

# Jetpack Compose 1.6 Essentials





# **Jetpack Compose 1.6 Essentials**

---

Jetpack Compose 1.6 Essentials

ISBN-13: 978-1-951442-91-0

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

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

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

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

Rev: 1.0



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

## Table of Contents

### 1. Start Here

- 1.1 For Kotlin programmers
- 1.2 For new Kotlin programmers
- 1.3 Downloading the code samples
- 1.4 Feedback
- 1.5 Errata
- 1.6 Find more books
- 1.7 Authors wanted

### 2. Setting up an Android Studio Development Environment

- 2.1 System requirements
- 2.2 Downloading the Android Studio package
- 2.3 Installing Android Studio
  - 2.3.1 Installation on Windows
  - 2.3.2 Installation on macOS
  - 2.3.3 Installation on Linux
- 2.4 The Android Studio setup wizard
- 2.5 Installing additional Android SDK packages
- 2.6 Installing the Android SDK Command-line Tools
  - 2.6.1 Windows 8.1
  - 2.6.2 Windows 10
  - 2.6.3 Windows 11
  - 2.6.4 Linux
  - 2.6.5 macOS
- 2.7 Android Studio memory management
- 2.8 Updating Android Studio and the SDK
- 2.9 Summary

### 3. A Compose Project Overview

- 3.1 About the project
- 3.2 Creating the project
- 3.3 Creating an activity
- 3.4 Defining the project and SDK settings
- 3.5 Enabling the New Android Studio UI
- 3.6 Previewing the example project
- 3.7 Reviewing the main activity
- 3.8 Preview updates
- 3.9 Bill of Materials and the Compose version
- 3.10 Summary

### 4. An Example Compose Project

- 4.1 Getting started
- 4.2 Removing the template Code
- 4.3 The Composable hierarchy

## Table of Contents

- 4.4 Adding the DemoText composable
- 4.5 Previewing the DemoText composable
- 4.6 Adding the DemoSlider composable
- 4.7 Adding the DemoScreen composable
- 4.8 Previewing the DemoScreen composable
- 4.9 Adjusting preview settings
- 4.10 Testing in interactive mode
- 4.11 Completing the project
- 4.12 Summary

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

- 5.1 About Android Virtual Devices
- 5.2 Starting the Emulator
- 5.3 Running the Application in the AVD
- 5.4 Real-time updates with Live Edit
- 5.5 Running on Multiple Devices
- 5.6 Stopping a Running Application
- 5.7 Supporting Dark Theme
- 5.8 Running the Emulator in a Separate Window
- 5.9 Removing the Device Frame
- 5.10 Summary

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

- 6.1 The Emulator Environment
- 6.2 Emulator Toolbar Options
- 6.3 Working in Zoom Mode
- 6.4 Resizing the Emulator Window
- 6.5 Extended Control Options
  - 6.5.1 Location
  - 6.5.2 Displays
  - 6.5.3 Cellular
  - 6.5.4 Battery
  - 6.5.5 Camera
  - 6.5.6 Phone
  - 6.5.7 Directional Pad
  - 6.5.8 Microphone
  - 6.5.9 Fingerprint
  - 6.5.10 Virtual Sensors
  - 6.5.11 Snapshots
  - 6.5.12 Record and Playback
  - 6.5.13 Google Play
  - 6.5.14 Settings
  - 6.5.15 Help
- 6.6 Working with Snapshots
- 6.7 Configuring Fingerprint Emulation
- 6.8 The Emulator in Tool Window Mode
- 6.9 Creating a Resizable Emulator
- 6.10 Summary

## 7. A Tour of the Android Studio User Interface

- 7.1 The Welcome Screen

- 7.2 The Menu Bar
- 7.3 The Main Window
- 7.4 The Tool Windows
- 7.5 The Tool Window Menus
- 7.6 Android Studio Keyboard Shortcuts
- 7.7 Switcher and Recent Files Navigation
- 7.8 Changing the Android Studio Theme
- 7.9 Summary

## **8. Testing Android Studio Apps on a Physical Android Device**

- 8.1 An Overview of the Android Debug Bridge (ADB)
- 8.2 Enabling USB Debugging ADB on Android Devices
  - 8.2.1 macOS ADB Configuration
  - 8.2.2 Windows ADB Configuration
  - 8.2.3 Linux adb Configuration
- 8.3 Resolving USB Connection Issues
- 8.4 Enabling Wireless Debugging on Android Devices
- 8.5 Testing the adb Connection
- 8.6 Device Mirroring
- 8.7 Summary

## **9. The Basics of the Android Studio Code Editor**

- 9.1 The Android Studio Editor
- 9.2 Splitting the Editor Window
- 9.3 Code Completion
- 9.4 Statement Completion
- 9.5 Parameter Information
- 9.6 Parameter Name Hints
- 9.7 Code Generation
- 9.8 Code Folding
- 9.9 Quick Documentation Lookup
- 9.10 Code Reformatting
- 9.11 Finding Sample Code
- 9.12 Live Templates
- 9.13 Summary

## **10. An Overview of the Android Architecture**

- 10.1 The Android Software Stack
- 10.2 The Linux Kernel
- 10.3 Hardware Abstraction Layer
- 10.4 Android Runtime – ART
- 10.5 Android Libraries
  - 10.5.1 C/C++ Libraries
- 10.6 Application Framework
- 10.7 Applications
- 10.8 Summary

## **11. An Introduction to Kotlin**

- 11.1 What is Kotlin?
- 11.2 Kotlin and Java
- 11.3 Converting from Java to Kotlin

## Table of Contents

- 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

- 12.1 Kotlin data types
  - 12.1.1 Integer data types
  - 12.1.2 Floating point data types
  - 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
  - 14.2.3 Using *if ... else if ...* Expressions
  - 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 Compose**

- 18.1 Development before Compose
- 18.2 Compose declarative syntax
- 18.3 Compose is data-driven

## Table of Contents

### 18.4 Summary

## 19. A Guide to Gradle Version Catalogs

### 19.1 Library and Plugin Dependencies

### 19.2 Project Gradle Build File

### 19.3 Module Gradle Build Files

### 19.4 Version Catalog File

### 19.5 Adding Dependencies

### 19.6 Library Updates

### 19.7 Summary

## 20. Composable Functions Overview

### 20.1 What is a composable function?

### 20.2 Stateful vs. stateless composables

### 20.3 Composable function syntax

### 20.4 Foundation and Material composables

### 20.5 Summary

## 21. An Overview of Compose State and Recomposition

### 21.1 The basics of state

### 21.2 Introducing recomposition

### 21.3 Creating the StateExample project

### 21.4 Declaring state in a composable

### 21.5 Unidirectional data flow

### 21.6 State hoisting

### 21.7 Saving state through configuration changes

### 21.8 Summary

## 22. An Introduction to Composition Local

### 22.1 Understanding CompositionLocal

### 22.2 Using CompositionLocal

### 22.3 Creating the CompLocalDemo project

### 22.4 Designing the layout

### 22.5 Adding the CompositionLocal state

### 22.6 Accessing the CompositionLocal state

### 22.7 Testing the design

### 22.8 Summary

## 23. An Overview of Compose Slot APIs

### 23.1 Understanding slot APIs

### 23.2 Declaring a slot API

### 23.3 Calling slot API composables

### 23.4 Summary

## 24. A Compose Slot API Tutorial

### 24.1 About the project

### 24.2 Creating the SlotApiDemo project

### 24.3 Preparing the MainActivity class file

### 24.4 Creating the mainScreen composable

### 24.5 Adding the ScreenContent composable

### 24.6 Creating the Checkbox composable

### 24.7 Implementing the ScreenContent slot API

- 24.8 Adding an Image drawable resource
- 24.9 Coding the TitleImage composable
- 24.10 Completing the MainScreen composable
- 24.11 Previewing the project
- 24.12 Summary

## **25. Using Modifiers in Compose**

- 25.1 An overview of modifiers
- 25.2 Creating the ModifierDemo project
- 25.3 Creating a modifier
- 25.4 Modifier ordering
- 25.5 Adding modifier support to a composable
- 25.6 Common built-in modifiers
- 25.7 Combining modifiers
- 25.8 Summary

## **26. Annotated Strings and Brush Styles**

- 26.1 What are annotated strings?
- 26.2 Using annotated strings
- 26.3 Brush Text Styling
- 26.4 Creating the example project
- 26.5 An example SpanStyle annotated string
- 26.6 An example ParagraphStyle annotated string
- 26.7 A Brush style example
- 26.8 Summary

## **27. Composing Layouts with Row and Column**

- 27.1 Creating the RowColDemo project
- 27.2 Row composable
- 27.3 Column composable
- 27.4 Combining Row and Column composables
- 27.5 Layout alignment
- 27.6 Layout arrangement positioning
- 27.7 Layout arrangement spacing
- 27.8 Row and Column scope modifiers
- 27.9 Scope modifier weights
- 27.10 Summary

## **28. Box Layouts in Compose**

- 28.1 An introduction to the Box composable
- 28.2 Creating the BoxLayout project
- 28.3 Adding the TextCell composable
- 28.4 Adding a Box layout
- 28.5 Box alignment
- 28.6 BoxScope modifiers
- 28.7 Using the clip() modifier
- 28.8 Summary

## **29. An Introduction to FlowRow and FlowColumn**

- 29.1 FlowColumn and FlowRow
- 29.2 Maximum number of items

## Table of Contents

- 29.3 Working with main axis arrangement
- 29.4 Understanding cross-axis arrangement
- 29.5 Item alignment
- 29.6 Controlling item size
- 29.7 Summary

### **30. A FlowRow and FlowColumn Tutorial**

- 30.1 Creating the FlowLayoutDemo project
- 30.2 Generating random height and color values
- 30.3 Adding the Box Composable
- 30.4 Modifying the Flow arrangement
- 30.5 Modifying item alignment
- 30.6 Switching to FlowColumn
- 30.7 Using cross-axis arrangement
- 30.8 Adding item weights
- 30.9 Summary

### **31. Custom Layout Modifiers**

- 31.1 Compose layout basics
- 31.2 Custom layouts
- 31.3 Creating the LayoutModifier project
- 31.4 Adding the ColorBox composable
- 31.5 Creating a custom layout modifier
- 31.6 Understanding default position
- 31.7 Completing the layout modifier
- 31.8 Using a custom modifier
- 31.9 Working with alignment lines
- 31.10 Working with baselines
- 31.11 Summary

### **32. Building Custom Layouts**

- 32.1 An overview of custom layouts
- 32.2 Custom layout syntax
- 32.3 Using a custom layout
- 32.4 Creating the CustomLayout project
- 32.5 Creating the CascadeLayout composable
- 32.6 Using the CascadeLayout composable
- 32.7 Summary

### **33. A Guide to ConstraintLayout in Compose**

- 33.1 An introduction to ConstraintLayout
- 33.2 How ConstraintLayout works
  - 33.2.1 Constraints
  - 33.2.2 Margins
  - 33.2.3 Opposing constraints
  - 33.2.4 Constraint bias
  - 33.2.5 Chains
  - 33.2.6 Chain styles
- 33.3 Configuring dimensions
- 33.4 Guideline helper
- 33.5 Barrier helper

### 33.6 Summary

## 34. Working with ConstraintLayout in Compose

- 34.1 Calling ConstraintLayout
- 34.2 Generating references
- 34.3 Assigning a reference to a composable
- 34.4 Adding constraints
- 34.5 Creating the ConstraintLayout project
- 34.6 Adding the ConstraintLayout library
- 34.7 Adding a custom button composable
- 34.8 Basic constraints
- 34.9 Opposing constraints
- 34.10 Constraint bias
- 34.11 Constraint margins
- 34.12 The importance of opposing constraints and bias
- 34.13 Creating chains
- 34.14 Working with guidelines
- 34.15 Working with barriers
- 34.16 Decoupling constraints with constraint sets
- 34.17 Summary

## 35. Working with IntrinsicSize in Compose

- 35.1 Intrinsic measurements
- 35.2 Max. vs Min. Intrinsic Size measurements
- 35.3 About the example project
- 35.4 Creating the IntrinsicSizeDemo project
- 35.5 Creating the custom text field
- 35.6 Adding the Text and Box components
- 35.7 Adding the top-level Column
- 35.8 Testing the project
- 35.9 Applying IntrinsicSize.Max measurements
- 35.10 Applying IntrinsicSize.Min measurements
- 35.11 Summary

## 36. Coroutines and LaunchedEffects in Jetpack Compose

- 36.1 What are coroutines?
- 36.2 Threads vs. coroutines
- 36.3 Coroutine Scope
- 36.4 Suspend functions
- 36.5 Coroutine dispatchers
- 36.6 Coroutine builders
- 36.7 Jobs
- 36.8 Coroutines – suspending and resuming
- 36.9 Coroutine channel communication
- 36.10 Understanding side effects
- 36.11 Summary

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

- 37.1 Standard vs. lazy lists
- 37.2 Working with Column and Row lists
- 37.3 Creating lazy lists

## Table of Contents

- 37.4 Enabling scrolling with ScrollState
- 37.5 Programmatic scrolling
- 37.6 Sticky headers
- 37.7 Responding to scroll position
- 37.8 Creating a lazy grid
- 37.9 Summary

### **38. A Compose Row and Column List Tutorial**

- 38.1 Creating the ListDemo project
- 38.2 Creating a Column-based list
- 38.3 Enabling list scrolling
- 38.4 Manual scrolling
- 38.5 A Row list example
- 38.6 Summary

### **39. A Compose Lazy List Tutorial**

- 39.1 Creating the LazyListDemo project
- 39.2 Adding list data to the project
- 39.3 Reading the XML data
- 39.4 Handling image loading
- 39.5 Designing the list item composable
- 39.6 Building the lazy list
- 39.7 Testing the project
- 39.8 Making list items clickable
- 39.9 Summary

### **40. Lazy List Sticky Headers and Scroll Detection**

- 40.1 Grouping the list item data
- 40.2 Displaying the headers and items
- 40.3 Adding sticky headers
- 40.4 Reacting to scroll position
- 40.5 Adding the scroll button
- 40.6 Testing the finished app
- 40.7 Summary

### **41. A Compose Lazy Staggered Grid Tutorial**

- 41.1 Lazy Staggered Grids
- 41.2 Creating the StaggeredGridDemo project
- 41.3 Adding the Box composable
- 41.4 Generating random height and color values
- 41.5 Creating the Staggered List
- 41.6 Testing the project
- 41.7 Switching to a horizontal staggered grid
- 41.8 Summary

### **42. VerticalPager and HorizontalPager in Compose**

- 42.1 The Pager composables
- 42.2 Working with pager state
- 42.3 About the PagerDemo project
- 42.4 Creating the PagerDemo project
- 42.5 Adding the book cover images

- 42.6 Adding the HorizontalPager
- 42.7 Creating the page content
- 42.8 Testing the pager
- 42.9 Adding the arrow buttons
- 42.10 Summary

### **43. Compose Visibility Animation**

- 43.1 Creating the AnimateVisibility project
- 43.2 Animating visibility
- 43.3 Defining enter and exit animations
- 43.4 Animation specs and animation easing
- 43.5 Repeating an animation
- 43.6 Different animations for different children
- 43.7 Auto-starting an animation
- 43.8 Implementing crossfading
- 43.9 Summary

### **44. Compose State-Driven Animation**

- 44.1 Understanding state-driven animation
- 44.2 Introducing animate as state functions
- 44.3 Creating the AnimateState project
- 44.4 Animating rotation with animateFloatAsState
- 44.5 Animating color changes with animateColorAsState
- 44.6 Animating motion with animateDpAsState
- 44.7 Adding spring effects
- 44.8 Working with keyframes
- 44.9 Combining multiple animations
- 44.10 Using the Animation Inspector
- 44.11 Summary

### **45. Canvas Graphics Drawing in Compose**

- 45.1 Introducing the Canvas component
- 45.2 Creating the CanvasDemo project
- 45.3 Drawing a line and getting the canvas size
- 45.4 Drawing dashed lines
- 45.5 Drawing a rectangle
- 45.6 Applying rotation
- 45.7 Drawing circles and ovals
- 45.8 Drawing gradients
- 45.9 Drawing arcs
- 45.10 Drawing paths
- 45.11 Drawing points
- 45.12 Drawing an image
- 45.13 Drawing text
- 45.14 Summary

### **46. Working with ViewModels in Compose**

- 46.1 What is Android Jetpack?
- 46.2 The “old” architecture
- 46.3 Modern Android architecture
- 46.4 The ViewModel component

## Table of Contents

- 46.5 ViewModel implementation using state
- 46.6 Connecting a ViewModel state to an activity
- 46.7 ViewModel implementation using LiveData
- 46.8 Observing ViewModel LiveData within an activity
- 46.9 Summary

### **47. A Compose ViewModel Tutorial**

- 47.1 About the project
- 47.2 Creating the ViewModelDemo project
- 47.3 Adding the ViewModel
- 47.4 Accessing DemoViewModel from MainActivity
- 47.5 Designing the temperature input composable
- 47.6 Designing the temperature input composable
- 47.7 Completing the user interface design
- 47.8 Testing the app
- 47.9 Summary

### **48. An Overview of Android SQLite Databases**

- 48.1 Understanding database tables
- 48.2 Introducing database schema
- 48.3 Columns and data types
- 48.4 Database rows
- 48.5 Introducing primary keys
- 48.6 What is SQLite?
- 48.7 Structured Query Language (SQL)
- 48.8 Trying SQLite on an Android Virtual Device (AVD)
- 48.9 The Android Room persistence library
- 48.10 Summary

### **49. Room Databases and Compose**

- 49.1 Revisiting modern app architecture
- 49.2 Key elements of Room database persistence
  - 49.2.1 Repository
  - 49.2.2 Room database
  - 49.2.3 Data Access Object (DAO)
  - 49.2.4 Entities
  - 49.2.5 SQLite database
- 49.3 Understanding entities
- 49.4 Data Access Objects
- 49.5 The Room database
- 49.6 The Repository
- 49.7 In-Memory databases
- 49.8 Database Inspector
- 49.9 Summary

### **50. A Compose Room Database and Repository Tutorial**

- 50.1 About the RoomDemo project
- 50.2 Creating the RoomDemo project
- 50.3 Modifying the build configuration
- 50.4 Building the entity
- 50.5 Creating the Data Access Object



- 50.6 Adding the Room database
- 50.7 Adding the repository
- 50.8 Adding the ViewModel
- 50.9 Designing the user interface
- 50.10 Writing a ViewModelProvider Factory class
- 50.11 Completing the mainScreen function
- 50.12 Testing the RoomDemo app
- 50.13 Using the Database Inspector
- 50.14 Summary

## **51. An Overview of Navigation in Compose**

- 51.1 Understanding navigation
- 51.2 Declaring a navigation controller
- 51.3 Declaring a navigation host
- 51.4 Adding destinations to the navigation graph
- 51.5 Navigating to destinations
- 51.6 Passing arguments to a destination
- 51.7 Working with bottom navigation bars
- 51.8 Summary

## **52. A Compose Navigation Tutorial**

- 52.1 Creating the NavigationDemo project
- 52.2 About the NavigationDemo project
- 52.3 Declaring the navigation routes
- 52.4 Adding the home screen
- 52.5 Adding the welcome screen
- 52.6 Adding the profile screen
- 52.7 Creating the navigation controller and host
- 52.8 Implementing the screen navigation
- 52.9 Passing the user name argument
- 52.10 Testing the project
- 52.11 Summary

## **53. A Compose Navigation Bar Tutorial**

- 53.1 Creating the BottomBarDemo project
- 53.2 Declaring the navigation routes
- 53.3 Designing bar items
- 53.4 Creating the bar item list
- 53.5 Adding the destination screens
- 53.6 Creating the navigation controller and host
- 53.7 Designing the navigation bar
- 53.8 Working with the Scaffold component
- 53.9 Testing the project
- 53.10 Summary

## **54. Detecting Gestures in Compose**

- 54.1 Compose gesture detection
- 54.2 Creating the GestureDemo project
- 54.3 Detecting click gestures
- 54.4 Detecting taps using PointerInputScope
- 54.5 Detecting drag gestures

## Table of Contents

- 54.6 Detecting drag gestures using PointerInputScope
- 54.7 Scrolling using the scrollable modifier
- 54.8 Scrolling using the scroll modifiers
- 54.9 Detecting pinch gestures
- 54.10 Detecting rotation gestures
- 54.11 Detecting translation gestures
- 54.12 Summary

### **55. Working with Anchored Draggable Components**

- 55.1 Dragging and anchors
- 55.2 Detecting dragging gestures
- 55.3 Declaring the anchor points
- 55.4 Declaring thresholds
- 55.5 Declaring draggable state
- 55.6 Moving a component in response to a drag
- 55.7 About the DraggableDemo project
- 55.8 Creating the DraggableDemo project
- 55.9 Adding Foundation library
- 55.10 Adding the anchors enumeration
- 55.11 Setting up the draggable state and anchors
- 55.12 Designing the parent Box
- 55.13 Adding the draggable box
- 55.14 Testing the project
- 55.15 Summary

### **56. An Introduction to Kotlin Flow**

- 56.1 Understanding Flows
- 56.2 Creating the sample project
- 56.3 Adding a view model to the project
- 56.4 Declaring the flow
- 56.5 Emitting flow data
- 56.6 Collecting flow data as state
- 56.7 Transforming data with intermediaries
- 56.8 Collecting flow data
- 56.9 Adding a flow buffer
- 56.10 More terminal flow operators
- 56.11 Flow flattening
- 56.12 Combining multiple flows
- 56.13 Hot and cold flows
- 56.14 StateFlow
- 56.15 SharedFlow
- 56.16 Converting a flow from cold to hot
- 56.17 Summary

### **57. A Jetpack Compose SharedFlow Tutorial**

- 57.1 About the project
- 57.2 Creating the SharedFlowDemo project
- 57.3 Adding a view model to the project
- 57.4 Declaring the SharedFlow
- 57.5 Collecting the flow values
- 57.6 Testing the SharedFlowDemo app

57.7 Handling flows in the background

57.8 Summary

## **58. An Android Biometric Authentication Tutorial**

58.1 An overview of biometric authentication

58.2 Creating the biometric authentication project

58.3 Adding the biometric dependency

58.4 Configuring device fingerprint authentication

58.5 Adding the biometric permissions to the manifest file

58.6 Checking the security settings

58.7 Designing the user interface

58.8 Configuring the authentication callbacks

58.9 Starting the biometric prompt

58.10 Testing the project

58.11 Summary

## **59. Working with the Google Maps Android API in Android Studio**

59.1 The elements of the Google Maps Android API

59.2 Creating the Google Maps project

59.3 Creating a Google Cloud billing account

59.4 Creating a new Google Cloud project

59.5 Enabling the Google Maps SDK

59.6 Generating a Google Maps API key

59.7 Adding the API key to the Android Studio project

59.8 Adding the compose map dependency

59.9 Creating a map

59.10 Testing the application

59.11 Understanding geocoding and reverse geocoding

59.12 Specifying a map location

59.13 Changing the map type

59.14 Displaying map controls to the user

59.15 Handling map gesture interaction

59.15.1 Map zooming gestures

59.15.2 Map scrolling/panning gestures

59.15.3 Map tilt gestures

59.15.4 Map rotation gestures

59.16 Creating map markers

59.17 Controlling the map camera

59.18 Summary

## **60. Creating, Testing, and Uploading an Android App Bundle**

60.1 The Release Preparation Process

60.2 Android App Bundles

60.3 Register for a Google Play Developer Console Account

60.4 Configuring the App in the Console

60.5 Enabling Google Play App Signing

60.6 Creating a Keystore File

60.7 Creating the Android App Bundle

60.8 Generating Test APK Files

60.9 Uploading the App Bundle to the Google Play Developer Console

60.10 Exploring the App Bundle

## Table of Contents

- 60.11 Managing Testers
- 60.12 Rolling the App Out for Testing
- 60.13 Uploading New App Bundle Revisions
- 60.14 Analyzing the App Bundle File
- 60.15 Summary

### **61. An Overview of Android In-App Billing**

- 61.1 Preparing a project for In-App purchasing
- 61.2 Creating In-App products and subscriptions
- 61.3 Billing client initialization
- 61.4 Connecting to the Google Play Billing library
- 61.5 Querying available products
- 61.6 Starting the purchase process
- 61.7 Completing the purchase
- 61.8 Querying previous purchases
- 61.9 Summary

### **62. An Android In-App Purchasing Tutorial**

- 62.1 About the In-App purchasing example project
- 62.2 Creating the InAppPurchase project
- 62.3 Adding libraries to the project
- 62.4 Adding the App to the Google Play Store
- 62.5 Creating an In-App product
- 62.6 Enabling license testers
- 62.7 Creating a purchase helper class
- 62.8 Adding the StateFlow streams
- 62.9 Initializing the billing client
- 62.10 Querying the product
- 62.11 Handling purchase updates
- 62.12 Launching the purchase flow
- 62.13 Consuming the product
- 62.14 Restoring a previous purchase
- 62.15 Completing the MainActivity
- 62.16 Testing the app
- 62.17 Troubleshooting
- 62.18 Summary

### **63. Working with Compose Theming**

- 63.1 Material Design 2 vs. Material Design 3
- 63.2 Material Design 3 theming
- 63.3 Building a custom theme
- 63.4 Summary

### **64. A Material Design 3 Theming Tutorial**

- 64.1 Creating the ThemeDemo project
- 64.2 Designing the user interface
- 64.3 Building a new theme
- 64.4 Adding the theme to the project
- 64.5 Enabling dynamic colors
- 64.6 Summary

### **65. An Overview of Gradle in Android Studio**

- 65.1 An Overview of Gradle
- 65.2 Gradle and Android Studio
  - 65.2.1 Sensible Defaults
  - 65.2.2 Dependencies
  - 65.2.3 Build Variants
  - 65.2.4 Manifest Entries
  - 65.2.5 APK Signing
  - 65.2.6 ProGuard Support
- 65.3 The Property and Settings Gradle Build File
- 65.4 The Top-level Gradle Build File
- 65.5 Module Level Gradle Build Files
- 65.6 Configuring Signing Settings in the Build File
- 65.7 Running Gradle Tasks from the Command Line
- 65.8 Summary

**Index**



## 1. Start Here

This book teaches you how to build Android applications using Jetpack Compose 1.6, Android Studio Iguana (2023.2.1), Material Design 3, and the Kotlin programming language.

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

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

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

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

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

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

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

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

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

### 1.1 For Kotlin programmers

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

### 1.2 For new Kotlin programmers

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

### 1.3 Downloading the code samples

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

<https://www.payloadbooks.com/product/compose16/>

## Start Here

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

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

## 1.4 Feedback

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

## 1.5 Errata

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

[https://www.payloadbooks.com/compose16\\_errata](https://www.payloadbooks.com/compose16_errata)

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

## 1.6 Find more books

Visit <https://www.payloadbooks.com> to view our complete book catalog.

## 1.7 Authors wanted

Payload Publishing is looking for authors.

Are you an aspiring author with a book idea in mind? When you publish with us, you'll receive our full support every step of the way. We offer guidance and technical and editorial assistance to help you bring your book to life. Once your book is completed, we will publish and market it worldwide through our distribution and channel partnerships while paying you higher royalties than traditional publishers.

Find out more at:

<https://www.payloadbooks.com/authors-wanted>

or email us at:

[authors@payloadbooks.com](mailto:authors@payloadbooks.com)



## 3. A Compose Project Overview

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

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

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

### 3.1 About the project

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

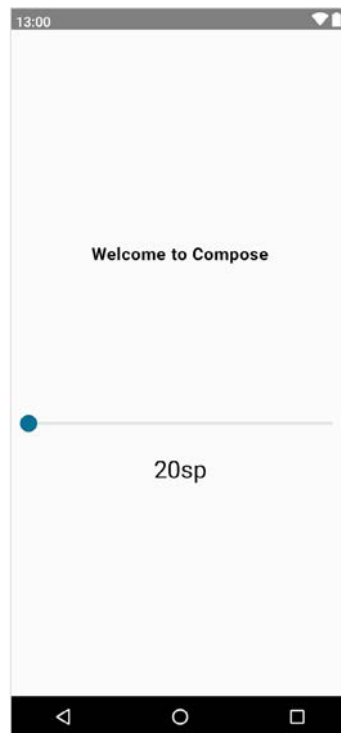


Figure 3-1

## 3.2 Creating the project

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

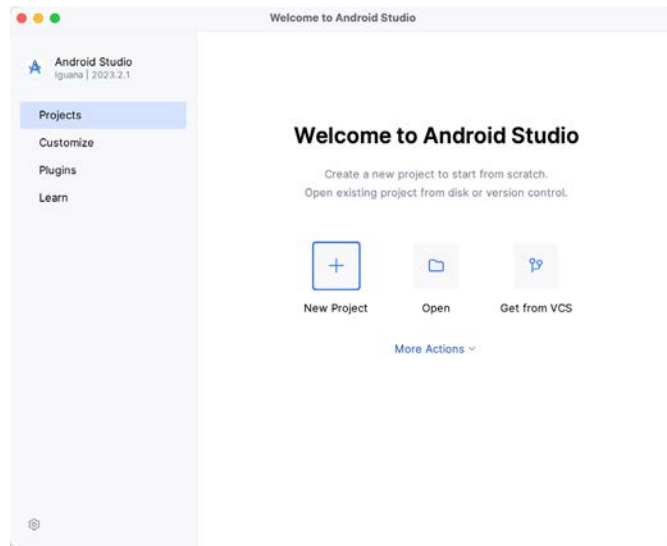


Figure 3-2

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

## 3.3 Creating an activity

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

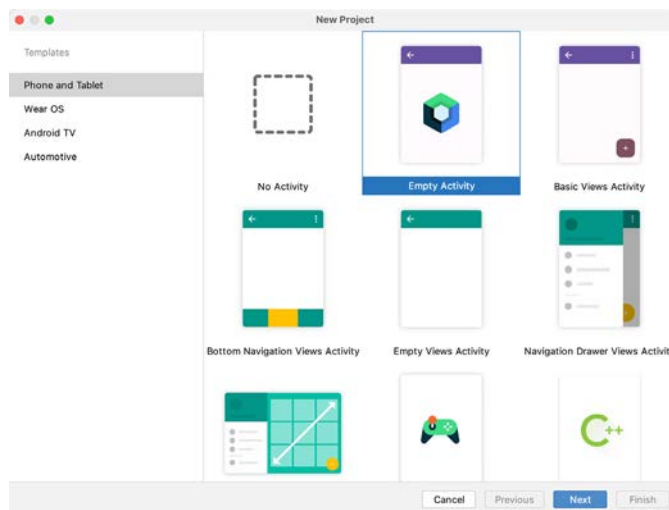


Figure 3-3

### 3.4 Defining the project and SDK settings

In the project configuration window (Figure 3-4), set the *Name* field to *ComposeDemo*. The application name is the name by which the application will be referenced and identified within Android Studio and is also the name that would be used if the completed application were to go on sale in the Google Play store:

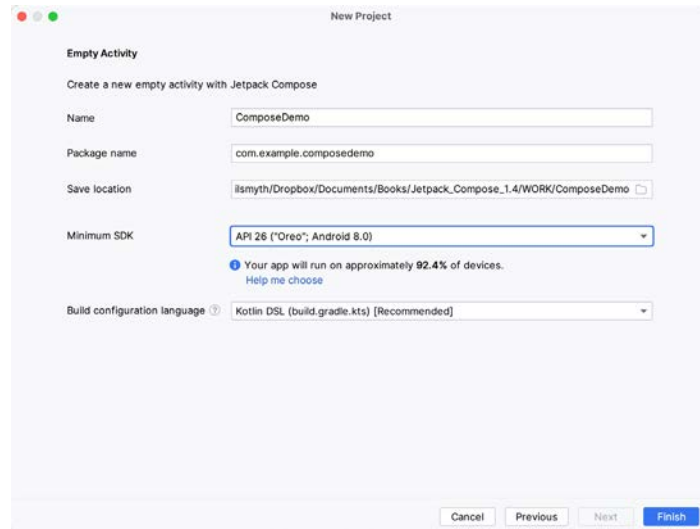


Figure 3-4

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

```
com.mycompany.composedemo
```

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

```
com.example.composedemo
```

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

Set the minimum SDK setting to API 26: Android 8.0 (Oreo). This is the minimum SDK that will be used in most projects created in this book unless a necessary feature is only available in a more recent version. The objective here is to build an app using the latest Android SDK, while also retaining compatibility with devices running older versions of Android (in this case as far back as Android 8.0). The text beneath the Minimum SDK setting will outline the percentage of Android devices currently in use on which the app will run. Click on the *Help me choose* link to see a full breakdown of the various Android versions still in use:

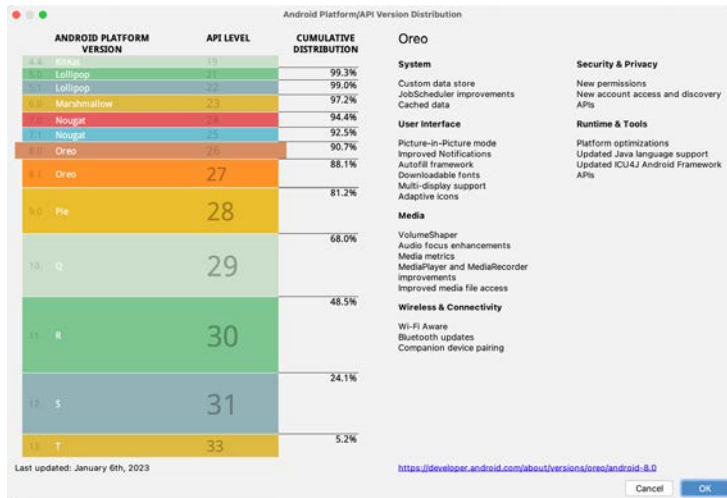


Figure 3-5

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

### 3.5 Enabling the New Android Studio UI

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

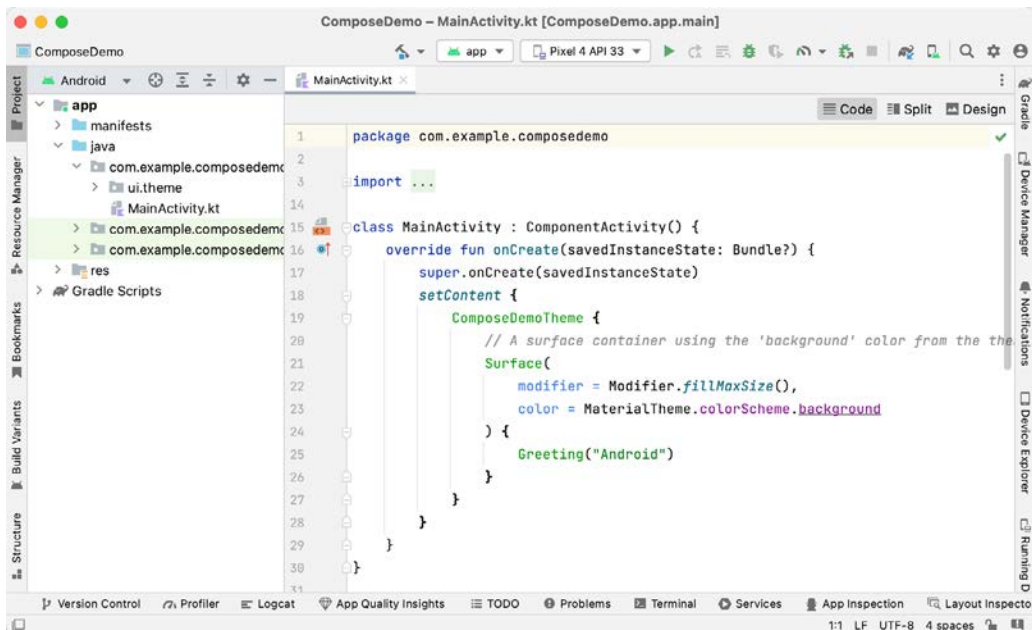


Figure 3-6

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

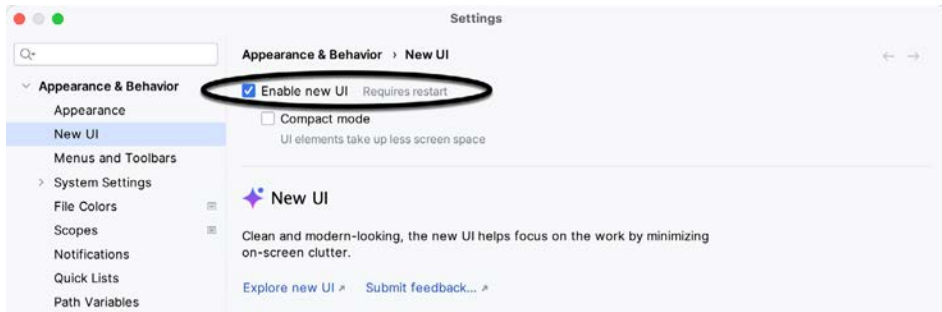


Figure 3-7

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

### 3.6 Previewing the example project

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

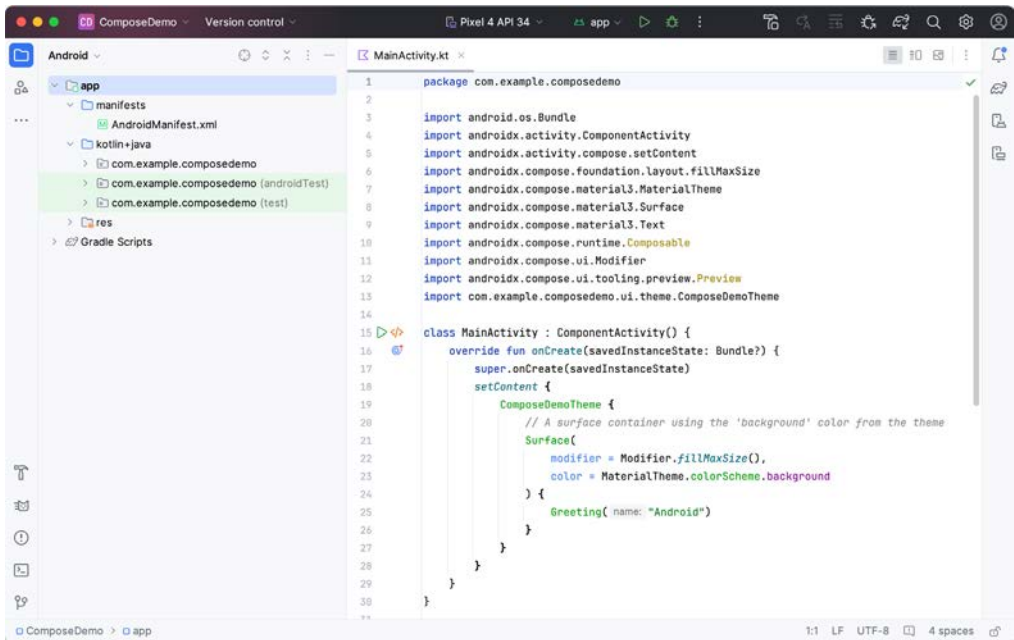


Figure 3-8

The newly created project and references to associated files are listed in the *Project* tool window located on the left-hand side of the main project window. The Project tool window has several modes in which information can be displayed. By default, this panel should be in *Android* mode. This setting is controlled by the menu at the top of the panel as highlighted in Figure 3-9. If the panel is not currently in Android mode, use the menu to switch mode:

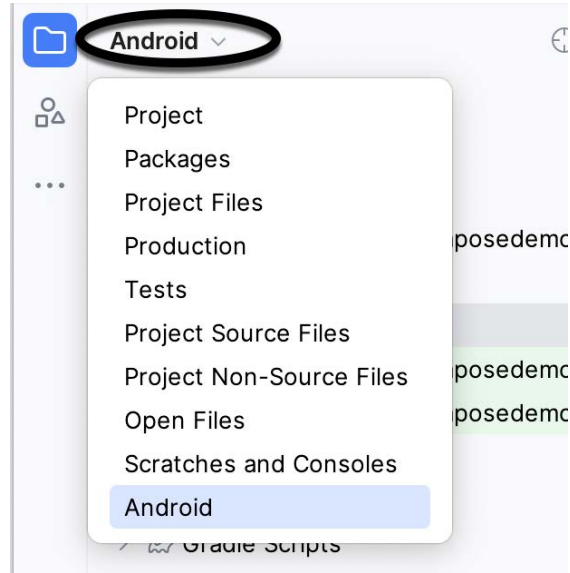


Figure 3-9

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

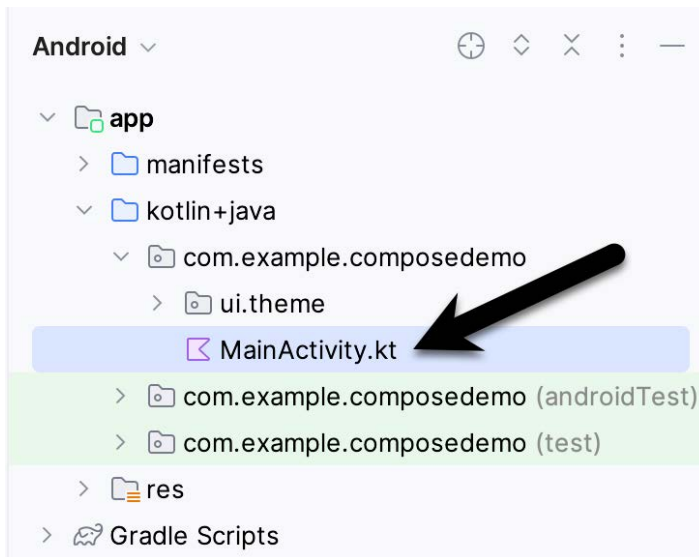


Figure 3-10

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

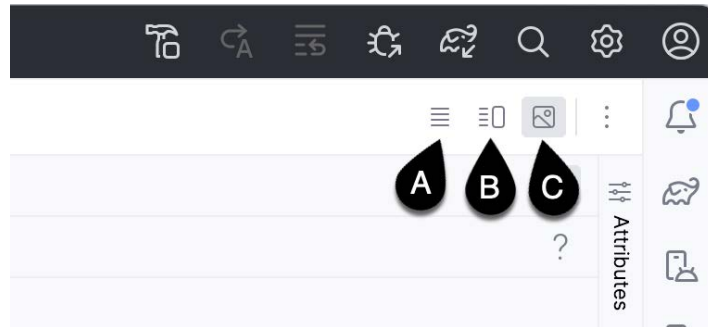


Figure 3-11

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

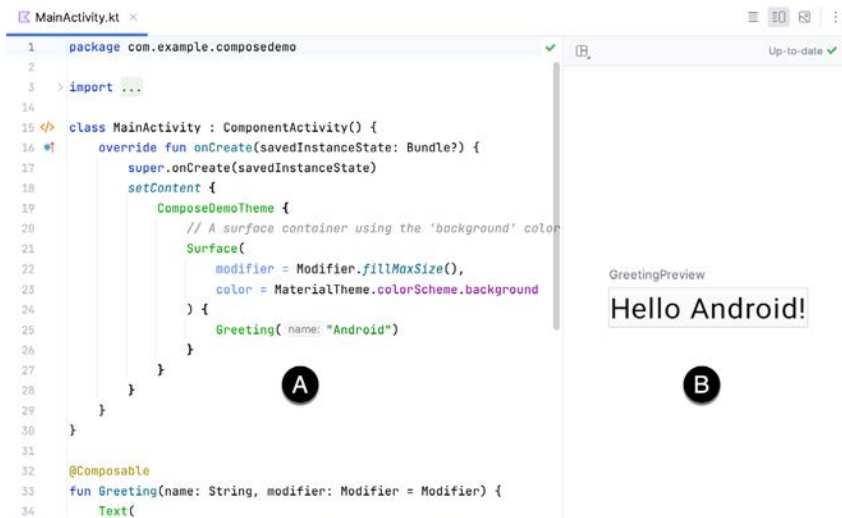


Figure 3-12

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

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

If the project has not yet been built, the Preview panel will display the message shown in Figure 3-13:

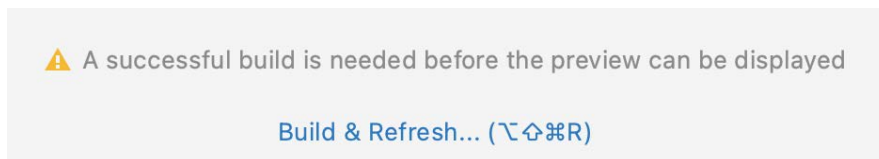


Figure 3-13

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

GreetingPreview



Hello Android!

Figure 3-14

### 3.7 Reviewing the main activity

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

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

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

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

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

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

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
.
.
```

Initially, the list of import directives will most likely be “folded” to save space. To unfold the list, click on the small disclosure button indicated by the arrow in Figure 3-15 below:



```

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

```

Figure 3-15

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

```

class MainActivity : ComponentActivity() {
    .
    .
}

```

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

```

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

```

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

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

```

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

```

In this case, the *Surface* component is configured to fill the entire screen and with the background set to the standard background color defined by the Android Material Design theme. Material Design is a set of design guidelines developed by Google to provide a consistent look and feel across all Android apps. It includes a theme

## A Compose Project Overview

(including fonts and colors), a set of user interface components (such as button, text, and a range of text fields), icons, and generally defines how an Android app should look, behave and respond to user interactions.

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

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

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

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

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

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

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

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

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

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

Once the preview has been updated, it should now be rendered as shown in Figure 3-16:

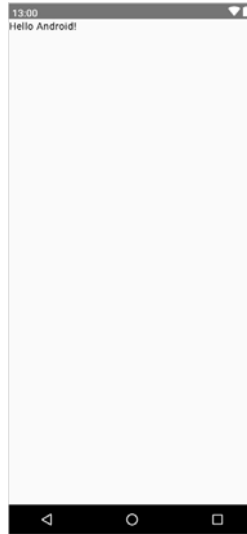


Figure 3-16

### 3.8 Preview updates

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

More significant changes will require a build and refresh before being reflected in the preview. When this is required, Android Studio will display the following “Out of date” notice at the top of the Preview panel and a *Build & Refresh* button (indicated by the arrow in Figure 3-17):

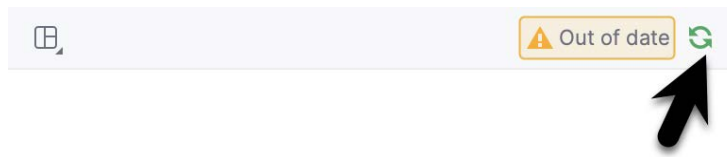


Figure 3-17

Simply click on the button to update the preview for the latest changes. Occasionally, Android Studio will fail to update the preview after code changes. If you believe that the preview no longer matches your code, hover the mouse pointer over the Up-to-date status text and select *Build & Refresh* from the resulting menu, as illustrated in Figure 3-18:



Figure 3-18

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

### 3.9 Bill of Materials and the Compose version

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

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

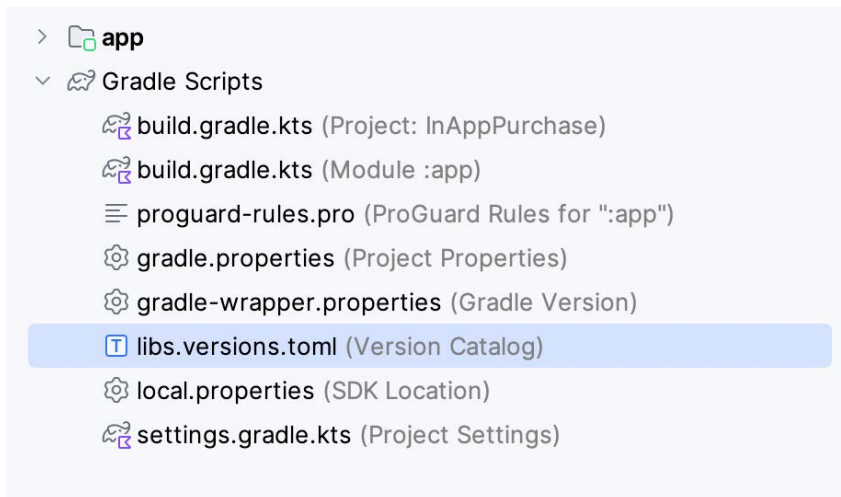


Figure 3-19

With the file loaded into the editor, locate the *composeBom* entry in the [versions] section:

```
[versions]
```

```
.
.
```

```
composeBom = "2023.08.00"
```

```
.
.
```

In the above example, we can see that the project is using BOM 2023.08.00. With this information, we can use the *BOM to library version mapping* web page at the following URL to identify the library versions being used to build our app:

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

Once the web page has loaded, select the BOM version from the menu highlighted in Figure 3-20 below. For example, the figure shows that BOM 2023.08.00 uses version 1.5.0 of the Compose libraries:



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

Figure 3-20

At the time of writing, Android Studio Iguana defaults to BOM 2023.08.00, while the latest stable BOM version is 2024.03.00, which maps to Jetpack Compose 1.6.4. Therefore, when working with the projects in this book, you should edit the *composeBom* entry in the *Gradle Scripts* -> *libs.versions.toml* and upgrade the BOM version to at least 2024.03.00.

Library versions and dependencies will be covered in greater detail in the “A Guide to Gradle Version Catalogs” chapter.

### 3.10 Summary

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

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



# 13. Kotlin Operators and Expressions

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

## 13.1 Expression syntax in Kotlin

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

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

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

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

## 13.2 The Basic assignment operator

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

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

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

## 13.3 Kotlin arithmetic operators

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

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

The following table lists the primary Kotlin arithmetic operators:

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

/	Division
+	Addition
-	Subtraction
%	Remainder/Modulo

Table 13-1

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

For example:

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

### 13.4 Augmented assignment operators

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

```
x = x + y
```

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

```
x += y
```

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

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

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

Table 13-2

### 13.5 Increment and decrement operators

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

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

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

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

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



value of 10:

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

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

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

## 13.6 Equality operators

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

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

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

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

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

result = x < y
```

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

Operator	Description
<code>x == y</code>	Returns true if <code>x</code> is equal to <code>y</code>
<code>x &gt; y</code>	Returns true if <code>x</code> is greater than <code>y</code>
<code>x &gt;= y</code>	Returns true if <code>x</code> is greater than or equal to <code>y</code>
<code>x &lt; y</code>	Returns true if <code>x</code> is less than <code>y</code>
<code>x &lt;= y</code>	Returns true if <code>x</code> is less than or equal to <code>y</code>
<code>x != y</code>	Returns true if <code>x</code> is not equal to <code>y</code>

Table 13-3

## 13.7 Boolean logical operators

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

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

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

## Kotlin Operators and Expressions

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

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

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

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

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

### 13.8 Range operator

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

The syntax for the range operator is as follows:

```
x..y
```

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

### 13.9 Bitwise operators

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

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

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

```
10101011
```

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

```
00000011
```

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

#### 13.9.1 Bitwise inversion

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

```
00000011 NOT
```

```
=====
```

```
11111100
```

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

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

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

### 13.9.2 Bitwise AND

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

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

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

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

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

### 13.9.3 Bitwise OR

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

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

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

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

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

### 13.9.4 Bitwise XOR

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

```
10101011 XOR
```

## Kotlin Operators and Expressions

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

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

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

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

When executed, we get the following output from print:

```
Result is 168
```

### 13.9.5 Bitwise left shift

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

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

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

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

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

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

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

### 13.9.6 Bitwise right shift

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

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

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

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

```
val x = 171
```

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

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

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

## 13.10 Summary

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



## 29. An Introduction to FlowRow and FlowColumn

The chapter entitled “*Composing Layouts with Row and Column*” used the Row and Column composables to present content elements uniformly within a user interface. One limitation of Row and Column-based layouts is that they are not well suited to organizing dynamic elements in terms of the quantity and sizes of the content. These composables are also less effective when designing layouts that are responsive to device screen orientation and size changes.

In this chapter, we will learn about the Flow layout composables and explore how they provide a more flexible way to organize content in rows and columns.

### 29.1 FlowColumn and FlowRow

The Row and Column composables work best when you know the number of items to be displayed and their respective sizes. This results in a spreadsheet-like layout with rows of aligned columns. The Flow layouts, however, are designed to flow content onto the next row or column when space runs out. These composables also discard the spreadsheet approach to organization, providing a more flexible approach to displaying items of varying sizes. Figure 29-1, for example, shows a typical FlowRow layout:

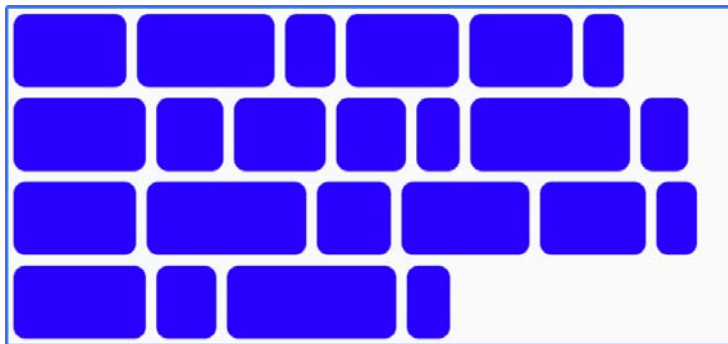


Figure 29-1

As we will explore later in this chapter, Flow layouts provide extensive options for configuring the layout and arrangement of child items, including weight, spacing, alignment, and the maximum number of items per row or column.

The FlowRow composable uses the following syntax:

```
FlowRow(  
    modifier: Modifier = Modifier,  
    horizontalArrangement: Arrangement.Horizontal,  
    verticalArrangement: Arrangement.Vertical,  
    maxItemsInEachRow: Int  
) {  
    // Content here
```

```
}
```

Figure 29-2 shows an example FlowColumn layout:

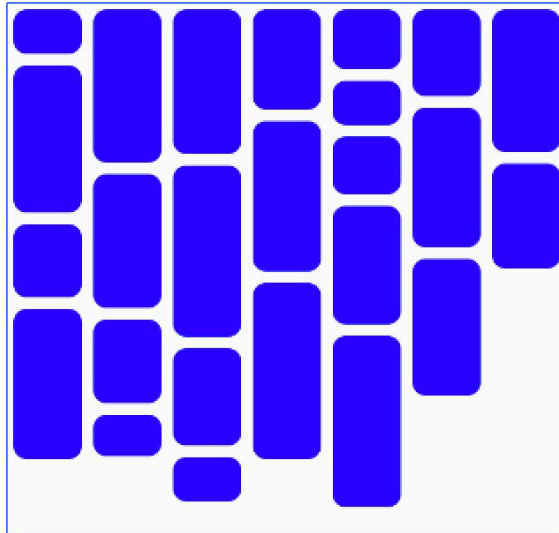


Figure 29-2

The FlowColumn composable uses the following syntax:

```
FlowColumn(  
    modifier: Modifier,  
    verticalArrangement: Arrangement.Vertical,  
    horizontalArrangement: Arrangement.Horizontal,  
    maxItemsInEachColumn: Int,  
) {  
    // Content here  
}
```

## 29.2 Maximum number of items

Without restrictions, the Flow layouts will fit as many items into a row or column as possible before flowing to the next one. The maximum number of items can be restricted using the *maxItemsInEachColumn* and *maxItemsInEachRow* properties of the FlowColumn and FlowRow. For example:

```
FlowRow(maxItemsInEachRow = 10) {  
    // Flow items here  
}  
  
FlowColumn(maxItemsInEachColumn = 5) {  
    // Flow items here  
}
```

## 29.3 Working with main axis arrangement

Main axis arrangement defines how the flow items are positioned along the main axis of the parent Flow layout. For example, the *horizontalArrangement* property controls the arrangement of flow items along the horizontal axis of the FlowRow composable. Table 29-1 shows the effects of the various *horizontalArrangement* options



when applied to a FlowRow instance:

	
Arrangement.Start	Arrangement.Center
	
Arrangement.End	Arrangement.SpaceBetween
	
Arrangement.SpaceAround	Arrangement.SpaceEvenly
	
Arrangement.spacedBy(10.dp)	

Table 29-1

Similarly, the *verticalArrangement* property controls the positioning of flow items along the vertical access of the FlowColumn. The same arrangement options are available as those listed above, except that *Arrangement.Start* and *Arrangement.End* are replaced by *Arrangement.Top* and *Arrangement.Bottom*.

## 29.4 Understanding cross-axis arrangement

Cross-axis arrangement controls the arrangement of a flow layout on the opposite axis to the main flow. In other words, the *verticalArrangement* property controls the vertical positioning of FlowRow items, while *horizontalArrangement* does the same along the horizontal axis of FlowColumn items. Table 29-2 demonstrates the three *horizontalArrangement* options applied to a FlowColumn instance:

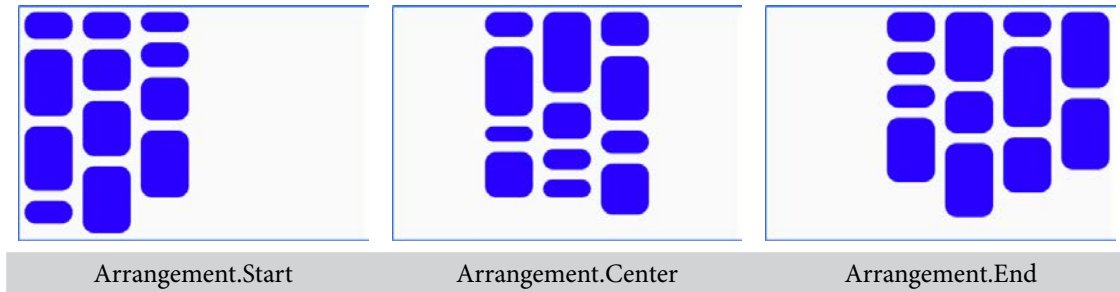


Table 29-2

## 29.5 Item alignment

The alignment of items within individual rows or columns can be controlled by passing an alignment value to the *align()* modifier of the child items of a Flow layout. This is useful when the Flow items vary in height (FlowRow) or width (FlowColumn). The following code, for example, specifies bottom alignment for a FlowRow item:

```
FlowRow {
    repeat(6) {
        MyFlowItem(modifier = Modifier.align(Alignment.Bottom))
    }
}
```

The following table illustrates the effect of applying *Alignment.Top*, *Alignment.CenterVertically*, and *Alignment.Bottom* to FlowRow items of varying height:

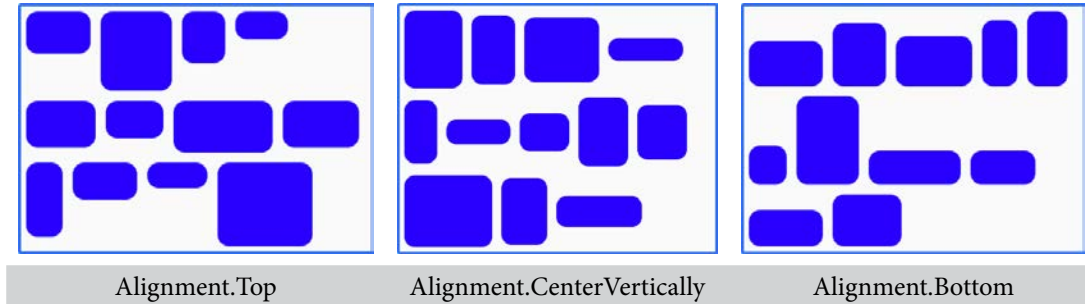


Table 29-3

Equivalent alignment effects can be achieved for FlowColumn items using *Alignment.Start*, *Alignment.CenterHorizontally*, and *Alignment.End*

## 29.6 Controlling item size

Weight factors can be applied to individual Flow items to specify the size relative to the overall space available and the weights of other items in the same row or column. Weights are expressed as Float values and applied to individual Flow items using the *weight()* modifier. Consider, for example, a FlowRow containing a single item with a weight of 1f:

```
FlowRow {
    MyFlowItem(
        Modifier
            .weight(1f)
    )
}
```

```

    )
}

```

When the layout is rendered, the item will occupy all the available space because it is the only item in the row:



Figure 29-3

If we add a second item, also with a weight of 1f, the two items will share the row equally:



Figure 29-4

If we add a third item with a weight of 1f, each item would occupy a third of the space. However, suppose that the third item has a weight of 2f, giving us a weight combination of 1f, 1f, and 2f. In this case, the first two items occupy half of the available space, while the third occupies the other half:



Figure 29-5

To calculate an item's when using weights, the Flow composables divide the amount of space remaining in the row or column by the total item weights, multiplied by the weight of the current item.

Another way to control the size of the items in a Flow layout is to use fractional sizing. Fractional sizing involves specifying the percentage of the overall space in a row or column that an item is to occupy. The fraction is declared as a Float value and applied to FlowRow and FlowColumn items using the *fillMaxWidth()* and *fillMaxHeight()* modifiers, respectively. For example:

```

FlowRow {
    MyFlowItem(Modifier.width(50.dp))
    MyFlowItem(Modifier.fillMaxWidth(0.7f))
    MyFlowItem(Modifier.width(50.dp))
}

```

Regardless of the sizes of the other items, the fractional item in the above code example will always occupy 70% of the row:



Figure 29-6

If there is insufficient room for the fractional item, items will flow onto the next row to make room:



Figure 29-7

## 29.7 Summary

The FlowRow and FlowColumn composables are ideal for arranging groups of items of varying sizes and quantities into flexible rows and columns. When a Flow layout runs out of space to display items, the remaining content flows to the next row or column. Combined with an extensive collection of alignment, spacing, and arrangement options, these composables provide a flexible and easy layout solution for presenting content within apps.

## 46. Working with ViewModels in Compose

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

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

### 46.1 What is Android Jetpack?

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

### 46.2 The “old” architecture

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

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

### 46.3 Modern Android architecture

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

Modern architecture guidelines also recommend separating different areas of responsibility within an app into entirely separate modules (a concept called “separation of concerns”). One of the keys to this approach is the ViewModel component.

### 46.4 The ViewModel component

The purpose of ViewModel is to separate the user interface-related data model and logic of an app from the code responsible for displaying and managing the user interface and interacting with the operating system.

When designed in this way, an app will consist of one or more *UI Controllers*, such as an activity, together with ViewModel instances responsible for handling the data needed by those controllers.

A ViewModel is implemented as a separate class and contains *state* values containing the model data and functions that can be called to manage that data. The activity containing the user interface *observes* the model state values such that any value changes trigger a recomposition. User interface events relating to the model data such as a button click are configured to call the appropriate function within the ViewModel. This is, in fact, a direct implementation of the *unidirectional data flow* concept described in the chapter entitled “*An Overview of Compose State and Recomposition*”. The diagram in Figure 46-1 illustrates this concept as it relates to activities and ViewModels:

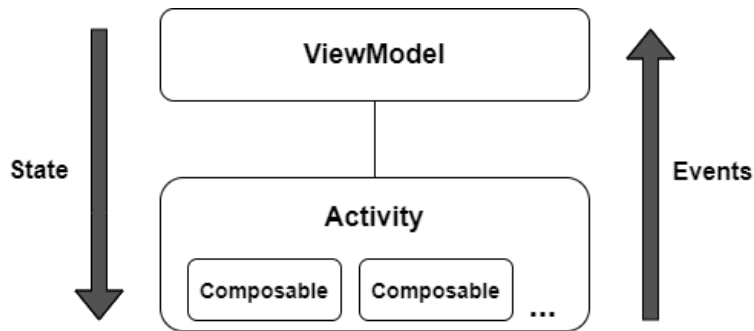


Figure 46-1

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

In addition to using ViewModels, the code responsible for gathering data from data sources such as web services or databases should be built into a separate *repository* module instead of being bundled with the view model. This topic will be covered in detail beginning with the chapter entitled “*Room Databases and Compose*”.

### 46.5 ViewModel implementation using state

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

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

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

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

```
class MyViewModel : ViewModel() {

    var customerCount by mutableStateOf(0)

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

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

## 46.6 Connecting a ViewModel state to an activity

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

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

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

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

```
        }  
    }  
}
```

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

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

The `viewModel()` function is provided by the Compose view model lifecycle library which needs to be added to the project's build dependencies when working with view models. This requires the following additions to the *Gradle Scripts* -> *libs.version.toml* file:

```
[versions]  
activityCompose = "1.8.2"  
.  
.  
[libraries]  
androidx-lifecycle-viewmodel-compose = { module = "androidx.lifecycle:lifecycle-  
viewmodel-compose", version.ref = "lifecycleRuntimeKtx" }  
.  
.
```

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

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

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

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

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

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

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

## 46.7 ViewModel implementation using LiveData

The Jetpack `LiveData` component predates the introduction of Compose and can be used as a wrapper around data values within a view model. Once contained in a `LiveData` instance, those variables become observable to composables within an activity. `LiveData` instances can be declared as being mutable using the `MutableLiveData`



class, allowing the ViewModel functions to make changes to the underlying data value. An example view model designed to store a customer name could, for example, be implemented as follows using `MutableLiveData` instead of state:

```
class MyViewModel : ViewModel() {

    var customerName: MutableLiveData<String> = MutableLiveData("")

    fun setName(name: String) {
        customerName.value = name
    }
}
```

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

## 46.8 Observing ViewModel LiveData within an activity

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

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

}
```

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

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

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

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

## 46.9 Summary

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



## 61. An Overview of Android In-App Billing

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

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

### 61.1 Preparing a project for In-App purchasing

Building in-app purchasing into an app will require a Google Play Developer Console account, which was covered previously in the “*Creating, Testing and Uploading an Android App Bundle*” chapter. In addition, you must also register a Google merchant account and configure your payment settings. You can find these settings by navigating to *Setup -> Payments profile* in the Play Console. Note that merchant registration is not available in all countries. For details, refer to the following page:

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

The app will then need to be uploaded to the console and enabled for in-app purchasing. The console will not activate in-app purchasing support for an app, however, unless the Google Play Billing Library has been added to the module-level *build.gradle.kts* file. When working with Kotlin, the Google Play Kotlin Extensions Library is also recommended:

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

The corresponding entries in the *libs.versions.toml* file for the above libraries will read as follows:

```
[versions]  
billing = "<latest version>"  
.  
.  
[libraries]
```

```
billing = { module = "com.android.billingclient:billing", version.ref = "billing"
}
billing-ktx = { module = "com.android.billingclient:billing-ktx", version.ref =
"billing" }
.
.
```

Once the build files have 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.

### 61.2 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 61-1 below:

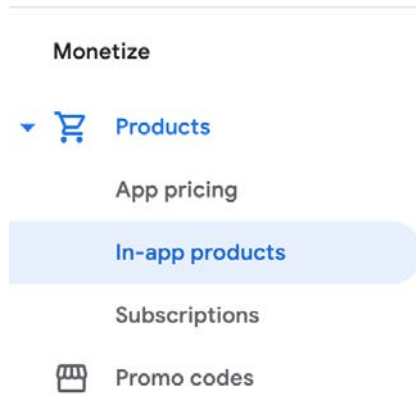


Figure 61-1

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

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

### 61.3 Billing client initialization

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

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

```
private val purchasesUpdatedListener =
    PurchasesUpdatedListener { billingResult, purchases ->
```

```

    if (billingResult.responseCode ==
        BillingClient.BillingResponseCode.OK
        && purchases != null
    ) {
        for (purchase in purchases) {
            // Process the purchases
        }
    } else if (billingResult.responseCode ==
        BillingClient.BillingResponseCode.USER_CANCELED
    ) {
        // Purchase cancelled by user
    } else {
        // Handle errors here
    }
}

```

```

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

```

## 61.4 Connecting to the Google Play Billing library

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

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

```

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

override fun onBillingServiceDisconnected() {
    // Connection to billing service lost
}

```

```
})
```

## 61.5 Querying available products

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

```
val queryProductDetailsParams = QueryProductDetailsParams.newBuilder()
    .setProductList(
        ImmutableList.of(
            QueryProductDetailsParams.Product.newBuilder()
                .setProductId(productId)
                .setProductType(
                    BillingClient.ProductType.INAPP
                )
                .build()
        )
    )
    .build()

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

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

## 61.6 Starting the purchase process

Once a product or subscription has been queried and selected for purchase by the user, the purchase process is ready to be launched. We do this by calling the `launchBillingFlow()` method of the `BillingClient`, passing through as arguments the current activity and a `BillingFlowParams` instance configured with the `ProductDetail` object for the item being purchased.

```
val billingFlowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(
        ImmutableList.of(
            BillingFlowParams.ProductDetailsParams.newBuilder()
                .setProductDetails(productDetails)
                .build()
        )
    )
```

```

    )
    )
    .build()

```

```
billingClient.launchBillingFlow(this, billingFlowParams)
```

The success or otherwise of the purchase operation will be reported via a call to the `PurchasesUpdatedListener` callback handler outlined earlier in the chapter.

## 61.7 Completing the purchase

When purchases are successful, the `PurchasesUpdatedListener` handler will be passed a list containing a `Purchase` object for each item. You can verify that the item has been purchased by calling the `getPurchaseState()` method of the `Purchase` instance as follows:

```

if (purchase.getPurchaseState() == Purchase.PurchaseState.PURCHASED) {
    // Purchase completed.
} else if (purchase.getPurchaseState() == Purchase.PurchaseState.PENDING) {
    // Payment is still pending
}

```

Note that your app will only support pending purchases if a call is made to the `enablePendingPurchases()` method during initialization. A pending purchase will remain so until the user completes the payment process.

When the purchase of a non-consumable item is complete, it will need to be acknowledged to prevent a refund from being issued to the user. This requires the *purchase token* for the item which is obtained via a call to the `getPurchaseToken()` method of the `Purchase` object. This token is used to create an `AcknowledgePurchaseParams` instance together with an `AcknowledgePurchaseResponseListener` handler. Managed product purchases and subscriptions are acknowledged by calling the `BillingClient`'s `acknowledgePurchase()` method as follows:

```

billingClient.acknowledgePurchase(acknowledgePurchaseParams,
                                acknowledgePurchaseResponseListener);
val acknowledgePurchaseParams = AcknowledgePurchaseParams.newBuilder()
    .setPurchaseToken(purchase.purchaseToken)
    .build()

val acknowledgePurchaseResponseListener = AcknowledgePurchaseResponseListener {
    // Check acknowledgement result
}

billingClient.acknowledgePurchase(
    acknowledgePurchaseParams,
    acknowledgePurchaseResponseListener
)

```

For consumable purchases, you will need to notify Google Play when the item has been consumed so that it is available to be repurchased by the user. This requires a configured `ConsumeParams` instance containing a purchase token and a call to the billing client's `consumePurchase()` method:

```

val consumeParams = ConsumeParams.newBuilder()
    .setPurchaseToken(purchase.purchaseToken)
    .build()

```

## An Overview of Android In-App Billing

```
coroutineScope.launch {
    val result = billingClient.consumePurchase(consumeParams)

    if (result.billingResult.responseCode ==
        BillingClient.BillingResponseCode.OK) {
        // Purchase successfully consumed
    }
}
```

### 61.8 Querying previous purchases

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



## 61.9 Summary

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



# Index

## Symbols

?. 107  
 2D graphics 387  
 @Composable 24, 155  
 @ExperimentalFoundationApi 334  
 :: operator 109  
 @Preview 25  
   showSystemUi 25

## A

acknowledgePurchase() method 581  
 Activity Manager 96  
 adb  
   command-line tool 73  
   connection testing 79  
   device pairing 77  
   enabling on Android devices 73  
   Linux configuration 76  
   list devices 73  
   macOS configuration 74  
   overview 73  
   restart server 74  
   testing connection 79  
   WiFi debugging 77  
   Windows configuration 75  
   Wireless debugging 77  
   Wireless pairing 77  
 AlertDialog 159  
 align() 233  
 alignByBaseline() 225  
 Alignment.Bottom 219, 223  
 Alignment.BottomCenter 231  
 Alignment.BottomEnd 231  
 Alignment.BottomStart 231  
 Alignment.Center 231

Alignment.CenterEnd 231  
 Alignment.CenterHorizontally 219  
 Alignment.CenterStart 231  
 Alignment.CenterVertically 219, 223  
 Alignment.End 219  
 alignment lines 255  
 Alignment.Start 219  
 Alignment.Top 219, 223  
 Alignment.TopCenter 231  
 Alignment.TopEnd 231  
 Alignment.TopStart 231  
 Anchored Draggable Components 499  
   anchoredDraggable() modifier 499  
   AnchoredDraggableState 500  
   anchor points 499  
   DraggableAnchors 500  
   threshold 499  
 anchoredDraggable() modifier 499, 508  
   orientation 499  
   reverseDirection 499  
   state 499  
 AnchoredDraggableState 500  
   anchors 500  
   animationSpec 501  
   initialValue 500  
   positionalThreshold 500  
   requiredOffset() 501, 507  
   velocityThreshold 501  
 Android  
   architecture 93  
   runtime 94  
   SDK Packages 6  
 android.app 94  
 Android Architecture Components 407  
 android.content 94  
 android.database 94  
 Android Debug Bridge. *See* ADB  
 Android Development  
   System Requirements 3

## Index

- android.graphics 95
- android.hardware 95
- android.hardware.camera 539
- Android Jetpack 407
- Android Libraries 94
- android.media 95
- Android Monitor tool window 46
- Android Native Development Kit 95
- android.net 95
- android.opengl 95
- android.os 95
- android.print 95
- android.provider 95
- Android SDK Location
  - identifying 10
- Android SDK Manager 8, 10
- Android SDK Packages
  - version requirements 8
- Android SDK Tools
  - command-line access 9
  - Linux 11
  - macOS 11
  - Windows 7 10
  - Windows 8 10
- Android Software Stack 93
- Android Studio
  - Animation Inspector 385
  - Asset Studio 190
  - changing theme 71
  - Database Inspector 455
  - downloading 3
  - Editor Window 66
  - installation 4
  - Layout Editor 149
  - Linux installation 5
  - macOS installation 4
  - Navigation Bar 65
  - Project tool window 66
  - setup wizard 5
  - Status Bar 66
  - Toolbar 65
  - Tool window bars 66
  - tool windows 66
  - updating 12
  - Welcome Screen 63
  - Windows installation 4
- Android Support Library, 407
- android.text 95
- android.util 95
- android.view 95
- Android Virtual Device. *See* AVD
  - overview 39
- Android Virtual Device Manager 39
- android.webkit 95
- android.widget 95
- AndroidX libraries 612
- animate as state functions 371
- animateColorAsState() function 371, 375, 377
- animateDpAsState() function 377, 382
- AnimatedVisibility 359
  - animation specs 363
  - enter and exit animations 362
  - expandHorizontally() 362
  - expandIn() 362
  - expandVertically() 362
  - fadeIn() 362
  - fadeOut() 363
  - MutableTransitionState 367
  - scaleIn() 363
  - scaleOut() 363
  - shrinkHorizontally() 363
  - shrinkOut() 363
  - shrinkVertically() 363
  - slideIn() 363
  - slideInHorizontally() 363
  - slideInVertically() 363
  - slideOut() 363
  - slideOutHorizontally() 363
  - slideOutVertically() 363
- animateEnterExit() modifier 366
- animateFloatAsState() function 372
- animateScrollTo() function 310, 321
- animateScrollToItem(index: Int) 310
- animateScrollTo(value: Int) 309

- Animation
    - auto-starting 366
    - combining animations 382
    - inspector 385
    - keyframes 381
    - KeyframesSpec 381
    - motion 377
    - spring effects 380
    - state-based 371
    - visibility 359
  - Animation damping
    - DampingRatioHighBouncy 380
    - DampingRatioLowBouncy 380
    - DampingRatioMediumBouncy 380
    - DampingRatioNoBouncy 380
  - Animation Inspector 385
  - AnimationSpec 363
    - tween() function 364
  - Animation specs 363
  - Animation stiffness
    - StiffnessHigh 381
    - StiffnessLow 381
    - StiffnessMedium 381
    - StiffnessMediumLow 381
    - StiffnessVeryLow 381
  - annotated strings 207, 403
    - append function 207
    - buildAnnotatedString function 207
    - ParagraphStyle 208
    - SpanStyle 207
  - API Key 549
  - APK analyzer 574
  - APK file 567
  - APK File
    - analyzing 574
  - APK Signing 612
  - APK Wizard dialog 566
  - App Bundles 563
    - creating 567
    - overview 563
    - revisions 573
    - uploading 570
  - append function 207
  - App Inspector 67
  - Application
    - stopping 46
  - Application Framework 96
  - Arrangement.Bottom 221
  - Arrangement.Center 220, 221
  - Arrangement.End 220
  - Arrangement.SpaceAround 222
  - Arrangement.SpaceBetween 222
  - Arrangement.SpaceEvenly 222
  - Arrangement.Start 220
  - Arrangement.Top 221
  - ART 94
  - as 109
  - as? 109
  - asFlow() builder 513
  - Asset Studio 190
  - asSharedFlow() 525
  - asStateFlow() 523
  - async 301
  - AVD
    - Change posture 61
    - cold boot 58
    - command-line creation 39
    - creation 39
    - device frame 50
    - Display mode 60
    - launch in tool window 50
    - overview 39
    - quickboot 58
    - Resizable 60
    - running an application 42
    - Snapshots 57
    - standalone 47
    - starting 41
    - Startup size and orientation 42
- ## B
- background modifier 204
  - barriers 285
  - Barriers 270

## Index

- constrained views 270
  - baseline
    - alignment 223
  - baselines 257
  - BaseTextField 158
  - BillingClient 582
    - acknowledgePurchase() method 581
    - consumeAsync() method 581
    - getPurchaseState() method 581
    - initialization 578, 589
    - launchBillingFlow() method 580
    - queryProductDetailsAsync() method 580
    - queryPurchasesAsync() method 582
    - startConnection() method 579
  - BillingResult 597
    - getDebugMessage() 597
  - Bill of Materials. *See* BOM
  - Biometric Authentication 537
    - callbacks 542
    - overview 537
    - tutorial 537
  - Biometric library 538
  - BiometricManager 537
  - BiometricPrompt 537
  - Bitwise AND 115
  - Bitwise Inversion 114
  - Bitwise Left Shift 116
  - Bitwise OR 115
  - Bitwise Right Shift 116
  - Bitwise XOR 115
  - BOM 26
    - build.gradle.kts 26
    - compose-bom 26
    - library version mapping 27
  - Boolean 102
  - BottomNavigation 159, 463
  - BottomNavigationItem 463
  - Box 158
    - align() 233
    - alignment 231
      - Alignment.BottomCenter 231
      - Alignment.BottomEnd 231
      - Alignment.BottomStart 231
      - Alignment.Center 231
      - Alignment.CenterEnd 231
      - Alignment.CenterStart 231
      - Alignment.TopCenter 231
      - Alignment.TopEnd 231
      - Alignment.TopStart 231
    - BoxScope 233
      - contentAlignment 231
      - matchParentSize() 233
      - overview 229
      - tutorial 229
  - BoxScope
    - align() 233
    - matchParentSize() 233
    - modifiers 233
  - BoxWithConstraints 158
  - Brush Text Styling 208
  - buffer() operator 519
  - buildAnnotatedString function 207
  - Build tool window 68
  - Build Variants , 68
    - tool window 68
  - Button 159
  - by keyword 164
- ## C
- CAMERA permission 539
  - CameraUpdateFactory class
    - methods 559
  - cancelAndJoin() 302
  - cancelChildren() 302
  - Canvas 158
    - DrawScope 387
    - inset() function 391
    - overview 387
    - size 387
  - Card 159
    - example 314
  - C/C++ Libraries 95
  - centerAround() function 274
  - chain head 268

- chaining modifiers 199
- chains 268
- chain styles 268
- Char 102
- Checkbox 159, 188
- Circle class 545
- CircleShape 233
- CircularProgressIndicator 159
- clickable 204
- clip 204
- Clip Art 191
- clip() modifier 233
  - CircleShape 233
  - CutCornerShape 233
  - RectangleShape 233
  - RoundedCornerShape 233
- close() function 399
- Code completion 84
- Code Editor
  - basics 81
  - Code completion 84
  - Code Generation 86
  - Code Reformatting 89
  - Document Tabs 82
  - Editing area 82
  - Gutter Area 82
  - Live Templates 90
  - Splitting 84
  - Statement Completion 86
  - Status Bar 83
- Code Generation 86
- Code Reformatting 89
- code samples
  - download 1
- Coil
  - rememberImagePainter() function 328
- cold boot 58
- Cold flow 523
  - convert to hot 526
- collectLatest() operator 518
- collect() operator 514
- ColorFilter 402
- color filtering 402
- Column 158
  - Alignment.CenterHorizontally 219
  - Alignment.End 219
  - Alignment.Start 219
  - Arrangement.Bottom 221
  - Arrangement.Center 221
  - Arrangement.SpaceAround 222
  - Arrangement.SpaceBetween 222
  - Arrangement.SpaceEvenly 222
  - Arrangement.Top 221
  - Layout alignment 218
  - list 307
  - list tutorial 317
  - overview 216
  - scope 223
  - scope modifiers 223
  - spacing 222
  - tutorial 215
  - verticalArrangement 220
- Column lists 307
- ColumnScope 223
  - Modifier.align() 223
  - Modifier.alignBy() 223
  - Modifier.weight() 223
- combine() operator 522
- combining modifiers 204
- Communicating Sequential Processes 299
- Companion Objects 139
- components 155
- Composable
  - adding a 30
  - previewing 32
- Composable function
  - syntax 156
- composable functions 155
- composables
  - add modifier support 200
- Composables
  - Foundation 158
  - Material 158
- Compose

## Index

- before 149
- components 155
- data-driven 150
- declarative syntax 149
- functions 155
- layout overview 251
- modifiers 197
- overview 149
- state 150
- compose-bom 26
- compose() method 459
- CompositionLocal
  - example 175
  - overview 173
  - state 176, 177
  - syntax 174
- compositionLocalOf() function 174
- conflate() operator 518
- constrainAs() modifier function 273
- constrain() function 290
- Constraint bias 279
- Constraint Bias 267
- ConstraintLayout 158
  - adding constraints 274
  - barriers 285
  - Barriers 270
  - basic constraints 276
  - centerAround() function 274
  - chain head 268
  - chains 268
  - chain styles 268
  - constrainAs() function 273
  - constrain() function 290
  - Constraint bias 279
  - Constraint Bias 267
  - Constraint margins 279
  - Constraints 265
  - constraint sets 289
  - createEndBarrier() 285, 286
  - createHorizontalChain() 283
  - createRefFor() function 290
  - createRef() function 273
  - createRefs() function 273
  - createStartBarrier() 285
  - createTopBarrier() 285
  - createVerticalChain() 283
  - creating chains 283
  - generating references 273
  - guidelines 284
  - Guidelines 269
  - how to call 273
  - layout() modifier 290
  - linkTo() function 274
  - Margins 266
  - Opposing constraints 277
  - Opposing Constraints 266, 281
  - overview of 265
  - Packed chain 269
  - reference assignment 273
  - Spread chain 268
  - Spread inside chain 268
  - Weighted chain 268
  - Widget Dimensions 269
- Constraint margins 279
- constraints 260
- constraint sets 289
- consumeAsync() method 581
- ConsumeParams 593
- contentAlignment 231
- Content Provider 96
- Coroutine Builders 301
  - async 301
  - coroutineScope 301
  - launch 301
  - runBlocking 301
  - supervisorScope 301
  - withContext 301
- Coroutine Dispatchers 300
- Coroutines 310, 511
  - channel communication 303
  - coroutine scope 310
  - CoroutineScope 310
  - GlobalScope 300
  - LaunchedEffect 304



- rememberCoroutineScope() 310
  - rememberCoroutineScope() function 300
  - SideEffect 304
  - Side Effects 304
  - Suspend Functions 300
  - suspending 302
  - ViewModelScope 300
  - vs Threads 299
  - vs. Threads 299
  - coroutineScope 301
  - CoroutineScope 300, 310
    - rememberCoroutineScope() 310
  - createEndBarrier() 285
  - createHorizontalChain() 283
  - createRefFor() function 290
  - createRef() function 273
  - createRefs() 273
  - createStartBarrier() 285
  - createTopBarrier() 285
  - createVerticalChain() 283
  - cross axis arrangement 249
  - Crossfading 367
  - currentBackStackEntryAsState() method 464, 480
  - Custom Accessors 137
  - Custom layout 259
    - building 259
    - constraints 260
    - Layout() composable 260
    - measurables 260
    - overview 259
    - Placeable 260
    - syntax 259
  - custom layout modifiers 251
    - alignment lines 255
    - baselines 257
    - creating 253
    - default position 253
  - Custom layouts
    - overview 251
    - tutorial 251
  - Custom Theme
    - building 603
  - CutCornerShape 233
- ## D
- DampingRatioHighBouncy 380
  - DampingRatioLowBouncy 380
  - DampingRatioMediumBouncy 380
  - DampingRatioNoBouncy 380
  - Dark Theme 46
    - enable on device 46
  - dashPathEffect() method 389
  - Data Access Object (DAO) 430, 443
  - Data Access Objects 433
  - Database Inspector 437, 455
    - live updates 455
    - SQL query 455
  - Database Rows 424
  - Database Schema 423
  - Database Tables 423
  - data-driven 150
  - DDMS 46
  - Debugging
    - enabling on device 73
  - declarative syntax 149
  - Default Function Parameters 129
  - default position 253
  - derivedStateOf 337
  - Device File Explorer 68
  - device frame 50
  - Device Mirroring 79
    - enabling 79
  - device pairing 77
  - Dispatchers.Default 301
  - Dispatchers.IO 301
  - Dispatchers.Main 300
  - DraggableAnchors 500
  - drag gestures 488
  - drawable
    - folder 190
  - drawArc() function 398
  - drawCircle() function 394
  - drawImage() function 401
  - Drawing

## Index

- arcs 398
- circle 394
- close() 399
- dashed lines 389
- dashPathEffect() 389
- drawArc() 398
- drawImage() 401
- drawPath() 399
- drawPoints() 400
- drawRect() 389
- drawRoundRect() 392
- gradients 395
- images 401
- line 387
- oval 394
- points 400
- rectangle 389
- rotate() 393
- rotation 393
- Drawing text 403
- drawLine() function 388
- drawPath() function 399
- drawPoints() function 400
- drawRect() function 389
- drawRoundRect() function 392
- DrawScope 387
- drawText() function 403, 404
- DropDownMenu 159
- DROP\_LATEST 525
- DROP\_OLDEST 525
- DurationBasedAnimationSpec 363
- Dynamic colors
  - enabling in Android 609
- E**
- Elvis Operator 109
- emit 155
- Empty Compose Activity
  - template 16
- Emulator
  - battery 56
  - cellular configuration 56
  - configuring fingerprints 58
  - directional pad 56
  - extended control options 55
  - Extended controls 55
  - fingerprint 56
  - location configuration 56
  - phone settings 56
  - Resizable 60
  - resize 55
  - rotate 54
  - Screen Record 57
  - Snapshots 57
  - starting 41
  - take screenshot 54
  - toolbar 53
  - toolbar options 53
  - tool window mode 59
  - Virtual Sensors 57
  - zoom 54
- enablePendingPurchases() method 581
- enabling ADB support 73
- enter animations 362
- EnterTransition.None 366
- Errata 2
- Escape Sequences 103
- exit animations 362
- ExitTransition.None 366
- expandHorizontally() 362
- expandIn() 362
- expandVertically() 362
- Extended Control
  - options 55
- F**
- fadeIn() 362
- fadeOut() 363
- Files
  - switching between 82
- fillMaxHeight 204
- fillMaxSize 204
- fillMaxWidth 204
- filter() operator 516

- findStartDestination() method 464
- Fingerprint
  - emulation 58
- Fingerprint authentication
  - device configuration 538
  - overview 537
  - steps to implement 537
  - tutorial 537
- firstVisibleItemIndex 312
- flatMapConcat() operator 521
- flatMapMerge() operator 521
- Float 102
- FloatingActionButton 159
- Flow 511
  - asFlow() builder 513
  - asSharedFlow() 525
  - asStateFlow() 523
  - backgroundn handling 533
  - buffering 518
  - buffer() operator 519
  - builder 513
  - cold 523
  - collect() 517
  - collecting data 517
  - collectLatest() operator 518
  - combine() operator 522
  - conflate() operator 518
  - emit() 513
  - emitting data 513
  - filter() operator 516
  - flatMapConcat() operator 521
  - flatMapMerge() operator 521
  - flattening 520
  - flowOf() builder 513
  - flow of flows 520
  - fold() operator 520
  - hot 523
  - MutableSharedFlow 525
  - MutableStateFlow 523
  - onEach() operator 522
  - reduce() operator 519, 520
  - repeatOnLifecycle 534
  - SharedFlow 524
  - shareIn() function 526
  - single() operator 518
  - StateFlow 523
  - transform() operator 516
  - try/finally 517
  - zip() operator 522
- flow builder 513
- FlowColumn 237, 243, 248
  - cross axis arrangement 249
  - maxItemsInEachColumn 238
  - tutorial 243
- Flow layout
  - arrangement 246
- Flow layouts
  - cross axis arrangement 239
  - fillMaxHeight() 241
  - fillMaxWidth() 241
  - Fractional sizing 241
  - horizontalArrangement 249
  - Item alignment 240
  - item weights 249
  - main axis arrangement 238
  - verticalArrangement 249
  - weight 240
- flowOf() builder 513
- flow of flows 520
- FlowRow 237, 243, 245
  - cross axis arrangement 249
  - horizontalArrangement 246
  - item alignment 246
  - maxItemsInEachRow 238
  - tutorial 243
- Flows
  - combining 522
  - Introduction to 511
- FontWeight 31
- forEachIndexed 249
- forEach loop 262
- Forward-geocoding 551
- Foundation components 158
- Foundation Composables 158

## Index

FragmentActivity 539  
Function Parameters  
    variable number of 129  
Functions 127

## G

Geocoder object 552  
Geocoding 551  
Gestures 485  
    click 485  
    drag 488  
    horizontalScroll() 492  
    overview 485  
    pinch gestures 494  
    PointerInputScope 487  
    rememberScrollableState() function 491  
    rememberScrollState() 492  
    rememberTransformableState() 494  
    rotation gestures 495  
    scrollable() modifier 491  
    scroll modifiers 492  
    taps 487  
    translation gestures 496  
    tutorial 485  
    verticalScroll() 492  
getDebugMessage() 597  
getFromLocation() method 552  
getPurchaseState() method 581  
getStringArray() method 325  
GlobalScope 300  
GNU/Linux 94  
Google Cloud  
    billing account 546  
    new project 547  
GoogleMap 545  
Google Maps Android API 545  
    Controlling the Map Camera 559  
    displaying controls 555  
    Map Markers 558  
    overview 545  
Google Maps SDK 545  
    API Key 549

    Credentials 548  
    enabling 548  
    Maps SDK for Android 548  
Google Play App Signing 566  
Google Play Billing Library 577  
Google Play Console 586  
    Creating an in-app product 586  
    License Testers 587  
Google Play Developer Console 564  
Google Play store 17  
Gradient drawing 395  
Gradle  
    APK signing settings 616  
    Build Variants 612  
    command line tasks 617  
    dependencies 611  
    Manifest Entries 612  
    overview 611  
    sensible defaults 611  
Gradle Build File  
    top level 613  
Gradle Build Files  
    module level 614  
gradle.properties file 612  
Graphics  
    drawing 387  
Grid  
    overview 307  
groupBy() function 311  
guidelines 284

## H

HAL 94  
Hardware Abstraction Layer 94  
Higher-order Functions 131  
horizontalArrangement 220, 222, 249  
HorizontalPager 349  
    animateScrollToPage() 351  
    scrollToPage() 351  
    state 350  
    syntax 349  
horizontalScroll() 492

Hot flows 523

## I

Image 158

- add drawable resource 190
- painterResource method 192

Immutable Variables 104

INAPP 582

In-App Products 578

In-App Purchasing 585

- acknowledgePurchase() method 581
- BillingClient 578
- BillingResult 597
- consumeAsync() method 581
- ConsumeParams 593
- Consuming purchases 592
- enablePendingPurchases() method 581
- getPurchaseState() method 581
- Google Play Billing Library 577
- launchBillingFlow() method 580
- Libraries 585
- newBuilder() method 578
- onBillingServiceDisconnected() callback 590
- onBillingServiceDisconnected() method 579
- onBillingSetupFinished() listener 590
- onProductDetailsResponse() callback 590
- Overview 577
- ProductDetail 580
- ProductDetails 591
- products 578
- ProductType 582
- Purchase Flow 592
- PurchaseResponseListener 582
- PurchasesUpdatedListener 581
- PurchaseUpdatedListener 591
- purchase updates 591
- queryProductDetailsAsync() 590
- queryProductDetailsAsync() method 580
- queryPurchasesAsync() 593
- queryPurchasesAsync() method 582
- startConnection() method 579
- subscriptions 578

tutorial 585

Initializer Blocks 137

In-Memory Database 436

Inner Classes 138

inset() function 391

IntrinsicSize.Max 297

IntrinsicSize.Min 297, 298

intelligent recomposition 161

IntelliJ IDEA 97

Interactive mode 36

Intrinsic measurements 293

IntrinsicSize 293

intrinsic measurements 293

Max 293

Min 293

tutorial 295

is 109

isInitialized property 109

isSystemInDarkTheme() function 176

item() function 308

items() function 308

itemsIndexed() function 308

## J

Java

convert to Kotlin 97

Java Native Interface 95

JetBrains 97

Jetpack Compose

see Compose 149

join() 302

## K

keyboardOptions 420

Keyboard Shortcuts 69

keyframe 364

keyframes 381

KeyframesSpec 381

keyframes() function 381

KeyframesSpec 381

Keystore File

creation 566

## Index

### Kotlin

- accessing class properties 137
- and Java 97
- arithmetic operators 111
- assignment operator 111
- augmented assignment operators 112
- bitwise operators 114
- Boolean 102
- break 122
- breaking from loops 121
- calling class methods 137
- Char 102
- class declaration 133
- class initialization 134
- class properties 134
- Companion Objects 139
- conditional control flow 123
- continue labels 122
- continue statement 122
- control flow 119
- convert from Java 97
- Custom Accessors 137
- data types 101
- decrement operator 112
- Default Function Parameters 129
- defining class methods 134
- do ... while loop 121
- Elvis Operator 109
- equality operators 113
- Escape Sequences 103
- expression syntax 111
- Float 102
- Flow 511
- for-in statement 119
- function calling 128
- Functions 127
- groupBy() function 311
- Higher-order Functions 131
- if ... else ... expressions 124
- if expressions 123
- Immutable Variables 104
- increment operator 112

- inheritance 143
- Initializer Blocks 137
- Inner Classes 138
- introduction 97
- Lambda Expressions 130
- let Function 107
- Local Functions 128
- logical operators 113
- looping 119
- Mutable Variables 104
- Not-Null Assertion 107
- Nullable Type 106
- Overriding inherited methods 146
- playground 98
- Primary Constructor 134
- properties 137
- range operator 114
- Safe Call Operator 106
- Secondary Constructors 134
- Single Expression Functions 128
- String 102
- subclassing 143
- substringBefore() method 327
- Type Annotations 105
- Type Casting 109
- Type Checking 109
- Type Inference 105
- variable parameters 129
- when statement 124
- while loop 120

### L

- Lambda Expressions 130
- lateinit 108
- Late Initialization 108
- launch 301
- launchBillingFlow() method 580
- LaunchedEffect 304
- launchSingleTop 461
- Layout alignment 218
- Layout arrangement 220
- Layout arrangement spacing 222

- Layout components 158
  - Layout() composable 260
  - Layout Editor 149
  - Layout Inspector 68
  - layout modifier 204
  - layout() modifier 290
  - LazyColumn 158, 307
    - creation 308
    - scroll position detection 312
  - LazyHorizontalStaggeredGrid 341, 346
    - syntax 342
  - LazyList
    - tutorial 323
  - Lazy lists 307
    - Scrolling 309
  - LazyListScope 308
    - item() function 308
    - items() function 308
    - itemsIndexed() function 308
    - stickyHeader() function 310
  - LazyListState 312
    - firstVisibleItemIndex 312
  - LazyRow 158, 307
    - creation 308
    - scroll position detection 312
  - LazyVerticalGrid 307
    - adaptive mode 312
    - fixed mode 312
  - LazyVerticalStaggeredGrid 341, 344
    - syntax 341
  - let Function 107
  - libc 95
  - libs.versions.toml file 152
  - License Testers 587
  - Lifecycle.State.CREATED 534
  - Lifecycle.State.DESTROYED 534
  - Lifecycle.State.INITIALIZED 534
  - Lifecycle.State.RESUMED 534
  - Lifecycle.State.STARTED 534
  - LinearProgressIndicator 159
  - lineTo() 399
  - lineTo() function 399
  - linkTo() function 274
  - Linux Kernel 94
  - list devices 73
  - Lists
    - clickable items 331
    - enabling scrolling 309
    - overview 307
  - literals
    - live editing 32
  - LiveData 410
    - observeAsState() 411
  - Live Edit 43
    - disabling 32
    - enabling 32
    - of literals 32
  - Live Templates 90
  - Local Functions 128
  - Location Manager 96
  - Logcat
    - tool window 67
- ## M
- MainActivity.kt file 20
    - template code 29
  - map method 260
  - Maps 545
  - MAP\_TYPE\_HYBRID 554
  - MAP\_TYPE\_NONE 554
  - MAP\_TYPE\_NORMAL 554
  - MAP\_TYPE\_SATELLITE 554
  - MAP\_TYPE\_TERRAIN 554
  - Marker class 545
  - matchParentSize() 233
  - Material Composables 158
  - Material Design 2 599
  - Material Design 2 Theming 599
  - Material Design 3 599
  - Material Design components 159
  - Material Theme Builder 603
  - Material You 599
  - maxValue property 321
  - measurables 260

## Index

- measure() function 404
  - measureTimeMillis() function 518
  - Memory Indicator 83
  - Minimum SDK
    - setting 17
  - ModalDrawer 159
  - Modern Android architecture 407
  - modifier
    - adding to composable 200
    - chaining 199
    - combining 204
    - creating a 198
    - ordering 200
    - tutorial 197
  - Modifier.align() 223
  - Modifier.alignBy() 223
  - modifiers
    - build-in 204
    - overview 197
  - Modifier.weight() 223
  - move() method 559
  - multiple devices
    - testing app on 45
  - MutableLiveData 410
  - MutableSharedFlow 525
  - MutableState 162
  - MutableStateFlow 523
  - mutableStateOf function 155
  - mutableStateOf() function 163
  - MutableTransitionState 367
  - Mutable Variables 104
  - My Location Layer 545
- ## N
- NavHost 459, 471, 479
  - NavHostController 457, 471, 479
  - navigate() method 461
  - Navigation 457
    - BottomNavigation 463
    - BottomNavigationItem 463
    - compose() method 459
    - currentBackStackEntryAsState() method 464
    - declaring routes 468
    - findStartDestination() method 464
    - graph 459
    - launchSingleTop 461
    - NavHost 459, 471
    - NavHostController 457, 471
    - navigate() method 461
    - navigation graph 457
    - NavType 462
    - overview 457
    - passing arguments
    - popUpTo() method 461
    - route 459
    - stack 457, 458
    - start destination 459
    - tutorial 467
  - Navigation Architecture Component 457
  - NavigationBar 480
  - NavigationBarItem 481
  - Navigation bars 463
  - navigation graph 457, 459
  - Navigation Host 459
  - NavType 462
  - newBuilder() method 578
  - Notifications Manager 96
  - Not-Null Assertion 107
  - Nullable Type 106
- ## O
- observeAsState() 411
  - Offset() function 388
  - offset modifier 204
  - onBillingServiceDisconnected() callback 590
  - onBillingServiceDisconnected() method 579
  - onBillingSetupFinished() listener 590
  - onCreate() method 24
  - onEach() operator 522
  - onProductDetailsResponse() callback 590
  - OpenJDK 3
  - Opposing constraints 277
  - OutlinedButton 337
  - OutlinedTextField 413



**P**

- Package Manager 96
- Package name 17
- Packed chain 269
- padding 204
- Pager 349
  - animateScrollToPage() 351
  - scrollToPage() 351
  - state 350
  - syntax , 238
- Pager state 350
- painterResource method 192
- ParagraphStyle 208
- PathEffect 389
- pinch gestures 494
- Placeable 260
- PointerInputScope 487
  - drag gestures 490
  - tap gestures 487
- popUpTo() method 461
- Preview configuration picker 35
- Preview panel 25
  - build and refresh 25
  - Interactive mode 36
  - settings 35
- Primary Constructor 134
- Problems
  - tool window 68
- ProductDetail 580
- ProductDetails 591
- ProductType 582
- Profiler
  - tool window 68
- proguard-rules.pro file 616
- ProGuard Support 612
- project
  - create new 16
  - package name 17
- Project tool window 19, 67
  - Android mode 19
- PurchaseResponseListener 582
- PurchasesUpdatedListener 581, 591

**Q**

- queryProductDetailsAsync() 590
- queryProductDetailsAsync() method 580
- queryPurchaseHistoryAsync() method 582
- queryPurchasesAsync() 593
- queryPurchasesAsync() method 582
- quickboot snapshot 58
- Quick Documentation 89

**R**

- RadioButton 159
- Random
  - nextInt() 244
- Random.nextInt() method 343, 244
- Range Operator 114
- Recent Files Navigation 70
- recomposition 150
  - intelligent recomposition 161
  - overview 161
- RectangleShape 233
- reduce() operator 519, 520
- relativeLineTo() function 399
- Release Preparation 563
- rememberCoroutineScope() function 300, 310, 319
- rememberDraggableState() function 488
- rememberImagePainter() function 328
- remember keyword 163
- rememberPagerState 350
- rememberSaveable keyword 170
- rememberScrollableState() function 491
- rememberScrollState() 492
- rememberScrollState() function 309, 319
- rememberTextMeasurer() function 403
- rememberTransformableState() 494
- rememberTransformationState() function 494
- repeatable() function 365
- RepeatableSpec
  - repeatable() 365
- RepeatMode.Reverse 365
- repeatOnLifecycle 534
- Repository
  - tutorial 439

## Index

- requiredOffset() 501, 507
- Resizable Emulator 60
- Resource Manager , 67
- Reverse-geocoding 551
- Reverse Geocoding 551
- Room
  - Data Access Object (DAO) 430
  - entities 430, 431
  - In-Memory Database 436
  - Repository 429
- Room Database 430
  - tutorial 439
- Room Database Persistence 429
- Room Persistence Library 427
- rotate modifier 204
- rotation gestures 495
- RoundedCornerShape 233
- Row 158
  - Alignment.Bottom 219
  - Alignment.CenterVertically 219
  - Alignment.Top 219
  - Arrangement.Center 220
  - Arrangement.End 220
  - Arrangement.SpaceAround 222
  - Arrangement.SpaceBetween 222
  - Arrangement.SpaceEvenly 222
  - Arrangement.Start 220
  - horizontalArrangement 220
  - Layout alignment 218
  - Layout arrangement 220
  - list 307
  - list example 322
  - overview 216
  - scope 223
  - scope modifiers 223
  - spacing 222
  - tutorial 215
- Row lists 307
- RowScope 223
  - Modifier.align() 223
  - Modifier.alignBy() 223
  - Modifier.alignByBaseline() 223
  - Modifier.paddingFrom() 224
  - Modifier.weight() 224
- Run
  - tool window 67
- runBlocking 301
- Running Devices
  - tool window 79
- S**
- Safe Call Operator 106
- Scaffold 159, 481
  - bottomBar 482
  - AppBar 482
- scaleIn() 363
- scale modifier 204
- scaleOut() 363
- Scope modifiers
  - weights 227
- scrollable modifier 204
- scrollable() modifier 491, 492
- Scroll detection
  - example 333
- scroll modifiers 492
- ScrollState
  - maxValue property 321
  - rememberScrollState() function 309
- scrollToItem(index: Int) 310
- scrollToPage() 351
- scrollTo(value: Int) 309
- SDK Packages 6
- SDK settings 17
- Secondary Constructors 134
- Secure Sockets Layer (SSL) 95
- settings.gradle file 612
- settings.gradle.kts file 612
- Shape 159
- Shapes
  - CircleShape 233
  - CutCornerShape 233
  - RectangleShape 233
  - RoundedCornerShape 233
- SharedFlow 524, 529

- backgroundn handling 533
- DROP\_LATEST 525
- DROP\_OLDEST 525
- in ViewModel 530
- repeatOnLifecycle 534
- SUSPEND 525
- tutorial 529
- shareIn() function 526
- SharingStarted.Eagerly() 526
- SharingStarted.Lazily() 526
- SharingStarted.WhileSubscribed() 526
- showSystemUi 25, 318
- shrinkHorizontally() 363
- shrinkOut() 363
- shrinkVertically() 363
- SideEffect 304
- Side Effects 304
- single() operator 518
- size modifier 204
- slideIn() 363
- slideInHorizontally() 363
- slideInVertically() 363
- slideOut() 363
- slideOutHorizontally() 363
- slideOutVertically() 363
- Slider 159
- Slider component 33
- Slot APIs
  - calling 182
  - declaring 182
  - overview 181
  - tutorial 185
- Snackbar 159
- Snapshots
  - emulator 57
- SpanStyle 207
- Spread chain 268
- Spread inside chain 268
- Spring effects 380
- spring() function 380
- SQL 424
- SQLite 423
  - AVD command-line use 425
  - Columns and Data Types 423
  - overview 424
  - Primary keys 424
- Staggered Grids 341
- startConnection() method 579
- start destination 459
- state 150
  - basics of 161
  - by keyword 164
  - configuration changes 169
  - declaring 162
  - hoisting 167
  - MutableState 162
  - mutableStateOf() function 163
  - overview 161
  - remember keyword 163
  - rememberSaveable 170
  - Unidirectional data flow 165
- StateFlow 523
- stateful 161
- stateful composables 155
- State hoisting 167
- stateless composables 155
- Statement Completion 86
- staticCompositionLocalOf() function 174, 176
- Status Bar Widgets 83
  - Memory Indicator 83
- stickyHeader 334
- stickyHeader() function 310
- Sticky headers
  - adding 334
  - example 333
  - stickyHeader() function 310
- StiffnessHigh 381
- StiffnessLow 381
- StiffnessMedium 381
- StiffnessMediumLow 381
- StiffnessVeryLow 381
- String 102
- Structure
  - tool window 68

## Index

Structured Query Language 424  
Structure tool window 68  
SUBS 582  
subscriptions 578  
subStringBefore() method 327  
supervisorScope 301  
Surface component 23, 231  
SUSPEND 525  
Suspend Functions 300  
Switch 159  
Switcher 70  
system requirements 3

## T

Telephony Manager 96  
Terminal  
    tool window 68  
Text 159  
Text component 156  
TextField 159  
TextMeasurer 403  
    measure() function 404  
TextStyle 420  
Theme  
    building a custom 603  
Theming 599  
    tutorial 605  
TODO  
    tool window 69  
Tool window bars 66  
Tool windows 66  
TopAppBar 159, 482  
trailingIcon 420  
TransformableState 494  
transform() operator 516  
translation gestures 496  
try/finally 517  
tween() function 364  
Type Annotations 105  
Type Casting 109  
Type Checking 109  
Type Inference 105

Type.kt file 602

## U

UI Controllers 408  
UI\_NIGHT\_MODE\_YES 177  
UiSettings class 545  
Unidirectional data flow 165  
updateTransition() function 372, 377, 382  
upload key 566  
USB connection issues  
    resolving 76  
USE\_BIOMETRIC permission 539

## V

Vector Asset  
    add to project 190  
velocityThreshold 501  
Version catalog 151  
    dependencies 153  
    libraries 152  
    libs.versions.toml file 152  
    plugins 152  
    versions 152  
verticalArrangement 220, 222  
VerticalPager  
    animateScrollToPage() 351  
    scrollToPage() 351  
    state 350  
    syntax , 238  
verticalScroll() 492  
verticalScroll() modifier 319  
ViewModel  
    example 414  
    lifecycle library 410, 414, 512, 529  
    LiveData 410  
    observeAsState() 411  
    overview 407  
    tutorial 413  
    using state 408  
viewModel() 410, 416, 450  
ViewModelProvider Factory 450  
ViewModelStoreOwner 450

- viewModel() function 410, 416, 450
- ViewModelProvider Factory 450
- ViewModelScope 300
- ViewModelStoreOwner 450
- View System 96
- Virtual Device Configuration dialog 40
- Virtual Sensors 57
- Visibility animation 359

## **W**

- Weighted chain 268
- Welcome screen 63
- while Loop 120
- Widget Dimensions 269
- WiFi debugging 77
- Wireless debugging 77
- Wireless pairing 77
- withContext 301

## **X**

- XML resource
  - reading an 323

## **Z**

- zip() operator 522

