 Take the knowledge test	553
57.9 Summary	553
58. Introducing Glance Widgets	555

Table of Contents

58.1 Glance Overview	555
58.2 Glance app widget.....	555
58.3 Broadcast receiver	556
58.4 Widget provider info data.....	557
58.5 Size modes.....	558
58.6 Responding to user interaction	558
58.7 Updating a widget	559
58.8 Take the knowledge test	560
58.9 Summary	560
59. A Glance Widget Tutorial.....	561
59.1 About the project.....	561
59.2 Creating the GlanceWidget project	561
59.3 Adding image resources	561
59.4 The price data repository.....	562
59.5 Declaring the repository.....	563
59.6 Adding the Glance app widget	564
59.7 Declaring the widget receiver	565
59.8 Configuring the widget provider info metadata	565
59.9 Adding the widget receiver to the manifest	566
59.10 Testing the widget	567
59.11 Simulating price changes.....	567
59.12 Designing the widget	569
59.13 Adding size mode support.....	571
59.14 Responding to clicks.....	573
59.15 Take the knowledge test	574
59.16 Summary	574
60. An Android Biometric Authentication Tutorial.....	575
60.1 An overview of biometric authentication	575
60.2 Creating the biometric authentication project	575
60.3 Adding the biometric dependency	576
60.4 Configuring device fingerprint authentication	576
60.5 Adding the biometric permissions to the manifest file.....	577
60.6 Checking the security settings.....	577
60.7 Designing the user interface	579
60.8 Configuring the authentication callbacks	580
60.9 Starting the biometric prompt.....	580
60.10 Testing the project.....	581
60.11 Take the knowledge test	582
60.12 Summary	582
61. Working with the Google Maps Android API in Android Studio	583
61.1 The elements of the Google Maps Android API.....	583
61.2 Creating the Google Maps project	584
61.3 Creating a Google Cloud billing account.....	584
61.4 Creating a new Google Cloud project	585
61.5 Enabling the Google Maps SDK.....	586
61.6 Generating a Google Maps API key	587
61.7 Adding the API key to the Android Studio project	587
61.8 Adding the compose map dependency	588

61.9 Creating a map.....	588
61.10 Testing the application.....	588
61.11 Understanding geocoding and reverse geocoding	589
61.12 Specifying a map location	590
61.13 Changing the map type	592
61.14 Displaying map controls to the user.....	593
61.15 Handling map gesture interaction	595
61.15.1 Map zooming gestures.....	595
61.15.2 Map scrolling/panning gestures.....	595
61.15.3 Map tilt gestures	595
61.15.4 Map rotation gestures.....	596
61.16 Creating map markers	596
61.17 Controlling the map camera.....	597
61.18 Take the knowledge test	599
61.19 Summary	599
62. Creating, Testing, and Uploading an Android App Bundle.....	601
62.1 The Release Preparation Process	601
62.2 Android App Bundles.....	601
62.3 Register for a Google Play Developer Console Account.....	602
62.4 Configuring the App in the Console	603
62.5 Enabling Google Play App Signing.....	604
62.6 Creating a Keystore File	604
62.7 Creating the Android App Bundle.....	605
62.8 Generating Test APK Files	607
62.9 Uploading the App Bundle to the Google Play Developer Console.....	608
62.10 Exploring the App Bundle	609
62.11 Managing Testers	610
62.12 Rolling the App Out for Testing.....	610
62.13 Uploading New App Bundle Revisions.....	611
62.14 Analyzing the App Bundle File	612
62.15 Take the knowledge test	613
62.16 Summary	613
63. An Overview of Android In-App Billing	615
63.1 Preparing a project for In-App purchasing.....	615
63.2 Creating In-App products and subscriptions.....	616
63.3 Billing client initialization.....	616
63.4 Connecting to the Google Play Billing library	617
63.5 Querying available products.....	618
63.6 Starting the purchase process	618
63.7 Completing the purchase	619
63.8 Querying previous purchases	620
63.9 Take the knowledge test	621
63.10 Summary	621
64. An Android In-App Purchasing Tutorial	623
64.1 About the In-App purchasing example project.....	623
64.2 Creating the InAppPurchase project	623
64.3 Adding libraries to the project.....	623
64.4 Adding the App to the Google Play Store	624

Table of Contents

64.5 Creating an In-App product	624
64.6 Enabling license testers.....	625
64.7 Creating a purchase helper class	626
64.8 Adding the StateFlow streams	627
64.9 Initializing the billing client.....	627
64.10 Querying the product.....	628
64.11 Handling purchase updates	629
64.12 Launching the purchase flow.....	630
64.13 Consuming the product	630
64.14 Restoring a previous purchase.....	631
64.15 Completing the MainActivity.....	632
64.16 Testing the app.....	634
64.17 Troubleshooting	636
64.18 Take the knowledge test	636
64.19 Summary	636
65. Working with Compose Theming	637
65.1 Material Design 2 vs. Material Design 3	637
65.2 Material Design 3 theming	637
65.3 Building a custom theme	640
65.4 Take the knowledge test	642
65.5 Summary	642
66. A Material Design 3 Theming Tutorial	645
66.1 Creating the ThemeDemo project	645
66.2 Designing the user interface	645
66.3 Building a new theme	647
66.4 Adding the Google font libraries.....	648
66.5 Adding the theme to the project	649
66.6 Testing dynamic colors.....	650
66.7 Take the knowledge test	650
66.8 Summary	651
67. An Overview of Gradle in Android Studio	653
67.1 An Overview of Gradle	653
67.2 Gradle and Android Studio	653
67.2.1 Sensible Defaults	653
67.2.2 Dependencies.....	653
67.2.3 Build Variants	654
67.2.4 Manifest Entries	654
67.2.5 APK Signing.....	654
67.2.6 ProGuard Support.....	654
67.3 The Property and Settings Gradle Build File	654
67.4 The Top-level Gradle Build File.....	655
67.5 Module Level Gradle Build Files.....	656
67.6 Configuring Signing Settings in the Build File.....	659
67.7 Running Gradle Tasks from the Command Line	660
67.8 Take the knowledge test	660
67.9 Summary	660
Index	661

1. Start Here

Welcome to this comprehensive guide to building Android applications using Jetpack Compose 1.7, Android Studio Ladybug, Material Design 3, and the Kotlin programming language. This book has been designed to teach developers the essential knowledge and skills to create modern, dynamic, and visually compelling applications.

We begin with foundational steps, including the setup of your development environment in Android Studio, followed by a detailed introduction to Kotlin, the language underpinning Android development. This section covers core aspects of Kotlin, such as data types, operators, control flow, functions, lambdas, and coroutines, establishing a strong basis in object-oriented programming principles.

With this foundation in place, we turn to Jetpack Compose, Google's innovative toolkit for building native user interfaces. This book presents an in-depth exploration of Compose components and layout structures, including rows, columns, boxes, flows, pagers, and lists, alongside an overview of Android project architecture and Android Studio's Compose development mode.

The following sections delve into more advanced topics, such as state management, modifiers, and navigation components—key elements in designing smooth and intuitive user interfaces. We also guide you through creating reusable layout components, securing applications with biometric authentication, and incorporating Google Maps functionality to enrich user engagement.

Furthermore, we cover specialized techniques, such as graphics rendering, animations, transitions, Kotlin Flows, and gesture handling. Practical data management solutions, including view models, Room database access, live data, and the Database Inspector, are discussed in depth. For developers interested in monetization, this guide also includes a dedicated section on implementing in-app billing.

The concluding chapters provide a comprehensive overview of app packaging and the publication process on the Google Play Store. Throughout the book, each concept is solidified through detailed, hands-on tutorials, complemented by downloadable source code to facilitate real-world application and access to over 55 online knowledge test quizzes.

With a basic understanding of programming, access to Android Studio and the Android SDK, and a Windows, Mac, or Linux system, you will be well-prepared to embark on this journey. It is our aim that, upon completion, you will be equipped with the skills necessary to design, build, and deploy professional-grade Android applications.

1.1 For Kotlin programmers

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

1.2 For new Kotlin programmers

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

1.3 Downloading the code samples

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

<https://www.payloadbooks.com/product/compose17/>

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

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

1.4 Feedback

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

1.5 Errata

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

https://www.payloadbooks.com/compose17_errata

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

1.6 Take the knowledge tests



Look for this section at the end of most chapters and use the link or scan the QR code to take a knowledge quiz to test and reinforce your understanding of the covered topic. Use the following link to review the full list of tests available for this book:

<https://www.answertopia.com/p9jf>



2. Setting up an Android Studio Development Environment

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

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

2.1 System requirements

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

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

2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio Ladybug 2024.2.1 using the Android API 35 SDK (VanillaIceCream), which, at the time of writing, are the latest stable releases.

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

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

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

If this page provides instructions for downloading a newer version of Android Studio, there may be differences between this book and the software. A web search for “Android Studio Ladybug” 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 Ladybug 2024.2.1 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 performed.

2.3.1 Installation on Windows

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

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

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

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

2.3.2 Installation on macOS

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



Figure 2-1

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

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

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

2.3.3 Installation on Linux

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

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

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

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

```
./studio.sh
```

2.4 The Android Studio setup wizard

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

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



Figure 2-2

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

Setting up an Android Studio Development Environment

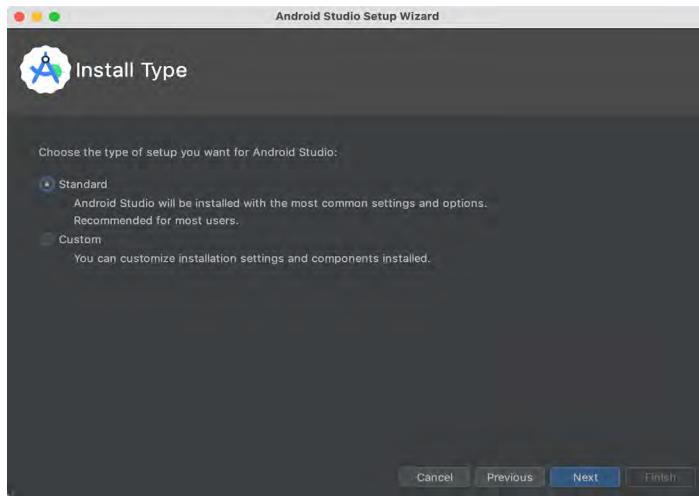


Figure 2-3

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

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

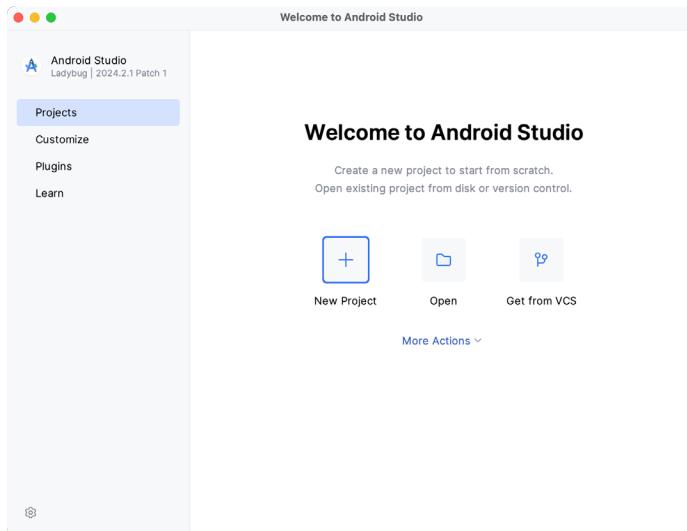


Figure 2-4

2.5 Installing additional Android SDK packages

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

This task can be performed by clicking on the *More Actions* link within the welcome dialog and selecting the

SDK Manager option from the drop-down menu. Once invoked, the *Android SDK* screen of the Settings dialog will appear as shown in Figure 2-5:

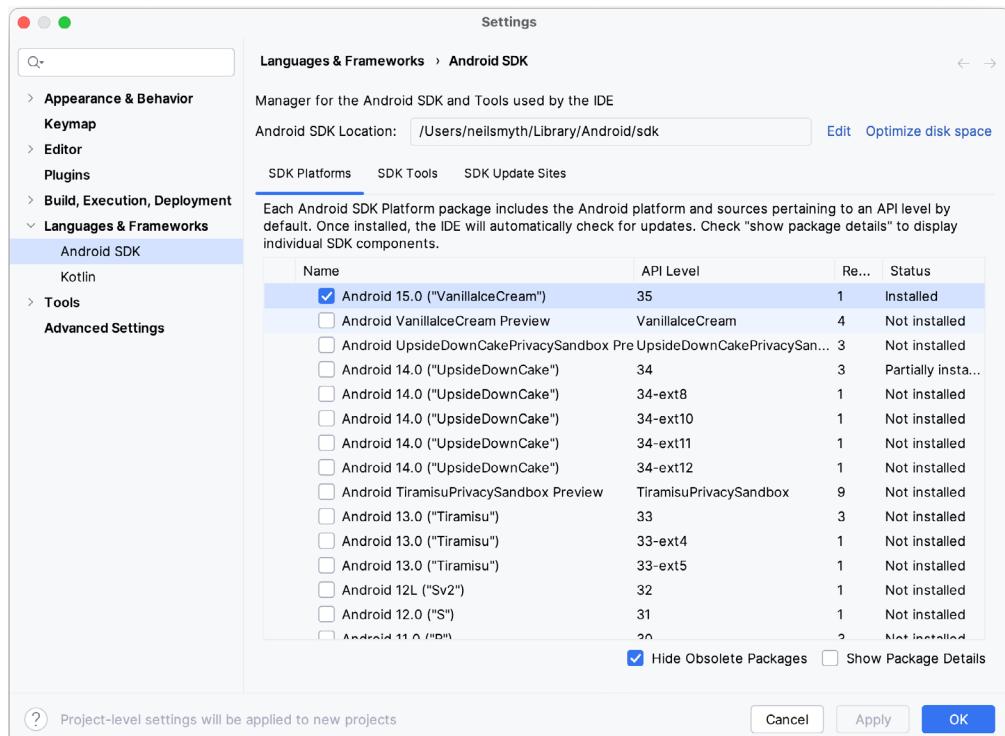


Figure 2-5

Google pairs each release of Android Studio with a maximum supported Application Programming Interface (API) level of the Android SDK. In the case of Android Studio Ladybug, this is Android VanillaIceCream (API Level 35). This information can be confirmed using the following link:

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

Immediately after installing Android Studio for the first time, it is likely that only the latest supported version of the Android SDK has been installed. To install older versions of the Android SDK, select the checkboxes corresponding to the versions and click the *Apply* button. The rest of this book assumes that the Android VanillaIceCream (API Level 35) SDK is installed.

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

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

Setting up an Android Studio Development Environment

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

Figure 2-6

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

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

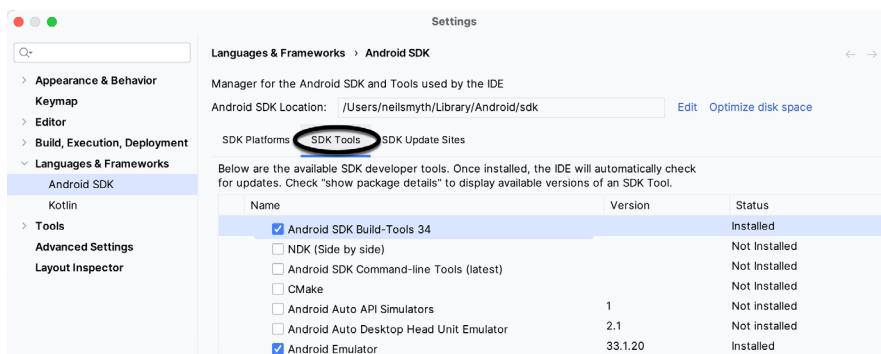


Figure 2-7

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

- Android SDK Build-tools
- Android Emulator
- Android SDK Platform-tools
- Google Play Services
- Intel x86 Emulator Accelerator (HAXM installer)*
- Google USB Driver (Windows only)
- Layout Inspector image server for API 31-35

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

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

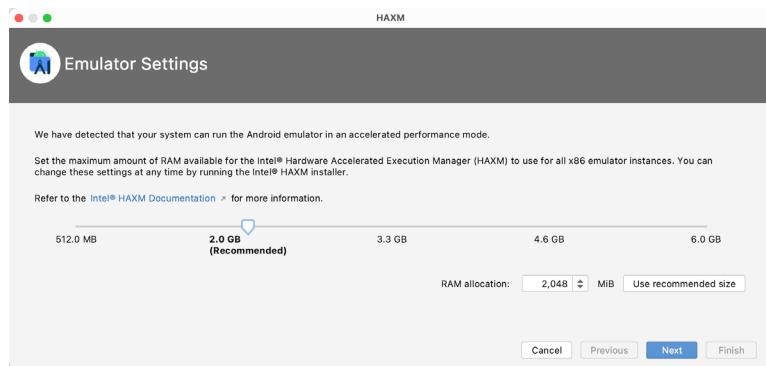


Figure 2-8

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

2.6 Installing the Android SDK Command-line Tools

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

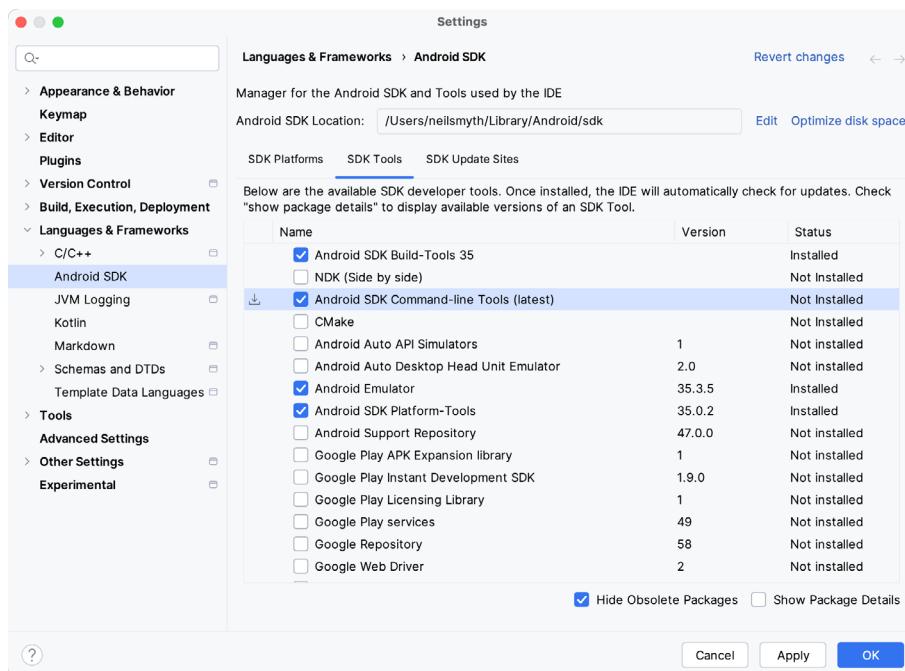


Figure 2-9

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

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

Setting up an Android Studio Development Environment

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

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

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

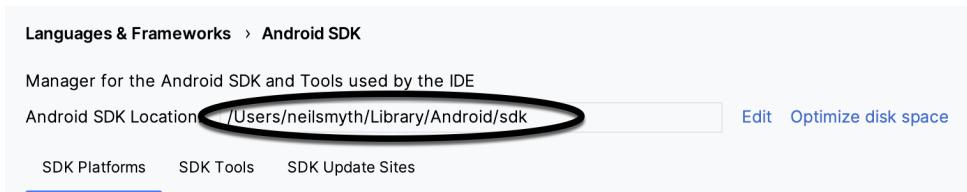


Figure 2-10

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

2.6.1 Windows 8.1

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

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

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

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

```
echo %Path%
```

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

```
adb
```

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

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

```
avdmanager
```

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

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

2.6.2 Windows 10

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

2.6.3 Windows 11

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

2.6.4 Linux

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

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

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

2.6.5 macOS

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

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

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

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

2.7 Android Studio memory management

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

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

Setting up an Android Studio Development Environment

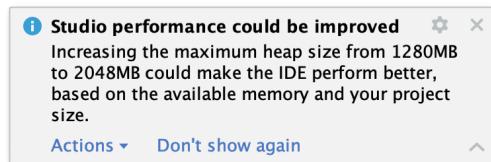


Figure 2-11

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

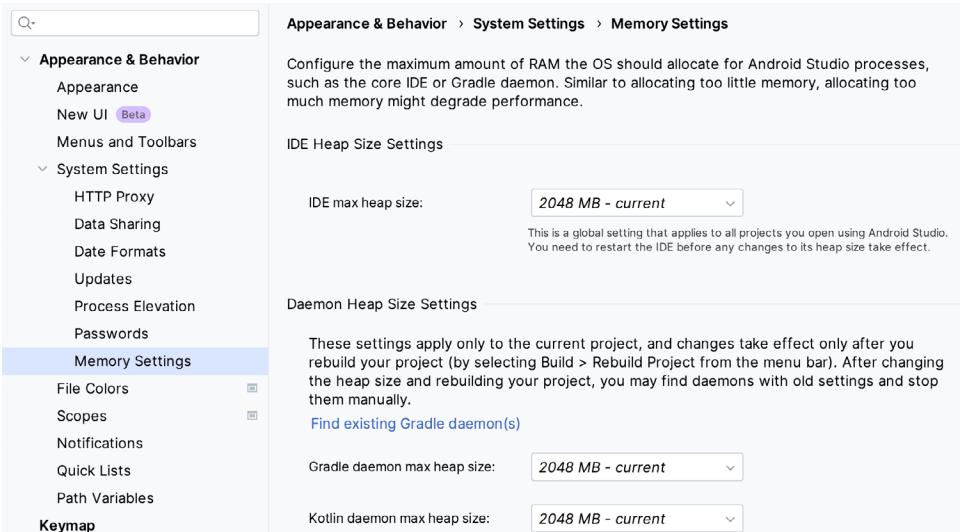


Figure 2-12

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

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

2.8 Updating Android Studio and the SDK

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

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

2.9 Summary

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

Chapter 13

13. Kotlin Operators and Expressions

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

13.1 Expression syntax in Kotlin

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

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

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

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

13.2 The Basic assignment operator

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

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

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

13.3 Kotlin arithmetic operators

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

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

The following table lists the primary Kotlin arithmetic operators:

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

/	Division
+	Addition
-	Subtraction
%	Remainder/Modulo

Table 13-1

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

For example:

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

13.4 Augmented assignment operators

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

```
x = x + y
```

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

```
x += y
```

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

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

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

Table 13-2

13.5 Increment and decrement operators

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

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

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

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

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

value of 10:

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

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

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

13.6 Equality operators

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

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

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

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

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

result = x < y
```

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

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

Table 13-3

13.7 Boolean logical operators

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

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

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

Kotlin Operators and Expressions

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

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

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

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

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

13.8 Range operator

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

The syntax for the range operator is as follows:

x..y

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

13.9 Bitwise operators

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

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

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

10101011

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

00000011

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

13.9.1 Bitwise inversion

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

00000011 NOT

```
=====
```

```
11111100
```

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

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

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

13.9.2 Bitwise AND

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

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

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

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

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

13.9.3 Bitwise OR

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

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

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

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

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

13.9.4 Bitwise XOR

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

```
10101011 XOR
```

Kotlin Operators and Expressions

```
00000011
```

```
=====
```

```
10101000
```

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

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

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

When executed, we get the following output from print:

```
Result is 168
```

13.9.5 Bitwise left shift

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

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

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

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

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

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

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

13.9.6 Bitwise right shift

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

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

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

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

```
val x = 171
```

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

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

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

13.10 Take the knowledge test



Click the link below or scan the QR code to test your knowledge and understanding of Kotlin operators and expressions:

<https://www.answertopia.com/3xrq>



13.11 Summary

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

22. 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*.

22.1 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.

22.2 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.

22.3 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.

22.4 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)
        enableEdgeToEdge()
        setContent {
            StateExampleTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    DemoScreen(Modifier.padding(innerPadding))
                }
            }
        }
    }
}

@Composable
fun DemoScreen(modifier: Modifier = Modifier) {
    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
    )
}
```

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

An Overview of Compose State and Recomposition

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.

22.5 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 DemoScreen(modifier: Modifier = Modifier) {

    var textState by remember { mutableStateOf("") }

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

    Column {
        MyTextField(text = textState, onTextChange = onTextChange)
        FunctionA()
    }
}

.

.

@Composable
fun FunctionA() {

    var switchState by remember { mutableStateOf(true) }

    val onSwitchChange = { value : Boolean ->
```

An Overview of Compose State and Recomposition

```
    switchState = value
}
}

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

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 22-1:

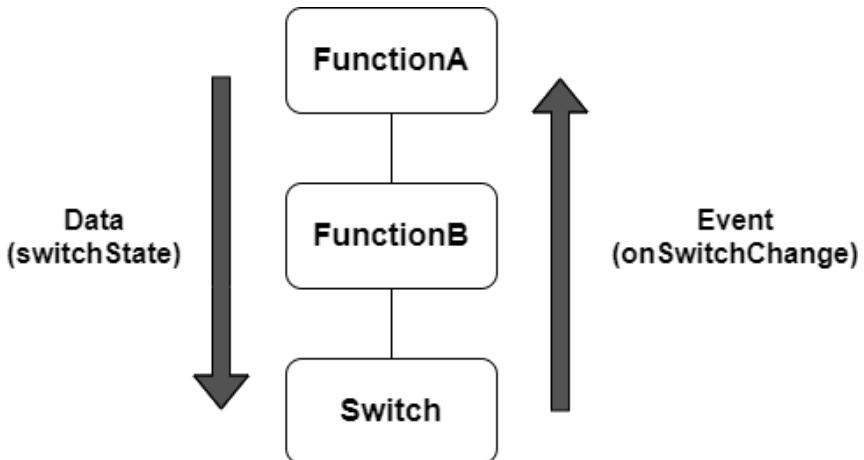


Figure 22-1

22.6 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(modifier: Modifier = Modifier) {
    MyTextField()
}

@Composable
fun MyTextField() {

    var textState by remember { mutableStateOf("") }

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

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

An Overview of Compose State and Recomposition

```
)  
}
```

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(modifier: Modifier = Modifier) {  
  
    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  
    )  
}  
.  
.
```

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 22-2 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:

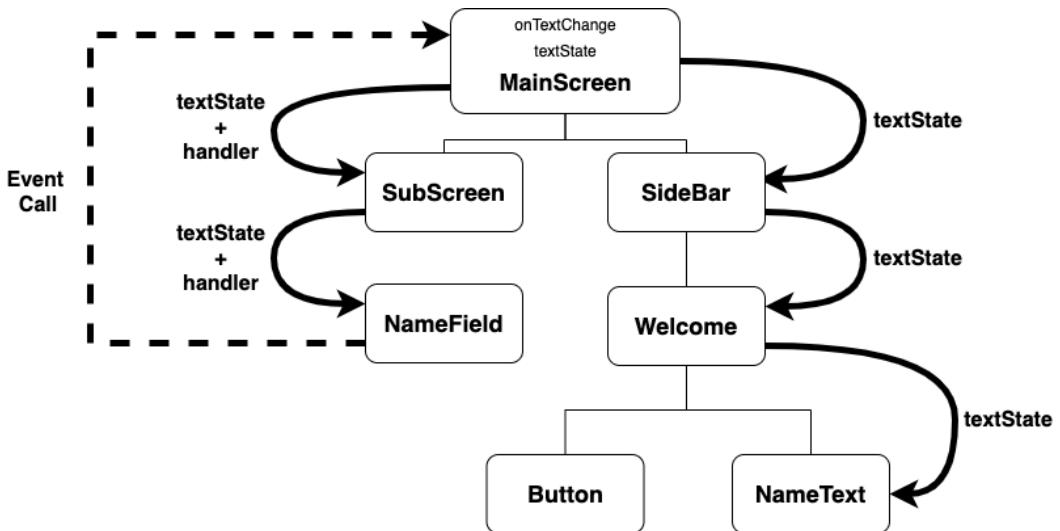


Figure 22-2

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.

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.

22.7 How high to hoist?

Hoisting is a good thing, but how high is too high? What goes up must come down, so remember that you must pass the state through each generation of descendants until you reach the lowest subscribing composable. Often, multiple children will subscribe to a single state, so the state should usually only be hoisted as high as the closest common ancestor.

Suppose functions F and J in Figure 22-3 below subscribe to the same state. Function D is the closest ancestor, so it is an excellent candidate to hold the state:

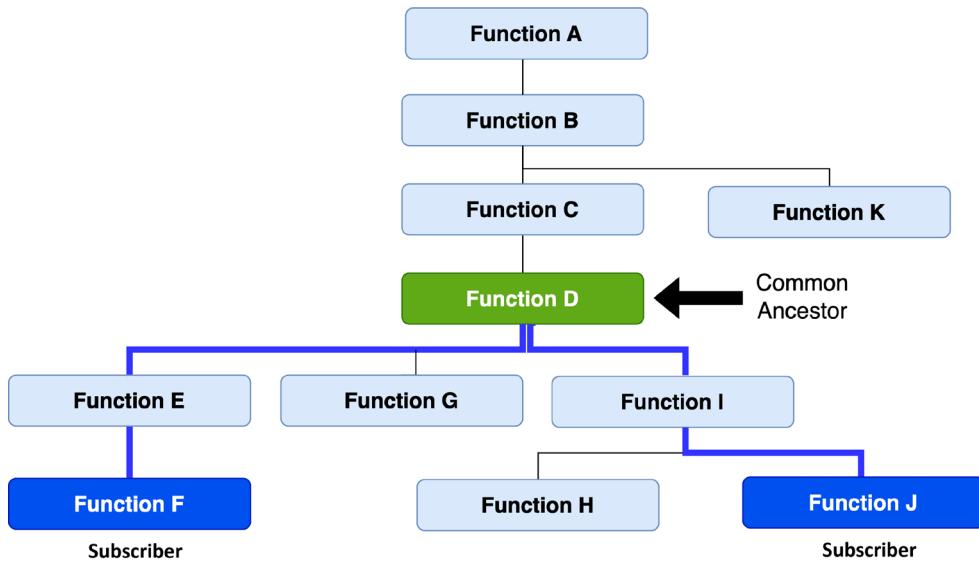


Figure 22-3

If function K also needs to subscribe, the state must be hoisted to function B:

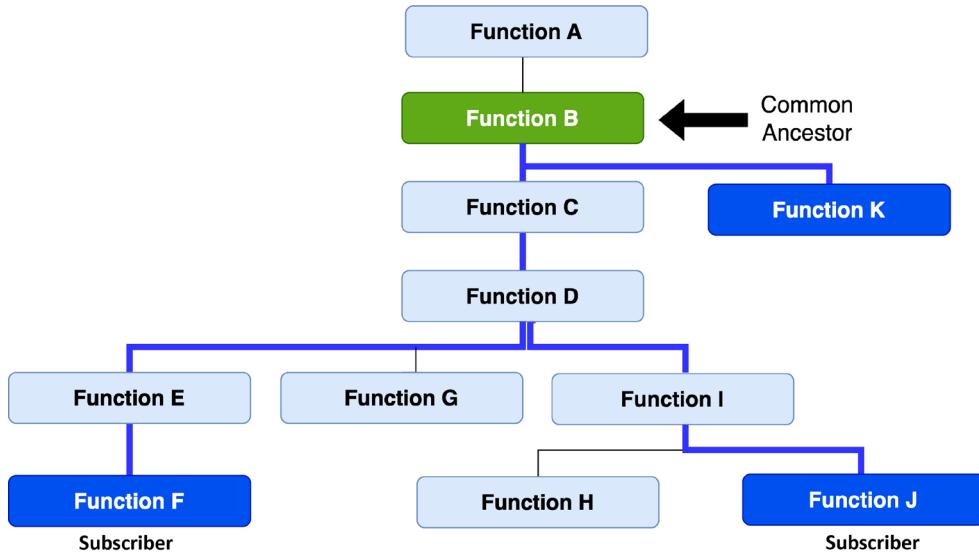


Figure 22-4

If you have declared a state in the lowest common ancestor and find yourself passing the state through an excessive number of descendants to reach all the subscribed children, consider using `CompositionLocalProvider`, a topic covered in the chapter entitled “*An Introduction to Composition Local*”.

22.8 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).

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 22-5:



Figure 22-5

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:

```

.
.
import androidx.compose.runtime.saveable.rememberSaveable
.
.
@Composable
fun DemoScreen(modifier: Modifier = Modifier) {
    var textState by rememberSaveable { mutableStateOf("") }
}
.
```

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.

22.9 Take the knowledge test



Click the link below or scan the QR code to test your knowledge and understanding of Compose state and recomposition:

<https://www.answertopia.com/c3sb>



22.10 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.

38. 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.

38.1 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.

38.2 Working with Column and Row lists

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

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

An Overview of Lists and Grids in Compose

```
}
```

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

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

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

38.3 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 38-1 below:

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

Figure 38-1

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.

38.4 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()
    }
}
```

38.5 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:

```
•  
•  
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:

```
•  
•  
Button(onClick = {  
    coroutineScope.launch {  
        scrollState.animateScrollTo(scrollState maxValue)  
    }  
    •  
    •  
})
```

38.6 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 38-2, 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 38-2

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)
```

```
.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 38-2 above.

38.7 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
}
```

38.8 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:

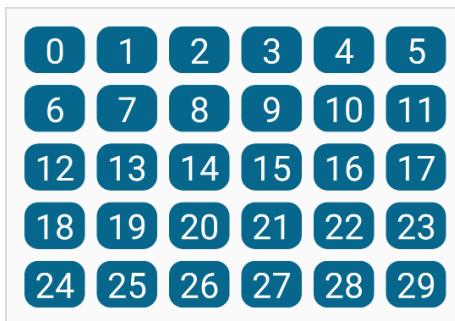


Figure 38-3

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 38-4 below:

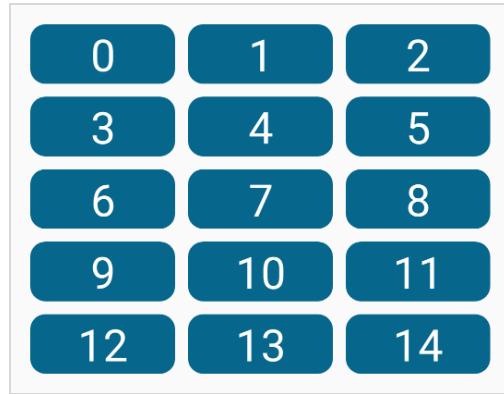


Figure 38-4

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 38-5:

Jetpack Compose

Card Example

Figure 38-5

38.9 Take the knowledge test



Click the link below or scan the QR code to test your knowledge and understanding of row and column lists:

<https://www.answertopia.com/iifq>



38.10 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.

47. 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.

47.1 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.

47.2 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.

47.3 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.

47.4 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.

Working with ViewModels in Compose

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 47-1 illustrates this concept as it relates to activities and ViewModels:

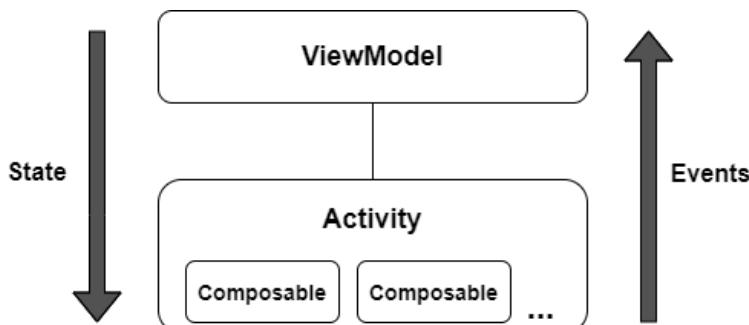


Figure 47-1

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*”.

47.5 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.

47.6 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)
        enableEdgeToEdge()
        setContent {
            ViewModelDemoTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    TopLevel(modifier = Modifier.padding(innerPadding))
                }
            }
        }
    }
}

@Composable
fun TopLevel(
    modifier: Modifier = Modifier,
    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))
```

Working with ViewModels in Compose

```
        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(
    modifier: Modifier = Modifier,
    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. This requires the following additions to the `Gradle Scripts -> libs.version.toml` file:

```
[versions]
activityCompose = "1.8.2"
.

.

[libraries]
androidx-lifecycle-viewmodel-compose = { module = "androidx.lifecycle:lifecycle-
viewmodel-compose", version.ref = "lifecycleRuntimeKtx" }
.
```

Once the library has been added to the version catalog, it must be added to the `dependencies` section of the `Gradle Scripts -> build.gradle.kts (Module :app)` file:

```
dependencies {
    .
    .
    implementation(libs.androidx.lifecycle.viewmodel.compose)
    .
    .
}
```

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*”.

47.7 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.

47.8 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(
    modifier: Modifier = Modifier,
    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(
    modifier: Modifier = Modifier,
    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*”.

47.9 Take the knowledge test



Click the link below or scan the QR code to test your knowledge and understanding of view models in Compose:

<https://www.answertopia.com/dooi>



47.10 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.

Index

Index

Symbols

? 111
2D graphics 413
@Composable 23, 171
@ExperimentalFoundationApi 358
:: operator 113
@Preview 23
 showSystemUi 23

A

acknowledgePurchase() method 619

actionSendBroadcast() 559

actionStartActivity() 559

actionStartService() 559

Activity Manager 98

adb

 command-line tool 73

 connection testing 79

 device pairing 77

 enabling on Android devices 73

 Linux configuration 76

 list devices 73

 macOS configuration 74

 overview 73

 restart server 74

 testing connection 79

 WiFi debugging 77

 Windows configuration 75

Wireless debugging 77

Wireless pairing 77

Ahead-of-Time 95

AlertDialog 176

align() 253

alignByBaseline() 245

Alignment.Bottom 240, 244

Alignment.BottomCenter 252
Alignment.BottomEnd 252
Alignment.BottomStart 252
Alignment.Center 251
Alignment.CenterEnd 252
Alignment.CenterHorizontally 240
Alignment.CenterStart 251
Alignment.CenterVertically 240, 244
Alignment.End 240
alignment lines 275
Alignment.Start 240
Alignment.Top 240, 244
Alignment.TopCenter 251
Alignment.TopEnd 251
Alignment.TopStart 251
Android
 architecture 93
 runtime 94
 SDK Packages 6
 android.app 96
 Android Application Package 95
 Android Architecture Components 433
 android.content 96
 android.database 96
 Android Debug Bridge. *See* ADB
 Android Development
 System Requirements 3
 android.graphics 96
 android.hardware 96
 android.hardware.camera 577
 Android Jetpack 433
 Android Libraries 96
 android.media 97
 Android Monitor tool window 44
 android.net 97
 android.opengl 96
 android.os 96
 android.print 97
 android.provider 97

Index

Android SDK Location
 identifying 10
Android SDK Manager 8, 10
Android SDK Packages
 version requirements 8
Android SDK Tools
 command-line access 9
 Linux 11
 macOS 11
 Windows 7 10
 Windows 8 10
Android Software Stack 93
Android Studio
 Animation Inspector 411
 Asset Studio 208
 changing theme 71
 Database Inspector 482
 downloading 3
 Editor Window 66
 installation 4
 Layout Editor 161
 Linux installation 5
 macOS installation 4
 Navigation Bar 65
 Project tool window 66
 setup wizard 5
 Status Bar 66
 Toolbar 65
 Tool window bars 66
 tool windows 66
 updating 12
 Welcome Screen 63
 Windows installation 4
Android Support Library, 433
android.text 97
android.util 97
android.view 97
Android Virtual Device. *See* AVD
 overview 37
Android Virtual Device Manager 37
android.webkit 97
android.widget 97
AndroidX libraries 654
animate as state functions 397
animateColorAsState() function 397, 401, 403
animateDpAsState() function 403, 408
AnimatedVisibility 385
 animation specs 389
 enter and exit animations 388
 expandHorizontally() 388
 expandIn() 388
 expandVertically() 388
 fadeIn() 388
 fadeOut() 389
 MutableTransitionState 393
 scaleIn() 389
 scaleOut() 389
 shrinkHorizontally() 389
 shrinkOut() 389
 shrinkVertically() 389
 slideIn() 389
 slideInHorizontally() 389
 slideInVertically() 389
 slideOut() 389
 slideOutHorizontally() 389
 slideOutVertically() 389
 animateEnterExit() modifier 392
 animateFloatAsState() function 398
 animateScrollTo() function 332, 343
 animateScrollToItem(index: Int) 332
 animateScrollTo(value: Int) 331
Animation
 auto-starting 392
 combining animations 408
 inspector 411
 keyframes 407
 KeyframesSpec 407
 motion 403
 spring effects 406
 state-based 397
 visibility 385
Animation damping
 DampingRatioHighBouncy 406
 DampingRatioLowBouncy 406

DampingRatioMediumBouncy 406
 DampingRatioNoBouncy 406
 Animation Inspector 411
 AnimationSpec 389
 tween() function 390
 Animation specs 389
 Animation stiffness
 StiffnessHigh 407
 StiffnessLow 407
 StiffnessMedium 407
 StiffnessMediumLow 407
 StiffnessVeryLow 407
 annotated strings 225, 429
 append function 225
 buildAnnotatedString function 225
 ParagraphStyle 226
 SpanStyle 225
 AOT 95
 API Key 587
 APK 95
 APK analyzer 612
 APK file 605
 APK File
 analyzing 612
 APK Signing 654
 APK Wizard dialog 604
 App Bundles 601
 creating 605
 overview 601
 revisions 611
 uploading 608
 append function 225
 App Inspector 67
 Application
 stopping 44
 Application Framework 98
 Arrangement.Bottom 241
 Arrangement.Center 241
 Arrangement.End 241
 Arrangement.SpaceAround 242
 Arrangement.SpaceBetween 242
 Arrangement.SpaceEvenly 242
 Arrangement.Start 241
 Arrangement.Top 241
 ART 94
 as 113
 as? 113
 asFlow() builder 532
 Asset Studio 208
 asSharedFlow() 544
 asStateFlow() 542
 async 323
 AVD
 Change posture 60
 cold boot 56
 command-line creation 37
 creation 37
 device frame 48
 Display mode 59
 launch in tool window 48
 overview 37
 quickboot 56
 Resizable 59
 running an application 40
 Snapshots 55
 standalone 45
 starting 39
 Startup size and orientation 40

B

background modifier 222
 barriers 305
 Barriers 290
 constrained views 290
 baseline
 alignment 244
 baselines 277
 BaseTextField 175
 BillingClient 620
 acknowledgePurchase() method 619
 consumeAsync() method 619
 getPurchaseState() method 619
 initialization 616, 627
 launchBillingFlow() method 618

Index

queryProductDetailsAsync() method 618
queryPurchasesAsync() method 620
startConnection() method 617
BillingResult 636
getDebugMessage() 636
Bill of Materials. *See* BOM
Biometric Authentication 575
 callbacks 580
 overview 575
 tutorial 575
Biometric library 576
BiometricManager 575
BiometricPrompt 575
Bitwise AND 119
Bitwise Inversion 118
Bitwise Left Shift 120
Bitwise OR 119
Bitwise Right Shift 120
Bitwise XOR 119
BOM 25
 build.gradle.kts 25
 compose-bom 25
 library version mapping 25
Boolean 106
BottomNavigation 176, 491
BottomNavigationItem 491
Box 175
 align() 253
 alignment 251
 Alignment.BottomCenter 252
 Alignment.BottomEnd 252
 Alignment.BottomStart 252
 Alignment.Center 251
 Alignment.CenterEnd 252
 Alignment.CenterStart 251
 Alignment.TopCenter 251
 Alignment.TopEnd 251
 Alignment.TopStart 251
BoxScope 253
 contentAlignment 251
 matchParentSize() 253
 overview 249
tutorial 249
BoxScope
 align() 253
 matchParentSize() 253
 modifiers 253
BoxWithConstraints 175
Broadcast Receiver 555, 556
Brush Text Styling 226
buffer() operator 537
buildAnnotatedString function 225
Build Variants , 68
 tool window 68
Button 176
by keyword 182

C

CAMERA permission 577
CameraUpdateFactory class
 methods 597
cancelAndJoin() 324
cancelChildren() 324
Canvas 175
 DrawScope 413
 inset() function 417
 overview 413
 size 413
Card 176
 example 336
C/C++ Libraries 97
centerAround() function 294
chain head 288
chaining modifiers 217
chains 288
chain styles 288
Char 106
Checkbox 176, 206
Circle class 583
CircleShape 254
CircularProgressIndicator 176
clickable 222
clickable() 558
clip 222

Clip Art 209
 clip() modifier 254
 CircleShape 254
 CutCornerShape 254
 RectangleShape 254
 RoundedCornerShape 254
 close() function 425
 Code completion 84
 Code Editor
 basics 81
 Code completion 84
 Code Generation 86
 Code Reformatting 89
 Document Tabs 82
 Editing area 82
 Gutter Area 82
 Live Templates 90
 Splitting 84
 Statement Completion 86
 Status Bar 83
 Code Generation 86
 Code Reformatting 89
 code samples
 download 2
 Coil
 rememberImagePainter() function 350
 cold boot 56
 Cold flow 542
 convert to hot 545
 collectLatest() operator 536
 collect() operator 532
 ColorFilter 428
 color filtering 428
 Column 175
 Alignment.CenterHorizontally 240
 Alignment.End 240
 Alignment.Start 240
 Arrangement.Bottom 241
 Arrangement.Center 241
 Arrangement.SpaceAround 242
 Arrangement.SpaceBetween 242
 Arrangement.SpaceEvenly 242
 Arrangement.Top 241
 Layout alignment 238
 list 329
 list tutorial 339
 overview 237
 scope 243
 scope modifiers 243
 spacing 242
 tutorial 235
 verticalArrangement 241
 Column lists 329
 ColumnScope 243
 Modifier.align() 244
 Modifier.alignBy() 244
 Modifier.weight() 244
 combine() operator 541
 combining modifiers 223
 Communicating Sequential Processes 321
 Companion Objects 143
 components 171
 Composable
 adding a 28
 previewing 30
 Composable function
 syntax 172
 composable functions 171
 composable 162
 add modifier support 218
 Composables
 Foundation 175
 Material 175
 Compose
 before 161
 components 171
 data-driven 162
 declarative syntax 161
 functions 171
 layout overview 271
 modifiers 215
 overview 161
 state 162
 compose-bom 25

Index

compose() method 487
CompositionLocal
example 193
overview 191
state 194, 195
syntax 192
compositionLocalOf() function 192
conflate() operator 537
constrainAs() modifier function 293
constrain() function 310
Constraint bias 299
Constraint Bias 287
ConstraintLayout 175
adding constraints 294
barriers 305
Barriers 290
basic constraints 296
centerAround() function 294
chain head 288
chains 288
chain styles 288
constrainAs() function 293
constrain() function 310
Constraint bias 299
Constraint Bias 287
Constraint margins 299
Constraints 285
constraint sets 308
createEndBarrier() 305
createHorizontalChain() 303
createRefFor() function 310
createRef() function 293
createRefs() function 293
createStartBarrier() 305
createTopBarrier() 305
createVerticalChain() 303
creating chains 303
generating references 293
guidelines 304
Guidelines 289
how to call 293
layout() modifier 310
linkTo() function 294
Margins 286
Opposing constraints 297
Opposing Constraints 286, 301
overview of 285
Packed chain 289
reference assignment 293
Spread chain 288
Spread inside chain 288
Weighted chain 288
Widget Dimensions 289
Constraint margins 299
constraints 280
constraint sets 308
consumeAsync() method 619
ConsumeParams 631
contentAlignment 251
Content Provider 98
Coroutine Builders 323
 async 323
 coroutineScope 323
 launch 323
 runBlocking 323
 supervisorScope 323
 withContext 323
Coroutine Dispatchers 322
Coroutines 332, 529
 channel communication 325
 coroutine scope 332
 CoroutineScope 332
 GlobalScope 322
 LaunchedEffect 326
 rememberCoroutineScope() 332
 rememberCoroutineScope() function 322
 SideEffect 326
 Side Effects 326
 Suspend Functions 322
 suspending 324
 ViewModelScope 322
 vs Threads 321
 vs. Threads 321
 coroutineScope 323

CoroutineScope 322, 332
 rememberCoroutineScope() 332
 createEndBarrier() 305
 createHorizontalChain() 303
 createRefFor() function 310
 createRef() function 293
 createRefs() 293
 createStartBarrier() 305
 createTopBarrier() 305
 createVerticalChain() 303
 cross axis arrangement 268
 Crossfading 393
 currentBackStackEntryAsState() method 492, 510
 Custom Accessors 141
 Custom layout 279
 building 279
 constraints 280
 Layout() composable 280
 measurables 280
 overview 279
 Placeable 280
 syntax 279
 custom layout modifiers 271
 alignment lines 275
 baselines 277
 creating 273
 default position 273
 Custom layouts
 overview 271
 tutorial 271
 Custom Theme
 building 640
 CutCornerShape 254

D

Dalvik 95
 DampingRatioHighBouncy 406
 DampingRatioLowBouncy 406
 DampingRatioMediumBouncy 406
 DampingRatioNoBouncy 406
 Dark Theme 44
 enable on device 44

dashPathEffect() method 415
 Data Access Object (DAO) 456, 470
 Data Access Objects 459
 Database Inspector 463, 482
 live updates 482
 SQL query 482
 Database Rows 450
 Database Schema 449
 Database Tables 449
 data-driven 162
 DDMS 44
 Debugging
 enabling on device 73
 declarative syntax 161
 Default Function Parameters 133
 default position 273
 derivedStateOf 361
 Device File Explorer 68
 device frame 48
 Device Mirroring 79
 enabling 79
 device pairing 77
 DEX code 95
 Dispatchers.Default 323
 Dispatchers.IO 323
 Dispatchers.Main 322
 drag gestures 518
 drawable
 folder 208
 drawArc() function 424
 drawCircle() function 420
 drawImage() function 427
 Drawing
 arcs 424
 circle 420
 close() 425
 dashed lines 415
 dashPathEffect() 415
 drawArc() 424
 drawImage() 427
 drawPath() 425
 drawPoints() 426

Index

drawRect() 415
drawRoundRect() 418
gradients 421
images 427
line 413
oval 420
points 426
rectangle 415
rotate() 419
rotation 419
Drawing text 429
drawLine() function 414
drawPath() function 425
drawPoints() function 426
drawRect() function 415
drawRoundRect() function 418
DrawScope 413
drawText() function 429, 430
DropdownMenu 176
DROP_LATEST 544
DROP_OLDEST 544
DurationBasedAnimationSpec 389
Dynamic colors
enabling in Android 650

E

Elvis Operator 113
emit 171
Empty Compose Activity
template 16
Emulator
battery 54
cellular configuration 54
configuring fingerprints 56
directional pad 54
extended control options 53
Extended controls 53
fingerprint 54
location configuration 54
phone settings 54
Resizable 59
resize 53

rotate 52
Screen Record 55
Snapshots 55
starting 39
take screenshot 52
toolbar 51
toolbar options 51
tool window mode 58
Virtual Sensors 55
zoom 52
enablePendingPurchases() method 619
enabling ADB support 73
enter animations 388
EnterTransition.None 392
Errata 2
Escape Sequences 107
exit animations 388
ExitTransition.None 392
expandHorizontally() 388
expandIn() 388
expandVertically() 388
Extended Control
options 53

F

fadeIn() 388
fadeOut() 389
Files
switching between 82
fillMaxHeight 222
fillMaxSize 222
fillMaxWidth 222
filter() operator 535
findStartDestination() method 492
Fingerprint
emulation 56
Fingerprint authentication
device configuration 576
overview 575
steps to implement 575
tutorial 575
firstVisibleItemIndex 334

flatMapConcat() operator 540
 flatMapMerge() operator 540
 Float 106
 FloatingActionButton 176
 Flow 529
 asFlow() builder 532
 asSharedFlow() 544
 asStateFlow() 542
 backgroudn handling 551
 buffering 537
 buffer() operator 537
 builder 531
 cold 542
 collect() 535
 collecting data 535
 collectLatest() operator 536
 combine() operator 541
 conflate() operator 537
 emit() 532
 emitting data 532
 filter() operator 535
 flatMapConcat() operator 540
 flatMapMerge() operator 540
 flattening 539
 flowOf() builder 532
 flow of flows 539
 fold() operator 539
 hot 542
 MutableSharedFlow 544
 MutableStateFlow 542
 onEach() operator 541
 reduce() operator 538, 539
 repeatOnLifecycle 552
 SharedFlow 543
 shareIn() function 545
 single() operator 537
 StateFlow 542
 transform() operator 535
 try/finally 536
 zip() operator 541
 flow builder 531
 FlowColumn 257, 263, 267
 cross axis arrangement 268
 maxItemsInEachColumn 258
 tutorial 263
 Flow layout
 arrangement 265
 Flow layouts
 cross axis arrangement 259
 fillMaxHeight() 261
 fillMaxWidth() 261
 Fractional sizing 261
 horizontalArrangement 268
 Item alignment 260
 item weights 269
 main axis arrangement 258
 verticalArrangement 268
 weight 260
 flowOf() builder 532
 flow of flows 539
 FlowRow 257, 263, 264
 cross axis arrangement 268
 horizontalArrangement 265
 item alignment 266
 maxItemsInEachRow 258
 tutorial 263
 Flows
 combining 541
 Introduction to 529
 FontWeight 29
 forEachIndexed 269
 forEach loop 282
 Forward-geocoding 590
 Foundation components 175
 Foundation Composables 175
 FragmentActivity 577
 Function Parameters
 variable number of 133
 Functions 131

G

Gemini 153
 asking questions 156
 configuration 155

Index

enabling 153
in Android Studio 153
inline code completion 157
overview 153
playground 156
proposed changes 159
question context 157
tool window 154
transforming code 158
Geocoder object 590
Geocoding 589
Gestures 515
click 515
drag 518
horizontalScroll() 522
overview 515
pinch gestures 524
PointerInputScope 517
rememberScrollableState() function 521
rememberScrollState() 522
rememberTransformableState() 524
rotation gestures 525
scrollable() modifier 521
scroll modifiers 522
taps 517
translation gestures 526
tutorial 515
verticalScroll() 522
getDebugMessage() 636
getFromLocation() method 590
getPurchaseState() method 619
getStringArray() method 347
Glance
actionSendBroadcast() 559
actionStartActivity() 559
actionStartService() 559
Broadcast Receiver 555, 556, 565
clickable() 558
composables 555
GlanceAppWidget 555, 564
glanceAppWidget property 556
GlanceAppWidgetReceiver 556
intents 556
libraries 561
LocalSize.current 558, 572
minHeight 557
minWidth 557
overview 555
provideGlance() 555
provider info data 557
RemoteViews 555
resizeMode 558
sizeMode 569
size modes 558
targetCellHeight 557
targetCellWidth 557
tutorial 561
updateAll() 559
user interaction 558
GlanceAppWidget 555
glanceAppWidget property 556
GlanceAppWidgetReceiver 556
glanceAppWidget property 556
Glance Widgets 555
GlobalScope 322
GLSurfaceView 97
GNU/Linux 94
Google Cloud
billing account 584
new project 585
Google font libraries 648
GoogleMap 583
Google Maps Android API 583
Controlling the Map Camera 597
displaying controls 593
Map Markers 596
overview 583
Google Maps SDK 583
API Key 587
Credentials 586
enabling 586
Maps SDK for Android 586
Google Play App Signing 604
Google Play Billing Library 615

Google Play Console 624
 Creating an in-app product 624
 License Testers 625
 Google Play Developer Console 602
 Google Play store 17
 Gradient drawing 421
Gradle
 APK signing settings 659
 Build Variants 654
 command line tasks 660
 dependencies 653
 Manifest Entries 654
 overview 653
 sensible defaults 653
Gradle Build File
 top level 655
Gradle Build Files
 module level 656
gradle.properties file 654
Graphics
 drawing 413
Grid
 overview 329
groupBy() function 333
guidelines 304

H

HAL 94
 Hardware Abstraction Layer 94
 Higher-order Functions 135
 horizontalArrangement 241, 242, 268
HorizontalPager 373
 animateScrollToPage() 375
 scrollToPage() 375
 state 374
 syntax 373
horizontalScroll() 522
Hot flows 542

I

Image 175
 add drawable resource 208

painterResource method 210
Immutable Variables 108
INAPP 620
In-App Products 616
In-App Purchasing 623
 acknowledgePurchase() method 619
 BillingClient 616
 BillingResult 636
 consumeAsync() method 619
 ConsumeParams 631
 Consuming purchases 630
 enablePendingPurchases() method 619
 getPurchaseState() method 619
 Google Play Billing Library 615
 launchBillingFlow() method 618
 Libraries 623
 newBuilder() method 616
 onBillingServiceDisconnected() callback 628
 onBillingServiceDisconnected() method 617
 onBillingSetupFinished() listener 628
 onProductDetailsResponse() callback 628
 Overview 615
 ProductDetail 618
 ProductDetails 629
 products 616
 ProductType 620
 Purchase Flow 630
 PurchaseResponseListener 620
 PurchasesUpdatedListener 619
 PurchaseUpdatedListener 629
 purchase updates 629
 queryProductDetailsAsync() 628
 queryProductDetailsAsync() method 618
 queryPurchasesAsync() 631
 queryPurchasesAsync() method 620
 startConnection() method 617
 subscriptions 616
 tutorial 623

Initializer Blocks 141
In-Memory Database 462
Inner Classes 142
 innerPadding 22

Index

inset() function 417
IntrinsicSize.Max 317
IntrinsicSize.Min 317, 318
intelligent recomposition 179
IntelliJ IDEA 101
Interactive mode 34
Intrinsic measurements 313
IntrinsicSize 313
 intrinsic measurements 313
 Max 313
 Min 313
 tutorial 315
is 113
isInitialized property 113
isSystemInDarkTheme() function 195
item() function 330
items() function 330
itemsIndexed() function 330

J

Java
 convert to Kotlin 101
JetBrains 101
Jetpack Compose
 see Compose 161
JIT 95
join() 324
Just-in-Time 95

K

K2 mode 466
keyboardOptions 446
Keyboard Shortcuts 70
keyframe 390
keyframes 407
 KeyframesSpec 407
keyframes() function 407
KeyframesSpec 407
Keystore File
 creation 604
Kotlin
 accessing class properties 141
 and Java 101
 arithmetic operators 115
 assignment operator 115
 augmented assignment operators 116
 bitwise operators 118
 Boolean 106
 break 126
 breaking from loops 125
 calling class methods 141
 Char 106
 class declaration 137
 class initialization 138
 class properties 138
 Companion Objects 143
 conditional control flow 127
 continue labels 126
 continue statement 126
 control flow 123
 convert from Java 101
 Custom Accessors 141
 data types 105
 decrement operator 116
 Default Function Parameters 133
 defining class methods 138
 do ... while loop 125
 Elvis Operator 113
 equality operators 117
 Escape Sequences 107
 expression syntax 115
 Float 106
 Flow 529
 for-in statement 123
 function calling 132
 Functions 131
 groupBy() function 333
 Higher-order Functions 135
 if ... else ... expressions 128
 if expressions 127
 Immutable Variables 108
 increment operator 116
 inheritance 147
 Initializer Blocks 141

Inner Classes 142
 introduction 101
 Lambda Expressions 134
 let Function 111
 Local Functions 132
 logical operators 117
 looping 123
 Mutable Variables 108
 Not-Null Assertion 111
 Nullable Type 110
 object 562
 Overriding inherited methods 150
 playground 102
 Primary Constructor 138
 properties 141
 range operator 118
 Safe Call Operator 110
 Secondary Constructors 138
 Single Expression Functions 132
 singleton 563
 String 106
 subclassing 147
 subStringBefore() method 349
 Type Annotations 109
 Type Casting 113
 Type Checking 113
 Type Inference 109
 variable parameters 133
 when statement 128
 while loop 124

L

Lambda Expressions 134
 Large Language Model 153
 lateinit 112
 Late Initialization 112
 launch 323
 launchBillingFlow() method 618
 LaunchedEffect 326
 launchSingleTop 489
 Layout alignment 238
 Layout arrangement 241
 Layout arrangement spacing 242
 Layout components 175
 Layout() composable 280
 Layout Editor 161
 Layout Inspector 69
 layout modifier 222
 layout() modifier 310
 LazyColumn 175, 329
 creation 330
 scroll position detection 334
 LazyHorizontalStaggeredGrid 365, 369
 syntax 366
 LazyList
 tutorial 345
 Lazy lists 329
 Scrolling 331
 LazyListScope 330
 item() function 330
 items() function 330
 itemsIndexed() function 330
 stickyHeader() function 332
 LazyListState 334
 firstVisibleItemIndex 334
 LazyRow 175, 329
 creation 330
 scroll position detection 334
 LazyVerticalGrid 329
 adaptive mode 334
 fixed mode 334
 LazyVerticalStaggeredGrid 365, 368
 syntax 365
 let Function 111
 libs.versions.toml file 166
 License Testers 625
 Lifecycle.State.CREATED 552
 Lifecycle.State.DESTROYED 553
 Lifecycle.State.INITIALIZED 552
 Lifecycle.State.RESUMED 553
 Lifecycle.State.STARTED 553
 LinearProgressIndicator 176
 lineTo() 425
 lineTo() function 425

Index

linkTo() function 294
Linux Kernel 94
list devices 73
Lists
 clickable items 353
 enabling scrolling 331
 overview 329
literals
 live editing 30
LiveData 437
 observeAsState() 437
Live Edit 41
 disabling 30
 enabling 30
 of literals 30
Live Templates 90
LLM 153
Local Functions 132
LocalSize.current 558, 572
Location Manager 98
Logcat
 tool window 68

M

MainActivity.kt file 19
 template code 27
map method 280
Maps 583
MAP_TYPE_HYBRID 592
MAP_TYPE_NONE 592
MAP_TYPE_NORMAL 592
MAP_TYPE_SATELLITE 592
MAP_TYPE_TERRAIN 592
Marker class 583
matchParentSize() 253
Material Composables 175
Material Design 22
Material Design 2 637
Material Design 2 Theming 637
Material Design 3 637
Material Design components 176
Material Theme Builder 640
Material You 637
maxValue property 343
measurables 280
measure() function 430
measureTimeMillis() function 537
Memory Indicator 83
Minimum SDK
 setting 17
ModalDrawer 176
Modern Android architecture 433
modifier
 adding to composable 218
 chaining 217
 combining 223
 creating a 216
 ordering 218
 tutorial 215
Modifier.align() 244
Modifier.alignBy() 244
modifiers
 build-in 222
 overview 215
Modifier.weight() 244
move() method 597
multiple devices
 testing app on 43
MutableLiveData 437
MutableSharedFlow 544
MutableState 180
MutableStateFlow 542
mutableStateOf function 171
mutableStateOf() function 181
MutableTransitionState 393
Mutable Variables 108
My Location Layer 583

N

Native Development Kit 98
NavHost 487, 499, 509
NavHostController 485, 499, 509
navigate() method 489
Navigation 485

BottomNavigation 491
 BottomNavigationItem 491
 compose() method 487
 currentBackStackEntryAsState() method 492
 declaring routes 496
 findStartDestination() method 492
 graph 487
 launchSingleTop 489
 NavHost 487, 499
 NavHostController 485, 499
 navigate() method 489
 navigation graph 485
 NavType 490
 overview 485
 passing arguments 490
 popUpTo() method 489
 route 487
 stack 485, 486
 start destination 487
 tutorial 495

Navigation Architecture Component 485
 NavigationBar 510
 NavigationBarItem 511
 Navigation bars 491
 navigation graph 485, 487
 Navigation Host 487
 NavType 490
 NDK 98
 newBuilder() method 616
 Notifications Manager 98
 Not-Null Assertion 111
 Nullable Type 110

O

object 562
 observeAsState() 437
 Offset() function 414
 offset modifier 222
 onBillingServiceDisconnected() callback 628
 onBillingServiceDisconnected() method 617
 onBillingSetupFinished() listener 628
 onCreate() method 23

onEach() operator 541
 onProductDetailsResponse() callback 628
 OpenJDK 3
 Opposing constraints 297
 OutlinedButton 361
 OutlinedTextField 439

P

Package Manager 98
 Package name 17
 Packed chain 289
 padding 222
 Pager 373

- animateScrollToPage() 375
- scrollToPage() 375
- state 374
- syntax 258, 373

Pager state 374
 painterResource method 210
 ParagraphStyle 226
 PathEffect 415
 pinch gestures 524
 Placeable 280
 PointerInputScope 517

- drag gestures 520
- tap gestures 517

popUpTo() method 489
 Preview configuration picker 33
 Preview panel 24

- build and refresh 24
- Interactive mode 34
- settings 33

Primary Constructor 138
 Problems

- tool window 68, 69

ProductDetail 618
 ProductDetails 629
 ProductType 620
 Profiler

- tool window 69

proguard-rules.pro file 658
 ProGuard Support 654

Index

project

 create new 16

 package name 17

Project tool window 18, 67

 Android mode 18

provideGlance() 555

PurchaseResponseListener 620

PurchasesUpdatedListener 619, 629

Q

queryProductDetailsAsync() 628

queryProductDetailsAsync() method 618

queryPurchaseHistoryAsync() method 620

queryPurchasesAsync() 631

queryPurchasesAsync() method 620

quickboot snapshot 56

Quick Documentation 89

R

RadioButton 176

Random

 nextInt() 264

Random.nextInt() method 367, 264

Range Operator 118

Recent Files Navigation 70

recomposition 162

 intelligent recombination 179

 overview 179

RectangleShape 254

reduce() operator 538, 539

relativeLineTo() function 425

Release Preparation 601

rememberCoroutineScope() function 322, 332, 341

rememberDraggableState() function 518

rememberImagePainter() function 350

remember keyword 181

rememberPagerState 374

rememberSaveable keyword 189

rememberScrollableState() function 521

rememberScrollState() 522

rememberScrollState() function 331, 341

rememberTextMeasurer() function 429

rememberTransformableState() 524

rememberTransformationState() function 524

RemoteViews 555

repeatable() function 391

RepeatableSpec

 repeatable() 391

RepeatMode.Reverse 391

repeatOnLifecycle 552

Repository

 tutorial 465

Resizable Emulator 59

Resource Manager 98, 67

Reverse-geocoding 590

Reverse Geocoding 589

Room

 Data Access Object (DAO) 456

 entities 456, 457

 In-Memory Database 462

 Repository 455

Room Database 456

 tutorial 465

Room Database Persistence 455

Room Persistence Library 453

rotate modifier 222

rotation gestures 525

RoundedCornerShape 254

Row 175

 Alignment.Bottom 240

 Alignment.CenterVertically 240

 Alignment.Top 240

 Arrangement.Center 241

 Arrangement.End 241

 Arrangement.SpaceAround 242

 Arrangement.SpaceBetween 242

 Arrangement.SpaceEvenly 242

 Arrangement.Start 241

 horizontalArrangement 241

 Layout alignment 238

 Layout arrangement 241

 list 329

 list example 344

 overview 236

scope 243
 scope modifiers 243
 spacing 242
 tutorial 235
 Row lists 329
 RowScope 243
 Modifier.align() 244
 Modifier.alignBy() 244
 Modifier.alignByBaseline() 244
 Modifier.paddingFrom() 244
 Modifier.weight() 244
 Run
 tool window 67
 runBlocking 323
 Running Devices
 tool window 79

S

Safe Call Operator 110
 Scaffold 22, 28, 176, 511
 bottomBar 512
 TopAppBar 512
 scaleIn() 389
 scale modifier 222
 scaleOut() 389
 Scope modifiers
 weights 247
 scrollable modifier 222
 scrollable() modifier 521, 522
 Scroll detection
 example 357
 scroll modifiers 522
 ScrollState
 maxValue property 343
 rememberScrollState() function 331
 scrollToItem(index: Int) 332
 scrollToPage() 375
 scrollTo(value: Int) 331
 SDK Packages 6
 SDK settings 17
 Secondary Constructors 138
 settings.gradle file 654
 settings.gradle.kts file 654
 Shape 176
 Shapes
 CircleShape 254
 CutCornerShape 254
 RectangleShape 254
 RoundedCornerShape 254
 SharedFlow 543, 547
 background handling 551
 DROP_LATEST 544
 DROP_OLDEST 544
 in ViewModel 548
 repeatOnLifecycle 552
 SUSPEND 544
 tutorial 547
 shareIn() function 545
 SharingStarted.Eagerly() 545
 SharingStarted.Lazily() 545
 SharingStarted.WhileSubscribed() 545
 showSystemUi 23, 340
 shrinkHorizontally() 389
 shrinkOut() 389
 shrinkVertically() 389
 SideEffect 326
 Side Effects 326
 single() operator 537
 singleton 563
 size modifier 222
 slideIn() 389
 slideInHorizontally() 389
 slideInVertically() 389
 slideOut() 389
 slideOutHorizontally() 389
 slideOutVertically() 389
 Slider 176
 Slider component 31
 Slot APIs
 calling 200
 declaring 200
 overview 199
 tutorial 203
 Snackbar 176

Index

Snapshots
 emulator 55

SpanStyle 225

Spread chain 288

Spread inside chain 288

Spring effects 406

spring() function 406

SQL 450

SQLite 449

 AVD command-line use 451

Columns and Data Types 449

 overview 450

Primary keys 450

Staggered Grids 365

startConnection() method 617

start destination 487

state 162

 basics of 179

 by keyword 182

 configuration changes 188

 declaring 180

 hoisting 185

 MutableState 180

 mutableStateOf() function 181

 overview 179

 remember keyword 181

 rememberSaveable 189

 Unidirectional data flow 183

StateFlow 542

stateful 179

stateful composables 171

State hoisting 185

stateless composables 171

Statement Completion 86

staticCompositionLocalOf() function 194, 195

Status Bar Widgets 83

 Memory Indicator 83

stickyHeader 358

stickyHeader() function 332

Sticky headers

 adding 358

 example 357

stickyHeader() function 332

StiffnessHigh 407

StiffnessLow 407

StiffnessMedium 407

StiffnessMediumLow 407

StiffnessVeryLow 407

String 106

Structure

 tool window 69

Structured Query Language 450

Structure tool window 69

SUBS 620

subscriptions 616

subStringBefore() method 349

supervisorScope 323

Surface component 251

SUSPEND 544

Suspend Functions 322

Switch 176

Switcher 70

system requirements 3

T

Telephony Manager 98

Terminal

 tool window 68

Text 176

Text component 172

TextField 176

TextMeasurer 429

 measure() function 430

TextStyle 447

Theme

 building a custom 640

Theming 637

 tutorial 645

TODO

 tool window 69

Tool window bars 66

Tool windows 66

TopAppBar 176, 512

trailingIcon 447

TransformableState 524
 transform() operator 535
 translation gestures 526
 try/finally 536
 tween() function 390
 Type Annotations 109
 Type Casting 113
 Type Checking 113
 Type Inference 109
 Type.kt file 640

U

UI Controllers 434
 UI_NIGHT_MODE_YES 195
 UiSettings class 583
 Unidirectional data flow 183
 updateAll() 559
 updateTransition() function 398, 403, 408
 upload key 604
 USB connection issues
 resolving 76
 USE_BIOMETRIC permission 577

V

Vector Asset
 add to project 208
 Version catalog 165
 dependencies 167
 libraries 167
 libs.versions.toml file 166
 plugins 167
 versions 167
 verticalArrangement 241, 242
 VerticalPager
 animateScrollToPage() 375
 scrollToPage() 375
 state 374
 syntax , 258
 verticalScroll() 522
 verticalScroll() modifier 341
 ViewModel
 example 441

lifecycle library 436, 440, 530, 547
 LiveData 437
 observeAsState() 437
 overview 433
 tutorial 439
 using state 434
 viewModel() 436, 443, 477
 ViewModelProvider Factory 476
 ViewModelStoreOwner 477
 viewModel() function 436, 443, 477
 ViewModelProvider Factory 476
 ViewModelScope 322
 ViewModelStoreOwner 477
 View System 98
 Virtual Device Configuration dialog 38
 Virtual Sensors 55
 Visibility animation 385

W

Weighted chain 288
 Welcome screen 63
 while Loop 124
 Widget Dimensions 289
 WiFi debugging 77
 Wireless debugging 77
 Wireless pairing 77
 withContext 323

X

XML resource
 reading an 345

Z

zip() operator 541

