Android Studio Ladybug Essentials





Android Studio Ladybug Essentials

Kotlin Edition

Android Studio Ladybug Essentials – Kotlin Edition

ISBN: 978-1-965764-05-3

© 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. Introduction	1
1.1 Downloading the Code Samples	1
1.2 Feedback	1
1.3 Errata	2
2. Setting up an Android Studio Development Environment	
2.1 System requirements	3
2.2 Downloading the Android Studio package	3
2.2 Downloading the Android Studio	
2.3 1 Installation on Windows	4
2.3.2 Installation on macOS	4
2.3.2 Installation on Linux	5
2.4 Installing additional Android SDK packages	
2.5 Installing the Android SDK Command-line Tools	
2.5.1 Windows 8.1	9
2.5.2 Windows 10	10
2.5.3 Windows 11	10
2.5.4 Linux	10
2.5.5 macOS	10
2.6 Android Studio memory management	10
2.7 Updating Android Studio and the SDK	11
2.8 Summary	12
3. Creating an Example Android App in Android Studio	
3.1 About the Project	13
3.2 Creating a New Android Project	13
3.3 Creating an Activity	14
3.4 Defining the Project and SDK Settings	14
3.5 Modifying the Example Application	15
3.6 Modifying the User Interface	16
3.7 Reviewing the Layout and Resource Files	22
3.8 Adding Interaction	25
3.9 Summary	26
4. Creating an Android Virtual Device (AVD) in Android Studio	
4.1 About Android Virtual Devices	27
4.2 Starting the Emulator	29
4.3 Running the Application in the AVD	30
4.4 Running on Multiple Devices	31
4.5 Stopping a Running Application	32
4.6 Running the Emulator in a Separate Window	32
4.7 Removing the Device Frame	35
4.8 Summary	37
5. Using and Configuring the Android Studio AVD Emulator	

5.1 The Emulator Environment	
5.2 Emulator Toolbar Options	
5.3 Working in Zoom Mode	41
5.4 Resizing the Emulator Window	
5.5 Extended Control Options	
5.5.1 Location	
5.5.2 Displays	
5.5.3 Cellular	
5.5.4 Battery	
5.5.5 Camera	
5.5.6 Phone	
5.5.7 Directional Pad	
5.5.8 Microphone	
5.5.9 Fingerprint	
5.5.10 Virtual Sensors	
5.5.11 Snapshots	
5.5.12 Record and Playback	
5.5.13 Google Play	
5.5.14 Settings	
5.5.15 Help	
5.6 Working with Snapshots	
5.7 Configuring Fingerprint Emulation	
5.8 The Emulator in Tool Window Mode	
5.9 Common Android Settings	
5.10 Creating a Resizable Emulator	47
5.11 Summary	
6. A Tour of the Android Studio User Interface	
6.1 The Welcome Screen	
6.2 The Menu Bar	
6.3 The Main Window	
6.4 The Tool Windows	
6.5 The Tool Window Menus	55
6.6 Android Studio Keyboard Shortcuts	
6.7 Switcher and Recent Files Navigation	
6.8 Changing the Android Studio Theme	57
6.9 Summary	58
7. Testing Android Studio Apps on a Physical Android Device	
7 1 An Overview of the Android Debug Bridge (ADB)	59
7.2 Fnabling USB Debugging ADB on Android Devices	59
7.2.1 macOS ADB Configuration	60
7.2.2 Windows ADB Configuration	61
7.2.2 Vindows TDD Configuration	62
7 3 Resolving USB Connection Issues	62
7.4 Enabling Wireless Debugging on Android Devices	
7.5 Testing the adb Connection	
7.6 Device Mirroring	
7.7 Summary	
8. The Basics of the Android Studio Code Editor	

8.1 The Android Studio Editor	67
8.2 Splitting the Editor Window	
8.3 Code Completion	
8.4 Statement Completion	
8.5 Parameter Information	
8.6 Parameter Name Hints	
8.7 Code Generation	
8.8 Code Folding	
8.9 Quick Documentation Lookup	
8.10 Code Reformatting	
8.11 Finding Sample Code	
8.12 Live Templates	
8 13 Summary	77
9. An Overview of the Android Architecture	
	70
9.1 The Android Software Stack	
9.2 The Linux Kernel	
9.3 Hardware Abstraction Layer	
9.4 Android Runtime – ART	
9.5 Android Libraries	
9.5.1 C/C++ Libraries	
9.6 Application Framework	
9.7 Applications	
9.8 Summary	
10. The Anatomy of an Android App	
, 11	
10.1 Android Activities	
10.1 Android Activities 10.2 Android Fragments	
10.1 Android Activities 10.2 Android Fragments 10.3 Android Intents	
10.1 Android Activities 10.2 Android Fragments 10.3 Android Intents 10.4 Broadcast Intents	
10.1 Android Activities 10.2 Android Fragments 10.3 Android Intents 10.4 Broadcast Intents 10.5 Broadcast Receivers	
10.1 Android Activities 10.2 Android Fragments 10.3 Android Intents 10.4 Broadcast Intents 10.5 Broadcast Receivers 10.6 Android Services	
10.1 Android Activities 10.2 Android Fragments 10.3 Android Intents 10.4 Broadcast Intents 10.5 Broadcast Receivers 10.6 Android Services 10.7 Content Providers	
10.1 Android Activities. 10.2 Android Fragments. 10.3 Android Intents. 10.4 Broadcast Intents. 10.5 Broadcast Receivers . 10.6 Android Services. 10.7 Content Providers . 10 8 The Application Manifest	
10.1 Android Activities. 10.2 Android Fragments. 10.3 Android Intents 10.4 Broadcast Intents. 10.5 Broadcast Receivers 10.6 Android Services. 10.7 Content Providers 10.8 The Application Manifest. 10 9 Application Resources	
10.1 Android Activities. 10.2 Android Fragments. 10.3 Android Intents 10.4 Broadcast Intents. 10.5 Broadcast Receivers 10.6 Android Services. 10.7 Content Providers 10.8 The Application Manifest. 10.9 Application Resources 10 10 Application Context	
10.1 Android Activities.10.2 Android Fragments.10.3 Android Intents10.4 Broadcast Intents.10.5 Broadcast Receivers10.6 Android Services.10.7 Content Providers.10.8 The Application Manifest.10.9 Application Resources10.10 Application Context.10 11 Summary	
10.1 Android Activities. 10.2 Android Fragments. 10.3 Android Intents 10.4 Broadcast Intents. 10.5 Broadcast Receivers 10.6 Android Services. 10.7 Content Providers 10.8 The Application Manifest. 10.9 Application Resources 10.10 Application Context. 10.11 Summary.	
10.1 Android Activities. 10.2 Android Fragments. 10.3 Android Intents 10.4 Broadcast Intents. 10.5 Broadcast Receivers 10.6 Android Services. 10.7 Content Providers 10.8 The Application Manifest. 10.9 Application Context. 10.10 Application Context. 10.11 Summary.	
10.1 Android Activities. 10.2 Android Fragments. 10.3 Android Intents 10.4 Broadcast Intents. 10.5 Broadcast Receivers 10.6 Android Services. 10.7 Content Providers 10.8 The Application Manifest. 10.9 Application Resources 10.10 Application Context. 10.11 Summary. 11. An Introduction to Kotlin	
10.1 Android Activities. 10.2 Android Fragments. 10.3 Android Intents 10.4 Broadcast Intents. 10.5 Broadcast Receivers 10.6 Android Services. 10.7 Content Providers. 10.8 The Application Manifest. 10.9 Application Resources. 10.10 Application Context. 10.11 Summary. 11. An Introduction to Kotlin. 11.1 What is Kotlin? 11.2 Kotlin and Java.	
10.1 Android Activities.10.2 Android Fragments.10.3 Android Intents10.4 Broadcast Intents10.5 Broadcast Receivers10.6 Android Services10.7 Content Providers10.8 The Application Manifest10.9 Application Resources10.10 Application Context10.11 Summary11. An Introduction to Kotlin11.2 Kotlin and Java11.3 Converting from Java to Kotlin	
10.1 Android Activities.10.2 Android Fragments.10.3 Android Intents10.4 Broadcast Intents.10.5 Broadcast Receivers10.6 Android Services10.7 Content Providers10.8 The Application Manifest.10.9 Application Resources10.10 Application Context.10.11 Summary.11. An Introduction to Kotlin11.2 Kotlin and Java.11.4 Kotlin and Android Studio	
10.1 Android Activities.10.2 Android Fragments.10.3 Android Intents10.4 Broadcast Intents.10.5 Broadcast Receivers10.6 Android Services.10.7 Content Providers10.8 The Application Manifest.10.9 Application Resources10.10 Application Context.10.11 Summary.11. An Introduction to Kotlin11.2 Kotlin and Java.11.3 Converting from Java to Kotlin11.4 Kotlin and Android Studio11.5 Experimenting with Kotlin	83 83 84 84 84 84 84 85 85 85 85 85 85 85 85 85 85 85 85 85
10.1 Android Activities10.2 Android Fragments10.3 Android Intents10.3 Android Intents10.4 Broadcast Intents10.5 Broadcast Receivers10.6 Android Services10.7 Content Providers10.8 The Application Manifest10.9 Application Resources10.10 Application Context10.11 Summary11. An Introduction to Kotlin11.2 Kotlin and Java11.3 Converting from Java to Kotlin11.4 Kotlin and Android Studio11.5 Experimenting with Kotlin11.6 Semi-colons in Kotlin	83 83 84 84 84 84 84 85 85 85 85 85 85 85 85 85 85 87 87 87 87 87 87 87 88 88 88 88 88 88
10.1 Android Activities10.2 Android Fragments10.3 Android Intents10.4 Broadcast Intents10.5 Broadcast Receivers10.6 Android Services10.7 Content Providers10.8 The Application Manifest10.9 Application Resources10.10 Application Context10.11 Summary11. An Introduction to Kotlin11.2 Kotlin and Java11.3 Converting from Java to Kotlin11.4 Kotlin and Android Studio11.5 Experimenting with Kotlin11.7 Summary	83 83 84 84 84 84 84 85 85 85 85 85 85 85 85 85 87 87 87 87 87 87 87 87 88 88 88 88 88
10.1 Android Activities 10.2 Android Fragments 10.3 Android Intents 10.4 Broadcast Intents 10.5 Broadcast Receivers 10.6 Android Services 10.7 Content Providers 10.8 The Application Manifest 10.9 Application Resources 10.10 Application Context 10.11 Summary 11. An Introduction to Kotlin 11.2 Kotlin and Java 11.3 Converting from Java to Kotlin 11.4 Kotlin and Android Studio 11.5 Experimenting with Kotlin 11.6 Semi-colons in Kotlin 11.7 Summary	83 83 84 84 84 84 84 85 85 85 85 85 85 85 85 87 87 87 87 87 87 87 87 87 87 87 87 87
10.1 Android Activities 10.2 Android Fragments 10.3 Android Intents 10.4 Broadcast Intents 10.5 Broadcast Receivers 10.6 Android Services 10.7 Content Providers 10.8 The Application Manifest 10.9 Application Resources 10.10 Application Context 10.11 Summary 11. An Introduction to Kotlin 11.2 Kotlin and Java 11.3 Converting from Java to Kotlin 11.4 Kotlin and Android Studio 11.5 Experimenting with Kotlin 11.6 Semi-colons in Kotlin 11.7 Summary 12. Kotlin Data Types, Variables, and Nullability	83 83 84 84 84 84 84 85 85 85 85 85 85 85 87 87 87 87 87 87 87 87 87 87 87 87 87
10.1 Android Activities 10.2 Android Fragments 10.3 Android Intents 10.4 Broadcast Intents 10.5 Broadcast Receivers 10.6 Android Services 10.7 Content Providers 10.8 The Application Manifest 10.9 Application Resources 10.10 Application Context 10.11 Summary 11. An Introduction to Kotlin 11.3 Converting from Java to Kotlin 11.4 Kotlin and Java. 11.5 Experimenting with Kotlin 11.6 Semi-colons in Kotlin 11.7 Summary 12. Kotlin Data Types, Variables, and Nullability 12.1 Kotlin Data Types	83 83 84 84 84 84 84 85 85 85 85 85 85 85 85 87 87 87 87 87 87 87 87 87 87 87 87 87

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

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

	19.2 Enabling Gemini in Android Studio	145	
	19.3 Gemini configuration	147	
	19.4 Asking Gemini questions	148	
	19.5 Question contexts	149	
	19.6 Inline code completion	149	
	19.7 Transforming and documenting code	150	
	19.8 Summary	152	
20.	Understanding Android Application and Activity Lifecycles		155
	20.1 Andraid Applications and Pasourca Management	155	
	20.1 Android Process States	155	
	20.2 1 Enground Process	155	
	20.2.2 Visible Process	150	
	20.2.2 VISIOLE TOCCSS	150	
	20.2.4 Background Process	150	
	20.2.5 Empty Process	150	
	20.2.5 Empty 110ccss	157	
	20.4 The Activity Lifecucle	137 157	
	20.5 The Activity Stack	137 157	
	20.5 The Activity States	137	
	20.0 A Civity States	150	
	20.7 Configuration Changes	156	
	20.8 Fundating State Change	159	
	20.9 Summary	139	
21.	Handling Android Activity State Changes		161
	21.1 New vs. Old Lifecycle Techniques	161	
	21.2 The Activity and Fragment Classes	161	
	21.3 Dynamic State vs. Persistent State	163	
	21.4 The Android Lifecycle Methods	163	
	21.5 Lifetimes	165	
	21.6 Foldable Devices and Multi-Resume	166	
	21.7 Disabling Configuration Change Restarts	166	
	21.8 Lifecycle Method Limitations	166	
	21.9 Summary	167	
22.	Android Activity State Changes by Example		169
	22.1 Creating the State Change Example Project	169	
	22.2 Designing the User Interface	170	
	22.3 Overriding the Activity Lifecycle Methods	171	
	22.4 Filtering the Logcat Panel	173	
	22.5 Running the Application	174	
	22.6 Experimenting with the Activity	175	
	22.7 Summary	176	
23.	Saving and Restoring the State of an Android Activity		177
	23.1 Saving Dynamic State	177	
	23.2 Default Saving of User Interface State	177	
	23.3 The Bundle Class	178	
	23.4 Saving the State	179	
	23.5 Restoring the State	180	
	23.6 Testing the Application	180	

23.7 Summary		
24. Understanding Android Views, View Groups and Layouts	•••••	181
24.1 Designing for Different Android Devices		
24.2 Views and View Groups		
24.3 Android Layout Managers		
24.4 The View Hierarchy		
24.5 Creating User Interfaces	184	
24.6 Summary		
25. A Guide to the Android Studio Layout Editor Tool		185
25.1 Basic vs. Empty Views Activity Templates		
25.2 The Android Studio Layout Editor		
25.3 Design Mode		
25.4 The Palette	190	
25.5 Design Mode and Layout Views		
25.6 Night Mode		
25.7 Code Mode	192	
25.8 Split Mode		
25.9 Setting Attributes		
25.10 Transforms		
25.11 Tools Visibility Toggles		
25.12 Converting Views		
25.13 Displaying Sample Data		
25.14 Creating a Custom Device Definition		
25.15 Changing the Current Device		
25.16 Lavout Validation		
25.17 Summary		
26. A Guide to the Android ConstraintLayout	••••••	203
26.1 How ConstraintLayout Works		
26.1.1 Constraints	203	
2612 Margins	204	
26.1.3 Opposing Constraints	204	
26.1.4 Constraint Bias	205	
2615 Chains	206	
2616 Chain Styles	206	
26.2 Baseline Alignment	207	
26.3 Configuring Widget Dimensions	207	
264 Guideline Helper	208	
26 5 Group Helper	208	
266 Barrier Helper	208	
267 Flow Helper	210	
26.8 Ratios		
26.9 ConstraintLavout Advantages	211	
26.10 ConstraintLayout Availability	211	
26 11 Summary	212	
27. A Guide to Using ConstraintLayout in Android Studio		213
27.1 Design and Levent Views	212	
27.2 Autocompact Made		
27.2 Autoconnect mode		

27.3 Inference Mode	
27.4 Manipulating Constraints Manually	
27.5 Adding Constraints in the Inspector	
27.6 Viewing Constraints in the Attributes Window	
27.7 Deleting Constraints	
27.8 Adjusting Constraint Bias	
27.9 Understanding ConstraintLayout Margins	
27.10 The Importance of Opposing Constraints and Bias	
27.11 Configuring Widget Dimensions	
27.12 Design Time Tools Positioning	
27.13 Adding Guidelines	
27.14 Adding Barriers	
27.15 Adding a Group	
27.16 Working with the Flow Helper	
27.17 Widget Group Alignment and Distribution	
27.18 Converting other Layouts to ConstraintLayout	
27.19 Summary	
28 Working with Constraint avout Chains and Datios in Andraid St	udio 222
28. Working with ConstraintLayout Chains and Ratios in Android St	uulo 255
28.1 Creating a Chain	
28.2 Changing the Chain Style	
28.3 Spread Inside Chain Style	
28.4 Packed Chain Style	
28.5 Packed Chain Style with Bias	
28.6 Weighted Chain	
28.7 Working with Ratios	
28.8 Summary	
29. An Android Studio Layout Editor ConstraintLayout Tutorial	
29.1 An Android Studio Layout Editor Tool Example	
29.2 Preparing the Layout Editor Environment	241
29.3 Adding the Widgets to the User Interface.	242
29.4 Adding the Constraints	245
29.5 Testing the Layout	247
29.6 Using the Layout Inspector	
29.7 Summary	
30 Manual XMI I avout Design in Android Studio	249
	240
30.1 Manually Creating an XML Layout	
30.2 Manual XML vs. Visual Layout Design	
30.3 Summary	
31. Managing Constraints using Constraint Sets	
31.1 Kotlin Code vs. XML Layout Files	
31.2 Creating Views	
31.3 View Attributes	
31.4 Constraint Sets	
31.4.1 Establishing Connections	
31.4.2 Applying Constraints to a Layout	
31.4.3 Parent Constraint Connections	
31.4.4 Sizing Constraints	
5	

31.4.5 Constraint Bias	
31.4.6 Alignment Constraints	
31.4.7 Copying and Applying Constraint Sets	
31.4.8 ConstraintLayout Chains	
31.4.9 Guidelines	
31.4.10 Removing Constraints	
31.4.11 Scaling	
31.4.12 Rotation	
31.5 Summary	
32. An Android ConstraintSet Tutorial	
32.1 Creating the Example Project in Android Studio	
32.2 Adding Views to an Activity	
32.3 Setting View Attributes	260
32.4 Creating View IDs	
32.5 Configuring the Constraint Set	262
32.6 Adding the EditText View	263
32.7 Converting Density Independent Pixels (dn) to Pixels (nx)	264
32.8 Summary	265
33. A Guide to Using Apply Changes in Android Studio	
22.1 Introducing Apply Changes	267
22.2 Linderstanding Apply Changes Ontions	
22.2 Ling Apply Changes Options	
22.4 Configuring Apply Changes Fallback Settings	
22.5 A. Austra Charges Tattarial	
22 C Ling Australia Charges Tutorial	
33.6 Using Apply Code Changes	
33.7 Using Apply Changes and Restart Activity	
33.8 Using Run App	
33.9 Summary	
34. A Guide to Gradle Version Catalogs	
34.1 Library and Plugin Dependencies	
34.2 Project Gradle Build File	
34.3 Module Gradle Build Files	
34.4 Version Catalog File	
34.5 Adding Dependencies	
34.6 Library Updates	
34.7 Summary	
35. An Overview and Example of Android Event Handling	
35.1 Understanding Android Events	
35.2 Using the android:onClick Resource	
35.3 Event Listeners and Callback Methods	
35.4 An Event Handling Example	
35.5 Designing the User Interface	2.77
35.6 The Event Listener and Callback Method	277
35.7 Consuming Events	279
35.8 Summary	
36. Android Touch and Multi-touch Event Handling	

36.1 Intercepting Touch Events	
36.2 The MotionEvent Object	
36.3 Understanding Touch Actions	
36.4 Handling Multiple Touches	
36.5 An Example Multi-Touch Application	
36.6 Designing the Activity User Interface	
36.7 Implementing the Touch Event Listener	
36.8 Running the Example Application	
36.9 Summary	
37. Detecting Common Gestures Using the Android Gesture Detector Class	
37.1 Implementing Common Gesture Detection	
37.2 Creating an Example Gesture Detection Project	
37.3 Implementing the Listener Class	
37.4 Creating the GestureDetector Instance	
37.5 Implementing the onTouchEvent() Method	
37.6 Testing the Application	
37.7 Summary	
38. Implementing Custom Gesture and Pinch Recognition on Android	
38.1 The Android Gesture Builder Application	
38.2 The GestureOverlayView Class	
38.3 Detecting Gestures.	
38.4 Identifying Specific Gestures	
38.5 Installing and Running the Gesture Builder Application	
38.6 Creating a Gestures File	
38.7 Creating the Example Project	
38.8 Extracting the Gestures File from the SD Card	
38.9 Adding the Gestures File to the Project	
38.10 Designing the User Interface	
38.11 Loading the Gestures File	
38.12 Registering the Event Listener	
38.13 Implementing the onGesturePerformed Method	
38.14 Testing the Application	
38.15 Configuring the GestureOverlavView	
38.16 Intercepting Gestures	
38.17 Detecting Pinch Gestures	
38.18 A Pinch Gesture Example Project	
38.19 Summary	
39. An Introduction to Android Fragments	
39.1 What is a Fragment?	
39.2 Creating a Fragment	
39.3 Adding a Fragment to an Activity using the Layout XML File	
39.4 Adding and Managing Fragments in Code	
39 5 Handling Fragment Events	307
39.6 Implementing Fragment Communication	
39.7 Summary	309
40. Using Fragments in Android Studio - An Example	
40.1 About the Example Fragment Application	311

40.2 Creating the Example Project	
40.3 Creating the First Fragment Layout	
40.4 Migrating a Fragment to View Binding	
40.5 Adding the Second Fragment	
40.6 Adding the Fragments to the Activity	
40.7 Making the Toolbar Fragment Talk to the Activity	
40.8 Making the Activity Talk to the Text Fragment	
40.9 Testing the Application	
40.10 Summary	
41. Modern Android App Architecture with Jetpack	
41.1 What is Android Jetpack?	
41.2 The "Old" Architecture	
41.3 Modern Android Architecture	
41.4 The ViewModel Component	
41.5 The LiveData Component	
41.6 ViewModel Saved State	
41.7 LiveData and Data Binding	
41.8 Android Lifecycles	
41.9 Repository Modules	
41.10 Summary	
42. An Android ViewModel Tutorial	
42.1 About the Project	
42.2 Creating the ViewModel Example Project	
42.3 Removing Unwanted Project Elements	
42.4 Designing the Fragment Layout	
42.5 Implementing the View Model	
42.6 Associating the Fragment with the View Model	
42.7 Modifying the Fragment	
42.8 Accessing the ViewModel Data	
42.9 Testing the Project	
42.10 Summary	
43. An Android Jetpack LiveData Tutorial	
43.1 LiveData - A Recap	
43.2 Adding LiveData to the ViewModel	
43.3 Implementing the Observer	
43.4 Summary	
44. An Overview of Android Jetpack Data Binding	
44.1 An Overview of Data Binding	
44.2 The Key Components of Data Binding	
44.2.1 The Project Build Configuration	
44.2.2 The Data Binding Layout File	
44.2.3 The Layout File Data Element	
44.2.4 The Binding Classes	
44.2.5 Data Binding Variable Configuration	
44.2.6 Binding Expressions (One-Way)	
44.2.7 Binding Expressions (Two-Way)	
44.2.8 Event and Listener Bindings	
0	

44.3 Summary	
45. An Android Jetpack Data Binding Tutorial	
45.1 Removing the Redundant Code	
45.2 Enabling Data Binding	
45.3 Adding the Lavout Element	
45.4 Adding the Data Element to Layout File	
45.5 Working with the Binding Class	
45.6 Assigning the ViewModel Instance to the Data Binding Variable	
45.7 Adding Binding Expressions	
45.8 Adding the Conversion Method	
45.9 Adding a Listener Binding	
45.10 Testing the App	
45.11 Summary	
46. An Android ViewModel Saved State Tutorial	
46.1 Understanding ViewModel State Saving	
46.2 Implementing ViewModel State Saving	
46.3 Saving and Restoring State	
46.4 Adding Saved State Support to the ViewModelDemo Project	
46.5 Summary	
47. Working with Android Lifecycle-Aware Components	
47.1 Lifecycle Awareness	357
47.1 Litecycle Awareness	357
47.3 Lifecycle Observers	358
47.5 Lifecycle Observers	358
47.5 Summary	359
48. An Android Jetpack Lifecycle Awareness Tutorial	
48.1 Creating the Example Lifecycle Project	361
48.2 Creating a Lifecycle Observer	361
48.2 Adding the Observer	362
48.4 Testing the Observer	363
48.5 Creating a Lifecycle Owner	363
48.5 Creating a Energy Cowner.	365
48.0 Testing the Custom Energie Owner	365
49 An Overview of the Navigation Architecture Component	367
40.1 Understanding Navigation	267
49.1 Understanding Navigation	
49.2 Declaring a Navigation Host.	
49.5 The Navigation Graph	
49.4 Accessing the Navigation Controller	
49.5 Inggering a Navigation Action	
49.0 Passing Arguments	
49.7 Summary	
50. 1. Creating the Numberti D. D. in t	
50.1 Creating the Navigation Demo Project	
50.2 Adding Inavigation to the Build Configuration	
50.5 Creating the Ivavigation Graph Resource File	
JU.4 Deciating a Navigation Host	

50.5 Adding Navigation Destinations	
50.6 Designing the Destination Fragment Layouts	
50.7 Adding an Action to the Navigation Graph	
50.8 Implement the OnFragmentInteractionListener	
50.9 Adding View Binding Support to the Destination Fragments	
50.10 Triggering the Action	
50.11 Passing Data Using Safeargs	
50.12 Summary	
51. An Introduction to MotionLayout	
51.1 An Overview of MotionLayout	
51.2 MotionLayout	
51.3 MotionScene	
51.4 Configuring ConstraintSets	
51.5 Custom Attributes	
51.6 Triggering an Animation	
51.7 Arc Motion	
51.8 Keyframes	
51.8.1 Attribute Keyframes	
51.8.2 Position Keyframes	
51.9 Time Linearity	
51.10 KevTrigger	
51.11 Cycle and Time Cycle Keyframes	
51.12 Starting an Animation from Code	
51.13 Summary	
52. An Android MotionLayout Editor Tutorial	
52. An Android MotionLayout Editor Tutorial	
 52. An Android MotionLayout Editor Tutorial. 52.1 Creating the MotionLayoutDemo Project	
 52. An Android MotionLayout Editor Tutorial	
 52. An Android MotionLayout Editor Tutorial	
 52. An Android MotionLayout Editor Tutorial	
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 404 404
 52. An Android MotionLayout Editor Tutorial	399 399 401 404 404 404 404 406 408
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 404 404 406 408 410
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 404 404 406 408 410 412
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 404 404 406 408 408 410 412 413
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 404 406 408 410 412 413
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 406 408 410 412 413 417
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 404 406 408 410 412 413 413 414
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 404 404 404 406 408 410 412 413 413 414 418
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 404 404 404 406 408 410 412 413 413 413 413 413 413 413 413 413 413 413 413 413
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 404 404 404 405 406 408 410 412 413 413 413 413 414 418 420 422
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 406 408 410 412 413 413 418 418 422 422
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 401 404 404 406 408 410 412 413 413 413 413 413 413 413 413 413 413
 52. An Android MotionLayout Editor Tutorial	399 399 399 401 404 404 404 404 404 406 408 410 412 413 413 413 413 414 413 414 413 414 413 414 413 414 414 415 416 417 418 420 422 424 424
 52. An Android MotionLayout Editor Tutorial	$\begin{array}{c} 399\\ 399\\ 399\\ 399\\ 401\\ 401\\ 404\\ 404\\ 404\\ 406\\ 408\\ 410\\ 412\\ 413\\ 417\\ 418\\ 413\\ 417\\ 418\\ 418\\ 420\\ 422\\ 422\\ 422\\ 424\\ 424\\ 425\\ 425\\ 425$
 52. An Android MotionLayout Editor Tutorial	$\begin{array}{c} 399\\ 399\\ 399\\ 399\\ 401\\ 404\\ 404\\ 404\\ 406\\ 408\\ 410\\ 412\\ 413\\ 413\\ 417\\ 418\\ 413\\ 417\\ 418\\ 420\\ 422\\ 422\\ 422\\ 422\\ 422\\ 422\\ 425\\ 425$
 52. An Android MotionLayout Editor Tutorial	$\begin{array}{c} 399\\ 399\\ 399\\ 399\\ 401\\ 404\\ 404\\ 404\\ 406\\ 408\\ 410\\ 412\\ 413\\ 413\\ 413\\ 413\\ 413\\ 413\\ 413\\ 413$

54.4 The Snackbar	
54.5 Creating the Example Project	
54.6 Reviewing the Project	
54.7 Removing Navigation Features	
54.8 Changing the Floating Action Button	
54.9 Adding an Action to the Snackbar	
54.10 Summary	
55. Creating a Tabbed Interface using the TabLayout Component	
55.1 An Introduction to the ViewPager?	/21
55.1 An Introduction to the View Page 2	
55.2 All Overview of the TabLayout Component	431
55.5 Creating the First Fragment	
55.4 Creating the First Fragments	
55.5 Duplicating the Flagments	
55.6 Adding the TabLayout and ViewPager2	
55.7 Creating the Pager Adapter	
55.8 Performing the Initialization Tasks	
55.9 Testing the Application	
55.10 Customizing the TabLayout	
55.11 Summary	
56. Working with the RecyclerView and CardView Widgets	
56.1 An Overview of the RecyclerView	
56.2 An Overview of the CardView	
56.3 Summary	
57. An Android RecyclerView and CardView Tutorial	
57. An Android RecyclerView and CardView Tutorial	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project 57.2 Modifying the Basic Views Activity Project 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the BecyclerView 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView. 57.5 Adding the Image Files 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView. 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView. 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView 57.5 Adding the Image Files 57.6 Creating the RecyclerView Adapter 57.7 Initializing the RecyclerView Component 57.8 Testing the Application 57.9 Responding to Card Selections 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application. 57.9 Responding to Card Selections. 57 10 Summary 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView Layout 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application. 57.9 Responding to Card Selections. 57.10 Summary. 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application. 57.9 Responding to Card Selections. 57.10 Summary. 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView 57.5 Adding the Image Files 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component 57.8 Testing the Application. 57.9 Responding to Card Selections. 57.10 Summary. 58. Working with the AppBar and Collapsing Toolbar Layouts	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application. 57.9 Responding to Card Selections. 57.10 Summary 58. Working with the AppBar and Collapsing Toolbar Layouts 58.1 The Anatomy of an AppBar 58.2 The Example Project. 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView Layout 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application. 57.9 Responding to Card Selections. 57.10 Summary. 58. Working with the AppBar and Collapsing Toolbar Layouts 58.1 The Anatomy of an AppBar 58.2 The Example Project. 58.3 Coordinating the RecyclerView and Toolbar 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView Layout 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application. 57.9 Responding to Card Selections. 57.10 Summary. 58. Working with the AppBar and Collapsing Toolbar Layouts 58.1 The Anatomy of an AppBar 58.2 The Example Project 58.3 Coordinating the RecyclerView and Toolbar 58.4 Introducing the Collapsing Toolbar Layout 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application. 57.9 Responding to Card Selections. 57.10 Summary. 58. Working with the AppBar and Collapsing Toolbar Layouts 58.1 The Anatomy of an AppBar 58.2 The Example Project 58.3 Coordinating the RecyclerView and Toolbar 58.4 Introducing the Collapsing Toolbar Layout 58.5 Changing the Title and Scrim Color 	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application. 57.9 Responding to Card Selections. 57.10 Summary. 58. Working with the AppBar and Collapsing Toolbar Layouts 58.1 The Anatomy of an AppBar 58.2 The Example Project 58.3 Coordinating the RecyclerView and Toolbar 58.4 Introducing the Collapsing Toolbar Layout 58.5 Changing the Title and Scrim Color 58.6 Summary 	445 445 445 445 446 447 447 447 447 447 447 447 449 450 450 451 452 453 453 454 454 454 456 459 460
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView Layout 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application. 57.9 Responding to Card Selections. 57.10 Summary. 58. Working with the AppBar and Collapsing Toolbar Layouts 58.1 The Anatomy of an AppBar 58.2 The Example Project 58.3 Coordinating the RecyclerView and Toolbar. 58.4 Introducing the Collapsing Toolbar Layout 58.5 Changing the Title and Scrim Color 58.6 Summary. 	
 57. An Android RecyclerView and CardView Tutorial	445 445 445 445 446 447 447 447 447 447 447 447 447 447 447 447 447 447 447 450 451 452 453 453 454 454 456 459 460 461
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project. 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView. 57.5 Adding the Image Files. 57.6 Creating the RecyclerView Adapter. 57.7 Initializing the RecyclerView Component. 57.8 Testing the Application. 57.9 Responding to Card Selections. 57.10 Summary. 58. Working with the AppBar and Collapsing Toolbar Layouts 58.1 The Anatomy of an AppBar 58.2 The Example Project 58.3 Coordinating the RecyclerView and Toolbar 58.4 Introducing the Collapsing Toolbar Layout 58.5 Changing the Title and Scrim Color 58.6 Summary. 59. An Overview of Intents	
 57. An Android RecyclerView and CardView Tutorial 57.1 Creating the CardDemo Project 57.2 Modifying the Basic Views Activity Project 57.3 Designing the CardView Layout 57.4 Adding the RecyclerView 57.5 Adding the Image Files 57.6 Creating the RecyclerView Adapter 57.7 Initializing the RecyclerView Component 57.8 Testing the Application 57.9 Responding to Card Selections 57.10 Summary 58. Working with the AppBar and Collapsing Toolbar Layouts 58.1 The Anatomy of an AppBar 58.2 The Example Project 58.3 Coordinating the RecyclerView and Toolbar 58.4 Introducing the Collapsing Toolbar Layout 58.5 Changing the Title and Scrim Color 58.6 Summary 59. An Overview of Android Intents 59.1 An Overview of Intents 59.3 Returning Data from an Activity 	445 445 445 445 446 447 450 451 452 453 454 454 456 459 460 461 461 461 462

59.5 Using Intent Filters	
59.6 Automatic Link Verification	
59.7 Manually Enabling Links	
59.8 Checking Intent Availability	
59.9 Summary	
60. Android Explicit Intents – A Worked Example	
60.1 Creating the Explicit Intent Example Application	
60.2 Designing the User Interface Layout for MainActivity	
60.3 Creating the Second Activity Class	
60.4 Designing the User Interface Layout for SecondActivity	
60.5 Reviewing the Application Manifest File	
60.6 Creating the Intent	
60.7 Extracting Intent Data	
60.8 Launching SecondActivity as a Sub-Activity	
60.9 Returning Data from a Sub-Activity	
60.10 Testing the Application	
60.11 Summary	
61. Android Implicit Intents – A Worked Example	
61.1 Creating the Android Studio Implicit Intent Example Project	
61.2 Designing the User Interface	
61.3 Creating the Implicit Intent	
61.4 Adding a Second Matching Activity	
61.5 Adding the Web View to the UI	
61.6 Obtaining the Intent URL	
61.7 Modifying the MyWebView Project Manifest File	
61.8 Installing the MyWebView Package on a Device	
61.9 Testing the Application	
61.10 Manually Enabling the Link	
61.11 Automatic Link Verification	
61.12 Summary	
62. Android Broadcast Intents and Broadcast Receivers	
62.1 An Overview of Broadcast Intents	
62.2 An Overview of Broadcast Receivers	
62.3 Obtaining Results from a Broadcast	
62.4 Sticky Broadcast Intents	
62.5 The Broadcast Intent Example	
62.6 Creating the Example Application	
62.7 Creating and Sending the Broadcast Intent	
62.8 Creating the Broadcast Receiver	
62.9 Registering the Broadcast Receiver	
62.10 Testing the Broadcast Example	
62.11 Listening for System Broadcasts	
62.12 Summary	
63. An Introduction to Kotlin Coroutines	499
63.1 What are Coroutines?	
63.2 Threads vs. Coroutines	
63.3 Coroutine Scope	500

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

	68.2 Creating the Sample Project	531
	68.3 Adding the Kotlin Lifecycle Library	532
	68.4 Declaring a Flow	532
	68.5 Emitting Flow Data	533
	68.6 Collecting Flow Data	533
	68.7 Adding a Flow Buffer	535
	68.8 Transforming Data with Intermediaries	536
	68.9 Terminal Flow Operators	538
	68.10 Flow Flattening	538
	68.11 Combining Multiple Flows	540
	68.12 Hot and Cold Flows	541
	68.13 StateFlow	541
	68.14 SharedFlow	542
	68.15 Summary	544
69.	An Android SharedFlow Tutorial	
	69.1 About the Project	545
	69.2 Creating the Shared Flow Demo Project	545
	69.3 Adding the Lifecycle Libraries	545
	69.4 Designing the User Interface I avout	545 546
	60.5 Adding the List Dow Layout	546 546
	69.6 Adding the DecyclerView Adapter	540 547
	60.7 Adding the ViewModel	547 547
	60.8 Configuring the ViewModelDrovider	347 549
	60.0 Collecting the Flow Values	540 540
	60.10 Testing the Shared Eleve Demo App	550
	60.11 Handling Elows in the Background	550
	69.12 Summary	
70.	An Overview of Android SOLite Databases	
	70.1 Un denoten ding Detabase Tables	FFF
	70.2 Interstanding Database Tables	555
	70.2 Columns on d Data Terrar	555
	70.4 Detahara Data Types	555
	70.5 Lutus du sin a Deire and Karr	556
	70.5 Introducing Primary Keys	556
	70.5 What is SQLifes	556
	70.9 Turciured Query Language (SQL)	556
	70.8 Trying SQLite on an Android Virtual Device (AVD)	557
	70.0.1 C	558
	70.9.1 Cursor	559
	70.9.2 SQLiteDatabase	559
	70.9.3 SQLiteOpenHelper	559
	70.9.4 Content Values.	560
	70.10 The Android Room Persistence Library	560
	70.11 Summary	560
71.	An Android SQLite Database Tutorial	
	/1.1 About the Database Example	561
	71.2 Creating the SQLDemo Project	561
	71.3 Designing the User interface	561
	71.4 Creating the Data Model	562

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

75.2.4 Entities	
75.2.5 SQLite Database	
75.3 Understanding Entities	
75.4 Data Access Objects	
75.5 The Room Database	
75.6 The Repository	
75.7 In-Memory Databases	
75.8 Database Inspector	
75.9 Summary	
76. An Android TableLayout and TableRow Tutorial	599
76.1 The TableLayout and TableRow Layout Views	
76.2 Creating the Room Database Project	
76.3 Converting to a LinearLayout	600
76.4 Adding the TableLayout to the User Interface	601
76.5 Configuring the TableRows	602
76.6 Adding the Button Bar to the Layout	
76.7 Adding the RecyclerView	604
76.8 Adjusting the Layout Margins	605
76.9 Summary	605
77. An Android Room Database and Repository Tutorial	
77.1 About the RoomDemo Project	607
77.2 Modifying the Build Configuration	607
77.3 Building the Entity	608
77.4 Creating the Data Access Object	610
77.5 Adding the Room Database	611
77.6 Adding the Repository	612
77.7 Adding the ViewModel	615
77.8 Creating the Product Item Layout	616
77.9 Adding the RecyclerView Adapter	616
77.10 Preparing the Main Activity	617
77.11 Adding the Button Listeners	618
77.12 Adding LiveData Observers	619
77.13 Initializing the RecyclerView	
77.14 Testing the RoomDemo App	
77.15 Using the Database Inspector	
77.16 Summary	
78. Video Playback on Android using the VideoView and MediaController Class	es 623
78.1 Introducing the Android VideoView Class	
78.2 Introducing the Android MediaController Class	
78.3 Creating the Video Playback Example	
78.4 Designing the VideoPlayer Layout	
78.5 Downloading the Video File	
78.6 Configuring the VideoView	
78.7 Adding the MediaController to the Video View	
78.8 Setting up the onPreparedListener	
78.9 Summary	
79. Android Picture-in-Picture Mode	

79.1 Picture-in-Picture Features	
79.2 Enabling Picture-in-Picture Mode	630
79.3 Configuring Picture-in-Picture Parameters	630
79.4 Entering Picture-in-Picture Mode	631
79.5 Detecting Picture-in-Picture Mode Changes	631
79.6 Adding Picture-in-Picture Actions	631
79.7 Summary	
80. An Android Picture-in-Picture Tutorial	
20.1 Adding Dicture in Dicture Support to the Manifest	622
80.1 Adding a Disture in Disture Support to the Mannest	
80.2 Fataning Disture in Disture Made	
80.4 Detecting Picture in Dicture Mode	
80.4 Detecting Picture-in-Picture Mode Changes	
80.5 Adding a Broadcast Receiver	
80.6 Adding the PiP Action.	
80.7 Testing the Picture-in-Picture Action	
80.8 Summary	
81. Making Runtime Permission Requests in Android	
81.1 Understanding Normal and Dangerous Permissions	641
81.2 Creating the Permissions Example Project	
81.3 Checking for a Permission	
81.4 Requesting Permission at Runtime	
81.5 Providing a Rationale for the Permission Request	
81.6 Testing the Permissions App	
81.7 Summary	
82. Android Audio Recording and Playback using MediaPlayer and MediaReco	order 649
82. Android Audio Recording and Playback using MediaPlayer and MediaReco	order 649
82. Android Audio Recording and Playback using MediaPlayer and MediaReco 82.1 Playing Audio	order 649
82. Android Audio Recording and Playback using MediaPlayer and MediaReco 82.1 Playing Audio	order
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order 649 650 651 651 651 651 651 652 653 654 654
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order 649
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order 649
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order 649 650 651 651 651 652 653 654 654 655 655
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order 649 650 651 651 651 652 653 654 654 655 655 657 657
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order 649 650 651 651 651 651 651 652 653 654 654 655 655 657 657
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio	order 649 650 651 651 651 652 653 654 654 655 655 657 657 659 659
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio 82.2 Recording Audio and Video using the MediaRecorder Class 82.3 About the Example Project 82.4 Creating the AudioApp Project 82.5 Designing the User Interface 82.6 Checking for Microphone Availability 82.7 Initializing the Activity 82.8 Implementing the recordAudio() Method 82.9 Implementing the stopAudio() Method 82.10 Implementing the playAudio() method 82.12 Testing the Application 82.13 Summary 83. An Android Notifications Tutorial 83.1 An Overview of Notifications 	order 649 650 651 651 651 651 651 652 653 654 654 655 655 657 657 659 659
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio 82.2 Recording Audio and Video using the MediaRecorder Class 82.3 About the Example Project 82.4 Creating the AudioApp Project 82.5 Designing the User Interface 82.6 Checking for Microphone Availability 82.7 Initializing the Activity 82.8 Implementing the recordAudio() Method 82.9 Implementing the stopAudio() Method 82.10 Implementing the playAudio() method 82.12 Testing the Application 82.13 Summary 83. An Android Notifications Tutorial 83.1 An Overview of Notifications 83.2 Creating the NotifyDemo Project 	order 649 650 651 651 651 652 653 654 654 655 655 657 657 659 661
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio 82.2 Recording Audio and Video using the MediaRecorder Class 82.3 About the Example Project 82.4 Creating the AudioApp Project 82.5 Designing the User Interface 82.6 Checking for Microphone Availability 82.7 Initializing the Activity 82.8 Implementing the recordAudio() Method 82.9 Implementing the stopAudio() Method 82.10 Implementing the playAudio() method 82.12 Testing the Application 82.13 Summary 83. An Android Notifications Tutorial 83.1 An Overview of Notifications 83.2 Creating the User Interface 	order 649 650 651 651 651 651 651 652 653 654 654 655 655 657 657 659 661
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecor 82.1 Playing Audio 82.2 Recording Audio and Video using the MediaRecorder Class 82.3 About the Example Project 82.4 Creating the AudioApp Project 82.5 Designing the User Interface 82.6 Checking for Microphone Availability 82.7 Initializing the Activity 82.8 Implementing the recordAudio() Method 82.9 Implementing the stopAudio() Method 82.10 Implementing the playAudio() method 82.11 Configuring and Requesting Permissions 82.12 Testing the Application 82.13 Summary 83. An Android Notifications Tutorial 83.1 An Overview of Notifications 83.2 Creating the User Interface 83.4 Creating the Second Activity 	order 649 650 651 651 651 651 651 652 653 654 654 655 655 657 657 659 661 661 661
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio 82.2 Recording Audio and Video using the MediaRecorder Class 82.3 About the Example Project 82.4 Creating the AudioApp Project 82.5 Designing the User Interface 82.6 Checking for Microphone Availability 82.7 Initializing the Activity. 82.8 Implementing the recordAudio() Method. 82.9 Implementing the stopAudio() Method. 82.10 Implementing the playAudio() method. 82.12 Testing the Application. 82.13 Summary. 83. An Android Notifications Tutorial 83.1 An Overview of Notifications 83.2 Creating the NotifyDemo Project. 83.3 Designing the User Interface. 83.4 Creating the Second Activity. 83.5 Creating a Notification Channel 	order 649 650 651 651 651 652 653 654 654 655 655 657 657 659 661 661 661
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio 82.2 Recording Audio and Video using the MediaRecorder Class 82.3 About the Example Project 82.4 Creating the AudioApp Project 82.5 Designing the User Interface 82.6 Checking for Microphone Availability 82.7 Initializing the Activity. 82.8 Implementing the recordAudio() Method. 82.9 Implementing the stopAudio() Method. 82.10 Implementing the playAudio() method. 82.12 Testing the Application. 82.13 Summary. 83. An Android Notifications Tutorial 83.1 An Overview of Notifications. 83.2 Creating the NotifyDemo Project. 83.3 Designing the User Interface 83.4 Creating the Second Activity. 83.5 Creating a Notification Permission 	order 649 650 651 651 651 651 651 652 653 654 654 655 655 657 657 659 661 661 661 662 663
 82. Android Audio Recording and Playback using MediaPlayer and MediaRecords 82.1 Playing Audio 82.2 Recording Audio and Video using the MediaRecorder Class 82.3 About the Example Project 82.4 Creating the AudioApp Project 82.5 Designing the User Interface 82.6 Checking for Microphone Availability 82.7 Initializing the Activity 82.8 Implementing the recordAudio() Method 82.9 Implementing the stopAudio() Method 82.10 Implementing the playAudio() method 82.12 Testing the Application 82.13 Summary 83. An Android Notifications Tutorial 83.1 An Overview of Notifications 83.2 Creating the NotifyDemo Project 83.3 Designing the User Interface 83.4 Creating the Second Activity 83.5 Creating a Notification Channel 83.6 Requesting Notification Permission 83.7 Creating and Issuing a Notification 	order 649 650 651 651 651 651 651 652 653 654 654 655 655 657 657 659 661 661 661 663 666

	83.9 Adding Actions to a Notification	670	
	83.10 Bundled Notifications	670	
	83.11 Summary		
84.	An Android Direct Reply Notification Tutorial		673
0 10	04.1 Creating the Direct Deriver Derivert	(72	
	84.1 Creating the DirectReply Project		
	84.2 Designing the User Interface		
	84.5 Requesting Notification Permission		
	84.4 Creating the Notification Channel.		
	84.5 Building the Remotemput Object.		
	84.6 Creating the Pendingintent.		
	84.7 Creating the Reply Action		
	84.8 Receiving Direct Reply Input		
	84.9 Updating the Notification		
	84.10 Summary		
85.	Working with the Google Maps Android API in Android Studio	•••••	683
	85.1 The Elements of the Google Maps Android API		
	85.2 Creating the Google Maps Project		
	85.3 Creating a Google Cloud Billing Account		
	85.4 Creating a New Google Cloud Project		
	85.5 Enabling the Google Maps SDK		
	85.6 Generating a Google Maps API Key		
	85.7 Adding the API Key to the Android Studio Project		
	85.8 Testing the Application		
	85.9 Understanding Geocoding and Reverse Geocoding		
	85.10 Adding a Map to an Application		
	85.11 Requesting Current Location Permission		
	85.12 Displaying the User's Current Location		
	85.13 Changing the Map Type		
	85.14 Displaying Map Controls to the User		
	85.15 Handling Map Gesture Interaction		
	85.15.1 Map Zooming Gestures		
	85.15.2 Map Scrolling/Panning Gestures		
	85.15.3 Map Tilt Gestures		
	85.15.4 Map Rotation Gestures		
	85.16 Creating Map Markers		
	85.17 Controlling the Map Camera		
	85.18 Summary		
86.	Printing with the Android Printing Framework	•••••	699
	86.1 The Android Printing Architecture	699	
	86.2 The Print Service Plugins		
	86.3 Google Cloud Print		
	86.4 Printing to Google Drive		
	86.5 Save as PDF		
	86.6 Printing from Android Devices		
	86.7 Options for Building Print Support into Android Apps		
	86.7.1 Image Printing		
	86.7.2 Creating and Printing HTML Content		
	86.7.3 Printing a Web Page		
	0		

86.7.4 Printing a Custom Document	
86.8 Summary	
87. An Android HTML and Web Content Printing Example	
87.1 Creating the HTML Printing Example Application	707
87.2 Printing Dynamic HTML Content	
87.3 Creating the Web Page Printing Example	
87.4 Removing the Floating Action Button	
87.5 Removing Navigation Features	
87.6 Designing the User Interface Layout	
87.7 Accessing the WebView from the Main Activity	
87.8 Loading the Web Page into the WebView	
87.9 Adding the Print Menu Option	
87.10 Summary	
88. A Guide to Android Custom Document Printing	
88.1 An Overview of Android Custom Document Printing	
88.1.1 Custom Print Adapters	717
88.2 Preparing the Custom Document Printing Project	718
88.3 Designing the UI	
88.4 Creating the Custom Print Adapter	719
88.5 Implementing the onLayout() Callback Method	
88.6 Implementing the onWrite() Callback Method	
88.7 Checking a Page is in Range	
88.8 Drawing the Content on the Page Canvas	
88.9 Starting the Print Job	
88.10 Testing the Application	
88.11 Summary	
89. An Introduction to Android App Links	
89.1 An Overview of Android App Links	
89.2 App Link Intent Filters	
89.3 Handling App Link Intents	
89.4 Associating the App with a Website	
89.5 Summary	
90. An Android Studio App Links Tutorial	
90.1 About the Example App	
90.2 The Database Schema	
90.3 Loading and Running the Project	
90.4 Adding the URL Mapping	737
90.5 Adding the Intent Filter	740
90.6 Adding Intent Handling Code	740
90.7 Testing the App	
90.8 Creating the Digital Asset Links File	743
90.9 Testing the App Link	744
90.10 Summary	
91. An Android Biometric Authentication Tutorial	
91.1 An Overview of Biometric Authentication	745
91.2 Creating the Biometric Authentication Project	745

91.3 Confi	guring Device Fingerprint Authentication	746
91.4 Addii	ng the Biometric Permission to the Manifest File	746
91.5 Desig	ning the User Interface	747
91.6 Addii	ng a Toast Convenience Method	747
91.7 Chec	king the Security Settings	748
91.8 Confi	guring the Authentication Callbacks	749
91.9 Addii	ng the CancellationSignal	750
91.10 Star	ting the Biometric Prompt	750
91.11 Test	ing the Project	751
91.12 Sum	ımary	752
92. Creating, T	Festing, and Uploading an Android App Bundle	
92.1 The R	Release Preparation Process	753
92.2 Andr	oid App Bundles	753
92.3 Regis	ter for a Google Play Developer Console Account	754
92.4 Confi	guring the App in the Console	755
92.5 Enabl	ling Google Play App Signing	756
92.6 Creat	ing a Keystore File	756
92.7 Creat	ing the Android App Bundle	757
92.8 Gene	rating Test APK Files	759
92.9 Uploa	ading the App Bundle to the Google Play Developer Console	
92.10 Expl	loring the App Bundle	
92.11 Man	laging Testers	
92.12 Roll	ing the App Out for Testing	
92.13 Uplo	oading New App Bundle Revisions	
92.14 Ana	lyzing the App Bundle File	
92.15 Sum	imary	
93. An Overvi	ew of Android In-App Billing	
93.1 Prepa	aring a Project for In-App Purchasing	
93.2 Creat	ing In-App Products and Subscriptions	
93.3 Billin	g Client Initialization	
93.4 Conn	lecting to the Google Play Billing Library	
93.5 Quer	ying Available Products	
93.6 Starti	ng the Purchase Process	770
93.7 Com		
	pleting the Purchase	771
93.8 Quer	pleting the Purchase ying Previous Purchases	771 772
93.8 Quer 93.9 Sumr	pleting the Purchase ying Previous Purchases nary	771 772 772
93.8 Quer 93.9 Sumr 94. An Andro i	pleting the Purchase ying Previous Purchases nary i d In-App Purchasing Tutorial	
93.8 Quer 93.9 Sumr 94. An Androi 94.1 Abou	pleting the Purchase ying Previous Purchases nary i d In-App Purchasing Tutorial t the In-App Purchasing Example Project	771 772 772 773
93.8 Quer 93.9 Sumr 94. An Androi 94.1 Abou 94.2 Creat	pleting the Purchase ying Previous Purchases nary id In-App Purchasing Tutorial t the In-App Purchasing Example Project ing the InAppPurchase Project.	771 772 772 773 773
93.8 Quer 93.9 Sumr 94. An Androi 94.1 Abou 94.2 Creat 94.3 Addiu	pleting the Purchase ying Previous Purchases nary id In-App Purchasing Tutorial t the In-App Purchasing Example Project ing the InAppPurchase Project ng Libraries to the Project	771 772 772 773 773 773
93.8 Quer 93.9 Sumr 94. An Androi 94.1 Abou 94.2 Creat 94.3 Addin 94.4 Desig	pleting the Purchase ying Previous Purchases nary id In-App Purchasing Tutorial t the In-App Purchasing Example Project ing the InAppPurchase Project ng Libraries to the Project ning the User Interface	771 772 772 773 773 773 773 774
93.8 Quer 93.9 Sum 94. An Androi 94.1 Abou 94.2 Creat 94.3 Addin 94.4 Desig 94.5 Addin	pleting the Purchase ying Previous Purchases nary id In-App Purchasing Tutorial It the In-App Purchasing Example Project ing the InAppPurchase Project ng Libraries to the Project ping the User Interface ng the App to the Google Play Store	771 772 772 773 773 773 773 774 775
93.8 Quer 93.9 Sumr 94. An Androi 94.1 Abou 94.2 Creat 94.3 Addin 94.4 Desig 94.5 Addin 94.6 Creat	pleting the Purchase ying Previous Purchases nary id In-App Purchasing Tutorial t the In-App Purchasing Example Project ing the InAppPurchase Project ng Libraries to the Project pling the User Interface ng the App to the Google Play Store ing an In-App Product	771 772 772 773 773 773 773 774 775 775
93.8 Quer 93.9 Sumr 94. An Androi 94.1 Abou 94.2 Creat 94.3 Addin 94.4 Desig 94.5 Addin 94.6 Creat 94.7 Enabl	pleting the Purchase ying Previous Purchases nary id In-App Purchasing Tutorial t the In-App Purchasing Example Project ing the InAppPurchase Project ng Libraries to the Project ng the User Interface ng the App to the Google Play Store ing an In-App Product ling License Testers	771 772 772 773 773 773 773 773 775 775 775
93.8 Quer 93.9 Sumr 94. An Androi 94.1 Abou 94.2 Creat 94.3 Addin 94.4 Desig 94.5 Addin 94.6 Creat 94.7 Enabl 94.8 Initia	pleting the Purchase	771 772 772 773 773 773 773 774 775 775 775 776
93.8 Quer 93.9 Sumr 94. An Androi 94.1 Abou 94.2 Creat 94.3 Addin 94.4 Desig 94.5 Addin 94.6 Creat 94.7 Enabl 94.8 Initia 94.9 Ouer	pleting the Purchase ying Previous Purchases nary id In-App Purchasing Tutorial it the In-App Purchasing Example Project ing the InAppPurchase Project ng Libraries to the Project pring the User Interface ng the App to the Google Play Store ing an In-App Product ling License Testers lizing the Billing Client	771 772 772 773 773 773 773 773 775 775 775 775 776 778
93.8 Quer 93.9 Sumr 94. An Androi 94.1 Abou 94.2 Creat 94.3 Addin 94.4 Desig 94.5 Addin 94.6 Creat 94.7 Enabl 94.8 Initia 94.9 Quer 94.10 Lau	pleting the Purchase	771 772 772 773 773 773 773 773 774 775 775 775 776 778 779
93.8 Quer 93.9 Sumr 94. An Androi 94.1 Abou 94.2 Creat 94.3 Addin 94.4 Desig 94.5 Addin 94.6 Creat 94.7 Enabl 94.8 Initia 94.9 Quer 94.10 Laun 94.11 Han	pleting the Purchase	771 772 772 773 773 773 773 773 774 775 775 775 776 778 779 779

	94.12 Consuming the Product	780
	94.13 Restoring a Previous Purchase	781
	94.14 Testing the App	782
	94.15 Troubleshooting	783
	94.16 Summary	784
95.	Accessing Cloud Storage using the Android Storage Access Framework	
	95.1 The Storage Access Framework	
	95.2. Working with the Storage Access Framework	
	95.3 Filtering Picker File Listings	
	95.4 Handling Intent Results	
	95.5 Reading the Content of a File	
	95.6 Writing Content to a File	
	95.7 Deleting a File	
	95.8 Gaining Persistent Access to a File	
	95.9 Summary	
96.	An Android Storage Access Framework Example	
	061 About the Storage Access Framework Example	701
	96.1 About the Storage Access Framework Example	701
	96.2 Designing the User Interface	701
	96.4 Adding the Activity Launchers	702
	96.5 Creating a New Storage File	703
	96.5 Creating to a Storage File	793
	96.0 Saving to a Storage File	705
	96.7 Opening and Reading a Storage Trie	796
	96.8 Summary	798
.		/ 90
97.	An Android Studio Primary/Detail Flow Iutorial	
	97.1 The Primary/Detail Flow	799
	97.2 Creating a Primary/Detail Flow Activity	800
	97.3 Adding the Primary/Detail Flow Activity	800
	97.4 Modifying the Primary/Detail Flow Template	801
	97.5 Changing the Content Model	801
	97.6 Changing the Detail Pane	803
	97.7 Modifying the ItemDetailFragment Class	804
	97.8 Modifying the ItemListFragment Class	805
	97.9 Adding Manifest Permissions	805
	97.10 Running the Application	806
	97.11 Summary	806
98.	Working with Material Design 3 Theming	807
	98.1 Material Design 2 vs. Material Design 3	807
	98.2 Understanding Material Design Theming	807
	98.3 Material Design 3 Theming	807
	98.4 Building a custom theme	809
	98.5 Summary	811
99.	A Material Design 3 Theming and Dynamic Color Tutorial	813
	99.1 Creating the ThemeDemo Project	813
	99.2 Designing the User Interface	813

99.3 Building a new theme	
99.4 Adding the Theme to the Project	
99.5 Enabling Dynamic Color Support	
99.6 Summary	
100. An Overview of Gradle in Android Studio	. 821
100.1 An Overview of Gradle	
100.2 Gradle and Android Studio	
100.2.1 Sensible Defaults	
100.2.2 Dependencies	
100.2.3 Build Variants	
100.2.4 Manifest Entries	
100.2.5 APK Signing	
100.2.6 ProGuard Support	
100.3 The Property and Settings Gradle Build File	
100.4 The Top-level Gradle Build File	
100.5 Module Level Gradle Build Files	
100.6 Configuring Signing Settings in the Build File	
100.7 Running Gradle Tasks from the Command Line	
100.8 Summary	
Index	829

Chapter 1

1. Introduction

This book, fully updated for Android Studio Ladybug and the new UI, teaches you how to develop Androidbased applications using the Kotlin programming language.

Beginning with the basics, the book outlines how to set up an Android development and testing environment, followed by an introduction to programming in Kotlin, including data types, control flow, functions, lambdas, and object-oriented programming. Asynchronous programming using Kotlin coroutines and flow is also covered in detail.

Chapters also cover the Android Architecture Components, including view models, lifecycle management, Room database access, content providers, the Database Inspector, app navigation, live data, and data binding.

More advanced topics such as intents are also covered, as are touch screen handling, gesture recognition, and the recording and playback of audio. This book edition also covers printing, transitions, and foldable device support.

The concepts of material design are also covered in detail, including the use of floating action buttons, Snackbars, tabbed interfaces, card views, navigation drawers, and collapsing toolbars.

Other key features of Android Studio and Android are also covered in detail, including the Layout Editor, the ConstraintLayout and ConstraintSet classes, MotionLayout Editor, view binding, constraint chains, barriers, and direct reply notifications.

Chapters also cover advanced features of Android Studio, such as App Links, Gradle build configuration, in-app billing, and submitting apps to the Google Play Developer Console.

Assuming you already have some programming experience, are ready to download Android Studio and the Android SDK, have access to a Windows, Mac, or Linux system, and have ideas for some apps to develop, you are ready to get started.

1.1 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/ladybugkotlin/

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

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

1.2 Feedback

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

Introduction

1.3 Errata

While we make every effort to ensure the accuracy of the content of this book, it is inevitable that 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/ladybugkotlin

If you find an error not listed in the errata, please let us know by emailing our technical support team at *info@ payloadbooks.com*. They are there to help you and will work to resolve any problems you may encounter.

Chapter 3

3. Creating an Example Android App in Android Studio

The preceding chapters of this book have explained how to configure an environment suitable for developing Android applications using the Android Studio IDE. Before moving on to slightly more advanced topics, now is a good time to validate that all required development packages are installed and functioning correctly. The best way to achieve this goal is to create an Android application and compile and run it. This chapter will cover creating an Android application project using Android Studio. Once the project has been created, a later chapter will explore using the Android emulator environment to perform a test run of the application.

3.1 About the Project

The project created in this chapter takes the form of a rudimentary currency conversion calculator (so simple, in fact, that it only converts from dollars to euros and does so using an estimated conversion rate). The project will also use one of the most basic Android Studio project templates. This simplicity allows us to introduce some key aspects of Android app development without overwhelming the beginner by introducing too many concepts, such as the recommended app architecture and Android architecture components, at once. When following the tutorial in this chapter, rest assured that the techniques and code used in this initial example project will be covered in much greater detail later.

3.2 Creating a New Android Project

The first step in the application development process is to create a new project within the Android Studio environment. Begin, therefore, by launching Android Studio so that the "Welcome to Android Studio" screen appears as illustrated in Figure 3-1:



Figure 3-1

Creating an Example Android App in Android Studio

Once this window appears, Android Studio is ready for a new project to be created. To create the new project, click on the *New Project* option to display the first screen of the *New Project* wizard.

3.3 Creating an Activity

The next step is to define the type of initial activity to be created for the application. Options are available to create projects for Phone and Tablet, Wear OS, Television, or Automotive. A range of different activity types is available when developing Android applications, many of which will be covered extensively in later chapters. For this example, however, select the *Phone and Tablet* option from the Templates panel, followed by the option to create an *Empty Views Activity*. The Empty Views Activity option creates a template user interface consisting of a single TextView object.





With the Empty Views Activity option selected, click Next to continue with the project configuration.

3.4 Defining the Project and SDK Settings

In the project configuration window (Figure 3-3), set the *Name* field to *AndroidSample*. The application name is the name by which the application will be referenced and identified within Android Studio and is also the name that would be used if the completed application were to go on sale in the Google Play store.

The *Package name* uniquely identifies the application within the Android application ecosystem. Although this can be set to any string that uniquely identifies your app, it is traditionally based on the reversed URL of your domain name followed by the application's name. For example, if your domain is *www.mycompany.com*, and the application has been named *AndroidSample*, then the package name might be specified as follows:

com.mycompany.androidsample

If you do not have a domain name, you can enter any other string into the Company Domain field, or you may use *example.com* for testing, though this will need to be changed before an application can be published:

com.example.androidsample

The *Save location* setting will default to a location in the folder named *AndroidStudioProjects* located in your home directory and may be changed by clicking on the folder icon to the right of the text field containing the current path setting.

Set the minimum SDK setting to API 26 (Oreo; Android 8.0). This minimum SDK will be used in most projects created in this book unless a necessary feature is only available in a more recent version. The objective here is to

build an app using the latest Android SDK while retaining compatibility with devices running older versions of Android (in this case, as far back as Android 8.0). The text beneath the Minimum SDK setting will outline the percentage of Android devices currently in use on which the app will run. Click on the *Help me choose* button (highlighted in Figure 3-3) to see a full breakdown of the various Android versions still in use:

	New Project	
Empty Views Activity		
Creates a new empty activity		
Name	AndroidSample	
Package name	com.example.androidsample	
Save location	/Users/neilsmyth/Dropbox/Documents/Books/Giraffe_Kotlin/AndroidSample	D
Language	Kotlin	¥
Minimum SDK	API 26 ("Oreo"; Android 8.0)	¥
	• Your app will run on approximately 92.4% of devices. Help me choose	
Build configuration language 🕐	Kotlin DSL (build.gradle.kts) [Recommended]	Ŧ
	Cancel Previous Next	Finist

Figure 3-3

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

3.5 Modifying the Example Application

Once the project has been created, the main window will appear containing our AndroidSample project, as illustrated in Figure 3-4 below:

•	🔹 🔼 AndroidSample 🗸 Version co	ontrol Y 🕒 Pixel 4 API 34 Y 🗠 app Y D 🏦 🗄 🖬 📬 式 🖧 🥰 Q 🖇	êj 🙁				
	Android \sim \bigcirc \diamondsuit $\stackrel{\scriptstyle <}{}$ $\stackrel{\scriptstyle }{}$	$\textcircled{\mbox{\footnotesize Cl}}$ MainActivity.kt $ imes$: Ç				
2	🗸 🕞 арр	1 package com.example.androidsample	× 67				
-	> 🗀 manifests	2					
	🗸 🛅 kotlin+java	3 > import	L				
	 com.example.androidsample 	11 10 N/h - Bass Maintabivity - AurConnettabivity() (c.				
C?	G MainActivity	13 6 [†] overnide fun onCreate(savedInstanceState: Rundle2) {	le				
~0	> 🗟 com.example.androidsample	14 super.onCreate(savedInstanceState)	+				
T	> icom.example.androidsample	15 enableEdgeToEdge()	-				
	> 🗋 res	16 setContentView(R.layout. <u>activity main</u>)					
10	> 🕾 Gradle Scripts	<pre>17 ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, insets -></pre>					
(!)		<pre>18 val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())</pre>					
0		19 v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.botto	n)				
>_		28 insets ^setOnApplyWindowInsetsListener					
0.0		21 }					
Ч		22 F					
o Ar	🗅 AndroidSample > 💿 app > src > 🗈 main > java > com > example > androidsample > @ MainActivity 30.1 LF UTF-8 🗔 🔶 4 spaces 🔗						



The newly created project and references to associated files are listed in the *Project* tool window on the left side of the main project window. The Project tool window has several modes in which information can be displayed. By default, this panel should be in *Android* mode. This setting is controlled by the menu at the top of the panel as highlighted in Figure 3-5. If the panel is not currently in Android mode, use the menu to switch mode:

Creating an Example Android App in Android Studio



Figure 3-5

3.6 Modifying the User Interface

The user interface design for our activity is stored in a file named *activity_main.xml* which, in turn, is located under *app -> res -> layout* in the Project tool window file hierarchy. Once located in the Project tool window, double-click on the file to load it into the user interface Layout Editor tool, which will appear in the center panel of the Android Studio main window:



Figure 3-6

In the toolbar across the top of the Layout Editor window is a menu (currently set to *Pixel* in the above figure) which is reflected in the visual representation of the device within the Layout Editor panel. A range of other
device options are available by clicking on this menu.

Use the System UI Mode button (\bigcirc) to turn Night mode on and off for the device screen layout. To change the orientation of the device representation between landscape and portrait, use the drop-down menu showing the \bigotimes icon.

As we can see in the device screen, the content layout already includes a label that displays a "Hello World!" message. Running down the left-hand side of the panel is a palette containing different categories of user interface components that may be used to construct a user interface, such as buttons, labels, and text fields. However, it should be noted that not all user interface components are visible to the user. One such category consists of *layouts*. Android supports a variety of layouts that provide different levels of control over how visual user interface components are positioned and managed on the screen. Though it is difficult to tell from looking at the visual representation of the user interface, the current design has been created using a ConstraintLayout. This can be confirmed by reviewing the information in the *Component Tree* panel, which, by default, is located in the lower left-hand corner of the Layout Editor panel and is shown in Figure 3-7:



Figure 3-7

As we can see from the component tree hierarchy, the user interface layout consists of a ConstraintLayout parent called *main* and a TextView child object.

Before proceeding, check that the Layout Editor's Autoconnect mode is enabled. This means that as components are added to the layout, the Layout Editor will automatically add constraints to ensure the components are correctly positioned for different screen sizes and device orientations (a topic that will be covered in much greater detail in future chapters). The Autoconnect button appears in the Layout Editor toolbar and is represented by a U-shaped icon. When disabled, the icon appears with a diagonal line through it (Figure 3-8). If necessary, re-enable Autoconnect mode by clicking on this button.



Figure 3-8

The next step in modifying the application is to add some additional components to the layout, the first of which will be a Button for the user to press to initiate the currency conversion.

The Palette panel consists of two columns, with the left-hand column containing a list of view component categories. The right-hand column lists the components contained within the currently selected category. In Figure 3-9, for example, the Button view is currently selected within the Buttons category:

Creating an Example Android App in Android Studio





Click and drag the *Button* object from the Buttons list and drop it in the horizontal center of the user interface design so that it is positioned beneath the existing TextView widget:



Figure 3-10

The next step is to change the text currently displayed by the Button component. The panel located to the right of the design area is the Attributes panel. This panel displays the attributes assigned to the currently selected component in the layout. Within this panel, locate the *text* property in the Common Attributes section and change the current value from "Button" to "Convert", as shown in Figure 3-11:

Creating an Example Android App in Android Studio

strokeWidth	
cornerRadius	@null
rippleColor	🖉 @color/m3_button_rip
text	Convert
ℬ text	
contentDescription	

Figure 3-11

The second text property with a wrench next to it allows a text property to be set, which only appears within the Layout Editor tool but is not shown at runtime. This is useful for testing how a visual component and the layout will behave with different settings without running the app repeatedly.

Just in case the Autoconnect system failed to set all of the layout connections, click on the Infer Constraints button (Figure 3-12) to add any missing constraints to the layout:



Figure 3-12

It is important to explain the warning button in the top right-hand corner of the Layout Editor tool, as indicated in Figure 3-13. This warning indicates potential problems with the layout. For details on any problems, click on the button:



Figure 3-13

When clicked, the Problems tool window (Figure 3-14) will appear, describing the nature of the problems:

Problems File 1 Project Errors Layout and Qualifiers 1	: -
 	Hardcoded text Hardcoded string "Convert", should use @string resource Hardcoding text attributes directly in layout files is bad for several reasons: * When creating configuration variations (for example for landscape or portrait) you have to repeat the actual text (and keep it up to date when making changes)
	19 <button< td=""> 20 android:id="@+id/button" 21 android:layout_width="wrap_content" 22 android:layout_height="wrap_content" 23 android:layout_marginTop="106dp" 24 android:text="Convert"</button<>

Figure 3-14

This tool window is divided into two panels. The left panel (marked A in the above figure) lists issues detected

Creating an Example Android App in Android Studio

within the layout file. In our example, only the following problem is listed:

button <Button>: Hardcoded text

When an item is selected from the list (B), the right-hand panel will update to provide additional detail on the problem (C). In this case, the explanation reads as follows:

Hardcoded string "Convert", should use @string resource

The tool window also includes a preview editor (D), allowing manual corrections to be made to the layout file.

This I18N message informs us that a potential issue exists concerning the future internationalization of the project ("I18N" comes from the fact that the word "internationalization" begins with an "I", ends with an "N" and has 18 letters in between). The warning reminds us that attributes and values such as text strings should be stored as *resources* wherever possible when developing Android applications. Doing so enables changes to the appearance of the application to be made by modifying resource files instead of changing the application source code. This can be especially valuable when translating a user interface to a different spoken language. If all of the text in a user interface is contained in a single resource file, for example, that file can be given to a translator, who will then perform the translation work and return the translated file for inclusion in the application. This enables multiple languages to be targeted without the necessity for any source code changes to be made. In this instance, we are going to create a new resource named *convert_string* and assign to it the string "Convert".

Begin by clicking on the Show Quick Fixes button (E) and selecting the *Extract string resource* option from the menu, as shown in Figure 3-15:





After selecting this option, the *Extract Resource* panel (Figure 3-16) will appear. Within this panel, change the resource name field to *convert_string* and leave the resource value set to *Convert* before clicking on the OK button:

• • •	Extract Resource
Resource name:	convert_string
Resource value:	Convert
Source set:	main src/main/res
File name:	strings.xml 💌
Create the resour	ce in directories:
$+$ - \boxtimes \square	
🗹 values	
values-night	
	Cancel OK

Figure 3-16

Chapter 12

12. Kotlin Data Types, Variables, and Nullability

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

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

12.1 Kotlin Data Types

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

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

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

```
val mynumber = 10
```

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

```
1010
```

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

val myletter = 'c'

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

Kotlin Data Types, Variables, and Nullability

converted to binary, it is stored as:

10101100011

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

12.1.1 Integer Data Types

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

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

12.1.2 Floating-Point Data Types

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

12.1.3 Boolean Data Type

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

12.1.4 Character Data Type

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

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

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

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

```
val myChar4 = ' \ 0058'
```

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

12.1.5 String Data Type

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

val message = "You have 10 new messages."

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

val message = """You have 10 new messages,

```
5 old messages
and 6 spam messages."""
```

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

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

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

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

println(message)

When executed, the code will output the following message:

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

12.1.6 Escape Sequences

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

var newline = '\n'

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

var backslash = $' \setminus \setminus '$

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

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

Kotlin Data Types, Variables, and Nullability

12.2 Mutable Variables

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

12.3 Immutable Variables

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

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

12.4 Declaring Mutable and Immutable Variables

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

var userCount = 10

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

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

Immutable variables are declared using the val keyword.

```
val maxUserCount = 20
```

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

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

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

12.5 Data Types are Objects

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

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

```
val myString = "The quick brown fox"
```

val uppercase = myString.toUpperCase()

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

val length = myString.length

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

val result = myString.contains("fox")

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

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

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

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

12.6 Type Annotations and Type Inference

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

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

val userCount: Int = 10

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

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

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

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

val bookTitle = "Android Studio Development Essentials"

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

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

Kotlin Data Types, Variables, and Nullability

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

12.7 Nullable Type

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

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

val username: String = null

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

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

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

val username: String? = null

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

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

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

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

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

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

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

12.8 The Safe Call Operator

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

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

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

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

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

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

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

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

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

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

val uppercase = username?.length

12.9 Not-Null Assertion

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

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

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

Exception in thread "main" kotlin.KotlinNullPointerException

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

12.10 Nullable Types and the let Function

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

```
val firstNumber = 10
val secondNumber = 20
val result = firstNumber.times(secondNumber)
print(result)
```

The above example works because the secondNumber variable is a non-null type. A problem, however, occurs if the secondNumber variable is declared as being of nullable type:

Kotlin Data Types, Variables, and Nullability

```
val firstNumber = 10
val secondNumber: Int? = 20
val result = firstNumber.times(secondNumber)
print(result)
```

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

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

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

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

```
if (secondNumber != null) {
```

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

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

secondNumber?.let {

```
val result = firstNumber.times(it)
print(result)
}
```

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

12.11 Late Initialization (lateinit)

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

```
var myName: String
```

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

lateinit var myName: String

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

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

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

```
lateinit var myName: String
print("My Name is " + myName)
Exception in thread "main" kotlin.UninitializedPropertyAccessException: lateinit
property myName has not been initialized
```

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

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

12.12 The Elvis Operator

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

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

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

return myString ?: "String is null"

12.13 Type Casting and Type Checking

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

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

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

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

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

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

```
if (keyMgr is KeyguardManager) {
    // It is a KeyguardManager object
```

}

Kotlin Data Types, Variables, and Nullability

12.14 Summary

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

It is challenging to think of an Android application concept that does not require some form of user interface. Most Android devices come equipped with a touch screen and keyboard (either virtual or physical), and taps and swipes are the primary interaction between the user and the application. Invariably these interactions take place through the application's user interface.

A well-designed and implemented user interface, an essential factor in creating a successful and popular Android application, can vary from simple to highly complex, depending on the design requirements of the individual application. Regardless of the level of complexity, the Android Studio Layout Editor tool significantly simplifies the task of designing and implementing Android user interfaces.

25.1 Basic vs. Empty Views Activity Templates

As outlined in the chapter entitled *"The Anatomy of an Android App*", Android applications comprise one or more activities. An activity is a standalone module of application functionality that usually correlates directly to a single user interface screen. As such, when working with the Android Studio Layout Editor, we are invariably work on the layout for an activity.

When creating a new Android Studio project, several templates are available to be used as the starting point for the user interface of the main activity. The most basic templates are the Basic Views Activity and Empty Views Activity templates. Although these seem similar at first glance, there are considerable differences between the two options. To see these differences within the layout editor, use the View Options menu to enable Show System UI, as shown in Figure 25-1 below:



Figure 25-1

The Empty Views Activity template creates a single layout file consisting of a ConstraintLayout manager instance containing a TextView object, as shown in Figure 25-2:



Figure 25-2

The Basic Views Activity, on the other hand, consists of multiple layout files. The top-level layout file has a CoordinatorLayout as the root view, a configurable app bar (which contains a toolbar) that appears across the top of the device screen (marked A in Figure 25-3), and a floating action button (the email button marked B). In addition to these items, the *activity_main.xml* layout file contains a reference to a second file named *content_main.xml* containing the content layout (marked C):



Figure 25-3

The Basic Views Activity contains layouts for two screens containing a button and a text view. This template aims to demonstrate how to implement navigation between multiple screens within an app. If an unmodified app using the Basic Views Activity template were to be run, the first of these two screens would appear (marked A in Figure 25-4). Pressing the Next button would navigate to the second screen (B), which, in turn, contains a button to return to the first screen:



This app behavior uses of two Android features referred to as *fragments* and *navigation*, which will be covered starting with the chapters entitled *"An Introduction to Android Fragments"* and *"An Overview of the Navigation Architecture Component"* respectively.

The *content_main.xml* file contains a special fragment, known as a Navigation Host Fragment which allows different content to be switched in and out of view depending on the settings configured in the *res -> layout -> nav_graph.xml* file. In the case of the Basic Views Activity template, the *nav_graph.xml* file is configured to switch between the user interface layouts defined in the *fragment_first.xml* and *fragment_second.xml* files based on the Next and Previous button selections made by the user.

The Empty Views Activity template is helpful if you need neither a floating action button nor a menu in your activity and do not need the special app bar behavior provided by the CoordinatorLayout, such as options to make the app bar and toolbar collapse from view during certain scrolling operations (a topic covered in the chapter entitled *"Working with the AppBar and Collapsing Toolbar Layouts"*). However, the Basic Views Activity is helpful because it provides these elements by default. In fact, it is often quicker to create a new activity using the Basic Views Activity template and delete the elements you do not require than to use the Empty Views Activity template and manually implement behavior such as collapsing toolbars, a menu, or a floating action button.

Since not all of the examples in this book require the features of the Basic Views Activity template, however, most of the examples in this chapter will use the Empty Views Activity template unless the example requires one or other of the features provided by the Basic Views Activity template.

For future reference, if you need a menu but not a floating action button, use the Basic Views Activity and follow these steps to delete the floating action button:

- 1. Double-click on the main *activity_main.xml* layout file in the Project tool window under *app -> res -> layout* to load it into the Layout Editor. With the layout loaded into the Layout Editor tool, select the floating action button and tap the keyboard *Delete* key to remove the object from the layout.
- 2. Locate and edit the Kotlin code for the activity (located under *app -> kotlin+java -> <package name> -> <activity class name>* and remove the floating action button code from the onCreate method as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    setSupportActionBar(binding.toolbar)
    val navController = findNavController(R.id.nav_host_fragment_content_main)
    appBarConfiguration = AppBarConfiguration(navController.graph)
    setupActionBarWithNavController(navController, appBarConfiguration)
    binding.fab.setOnClickListener { view ->
        Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
        .setAnchorView(R.id.fab)
        .setAction("Action", null).show()
    }
}
```

If you need a floating action button but no menu, use the Basic Views Activity template and follow these steps:

- 1. Edit the main activity class file and delete the onCreateOptionsMenu and onOptionsItemSelected methods.
- 2. Select the *res* -> *menu* item in the Project tool window and tap the keyboard *Delete* key to remove the folder and corresponding menu resource files from the project.

If you need to use the Basic Views Activity template but need neither the navigation features nor the second content fragment, follow these steps:

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

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

binding.buttonFirst.setOnClickListener {

findNavController().navigate(R.id.action_FirstFragment_to_SecondFragment)

}

 \rightarrow

25.2 The Android Studio Layout Editor

As demonstrated in previous chapters, the Layout Editor tool provides a "what you see is what you get" (WYSIWYG) environment in which views can be selected from a palette and then placed onto a canvas representing the display of an Android device. Once a view has been placed on the canvas, it can be moved, deleted, and resized (subject to the constraints of the parent view). Moreover, various properties relating to the selected view may be modified using the Attributes tool window.

Under the surface, the Layout Editor tool constructs an XML resource file containing the definition of the user interface that is being designed. As such, the Layout Editor tool operates in three distinct modes: Design, Code, and Split.

25.3 Design Mode

In design mode, the user interface can be visually manipulated by directly working with the view palette and the graphical representation of the layout. Figure 25-5 highlights the key areas of the Android Studio Layout Editor tool in design mode:



Figure 25-5

A – **Palette** – The palette provides access to the range of view components the Android SDK provides. These are grouped into categories for easy navigation. Items may be added to the layout by dragging a view component from the palette and dropping it at the desired position on the layout.

 \mathbf{B} – **Device Screen** – The device screen provides a visual "what you see is what you get" representation of the user interface layout as it is being designed. This layout allows direct design manipulation by allowing views to be selected, deleted, moved, and resized. The device model represented by the layout can be changed anytime using a menu in the toolbar.

C – **Component Tree** – As outlined in the previous chapter (*"Understanding Android Views, View Groups and Layouts"*), user interfaces are constructed using a hierarchical structure. The component tree provides a visual

overview of the hierarchy of the user interface design. Selecting an element from the component tree will cause the corresponding view in the layout to be selected. Similarly, selecting a view from the device screen layout will select that view in the component tree hierarchy.

D – **Attributes** – All of the component views listed in the palette have associated with them a set of attributes that can be used to adjust the behavior and appearance of that view. The Layout Editor's attributes panel provides access to the attributes of the currently selected view in the layout allowing changes to be made.

E - Toolbar – The Layout Editor toolbar provides quick access to a wide range of options, including, amongst other options, the ability to zoom in and out of the device screen layout, change the device model currently displayed, rotate the layout between portrait and landscape and switch to a different Android SDK API level. The toolbar also has a set of context-sensitive buttons which will appear when relevant view types are selected in the device screen layout.

F – **Mode Switching Controls** – These three buttons provide a way to switch back and forth between the Layout Editor tool's Design, Code, and Split modes.

G - **Zoom and Pan Controls** - This control panel allows you to zoom in and out of the design canvas, grab the canvas, and pan around to find obscured areas when zoomed in.

25.4 The Palette

The Layout Editor palette is organized into two panels designed to make it easy to locate and preview view components for addition to a layout design. The category panel (marked A in Figure 25-6) lists the different categories of view components supported by the Android SDK. When a category is selected from the list, the second panel (B) updates to display a list of the components that fall into that category:



Figure 25-6

To add a component from the palette onto the layout canvas, select the item from the component list or the preview panel, drag it to the desired location on the canvas, and drop it into place.

A search for a specific component within the selected category may be initiated by clicking the search button (marked C in Figure 25-6 above) in the palette toolbar and typing in the component name. As characters are typed, matching results will appear in the component list panel. If you are unsure of the component's category, select the All Results category before or during the search operation.

25.5 Design Mode and Layout Views

The layout editor will appear in Design mode by default, as shown in Figure 25-5 above. This mode provides a visual representation of the user interface. Design mode can be selected by clicking on the button marked C in Figure 25-7:





When the Layout Editor tool is in Design mode, the layout can be viewed in two ways. The view shown in Figure 25-5 above is the Design view and shows the layout and widgets as they will appear in the running app. A second mode, the Blueprint view, can be shown instead of or concurrently with the Design view. The toolbar menu in Figure 25-8 provides options to display the Design, Blueprint, or both views. Settings are also available to adjust for color blindness. A fifth option, *Force Refresh Layout*, causes the layout to rebuild and redraw. This can be useful when the layout enters an unexpected state or is not accurately reflecting the current design settings:



Figure 25-8

Whether to display the layout view, design view, or both is a matter of personal preference. A good approach is to begin with both displayed as shown in Figure 25-9:





25.6 Night Mode

To view the layout in night mode during the design work, select the menu shown in Figure 25-10 below and change the setting to *Night*:



Figure 25-10

The mode menu also includes options for testing dynamic colors, a topic covered in the chapter "A Material Design 3 Theming and Dynamic Color Tutorial".

25.7 Code Mode

It is important to remember when using the Android Studio Layout Editor tool that all it is doing is providing a user-friendly approach to creating XML layout resource files. The underlying XML can be viewed and directly edited during the design process by selecting the button marked A in Figure 25-7 above.

Figure 25-11 shows the Android Studio Layout Editor tool in Code mode, allowing changes to be made to the user interface declaration by modifying the XML:

> activi	ty_main.xml × ≣0 ∞ ÷
1	xml version="1.0" encoding="utf-8"?
2 🕞	<pre><androidx.constraintlayout.widget.constraintlayout <="" pre="" xmlns:android="http://schemas.android.com/apk/res/android"></androidx.constraintlayout.widget.constraintlayout></pre>
3	<pre>xmlns:app="http://schemas.android.com/apk/res-auto"</pre>
4	<pre>xmlns:tools="http://schemas.android.com/tools"</pre>
5	android:layout_width="match_parent"
6	android:layout_height="match_parent"
7	tools:context=".MainActivity">
8	
9	kTextView
10	android:layout_width="wrap_content"
11	android:layout_height="wrap_content"
12	android:text="Hello World!"
13	app:layout_constraintBottom_toBottomOf="parent"
14	app:layout_constraintEnd_toEndOf="parent"

Figure 25-11

25.8 Split Mode

In Split mode, the editor shows the Design and Code views side-by-side, allowing the user interface to be modified visually using the design canvas and making changes directly to the XML declarations. Split mode is selected using the button marked B Figure 25-7 above.

Any changes to the XML are automatically reflected in the design canvas and vice versa. Figure 25-12 shows the editor in Split mode:

1	xml version="1.0" encoding="utf-8"?	Palette	Q @ —	activity_main.xml - 😒 🔘 🕓	Pixel ~ 🖂 33 ~	>	θ	-
2 @	<pre><androidx.constraintlayout.widget.constraintlayout xmlr<br="">ymlns:ann="http://schemas.android.com/ank/res-auto"</androidx.constraintlayout.widget.constraintlayout></pre>	Common	Ab TextView				?	Attrit
4 5 6 7 8 9	<pre>xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent" android:layout_height="match_parent" tools:context=".MainActivity"> <textview android:id="@+id/textView"</textview </pre>	Text Buttons Widgets Layouts Containers Helpers	Button ImageView RecyclerView FragmentCon ScrollView Switch		*******			utes
11 12 13 14 15 16	<pre>android:layout_width="wrap_content" android:layout_height="wrap_content" android:tayout_cheight="wrap_content" app:layout_constraintBottom_toBottomOf="parent" app:layout_constraintEnd_toEndOf="parent" app:layout_constraintStart_toStartOf="parent"</pre>	Google Legacy Component Tree °\o ConstraintLayou Ab textView "Heil	® — t o World!"	solo votit	4	~~		
17 18 19	<pre>app:layout_constraintTop_toTopDf="parent" /> </pre>		o nond.		//			

Figure 25-12

25.9 Setting Attributes

The Attributes panel provides access to all available settings for the currently selected component. Figure 25-13, for example, shows some of the attributes for the TextView widget:

Attributes	Q @ —
Ab TextView	textView
id	textView
> Declared Attr	ibutes + -
✓ Layout	
Constraint Widg	et
50 0 -	
 Constraints 	
Start → Start	Ut parent (Udp)
\bigcirc End \rightarrow EndOf	parent (0dp)
° Bottom → Bo	ttomOf parent (0dp)
layout_width	wrap_content
layout_height	wrap_content
visibility	•



The Attributes tool window is divided into the following different sections.

- id Contains the id property, which defines the name by which the currently selected object will be referenced in the app's source code.
- Declared Attributes Contains all of the properties already assigned a value.
- Layout The settings that define how the currently selected view object is positioned and sized relative to the screen and other objects in the layout.
- Transforms Contains controls allowing the currently selected object to be rotated, scaled, and offset.
- **Common Attributes** A list of attributes that commonly need to be changed for the class of view object currently selected.
- All Attributes A complete list of all the attributes available for the currently selected object.

A search for a specific attribute may also be performed by selecting the search button in the toolbar of the attributes tool window and typing in the attribute name.

Some attributes contain a narrow button to the right of the value field. This indicates that the Resources dialog is available to assist in selecting a suitable property value. To display the dialog, click on the button. The appearance of this button changes to reflect whether or not the corresponding property value is stored in a resource file or hard-coded. If the value is stored in a resource file, the button to the right of the text property field will be filled in to indicate that the value is not hard-coded, as highlighted in Figure 25-14 below:

✓ Common Attributes			
text	@string/some_text		
ℬ text			
contentDescription	0		
\vee textAppearance	@android:style/Text/ ▼ [

Figure 25-14

Attributes for which a finite number of valid options are available will present a drop-down menu (Figure 25-15) from which a selection may be made.

typeface	▼
textSize	normal
lineSpacingExtra	sans
textColor	serif
textStyle	monospace



A dropper icon can be clicked to display the color selection palette. Similarly, when a flag icon appears, it can be clicked to display a list of options available for the attribute, while an image icon opens the resource manager panel allowing images and other resource types to be selected for the attribute.

25.10 Transforms

The transforms panel within the Attributes tool window (Figure 25-16) provides a set of controls and properties that control visual aspects of the currently selected object in terms of rotation, alpha (used to fade a view in and out), scale (size), and translation (offset from current position):



Figure 25-16

The panel contains a visual representation of the view, which updates as properties are changed. These changes are also reflected in the view within the layout canvas.

25.11 Tools Visibility Toggles

When reviewing the content of an Android Studio XML layout file in Code mode, you will notice that many attributes that define how a view appears and behaves begin with the *android*: prefix. This indicates that the attributes are set within the *android* namespace and will take effect when the app is run. The following excerpt from a layout file, for example, sets a variety of attributes on a Button view:

```
<Button
android:id="@+id/button"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Button"
```

·

In addition to the android namespace, Android Studio also provides a *tools* namespace. When attributes are set within this namespace, they only take effect within the layout editor preview. While designing a layout, you might find it helpful for an EditText view to display some text but require the view to be blank when the app runs. To achieve this, you would set the text property of the view using the tools namespace as follows:

<EditText

```
android:id="@+id/editTextTextPersonName"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:ems="10"
android:inputType="textPersonName"
tools:text="Sample Text"
```

l attribute of this type is set in the Attributes tool y

A tool attribute of this type is set in the Attributes tool window by entering the value into the property fields marked by the wrench icon, as shown in Figure 25-17:





Tools attributes are particularly useful for changing the visibility of a view during the design process. A layout may contain a view that is programmatically displayed and hidden when the app runs, depending on user actions. To simulate the hiding of the view, the following tools attribute could be added to the view XML declaration:

tools:visibility="invisible"

Although the view will no longer be visible when using the invisible setting, it is still present in the layout and occupies the same space it did when it was visible. To make the layout behave as though the view no longer exists, the visibility attribute should be set to *gone* as follows:

tools:visibility="gone"

In both examples above, the visibility settings only apply within the layout editor and will have no effect in the running app. To control visibility in both the layout editor and running app, the same attribute would be set using the *android* namespace:

android:visibility="gone"

While these visibility tools attributes are useful, having to manually edit the XML layout file is a cumbersome process. To make it easier to change these settings, Android Studio provides a set of toggles within the layout editor Component Tree panel. To access these controls, click in the margin to the right of the corresponding view in the panel. Figure 25-18, for example, shows the tools visibility toggle controls for a Button view named myButton:



Figure 25-18

These toggles control the visibility of the corresponding view for both the android and tools namespaces and provide *not set*, *visible*, *invisible* and *gone* options. When conflicting attributes are set (for example, an android namespace toggle is set to visible while the tools value is set to invisible), the tools namespace takes precedence within the layout preview. When a toggle selection is made, Android Studio automatically adds the appropriate attribute to the XML view element in the layout file.

In addition to the visibility toggles in the Component Tree panel, the layout editor also includes the *tools visibility and position* toggle button shown highlighted in Figure 25-19 below:



Figure 25-19

This button toggles the current tools visibility settings. If the Button view shown above currently has the tools visibility attribute set to *gone*, for example, toggling this button will make it visible. This makes it easy to quickly check the layout behavior as the view is added to and removed from the layout. This toggle is also useful for checking that the views in the layout are correctly constrained, a topic covered in the chapter entitled "A *Guide to Using ConstraintLayout in Android Studio*".

25.12 Converting Views

Changing a view in a layout from one type to another (such as converting a TextView to an EditText) can be performed easily within the Android Studio layout editor by right-clicking on the view either within the screen layout or Component tree window and selecting the *Convert view*... menu option (Figure 25-20):

Component Tree	\$ -	
್ಧಿ ConstraintLa	ayout	
🗆 button3	Ab Show Baseline	
Ab textViev	$\mathcal{J}^{\mathcal{S}}_{\mathbf{x}}$ Clear Constraints of Selection	
Chip	Constrain	
	Organize >	
	Align	
	Chains	
	Center	
	Add helpers	
	$\mathscr{O}_{\!$	
	Convert view	
	Refactor	

Figure 25-20

Once selected, a dialog containing a list of compatible view types to which the selected object is eligible for conversion will appear. Figure 25-21, for example, shows the types to which an existing TextView view may be converted:

Convert View to:	Button	
Ab TextView ➢ ImageView ✓ CheckBox S ToggleButton	Button Button EditText RadioButton	
		Apply



This technique is also helpful in converting layouts from one type to another (for example, converting a ConstraintLayout to a LinearLayout).

25.13 Displaying Sample Data

When designing layouts in Android Studio, situations will arise where the content to be displayed within the user interface will not be available until the app is completed and running. This can sometimes make it difficult to assess how the layout will appear at app runtime from within the layout editor. To address this issue, the layout editor allows sample data to be specified, which will populate views within the layout editor with sample images and data. This sample data only appears within the layout editor and is not displayed when the app runs. Sample data may be configured either by directly editing the XML for the layout or visually using the design-time helper by right-clicking on the widget in the design area and selecting the *Set Sample Data* menu option. The design-time helper panel will display a range of preconfigured options for sample data to be displayed on the selected view item, including combinations of text and images in various configurations. Figure 25-22, for example, shows the sample data options displayed when selecting sample data to appear in a RecyclerView list:

Design-time View Attributes		
Item te	emplate	
<	Default	>
Item co	ount	
10		\$

Figure 25-22

Alternatively, custom text and images may be provided for display during the layout design process. Since sample data is implemented as a *tools* attribute, the visibility of the data within the preview can be controlled using the toggle button highlighted in Figure 25-19 above.

25.14 Creating a Custom Device Definition

The device menu in the Layout Editor toolbar (Figure 25-23) provides a list of pre-configured device types, which, when selected, will appear as the device screen canvas. In addition to the pre-configured device types, any AVD instances previously configured within the Android Studio environment will also be listed within the menu. To add additional device configurations, display the device menu, select the *Add Device Definition* option and follow the steps outlined in the chapter entitled *"Creating an Android Virtual Device (AVD) in Android Studio*".



Figure 25-23

25.15 Changing the Current Device

As an alternative to the device selection menu, the current device format may be changed by selecting the *Custom* option from the device menu, clicking on the resize handle located next to the bottom right-hand corner of the device screen (Figure 25-24), and dragging to select an alternate device display format. As the screen resizes, markers will appear indicating the various size options and orientations available for selection:



Figure 25-24

25.16 Layout Validation

The layout validation option allows the user interface layout to be previewed simultaneously on a range of Pixelsized screens. To access the layout validation tool window, select the *View -> Tool Windows -> Layout Validation* menu option. Once loaded, the panel will appear as shown in Figure 25-25, with the layout rendered on multiple device screen configurations:



Figure 25-25

25.17 Summary

A key part of developing Android applications involves the creation of the user interface. This is performed within the Android Studio environment using the Layout Editor tool, which operates in three modes. In Design mode, view components are selected from a palette, positioned on a layout representing an Android device screen, and configured using a list of attributes. The underlying XML representing the user interface layout can be directly edited in Code mode. Split mode, on the other hand, allows the layout to be created and modified both visually and via direct XML editing. These modes combine to provide an extensive and intuitive user interface design environment.

The layout validation panel allows user interface layouts to be quickly previewed on various device screen sizes.

41. Modern Android App Architecture with Jetpack

For many years, Google did not recommend a specific approach to building Android apps other than to provide tools and development kits while letting developers decide what worked best for a particular project or individual programming style. That changed in 2017 with the introduction of the Android Architecture Components, which, in turn, became part of Android Jetpack when it was released in 2018.

This chapter provides an overview of the concepts of Jetpack, Android app architecture recommendations, and some key architecture components. Once the basics have been covered, these topics will be covered in more detail and demonstrated through practical examples in later chapters.

41.1 What is Android Jetpack?

Android Jetpack consists of Android Studio, the Android Architecture Components, the Android Support Library, and a set of guidelines recommending how an Android App should be structured. The Android Architecture Components are designed to make it quicker and easier to perform common tasks when developing Android apps while also conforming to the key principle of the architectural guidelines.

While all Android Architecture Components will be covered in this book, this chapter will focus on the key architectural guidelines and the ViewModel, LiveData, and Lifecycle components while introducing Data Binding and Repositories.

Before moving on, it is important to understand that the Jetpack approach to app development is optional. While highlighting some of the shortcomings of other techniques that have gained popularity over the years, Google stopped short of completely condemning those approaches to app development. Google is taking the position that while there is no right or wrong way to develop an app, there is a recommended way.

41.2 The "Old" Architecture

In the chapter entitled *"Creating an Example Android App in Android Studio*", an Android project was created consisting of a single activity that contained all of the code for presenting and managing the user interface together with the back-end logic of the app. Until the introduction of Jetpack, the most common architecture followed this paradigm with apps consisting of multiple activities (one for each screen within the app), with each activity class to some degree mixing user interface and back-end code.

This approach led to a range of problems related to the lifecycle of an app (for example, an activity is destroyed and recreated each time the user rotates the device leading to the loss of any app data that had not been saved to some form of persistent storage) as well as issues such as inefficient navigation involving launching a new activity for each app screen accessed by the user.

41.3 Modern Android Architecture

At the most basic level, Google now advocates single-activity apps where different screens are loaded as content within the same activity.

Modern architecture guidelines also recommend separating different areas of responsibility within an app into entirely separate modules (a concept referred to as "separation of concerns"). One of the keys to this approach

Modern Android App Architecture with Jetpack

is the ViewModel component.

41.4 The ViewModel Component

The purpose of ViewModel is to separate the user interface-related data model and logic of an app from the code responsible for displaying and managing the user interface and interacting with the operating system. When designed this way, an app will consist of one or more UI Controllers, such as an activity, together with ViewModel instances responsible for handling the data those controllers need.

The ViewModel only knows about the data model and corresponding logic. It knows nothing about the user interface and does not attempt to directly access or respond to events relating to views within the user interface. When a UI controller needs data to display, it asks the ViewModel to provide it. Similarly, when the user enters data into a view within the user interface, the UI controller passes it to the ViewModel for handling.

This separation of responsibility addresses the issues relating to the lifecycle of UI controllers. Regardless of how often the UI controller is recreated during the lifecycle of an app, the ViewModel instances remain in memory, thereby maintaining data consistency. For example, a ViewModel used by an activity will remain in memory until the activity finishes, which, in the single activity app, is not until the app exits.



Figure 41-1

41.5 The LiveData Component

Consider an app that displays real-time data, such as the current price of a financial stock. The app could use a stock price web service to continuously update the data model within the ViewModel with the latest information. This real-time data is of use only if it is displayed to the user promptly. There are only two ways that the UI controller can ensure that the latest data is displayed in the user interface. One option is for the controller to continuously check with the ViewModel to determine if the data has changed since it was last displayed. However, the problem with this approach is that it could be more efficient. To maintain the real-time nature of the data feed, the UI controller would have to run on a loop, continuously checking for the data to change.

A better solution would be for the UI controller to receive a notification when a specific data item within a ViewModel changes. This is made possible by using the LiveData component. LiveData is a data holder that allows a value to become *observable*. In basic terms, an observable object can notify other objects when changes to its data occur, thereby solving the problem of ensuring that the user interface always matches the data within the ViewModel.

This means, for example, that a UI controller interested in a ViewModel value can set up an observer, which will, in turn, be notified when that value changes. In our hypothetical application, for example, the stock price would

be wrapped in a LiveData object within the ViewModel, and the UI controller would assign an observer to the value, declaring a method to be called when the value changes. When triggered by data change, this method will read the updated value from the ViewModel and use it to update the user interface.



Figure 41-2

A LiveData instance may also be declared as mutable, allowing the observing entity to update the underlying value held within the LiveData object. The user might, for example, enter a value in the user interface that needs to overwrite the value stored in the ViewModel.

Another of the key advantages of using LiveData is that it is aware of the *lifecycle state* of its observers. If, for example, an activity contains a LiveData observer, the corresponding LiveData object will know when the activity's lifecycle state changes and respond accordingly. If the activity is paused (perhaps the app is put into the background), the LiveData object will stop sending events to the observer. Suppose the activity has just started or resumes after being paused. In that case, the LiveData object will send a LiveData event to the observer so that the activity has the most up-to-date value. Similarly, the LiveData instance will know when the activity is destroyed and remove the observer to free up resources.

So far, we've only talked about UI controllers using observers. In practice, however, an observer can be used within any object that conforms to the Jetpack approach to lifecycle management.

41.6 ViewModel Saved State

Android allows the user to place an active app in the background and return to it after performing other tasks on the device (including running other apps). When a device runs low on resources, the operating system will rectify this by terminating background app processes, starting with the least recently used app. However, when the user returns to the terminated background app, it should appear in the same state as when it was placed in the background, regardless of whether it was terminated. In terms of the data associated with a ViewModel, this can be implemented using the ViewModel Saved State module. This module allows values to be stored in the app's *saved state* and restored in case of system-initiated process termination. This topic will be covered later in the *"An Android ViewModel Saved State Tutorial"* chapter.

41.7 LiveData and Data Binding

Android Jetpack includes the Data Binding Library, which allows data in a ViewModel to be mapped directly to specific views within the XML user interface layout file. In the AndroidSample project created earlier, code had to be written to obtain references to the EditText and TextView views and to set and get the text properties to

Modern Android App Architecture with Jetpack

reflect data changes. Data binding allows the LiveData value stored in the ViewModel to be referenced directly within the XML layout file avoiding the need to write code to keep the layout views updated.



Figure 41-3

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

41.8 Android Lifecycles

The duration from when an Android component is created to the point that it is destroyed is called the *lifecycle*. During this lifecycle, the component will change between different lifecycle states, usually under the operating system's control and in response to user actions. An activity, for example, will begin in the *initialized* state before transitioning to the *created* state. Once the activity runs, it will switch to the *started* state, from which it will cycle through various states, including *created*, *started*, *resumed*, and *destroyed*.

Many Android Framework classes and components allow other objects to access their current state. *Lifecycle observers* may also be used so that an object receives a notification when the lifecycle state of another object changes. The ViewModel component uses this technique behind the scenes to identify when an observer has restarted or been destroyed. This functionality is not limited to Android framework and architecture components. It may also be built into any other classes using a set of lifecycle components included with the architecture components.

Objects that can detect and react to lifecycle state changes in other objects are said to be *lifecycle-aware*. In contrast, objects that provide access to their lifecycle state are called *lifecycle owners*. The chapter entitled *"Working with Android Lifecycle-Aware Components"* will cover Lifecycles in greater detail.

41.9 Repository Modules

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

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


Figure 41-4

41.10 Summary

Until recently, Google has tended not to recommend any particular approach to structuring an Android app. That has now changed with the introduction of Android Jetpack, consisting of tools, components, libraries, and architecture guidelines. Google now recommends that an app project be divided into separate modules, each responsible for a particular area of functionality, otherwise known as "separation of concerns".

In particular, the guidelines recommend separating the view data model of an app from the code responsible for handling the user interface. In addition, the code responsible for gathering data from data sources such as web services or databases should be built into a separate repository module instead of being bundled with the view model.

Android Jetpack includes the Android Architecture Components, designed to make developing apps that conform to the recommended guidelines easier. This chapter has introduced the ViewModel, LiveData, and Lifecycle components. These will be covered in more detail, starting with the next chapter. Other architecture components not mentioned in this chapter will be covered later in the book.

Chapter 51

51. An Introduction to MotionLayout

The MotionLayout class provides an easy way to add animation effects to the views of a user interface layout. This chapter will begin by providing an overview of MotionLayout and introduce the concepts of MotionScenes, Transitions, and Keyframes. Once these basics have been covered, the next two chapters (entitled "An Android MotionLayout Editor Tutorial" and "A MotionLayout KeyCycle Tutorial") will provide additional detail and examples of MotionLayout animation in action through the creation of example projects.

51.1 An Overview of MotionLayout

MotionLayout is a layout container, the primary purpose of which is to animate the transition of views within a layout from one state to another. MotionLayout could, for example, animate the motion of an ImageView instance from the top left-hand corner of the screen to the bottom right-hand corner over a specified time. In addition to the position of a view, other attribute changes may also be animated, such as the color, size, or rotation angle. These state changes can also be interpolated (such that a view moves, rotates, and changes size throughout the animation).

The motion of a view using MotionLayout may be performed in a straight line between two points or implemented to follow a path comprising intermediate points at different positions between the start and end points. MotionLayout also supports using touches and swipes to initiate and control animation.

MotionLayout animations are declared entirely in XML and do not typically require writing code. These XML declarations may be implemented manually in the Android Studio code editor, visually using the MotionLayout editor, or combining both approaches.

51.2 MotionLayout

When implementing animation, the ConstraintLayout container typically used in a user interface must first be converted to a MotionLayout instance (a task which can be achieved by right-clicking on the ConstraintLayout in the layout editor and selecting the *Convert to MotionLayout* menu option). MotionLayout also requires at least version 2.0.0 of the ConstraintLayout library.

Unsurprisingly since it is a subclass of ConstraintLayout, MotionLayout supports all of the layout features of the ConstraintLayout. Therefore, a user interface layout can be similarly designed when using MotionLayout for views that do not require animation.

For views that are to be animated, two ConstraintSets are declared, defining the appearance and location of the view at the start and end of the animation. A *transition* declaration defines *keyframes* to apply additional effects to the target view between these start and end states and click and swipe handlers used to start and control the animation.

The start and end ConstraintSets and the transitions are declared within a MotionScene XML file.

51.3 MotionScene

As we have seen in earlier chapters, an XML layout file contains the information necessary to configure the appearance and layout behavior of the static views presented to the user, and this is still the case when using MotionLayout. For non-static views (in other words, the views that will be animated), those views are still declared within the layout file, but the start, end, and transition declarations related to those views are stored in a separate XML file referred to as the MotionScene file (so called because all of the declarations are defined

An Introduction to MotionLayout

within a MotionScene element). This file is imported into the layout XML file and contains the start and end ConstraintSets and Transition declarations (a single file can contain multiple ConstraintSet pairs and Transition declarations, allowing different animations to be targeted to specific views within the user interface layout).

The following listing shows a template for a MotionScene file:

```
<?xml version="1.0" encoding="utf-8"?>
<MotionScene
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:motion="http://schemas.android.com/apk/res-auto">
    <Transition
        motion:constraintSetEnd="@+id/end"
        motion:constraintSetStart="@id/start"
        motion:duration="1000">
       <KeyFrameSet>
       </KeyFrameSet>
    </Transition>
    <ConstraintSet android:id="@+id/start">
    </ConstraintSet>
    <ConstraintSet android:id="@+id/end">
    </ConstraintSet>
</MotionScene>
```

In the above XML, ConstraintSets named *start* and *end* (though any name can be used) have been declared, which, at this point, are yet to contain any constraint elements. The Transition element defines that these ConstraintSets represent the animation start and end points and contain an empty KeyFrameSet element ready to be populated with additional animation keyframe entries. The Transition element also includes a millisecond duration property to control the running time of the animation.

ConstraintSets do not have to imply the motion of a view. It is possible to have the start and end sets declare the same location on the screen and then use the transition to animate other property changes, such as scale and rotation angle.

ConstraintSets do not have to imply the motion of a view. It is possible, for example, to have the start and end sets declare the same location on the screen and then use the transition to animate other property changes, such as scale and rotation angle.

51.4 Configuring ConstraintSets

The ConstraintSets in the MotionScene file allow the full set of ConstraintLayout settings to be applied to a view regarding positioning, sizing, and relation to the parent and other views. In addition, the following attributes may also be included within the ConstraintSet declarations:

- alpha
- visibility
- elevation
- rotation

- rotationX
- rotationY
- translationX
- translationY
- translationZ
- scaleX
- scaleY

For example, to rotate the view by 180° during the animation, the following could be declared within the start and end constraints:

```
<ConstraintSet android:id="@+id/start">

<Constraint

.

.

.

motion:layout_constraintStart_toStartOf="parent"

android:rotation="0">

</ConstraintSet

</ConstraintSet

<ConstraintSet android:id="@+id/end">

<ConstraintSet

<ConstraintSet android:id="@+id/end">

<ConstraintSet

</constraintSet

.

motion:layout_constraintBottom_toBottomOf="parent"

android:rotation="180">

</ConstraintSet>
```

The above changes tell MotionLayout that the view is to start at 0° and then, during the animation, rotate a full 180° before coming to rest upside-down.

51.5 Custom Attributes

In addition to the standard attributes listed above, it is possible to specify a range of *custom attributes* (declared using CustomAttribute). In fact, just about any property available on the view type can be specified as a custom attribute for inclusion in an animation. To identify the attribute's name, find the getter/setter name from the documentation for the target view class, remove the get/set prefix, and lower the case of the first remaining character. For example, to change the background color of a Button view in code, we might call the *setBackgroundColor()* setter method as follows:

myButton.setBackgroundColor(Color.RED)

When setting this attribute in a constraint set or keyframe, the attribute name will be *backgroundColor*. In addition to the attribute name, the value must also be declared using the appropriate type from the following list of options:

• motion:customBoolean - Boolean attribute values.

An Introduction to MotionLayout

- motion:customColorValue Color attribute values.
- motion:customDimension Dimension attribute values.
- motion:customFloatValue Floating point attribute values.
- motion:customIntegerValue Integer attribute values.
- motion:customStringValue String attribute values

For example, a color setting will need to be assigned using the customColorValue type:

```
<CustomAttribute
motion:attributeName="backgroundColor"
motion:customColorValue="#43CC76" />
```

The following excerpt from a MotionScene file, for example, declares start and end constraints for a view in addition to changing the background color from green to red:

```
<ConstraintSet android:id="@+id/start">
     <Constraint
         android: layout width="wrap content"
         android: layout height="wrap content"
         motion:layout_editor absoluteX="21dp"
         android:id="@+id/button"
         motion:layout constraintTop toTopOf="parent"
         motion:layout constraintStart toStartOf="parent" >
         <CustomAttribute
             motion:attributeName="backgroundColor"
             motion:customColorValue="#33CC33" />
     </Constraint>
 </ConstraintSet>
 <ConstraintSet android:id="@+id/end">
     <Constraint
         android: layout width="wrap content"
         android: layout height="wrap content"
         motion:layout editor absoluteY="21dp"
         android:id="@+id/button"
         motion:layout constraintEnd toEndOf="parent"
         motion:layout constraintBottom toBottomOf="parent" >
         <CustomAttribute
             motion:attributeName="backgroundColor"
             motion:customColorValue="#F80A1F" />
     </Constraint>
 </ConstraintSet>
```

51.6 Triggering an Animation

Without some event to tell MotionLayout to start the animation, none of the settings in the MotionScene file will affect the layout (except that the view will be positioned based on the setting in the start ConstraintSet).

The animation can be configured to start in response to either screen tap (OnClick) or swipe motion (OnSwipe) gesture. The OnClick handler causes the animation to start and run until completion, while OnSwipe will synchronize the animation to move back and forth along the timeline to match the touch motion. The OnSwipe handler will also respond to "flinging" motions on the screen. The OnSwipe handler also provides options to configure how the animation reacts to dragging in different directions and the side of the target view to which the swipe is to be anchored. This allows, for example, left-ward dragging motions to move a view in the corresponding direction while preventing an upward motion from causing a view to move sideways (unless, of course, that is the required behavior).

The OnSwipe and OnClick declarations are contained within the Transition element of a MotionScene file. In both cases, the view id must be specified. For example, to implement an OnSwipe handler responding to downward drag motions anchored to the bottom edge of a view named *button*, the following XML would be placed in the Transition element:

```
.
<Transition
motion:constraintSetEnd="@+id/end"
motion:constraintSetStart="@id/start"
motion:duration="1000">
<KeyFrameSet>
</KeyFrameSet>
</KeyFrameSet>
<OnSwipe
motion:touchAnchorId="@+id/button"
motion:dragDirection="dragDown"
motion:touchAnchorSide="bottom" />
</Transition>
.
```

Alternatively, to add an OnClick handler to the same button:

```
<OnClick motion:targetId="@id/button"
motion:clickAction="toggle" />
```

In the above example, the action has been set to *toggle* mode. This mode and the other available options can be summarized as follows:

- **toggle** Animates to the opposite state. For example, if the view is currently at the transition start point, it will transition to the end point, and vice versa.
- jumpToStart Changes immediately to the start state without animation.
- jumpToEnd Changes immediately to the end state without animation.
- transitionToStart Transitions with animation to the start state.
- transitionToEnd Transitions with animation to the end state.

An Introduction to MotionLayout

51.7 Arc Motion

By default, a movement of view position will travel in a straight line between the start and end points. To change the motion to an arc path, use the *pathMotionArc* attribute as follows within the start constraint, configured with either a *startHorizontal* or *startVertical* setting to define whether the arc is to be concave or convex:

```
<ConstraintSet android:id="@+id/start">
<Constraint
android:layout_width="wrap_content"
android:layout_height="wrap_content"
motion:layout_editor_absoluteX="21dp"
android:id="@+id/button"
motion:layout_constraintTop_toTopOf="parent"
motion:layout_constraintStart_toStartOf="parent"
motion:layout_constraintStart_toStartOf="parent"
```

Figure 51-1 illustrates startVertical and startHorizontal arcs in comparison to the default straight line motion:





51.8 Keyframes

All of the ConstraintSet attributes outlined so far only apply to the start and end points of the animation. In other words, if the rotation property were set to 180° on the end point, the rotation would begin when the animation starts and complete when the end point is reached. It is not, therefore, possible to configure the rotation to reach the full 180° at a point 50% of the way through the animation and then rotate back to the original orientation by the end. Fortunately, this type of effect is available using Keyframes.

Keyframes are used to define intermediate points during the animation at which state changes are to occur. Keyframes could, for example, be declared such that the background color of a view is to have transitioned to blue at a point 50% of the way through the animation, green at the 75% point, and then back to the original color by the end of the animation. Keyframes are implemented within the Transition element of the MotionScene file embedded into the KeyFrameSet element.

MotionLayout supports several types of Keyframe which can be summarized as follows:

51.8.1 Attribute Keyframes

Attribute Keyframes (declared using KeyAttribute) allow view attributes to be changed at intermediate points in the animation timeline. KeyAttribute supports the attributes listed above for ConstraintSets combined with the ability to specify where the change will take effect in the animation timeline. For example, the following Keyframe declaration will gradually cause the button view to double in size horizontally (scaleX) and vertically (scaleY), reaching full size at 50% through the timeline. For the remainder of the timeline, the view will decrease in size to its original dimensions:

```
<Transition

motion:constraintSetEnd="@+id/end"

motion:constraintSetStart="@id/start"

motion:duration="1000">

<KeyFrameSet>

<KeyAttribute

motion:motionTarget="@+id/button"

motion:framePosition="50"

android:scaleX="2.0" />

<KeyAttribute
```

```
motion:motionTarget="@+id/button"
motion:framePosition="50"
android:scaleY="2.0" />
```

```
</KeyFrameSet>
```

51.8.2 Position Keyframes

Position keyframes (KeyPosition) modify the path followed by a view as it moves between the start and end locations. By placing key positions at different points on the timeline, a path of just about any level of complexity can be applied to an animation. Positions are declared using x and y coordinates combined with the corresponding points in the transition timeline. These coordinates must be declared relative to one of the following coordinate systems:

• **parentRelative** - The x and y coordinates are relative to the parent container where the coordinates are specified as a percentage (represented as a value between 0.0 and 1.0):



Figure 51-2

An Introduction to MotionLayout

• **deltaRelative** - Instead of relative to the parent, the x and y coordinates are relative to the start and end positions. For example, the start point is (0, 0) the end point (1, 1). Keep in mind that the x and y coordinates can be negative values):



Figure 51-3

• **pathRelative** - The x and y coordinates are relative to the path, where the straight line between the start and end points serves as the graph's X-axis. Once again, coordinates are represented as a percentage (0.0 to 1.0). This is similar to the deltaRelative coordinate space but takes into consideration the angle of the path. Once again coordinates may be negative:



Figure 51-4

Chapter 63

63. An Introduction to Kotlin Coroutines

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

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

63.1 What are Coroutines?

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

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

63.2 Threads vs. Coroutines

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

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

An Introduction to Kotlin Coroutines

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

63.3 Coroutine Scope

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

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

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

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

private val myCoroutineScope = CoroutineScope(Dispatchers.Main)

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

myCoroutineScope.cancel()

63.4 Suspend Functions

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

The following is an example suspend function:

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

63.5 Coroutine Dispatchers

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

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

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

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

63.6 Coroutine Builders

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

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

63.7 Jobs

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

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

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

An Introduction to Kotlin Coroutines

method.

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

63.8 Coroutines - Suspending and Resuming

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

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

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

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

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

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

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

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

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

Chapter 70

70. An Overview of Android SQLite Databases

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

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

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

70.1 Understanding Database Tables

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

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

70.2 Introducing Database Schema

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

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

70.3 Columns and Data Types

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

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

Each column, in turn, is defined to contain a certain type of data. Therefore, a column designed to store numbers would be defined as containing numerical data.

An Overview of Android SQLite Databases

70.4 Database Rows

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

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

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

70.5 Introducing Primary Keys

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

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

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

70.6 What is SQLite?

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

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

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

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

70.7 Structured Query Language (SQL)

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

SQL is a straightforward and easy-to-use language designed specifically to enable the reading and writing of database data. Because SQL contains a small set of keywords, it can be learned quickly. In addition, SQL syntax is

more or less identical between most DBMS implementations, so having learned SQL for one system, your skills will likely transfer to other database management systems.

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

70.8 Trying SQLite on an Android Virtual Device (AVD)

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

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

Once the AVD is up and running, open a Terminal or Command-Prompt window and connect to the emulator using the *adb* command-line tool as follows:

```
adb shell
```

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

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

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

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

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

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

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

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

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

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

```
root@android:/data/data/databases # sqlite3 ./mydatabase.db
sqLite version 3.8.10.2 2015-05-20 18:17:19
```

An Overview of Android SQLite Databases

```
Enter ".help" for usage hints.
sqlite>
```

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

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

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

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

```
sqlite> .tables contacts
```

To insert records into the table:

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

To retrieve all rows from a table:

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

To extract a row that meets specific criteria:

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

To exit from the sqlite3 interactive environment:

sqlite> .exit

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

It is also important to note that while connecting with an adb shell to a physical Android device is possible, the shell is not granted sufficient privileges by default to create and manage SQLite databases. Therefore, database problem debugging is best performed using an AVD session.

70.9 Android SQLite Classes

As previously mentioned, SQLite is written in the C programming language, while Android applications are primarily developed using Java or Kotlin. To bridge this "language gap", the Android SDK includes a set of classes that provide a programming layer on top of the SQLite database management system. The remainder of this chapter will provide a basic overview of each of the major classes within this category.

70.9.1 Cursor

A class provided specifically to access the results of a database query. For example, a SQL SELECT operation performed on a database will potentially return multiple matching rows from the database. A Cursor instance can be used to step through these results, which may then be accessed from within the application code using a variety of methods. Some key methods of this class are as follows:

- close() Releases all resources used by the cursor and closes it.
- getCount() Returns the number of rows contained within the result set.
- moveToFirst() Moves to the first row within the result set.
- moveToLast() Moves to the last row in the result set.
- moveToNext() Moves to the next row in the result set.
- move() Moves by a specified offset from the current position in the result set.
- get<type>() Returns the value of the specified <type> contained at the specified column index of the row at the current cursor position (variations consist of getString(), getInt(), getShort(), getFloat(), and getDouble()).

70.9.2 SQLiteDatabase

This class provides the primary interface between the application code and underlying SQLite databases including the ability to create, delete, and perform SQL-based operations on databases. Some key methods of this class are as follows:

- insert() Inserts a new row into a database table.
- delete() Deletes rows from a database table.
- query() Performs a specified database query and returns matching results via a Cursor object.
- execSQL() Executes a single SQL statement that does not return result data.
- rawQuery() Executes a SQL query statement and returns matching results in the form of a Cursor object.

70.9.3 SQLiteOpenHelper

A helper class designed to make it easier to create and update databases. This class must be subclassed within the code of the application seeking database access and the following callback methods implemented within that subclass:

- **onCreate()** Called when the database is created for the first time. This method is passed the SQLiteDatabase object as an argument for the newly created database. This is the ideal location to initialize the database in terms of creating a table and inserting any initial data rows.
- **onUpgrade**() Called in the event that the application code contains a more recent database version number reference. This is typically used when an application is updated on the device and requires that the database schema also be updated to handle storage of additional data.

In addition to the above mandatory callback methods, the *onOpen()* method, called when the database is opened, may also be implemented within the subclass.

The constructor for the subclass must also be implemented to call the super class, passing through the application context, the name of the database and the database version.

An Overview of Android SQLite Databases

Notable methods of the SQLiteOpenHelper class include:

- getWritableDatabase() Opens or creates a database for reading and writing. Returns a reference to the database in the form of a SQLiteDatabase object.
- getReadableDatabase() Creates or opens a database for reading only. Returns a reference to the database in the form of a SQLiteDatabase object.
- **close()** Closes the database.

70.9.4 ContentValues

ContentValues is a convenience class that allows key/value pairs to be declared consisting of table column identifiers and the values to be stored in each column. This class is of particular use when inserting or updating entries in a database table.

70.10 The Android Room Persistence Library

A limitation of the Android SDK SQLite classes is that they require moderate coding effort and don't take advantage of the new architecture guidelines and features such as LiveData and lifecycle management. The Android Jetpack Architecture Components include the Room persistent library to address these shortcomings. This library provides a high-level interface on top of the SQLite database system, making it easy to store data locally on Android devices with minimal coding while also conforming to the recommendations for modern application architecture.

The following chapters will provide an overview and tutorial on SQLite database management using SQLite and the Room persistence library.

70.11 Summary

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

This chapter provided an overview of databases in general and SQLite in particular within the context of Android application development.

Chapter 93

93. An Overview of Android In-App Billing

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

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

93.1 Preparing a Project for In-App Purchasing

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

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

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

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

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

93.2 Creating In-App Products and Subscriptions

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



Figure 93-1

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

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

93.3 Billing Client Initialization

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

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

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

```
// Handle errors here
}
billingClient = BillingClient.newBuilder(this)
.setListener(purchasesUpdatedListener)
.enablePendingPurchases(
        PendingPurchasesParams.newBuilder()
        .enableOneTimeProducts().build()
)
.build()
```

93.4 Connecting to the Google Play Billing Library

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

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

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

93.5 Querying Available Products

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

```
val queryProductDetailsParams = QueryProductDetailsParams.newBuilder()
```

An Overview of Android In-App Billing

}

93.8 Querying Previous Purchases

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

```
val queryPurchasesParams = QueryPurchasesParams.newBuilder()
    .setProductType(BillingClient.ProductType.INAPP)
    .build()
billingClient.queryPurchasesAsync(
    queryPurchasesParams,
    purchasesListener
)
.
.
private val purchasesListener =
PurchasesResponseListener { billingResult, purchases ->
    if (!purchases.isEmpty()) {
        // Access existing active purchases
    } else {
        // No
     }
}
```

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

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

```
val queryPurchaseHistoryParams = QueryPurchaseHistoryParams.newBuilder()
    .setProductType(BillingClient.ProductType.INAPP)
    .build()
billingClient.queryPurchaseHistoryAsync(queryPurchaseHistoryParams) {
    billingResult, historyList ->
        // Process purchase history list
}
```

93.9 Summary

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

Index

Symbols

?. 97
<application> 514
<fragment> 305
<fragment> element 305
<provider> 571
<receiver> 492
<service> 514, 520, 527
:: operator 99
.well-known folder 465, 488

A

AbsoluteLayout 182 ACCESS_COARSE_LOCATION permission 642 ACCESS_FINE_LOCATION permission 642 acknowledgePurchase() method 701 ACTION_DOWN 282 ACTION_MOVE 282 ACTION_POINTER_DOWN 282 ACTION_POINTER_UP 282 ACTION_UP 282 ACTION_VIEW 483 Active / Running state 158 Activity 83, 161 adding views in code 259 class 161 creation 14 Entire Lifetime 165 Foreground Lifetime 165 lifecycle methods 163 lifecycles 155 returning data from 462 state change example 169 state changes 161 states 158

Visible Lifetime 165 Activity Lifecycle 157 Activity Manager 82 ActivityResultLauncher 463 Activity Stack 157 Actual screen pixels 250 adb command-line tool 59 connection testing 65 device pairing 63 enabling on Android devices 59 Linux configuration 62 list devices 59 macOS configuration 60 overview 59 restart server 60 testing connection 65 WiFi debugging 63 Windows configuration 61 Wireless debugging 63 Wireless pairing 63 addCategory() method 491 addView() method 253 ADD_VOICEMAIL permission 642 android exported 515 gestureColor 298 layout_behavior property 455 onClick 307 process 515, 527 uncertainGestureColor 298 Android Activity 83 architecture 79 events 275 intents 84 onClick Resource 275 runtime 80

android.app 80 Android Architecture Components 321 android.content 80 android.content.Intent 461 android.database 80 Android Debug Bridge. See ADB Android Development System Requirements 3 Android Devices designing for different 181 android.graphics 81 android.hardware 81 android.intent.action 497 android.intent.action.BOOT_COMPLETED 515 android.intent.action.MAIN 483 android.intent.category.LAUNCHER 483 Android Libraries 80 android.media 81 Android Monitor tool window 32 Android Native Development Kit 81 android.net 81 android.opengl 81 android.os 81 android.permission.RECORD_AUDIO 651 android.print 81 Android Project create new 13 android.provider 81 Android SDK Location identifying 9 Android SDK Manager 7, 9 Android SDK Packages version requirements 7 Android SDK Tools command-line access 8 Linux 10 macOS 10 Windows 7 9 Windows 8 9 Android Software Stack 79 Android Studio changing theme 57

downloading 3 Editor Window 52 installation 4 Linux installation 5 macOS installation 4 Navigation Bar 51 Project tool window 52 Status Bar 52 Toolbar 51 Tool window bars 52 tool windows 52 updating 11 Welcome Screen 49 Windows installation 4 android.text 81 android.util 81 android.view 81 android.view.View 184 android.view.ViewGroup 181, 184 Android Virtual Device. See AVD overview 27 Android Virtual Device Manager 27 android.webkit 81 android.widget 81 AndroidX libraries 730 APK analyzer 694 APK file 687 APK File analyzing 694 APK Signing 730 APK Wizard dialog 686 App Architecture modern 321 AppBar anatomy of 453 appbar_scrolling_view_behavior 455 App Bundles 683 creating 687 overview 683 revisions 693 uploading 690 AppCompatActivity class 162

App Inspector 53 Application stopping 32 Application Context 85 Application Framework 82 Application Manifest 85 Application Resources 85 App Link Digital Asset Links file 465 App Links auto verification 464 autoVerify 465 Apply Changes 267 Apply Changes and Restart Activity 267 Apply Code Changes 267 fallback settings 269 options 267 Run App 267 tutorial 269 applyToActivitiesIfAvailable() method 726 Architecture Components 321 ART 80 as 99 as? 99 asFlow() builder 533 assetlinks.json, 465 asSharedFlow() 542 asStateFlow() 541 async 501 Attribute Keyframes 392 Audio supported formats 649 Audio Playback 649 Audio Recording 649 Auto Blocker 60 Autoconnect Mode 215 Automatic Link Verification 464, 487 autoVerify 465 AVD Change posture 48 cold boot 44 command-line creation 27

creation 27 device frame 35 Display mode 47 launch in tool window 35 overview 27 quickboot 44 Resizable 47 running an application 30 Snapshots 43 standalone 32 starting 29 Startup size and orientation 30

B

Background Process 156 Barriers 208 adding 227 constrained views 208 Baseline Alignment 207 beginTransaction() method 306 BillingClient 702 acknowledgePurchase() method 701 consumeAsync() method 701 getPurchaseState() method 701 initialization 698, 706 launchBillingFlow() method 700 queryProductDetailsAsync() method 699 queryPurchasesAsync() method 702 BillingResult 713 getDebugMessage() 713 Binding Expressions 341 one-way 341 two-way 342 BIND_JOB_SERVICE permission 515 bindService() method 513, 517, 521 Bitwise AND 105 Bitwise Inversion 104 Bitwise Left Shift 106 Bitwise OR 105 Bitwise Right Shift 106 Bitwise XOR 105 black activity 14

Blank template 185 Blueprint view 213 BODY_SENSORS permission 642 Boolean 92 Bound Service 513, 517 adding to a project 518 Implementing the Binder 518 Interaction options 517 BoundService class 519 Broadcast Intent 491 example 493 overview 84, 491 sending 494 Sticky 493 Broadcast Receiver 491 adding to manifest file 496 creation 495 overview 84, 492 BroadcastReceiver class 492 BroadcastReceiver superclass 495 buffer() operator 535 Build Variants, 54 tool window 54 Bundle class 178 Bundled Notifications 670

С

Calendar permissions 642 CALL_PHONE permission 642 CAMERA permission 642 Camera permissions 642 cancelAndJoin() 501 cancelChildren() 501 CardView layout file 443 responding to selection of 451 CardView class 443 C/C++ Libraries 81 Chain bias 236 chain head 206 chains 206 Chains

creation of 233 Chain style changing 235 chain styles 206 Char 92 CheckBox 181 checkSelfPermission() method 646 Code completion 70 Code Editor basics 67 Code completion 70 Code Generation 72 Code Reformatting 75 Document Tabs 68 Editing area 68 Gutter Area 68 Live Templates 76 Splitting 70 Statement Completion 72 Status Bar 69 Code Generation 72 Code Reformatting 75 code samples download 1 cold boot 44 Cold flows 541 CollapsingToolbarLayout example 456 introduction 456 parallax mode 456 pin mode 456 setting scrim color 459 setting title 459 with image 456 collectLatest() operator 534 combine() operator 540 Common Gestures 287 detection 287 Communicating Sequential Processes 499 Companion Objects 129 Component tree 17 conflate() operator 535

Constraint Bias 205 adjusting 219 ConstraintLayout advantages of 211 Availability 212 Barriers 208 Baseline Alignment 207 chain bias 236 chain head 206 chains 206 chain styles 206 Constraint Bias 205 Constraints 203 conversion to 231 convert to MotionLayout 399 deleting constraints 218 guidelines 225 Guidelines 208 manual constraint manipulation 215 Margins 204, 219 Opposing Constraints 204, 221 overview of 203 Packed chain 207, 236 ratios 211, 237 Spread chain 206 Spread inside 236 Spread inside chain 206 tutorial 241 using in Android Studio 213 Weighted chain 206, 236 Widget Dimensions 207, 223 Widget Group Alignment 229 ConstraintLayout chains creation of 233 in layout editor 233 ConstraintLayout Chain style changing 235 Constraints deleting 218 ConstraintSet addToHorizontalChain() method 256 addToVerticalChain() method 256

alignment constraints 255 apply to layout 254 applyTo() method 254 centerHorizontally() method 255 centerVertically() method 255 chains 255 clear() method 256 clone() method 255 connect() method 254 connect to parent 254 constraint bias 255 copying constraints 255 create 254 create connection 254 createHorizontalChain() method 255 createVerticalChain() method 255 guidelines 256 removeFromHorizontalChain() method 256 removeFromVerticalChain() method 256 removing constraints 256 rotation 257 scaling 256 setGuidelineBegin() method 256 setGuidelineEnd() method 256 setGuidelinePercent() method 256 setHorizonalBias() method 255 setRotationX() method 257 setRotationY() method 257 setScaleX() method 256 setScaleY() method 256 setTransformPivot() method 257 setTransformPivotX() method 257 setTransformPivotY() method 257 setVerticalBias() method 255 sizing constraints 255 tutorial 259 view IDs 261 ConstraintSet class 253, 254 Constraint Sets 254 ConstraintSets configuring 388 consumeAsync() method 701

ConsumeParams 711 Contacts permissions 642 container view 181 Content Provider 82, 569, 585 <provider> 571 accessing 585 Authority 575 client tutorial 585 ContentProvider class 569 Content Resolver 570 ContentResolver 582 content URI 570 Content URI 575, 585 ContentValues 577 delete() 570, 580 getType() 570 insert() 569, 577 onCreate() 569, 577 overview 85 query() 569, 578 tutorial 573 update() 570, 579 UriMatcher 576 UriMatcher class 570 ContentProvider class 569 Content Resolver 570 getContentResolver() 570 ContentResolver 582 getContentResolver() 570 content URI 570 Content URI 570, 575 ContentValues 577 Context class 85 CoordinatorLayout 182, 455 Coroutine Builders 501 async 501 coroutineScope 501 launch 501 runBlocking 501 supervisorScope 501 withContext 501 Coroutine Dispatchers 500

Coroutines 499, 531 channel communication 505 GlobalScope 500 returning results 503 Suspend Functions 500 suspending 502 tutorial 507 ViewModelScope 500 vs. Threads 499 coroutineScope 501 Coroutine Scope 500 Custom Accessors 127 Custom Attribute 389 Custom Gesture recognition 293 Custom Theme building 717 Cycle Editor 417 Cycle Keyframe 397 Cycle Keyframes overview 413

D

dangerous permissions list of 642 Data Access Object (DAO) 590 Database Inspector 596, 620 live updates 620 SQL query 620 Database Rows 556 Database Schema 555 Database Tables 555 Data binding binding expressions 341 Data Binding 323 binding classes 340 enabling 346 event and listener binding 342 key components 337 overview 337 tutorial 345 variables 340

with LiveData 323 DDMS 32 Debugging enabling on device 59 debug.keystore file 465, 487 Default Function Parameters 119 DefaultLifecycleObserver 358, 361 deltaRelative 394 Density-independent pixels 249 Density Independent Pixels converting to pixels 264 Device Definition custom 199 Device File Explorer 54 device frame 35 Device Mirroring 65 enabling 65 device pairing 63 Digital Asset Links file 465 Direct Reply Input 679 Dispatchers.Default 501 Dispatchers.IO 500 Dispatchers.Main 500 dp 249 DROP_LATEST 543 DROP_OLDEST 543 Dynamic Colors applyToActivitiesIfAvailable() method 726 enabling in Android 725 Dynamic State 163 saving 177

E

Elvis Operator 99 Empty Process 157 Empty template 185 Emulator battery 42 cellular configuration 42 configuring fingerprints 44 directional pad 42 extended control options 41

Extended controls 41 fingerprint 42 location configuration 42 phone settings 42 Resizable 47 resize 41 rotate 40 Screen Record 43 Snapshots 43 starting 29 take screenshot 40 toolbar 39 toolbar options 39 tool window mode 46 Virtual Sensors 43 zoom 40 enablePendingPurchases() method 701 enabling ADB support 59 Escape Sequences 93 Event Handling 275 example 276 Event Listener 277 Event Listeners 276 Events consuming 279 execSQL() 564 explicit intent 84 explicit intent 461 Explicit Intent 461 Extended Control options 41

F

Files switching between 68 filter() operator 536 findPointerIndex() method 282 findViewById() 139 Fingerprint emulation 44 FLAG_INCLUDE_STOPPED_PACKAGES 491

flatMapConcat() operator 539 flatMapMerge() operator 539 flexible space area 453 Float 92 floating action button 14, 186 changing appearance of 428 margins 426 removing 187 sizes 426 Flow 531 asFlow() builder 533 asSharedFlow() 542 asStateFlow() 541 background handling 551 buffering 535 buffer() operator 535 cold 541 collect() 533 collecting data 533 collectLatest() operator 534 combine() operator 540 conflate() operator 535 declaring 532 emit() 533 emitting data 533 filter() operator 536 flatMapConcat() operator 539 flatMapMerge() operator 539 flattening 538 flowOf() builder 533 flow of flows 538 fold() operator 538 hot 541 intermediate operators 536 library requirements 532 map() operator 536 MutableSharedFlow 542 MutableStateFlow 541 onEach() operator 540 reduce() operator 538 repeatOnLifecycle 552 SharedFlow 542

single() operator 535 StateFlow 541 terminal flow operators 538 transform() operator 537 try/finally 534 zip() operator 540 flowOf() builder 533 flow of flows 538 Flow operators 536 Flows combining 540 Introduction to 531 Foldable Devices 166 multi-resume 166 Foreground Process 156 Fragment creation 303 event handling 307 XML file 304 FragmentActivity class 162 Fragment Communication 307 Fragments 303 adding in code 306 duplicating 434 example 311 overview 303 FragmentStateAdapter class 437 FrameLayout 182 **Function Parameters** variable number of 119 Functions 117

G

Gemini 145 asking questions 148 configuration 147 enabling 145 in Android Studio 145 inline code completion 149 overview 145 playground 148 proposed changes 151

question context 149 tool window 146 transforming code 150 Gesture Builder Application 293 building and running 293 Gesture Detector class 287 GestureDetectorCompat instance creation 290 GestureDetectorCompat class 287 GestureDetector.OnDoubleTapListener 287, 288 GestureDetector.OnGestureListener 288 GestureLibrary 293 GestureOverlayView 293 configuring color 298 configuring multiple strokes 298 GestureOverlayView class 293 GesturePerformedListener 293 Gestures interception of 298 Gestures File creation 294 extract from SD card 294 loading into application 296 GET_ACCOUNTS permission 642 getAction() method 497 getContentResolver() 570 getDebugMessage() 713 getId() method 254 getIntent() method 462 getPointerCount() method 282 getPointerId() method 282 getPurchaseState() method 701 getService() method 521 getWritableDatabase() 564 GlobalScope 500 GNU/Linux 80 Google Play App Signing 686 Google Play Console 705 Creating an in-app product 705 License Testers 705 Google Play Developer Console 684 Gradle

APK signing settings 734 Build Variants 730 command line tasks 735 dependencies 729 Manifest Entries 730 overview 729 sensible defaults 729 Gradle Build File top level 731 Gradle Build Files module level 732 gradle.properties file 730 GridLayout 182 GridLayoutManager 441

Η

HAL 80 Handler class 526 Hardware Abstraction Layer 80 Higher-order Functions 121 Hot flows 541

I

IBinder 513, 519 IBinder object 517, 526 Immutable Variables 94 implicit intent 84 implicit intent 461 Implicit Intent 463 Implicit Intents example 479 importance hierarchy 155 in 249 INAPP 702 In-App Products 697 In-App Purchasing 703 acknowledgePurchase() method 701 BillingClient 698 BillingResult 713 consumeAsync() method 701 ConsumeParams 711

Consuming purchases 710 enablePendingPurchases() method 701 getPurchaseState() method 701 launchBillingFlow() method 700 Libraries 703 newBuilder() method 698 onBillingServiceDisconnected() callback 708 onBillingServiceDisconnected() method 699 onBillingSetupFinished() listener 708 onProductDetailsResponse() callback 708 Overview 697 ProductDetail 700 ProductDetails 709 products 697 ProductType 702 Purchase Flow 709 PurchaseResponseListener 702 PurchasesUpdatedListener 701 PurchaseUpdatedListener 709 purchase updates 709 queryProductDetailsAsync() 708 queryProductDetailsAsync() method 699 queryPurchasesAsync() 711 queryPurchasesAsync() method 702 runOnUiThread() 709 subscriptions 697 tutorial 703 Initializer Blocks 127 In-Memory Database 596 Inner Classes 128 IntelliJ IDEA 87 Intent 84 explicit 84 implicit 84 Intent Availability checking for 468 Intent Filters 464 Intents 461 ActivityResultLauncher 463 overview 461 registerForActivityResult() 463, 476 Intent Service 513

Intent URL 481 intermediate flow operators 536 is 99 isInitialized property 99

J

Java convert to Kotlin 87 Java Native Interface 81 JetBrains 87 Jetpack 321 overview 321 JobIntentService 513 BIND_JOB_SERVICE permission 515 onHandleWork() method 513 join() 501

K

K2 mode 607 KevAttribute 392 Keyboard Shortcuts 56 KeyCycle 413 Cycle Editor 417 tutorial 413 Keyframe 406 Keyframes 392 KeyFrameSet 422 KeyPosition 393 deltaRelative 394 parentRelative 393 pathRelative 394 Keystore File creation 686 KeyTimeCycle 413 keytool 465 KeyTrigger 396 Killed state 158 Kotlin accessing class properties 127 and Java 87 arithmetic operators 101 assignment operator 101

augmented assignment operators 102 bitwise operators 104 Boolean 92 break 112 breaking from loops 111 calling class methods 127 Char 92 class declaration 123 class initialization 124 class properties 124 Companion Objects 129 conditional control flow 113 continue labels 112 continue statement 112 control flow 109 convert from Java 87 Custom Accessors 127 data types 91 decrement operator 102 Default Function Parameters 119 defining class methods 124 do ... while loop 111 Elvis Operator 99 equality operators 103 Escape Sequences 93 expression syntax 101 Float 92 Flow 531 for-in statement 109 function calling 118 Functions 117 Higher-order Functions 121 if ... else ... expressions 114 if expressions 113 Immutable Variables 94 increment operator 102 inheritance 133 Initializer Blocks 127 Inner Classes 128 introduction 87 Lambda Expressions 120 let Function 97

Local Functions 118 logical operators 103 looping 109 Mutable Variables 94 Not-Null Assertion 97 Nullable Type 96 Overriding inherited methods 136 playground 88 Primary Constructor 124 properties 127 range operator 104 Safe Call Operator 96 Secondary Constructors 124 Single Expression Functions 118 String 92 subclassing 133 Type Annotations 95 Type Casting 99 Type Checking 99 Type Inference 95 variable parameters 119 when statement 114 while loop 110

L

Lambda Expressions 120 Large Language Model 145 lateinit 98 Late Initialization 98 launch 501 launchBillingFlow() method 700 layout_collapseMode parallax 458 pin 458 layout_constraintDimentionRatio 238 layout_constraintHorizontal_bias 236 layout_constraintVertical_bias 236 layout editor ConstraintLayout chains 233 Layout Editor 16, 241 Autoconnect Mode 215 code mode 192

Component Tree 189 design mode 189 device screen 189 example project 241 Inference Mode 215 palette 189 properties panel 190 Sample Data 198 Setting Properties 193 toolbar 190 user interface design 241 view conversion 197 Layout Editor Tool changing orientation 17 overview 189 Layout Inspector 55 Layout Managers 181 Layouts 181 layout_scrollFlags enterAlwaysCollapsed mode 455 enterAlways mode 455 exitUntilCollapsed mode 455 scroll mode 455 Layout Validation 200 let Function 97 libc 81 libs.versions.toml file 272 License Testers 705 Lifecycle awareness 357 components 324 observers 358 owners 357 states and events 358 tutorial 361 Lifecycle-Aware Components 357 Lifecycle library 532 Lifecycle Methods 163 Lifecycle Observer 361 creating a 361 Lifecycle Owner creating a 363

Lifecycles modern 324 Lifecycle.State.CREATED 553 Lifecycle.State.DESTROYED 553 Lifecycle.State.INITIALIZED 553 Lifecycle.State.RESUMED 553 Lifecycle.State.STARTED 553 LinearLayout 182 LinearLayoutManager 441 LinearLayoutManager layout 449 Linux Kernel 80 list devices 59 LiveData 322, 333 adding to ViewModel 333 observer 335 tutorial 333 Live Templates 76 LLM 145 Local Bound Service 517 example 517 Local Functions 118 Location Manager 82 Location permission 642 Logcat tool window 54 LogCat enabling 173

Μ

MANAGE_EXTERNAL_STORAGE 643 adb enabling 643 testing 643 Manifest File permissions 483 map() operator 536 match_parent properties 249 Material design 425 Material Design 2 715 Material Design 2 Theming 715 Material Design 3 715 Material Theme Builder 717 Material You 715
measureTimeMillis() function 535 MediaController adding to VideoView instance 627 MediaController class 624 methods 624 MediaPlayer class 649 methods 649 MediaRecorder class 649 methods 650 recording audio 650 Memory Indicator 69 Messenger object 526 Microphone checking for availability 652 Microphone permissions 642 mm 249 MotionEvent 281, 282, 301 getActionMasked() 282 MotionLayout 387 arc motion 392 Attribute Keyframes 392 ConstraintSets 388 Custom Attribute 408 Custom Attributes 389 Cycle Editor 417 Editor 399 KeyAttribute 392 KeyCycle 413 Keyframes 392 KeyFrameSet 422 KeyPosition 393 KeyTimeCycle 413 KeyTrigger 396 OnClick 391, 404 OnSwipe 391 overview 387 Position Keyframes 393 previewing animation 404 Trigger Keyframe 396 Tutorial 399 MotionScene ConstraintSets 388

Custom Attributes 389 file 388 overview 387 transition 388 multiple devices testing app on 31 Multiple Touches handling 282 multi-resume 166 Multi-Touch example 283 Multi-touch Event Handling 281 multi-window support 166 MutableSharedFlow 542 MutableStateFlow 541 Mutable Variables 94

N

Navigation 367 adding destinations 376 overview 367 pass data with safeargs 383 passing arguments 372 stack 367 tutorial 373 Navigation Action triggering 371 Navigation Architecture Component 367 Navigation Component tutorial 373 Navigation Controller accessing 371 Navigation Graph 370, 374 adding actions 380 creating a 374 Navigation Host 368 declaring 375 newBuilder() method 698 normal permissions 641 Notification adding actions 670 Direct Reply Input 679

Index

issuing a basic 666 launch activity from a 668 PendingIntent 676 Reply Action 678 updating direct reply 680 Notifications bundled 670 overview 659 Notifications Manager 82 Not-Null Assertion 97 Nullable Type 96

0

Observer

implementing a LiveData 335 onAttach() method 308 onBillingServiceDisconnected() callback 708 onBillingServiceDisconnected() method 699 onBillingSetupFinished() listener 708 onBind() method 514, 517, 525 onBindViewHolder() method 449 OnClick 391 onClickListener 276, 277, 280 onClick() method 275 onCreateContextMenuListener 276 onCreate() method 156, 163, 514 onCreateView() method 164 onDestroy() method 164, 514 onDoubleTap() method 287 onDown() method 287 onEach() operator 540 onFling() method 287 onFocusChangeListener 276 OnFragmentInteractionListener implementation 381 onGesturePerformed() method 293 onHandleWork() method 514 onKeyListener 276 onLongClickListener 276 onLongPress() method 287 onPause() method 164 onProductDetailsResponse() callback 708

onReceive() method 156, 492, 493, 495 onRequestPermissionsResult() method 645, 656, 664, 674 onRestart() method 163 onRestoreInstanceState() method 164 onResume() method 156, 164 onSaveInstanceState() method 164 onScaleBegin() method 299 onScaleEnd() method 299 onScale() method 299 onScroll() method 287 OnSeekBarChangeListener 318 onServiceConnected() method 517, 520, 527 onServiceDisconnected() method 517, 520, 527 onShowPress() method 287 onSingleTapUp() method 287 onStartCommand() method 514 onStart() method 164 onStop() method 164 onTouchEvent() method 287, 299 onTouchListener 276 onTouch() method 282 onUpgrade() 564 onViewCreated() method 164 onViewStatusRestored() method 164 OpenJDK 3

P

Package Explorer 15 Package Manager 82 PackageManager class 652 PackageManager.FEATURE_MICROPHONE 652 PackageManager.PERMISSION_DENIED 643 PackageManager.PERMISSION_GRANTED 643 Package Name 14 Packed chain 207, 236 parentRelative 393 parent view 183 pathRelative 394 Paused state 158 PendingIntent class 676 Permission checking for 643

permissions normal 641 Persistent State 163 Phone permissions 642 Pinch Gesture detection 299 example 299 Pinch Gesture Recognition 293 Position Keyframes 393 POST_NOTIFICATIONS permission 642, 674 Primary Constructor 124 Problems tool window 54, 55 process priority 155 state 155 PROCESS_OUTGOING_CALLS permission 642 Process States 155 ProductDetail 700 ProductDetails 709 ProductType 702 Profiler tool window 55 ProgressBar 181 proguard-rules.pro file 734 ProGuard Support 730 Project Name 14 Project tool window 15, 53 pt 249 PurchaseResponseListener 702 PurchasesUpdatedListener 701 PurchaseUpdatedListener 709 putExtra() method 461, 491 px 250

Q

queryProductDetailsAsync() 708
queryPurchasesAsync() 711
quickboot snapshot 44
Quick Documentation 75

RadioButton 181 Range Operator 104 ratios 237 READ_CALENDAR permission 642 READ_CALL_LOG permission 642 READ_CONTACTS permission 642 READ_EXTERNAL_STORAGE permission 643 READ_PHONE_STATE permission 642 READ_SMS permission 642 RECEIVE_MMS permission 642 RECEIVE_SMS permission 642 RECEIVE_WAP_PUSH permission 642 Recent Files Navigation 56 RECORD_AUDIO permission 642 Recording Audio permission 651 RecyclerView 441 adding to layout file 442 GridLayoutManager 441 initializing 449 LinearLayoutManager 441 StaggeredGridLayoutManager 441 RecyclerView Adapter creation of 447 RecyclerView.Adapter 442, 448 getItemCount() method 442 onBindViewHolder() method 442 onCreateViewHolder() method 442 RecyclerView.ViewHolder getAdapterPosition() method 452 reduce() operator 538 registerForActivityResult() 463 registerForActivityResult() method 462, 476 registerReceiver() method 493 RelativeLayout 182 Release Preparation 683 Remote Bound Service 525 client communication 525 implementation 525 manifest file declaration 527 RemoteInput.Builder() method 676 RemoteInput Object 676

Index

Remote Service launching and binding 527 sending a message 529 repeatOnLifecycle 552 Repository tutorial 607 Repository Modules 324 Resizable Emulator 47 Resource string creation 20 Resource File 22 Resource Management 155 Resource Manager 53, 82 result receiver 493 Room Data Access Object (DAO) 590 entities 590, 591 In-Memory Database 596 Repository 590 Room Database 590 tutorial 607 Room Database Persistence 589 Room Persistence Library 560, 589 root element 181 root view 183 Run tool window 53 runBlocking 501 Running Devices tool window 65 runOnUiThread() 709

S

safeargs 383 Safe Call Operator 96 Sample Data 198 Saved State 323, 353 SavedStateHandle 354 contains() method 355 keys() method 355 remove() method 355 Saved State module 353 SavedStateViewModelFactory 354 ScaleGestureDetector class 299 Scale-independent 249 SDK Packages 5 Secondary Constructors 124 Secure Sockets Layer (SSL) 81 SeekBar 311 sendBroadcast() method 491, 493 sendOrderedBroadcast() method 491, 493 SEND_SMS permission 642 sendStickyBroadcast() method 491 Sensor permissions 642 Service anatomy 514 launch at system start 515 manifest file entry 514 overview 84 run in separate process 515 ServiceConnection class 527 Service Process 156 Service Restart Options 514 setAudioEncoder() method 650 setAudioSource() method 650 setBackgroundColor() 254 setContentView() method 253, 259 setId() method 254 setOnClickListener() method 275, 277 setOnDoubleTapListener() method 287, 290 setOutputFile() method 650 setOutputFormat() method 650 setResult() method 463 setText() method 180 settings.gradle file 730 settings.gradle.kts file 730 setTransition() 397 setVideoSource() method 650 SHA-256 certificate fingerprint 465 SharedFlow 542, 545 backgroudn handling 551 DROP_LATEST 543 DROP_OLDEST 543 in ViewModel 547

repeatOnLifecycle 552 SUSPEND 543 tutorial 545 SimpleOnScaleGestureListener 299 SimpleOnScaleGestureListener class 300 single() operator 535 SMS permissions 642 Snackbar 425, 426, 427 Snapshots emulator 43 sp 249 Spread chain 206 Spread inside 236 Spread inside chain 206 SQL 556 SQL CREATE 564 SQLite 555 AVD command-line use 557 Columns and Data Types 555 overview 556 Primary keys 556 tutorial 561 SQLiteDatabase 564 SQLiteOpenHelper 562 SQL SELECT 565 StaggeredGridLayoutManager 441 startActivity() method 461 startForeground() method 156 START_NOT_STICKY 514 START_REDELIVER_INTENT 514 START_STICKY 514 State restoring 180 State Change handling 159 StateFlow 541 Statement Completion 72 Status Bar Widgets 69 Memory Indicator 69 Sticky Broadcast Intents 493 Stopped state 158 Storage permissions 643

String 92 strings.xml file 24 Structure tool window 55 Structured Query Language 556 Structure tool window 55 SUBS 702 subscriptions 697 supervisorScope 501 SUSPEND 543 Suspend Functions 500 Switcher 56 System Broadcasts 497 system requirements 3

Т

TabLayout adding to layout 435 app tabGravity property 440 tabMode property 440 example 432 fixed mode 439 getItemCount() method 431 overview 431 TableLayout 182, 599 TableRow 599 Telephony Manager 82 Templates blank vs. empty 185 Terminal tool window 54 terminal flow operators 538 Theme building a custom 717 Theme Builder 717 Theming 715 tutorial 721 Time Cycle Keyframes 397 TODO tool window 55 ToolbarListener 308

Index

tools layout 305 Tool window bars 52 Tool windows 52 Touch Actions 282 Touch Event Listener implementation 283 Touch Events intercepting 281 Touch handling 281 transform() operator 537 try/finally 534 Type Annotations 95 Type Casting 99 Type Checking 99 Type Inference 95

U

unbindService() method 513 unregisterReceiver() method 493 upload key 686 UriMatcher 570, 576 UriMatcher class 570 USB connection issues resolving 62 user interface state 163 USE_SIP permission 642

V

Version catalog 271 dependencies 273 libraries 273 libs.versions.toml file 272 plugins 273 versions 273 Video Playback 623 Video View class 623 methods 623 supported formats 623 view bindings enabling 140 using 140 View class setting properties 260 view conversion 197 ViewGroup 181 View Groups 181 View Hierarchy 183 ViewHolder class 442 sample implementation 448 ViewModel adding LiveData 333 data access 331 overview 322 saved state 353 Saved State 323, 353 tutorial 327 ViewModelProvider 330 ViewModel Saved State 353 ViewModelScope 500 ViewPager adding to layout 435 example 432 Views 181 Java creation 253 View System 82 Virtual Device Configuration dialog 28 Virtual Sensors 43 Visible Process 156

W

WebView view 481 Weighted chain 206, 236 Welcome screen 49 while Loop 110 Widget Dimensions 207 Widget Group Alignment 229 Widgets palette 242 WiFi debugging 63 Wireless debugging 63 Wireless pairing 63 withContext 501, 503 wrap_content properties 251 WRITE_CALENDAR permission 642 WRITE_CALL_LOG permission 642 WRITE_CONTACTS permission 642 WRITE_EXTERNAL_STORAGE permission 643

Х

XML Layout File manual creation 249 vs. Java Code 253

Ζ

zip() operator 540