

# **Android Studio Dolphin Essentials**

---

Kotlin Edition

Android Studio Dolphin Essentials – Kotlin Edition

ISBN-13: 978-1-951442-54-5

© 2022 Neil Smyth / Payload Media, Inc. All Rights Reserved.

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

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

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

Rev: 1.0

## Table of Contents

### **1. Introduction**

- 1.1 Downloading the Code Samples
- 1.2 Feedback
- 1.3 Errata

### **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 Making the Android SDK tools command-line accessible
  - 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. Creating an Example Android App in Android Studio**

- 3.1 About the Project
- 3.2 Creating a New Android Project
- 3.3 Creating an Activity
- 3.4 Defining the Project and SDK Settings
- 3.5 Modifying the Example Application
- 3.6 Modifying the User Interface
- 3.7 Reviewing the Layout and Resource Files
- 3.8 Adding the Kotlin Extensions Plugin
- 3.9 Adding Interaction
- 3.10 Summary

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

- 4.1 About Android Virtual Devices
- 4.2 Starting the Emulator
- 4.3 Running the Application in the AVD
- 4.4 Running on Multiple Devices
- 4.5 Stopping a Running Application
- 4.6 Supporting Dark Theme
- 4.7 Running the Emulator in a Separate Window

## Table of Contents

- 4.8 Enabling the Device Frame
- 4.9 AVD Command-line Creation
- 4.10 Android Virtual Device Configuration Files
- 4.11 Moving and Renaming an Android Virtual Device
- 4.12 Summary

## **5. Using and Configuring the Android Studio AVD Emulator**

- 5.1 The Emulator Environment
- 5.2 Emulator Toolbar Options
- 5.3 Working in Zoom Mode
- 5.4 Resizing the Emulator Window
- 5.5 Extended Control Options
  - 5.5.1 Location
  - 5.5.2 Displays
  - 5.5.3 Cellular
  - 5.5.4 Battery
  - 5.5.5 Camera
  - 5.5.6 Phone
  - 5.5.7 Directional Pad
  - 5.5.8 Microphone
  - 5.5.9 Fingerprint
  - 5.5.10 Virtual Sensors
  - 5.5.11 Snapshots
  - 5.5.12 Record and Playback
  - 5.5.13 Google Play
  - 5.5.14 Settings
  - 5.5.15 Help
- 5.6 Working with Snapshots
- 5.7 Configuring Fingerprint Emulation
- 5.8 The Emulator in Tool Window Mode
- 5.9 Summary

## **6. A Tour of the Android Studio User Interface**

- 6.1 The Welcome Screen
- 6.2 The Main Window
- 6.3 The Tool Windows
- 6.4 Android Studio Keyboard Shortcuts
- 6.5 Switcher and Recent Files Navigation
- 6.6 Changing the Android Studio Theme
- 6.7 Summary

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

- 7.1 An Overview of the Android Debug Bridge (ADB)
- 7.2 Enabling USB Debugging ADB on Android Devices
  - 7.2.1 macOS ADB Configuration
  - 7.2.2 Windows ADB Configuration
  - 7.2.3 Linux adb Configuration
- 7.3 Resolving USB Connection Issues
- 7.4 Enabling Wireless Debugging on Android Devices
- 7.5 Testing the adb Connection
- 7.6 Summary



**8. The Basics of the Android Studio Code Editor**

- 8.1 The Android Studio Editor
- 8.2 Splitting the Editor Window
- 8.3 Code Completion
- 8.4 Statement Completion
- 8.5 Parameter Information
- 8.6 Parameter Name Hints
- 8.7 Code Generation
- 8.8 Code Folding
- 8.9 Quick Documentation Lookup
- 8.10 Code Reformatting
- 8.11 Finding Sample Code
- 8.12 Live Templates
- 8.13 Summary

**9. An Overview of the Android Architecture**

- 9.1 The Android Software Stack
- 9.2 The Linux Kernel
- 9.3 Android Runtime – ART
- 9.4 Android Libraries
  - 9.4.1 C/C++ Libraries
- 9.5 Application Framework
- 9.6 Applications
- 9.7 Summary

**10. The Anatomy of an Android Application**

- 10.1 Android Activities
- 10.2 Android Fragments
- 10.3 Android Intents
- 10.4 Broadcast Intents
- 10.5 Broadcast Receivers
- 10.6 Android Services
- 10.7 Content Providers
- 10.8 The Application Manifest
- 10.9 Application Resources
- 10.10 Application Context
- 10.11 Summary

**11. An Introduction to Kotlin**

- 11.1 What is Kotlin?
- 11.2 Kotlin and Java
- 11.3 Converting from Java to Kotlin
- 11.4 Kotlin and Android Studio
- 11.5 Experimenting with Kotlin
- 11.6 Semi-colons in Kotlin
- 11.7 Summary

**12. Kotlin Data Types, Variables, and Nullability**

- 12.1 Kotlin Data Types
  - 12.1.1 Integer Data Types
  - 12.1.2 Floating-Point Data Types

## Table of Contents

- 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 Android View Binding**

- 18.1 Find View by Id and Synthetic Properties
- 18.2 View Binding
- 18.3 Converting the AndroidSample project
- 18.4 Enabling View Binding
- 18.5 Using View Binding
- 18.6 Choosing an Option
- 18.7 View Binding in the Book Examples
- 18.8 Migrating a Project to View Binding
- 18.9 Summary

**19. Understanding Android Application and Activity Lifecycles**

- 19.1 Android Applications and Resource Management

## Table of Contents

- 19.2 Android Process States
  - 19.2.1 Foreground Process
  - 19.2.2 Visible Process
  - 19.2.3 Service Process
  - 19.2.4 Background Process
  - 19.2.5 Empty Process
- 19.3 Inter-Process Dependencies
- 19.4 The Activity Lifecycle
- 19.5 The Activity Stack
- 19.6 Activity States
- 19.7 Configuration Changes
- 19.8 Handling State Change
- 19.9 Summary

## **20. Handling Android Activity State Changes**

- 20.1 New vs. Old Lifecycle Techniques
- 20.2 The Activity and Fragment Classes
- 20.3 Dynamic State vs. Persistent State
- 20.4 The Android Lifecycle Methods
- 20.5 Lifetimes
- 20.6 Foldable Devices and Multi-Resume
- 20.7 Disabling Configuration Change Restarts
- 20.8 Lifecycle Method Limitations
- 20.9 Summary

## **21. Android Activity State Changes by Example**

- 21.1 Creating the State Change Example Project
- 21.2 Designing the User Interface
- 21.3 Overriding the Activity Lifecycle Methods
- 21.4 Filtering the Logcat Panel
- 21.5 Running the Application
- 21.6 Experimenting with the Activity
- 21.7 Summary

## **22. Saving and Restoring the State of an Android Activity**

- 22.1 Saving Dynamic State
- 22.2 Default Saving of User Interface State
- 22.3 The Bundle Class
- 22.4 Saving the State
- 22.5 Restoring the State
- 22.6 Testing the Application
- 22.7 Summary

## **23. Understanding Android Views, View Groups and Layouts**

- 23.1 Designing for Different Android Devices
- 23.2 Views and View Groups
- 23.3 Android Layout Managers
- 23.4 The View Hierarchy
- 23.5 Creating User Interfaces
- 23.6 Summary

## **24. A Guide to the Android Studio Layout Editor Tool**

- 24.1 Basic vs. Empty Activity Templates
- 24.2 The Android Studio Layout Editor
- 24.3 Design Mode
- 24.4 The Palette
- 24.5 Design Mode and Layout Views
- 24.6 Night Mode
- 24.7 Code Mode
- 24.8 Split Mode
- 24.9 Setting Attributes
- 24.10 Transforms
- 24.11 Tools Visibility Toggles
- 24.12 Converting Views
- 24.13 Displaying Sample Data
- 24.14 Creating a Custom Device Definition
- 24.15 Changing the Current Device
- 24.16 Layout Validation (Multi Preview)
- 24.17 Summary

## **25. A Guide to the Android ConstraintLayout**

- 25.1 How ConstraintLayout Works
  - 25.1.1 Constraints
  - 25.1.2 Margins
  - 25.1.3 Opposing Constraints
  - 25.1.4 Constraint Bias
  - 25.1.5 Chains
  - 25.1.6 Chain Styles
- 25.2 Baseline Alignment
- 25.3 Configuring Widget Dimensions
- 25.4 Guideline Helper
- 25.5 Group Helper
- 25.6 Barrier Helper
- 25.7 Flow Helper
- 25.8 Ratios
- 25.9 ConstraintLayout Advantages
- 25.10 ConstraintLayout Availability
- 25.11 Summary

## **26. A Guide to Using ConstraintLayout in Android Studio**

- 26.1 Design and Layout Views
- 26.2 Autoconnect Mode
- 26.3 Inference Mode
- 26.4 Manipulating Constraints Manually
- 26.5 Adding Constraints in the Inspector
- 26.6 Viewing Constraints in the Attributes Window
- 26.7 Deleting Constraints
- 26.8 Adjusting Constraint Bias
- 26.9 Understanding ConstraintLayout Margins
- 26.10 The Importance of Opposing Constraints and Bias
- 26.11 Configuring Widget Dimensions
- 26.12 Design Time Tools Positioning

## Table of Contents

- 26.13 Adding Guidelines
- 26.14 Adding Barriers
- 26.15 Adding a Group
- 26.16 Working with the Flow Helper
- 26.17 Widget Group Alignment and Distribution
- 26.18 Converting other Layouts to ConstraintLayout
- 26.19 Summary

## **27. Working with ConstraintLayout Chains and Ratios in Android Studio**

- 27.1 Creating a Chain
- 27.2 Changing the Chain Style
- 27.3 Spread Inside Chain Style
- 27.4 Packed Chain Style
- 27.5 Packed Chain Style with Bias
- 27.6 Weighted Chain
- 27.7 Working with Ratios
- 27.8 Summary

## **28. An Android Studio Layout Editor ConstraintLayout Tutorial**

- 28.1 An Android Studio Layout Editor Tool Example
- 28.2 Creating a New Activity
- 28.3 Preparing the Layout Editor Environment
- 28.4 Adding the Widgets to the User Interface
- 28.5 Adding the Constraints
- 28.6 Testing the Layout
- 28.7 Using the Layout Inspector
- 28.8 Summary

## **29. Manual XML Layout Design in Android Studio**

- 29.1 Manually Creating an XML Layout
- 29.2 Manual XML vs. Visual Layout Design
- 29.3 Summary

## **30. Managing Constraints using Constraint Sets**

- 30.1 Kotlin Code vs. XML Layout Files
- 30.2 Creating Views
- 30.3 View Attributes
- 30.4 Constraint Sets
  - 30.4.1 Establishing Connections
  - 30.4.2 Applying Constraints to a Layout
  - 30.4.3 Parent Constraint Connections
  - 30.4.4 Sizing Constraints
  - 30.4.5 Constraint Bias
  - 30.4.6 Alignment Constraints
  - 30.4.7 Copying and Applying Constraint Sets
  - 30.4.8 ConstraintLayout Chains
  - 30.4.9 Guidelines
  - 30.4.10 Removing Constraints
  - 30.4.11 Scaling
  - 30.4.12 Rotation
- 30.5 Summary

**31. An Android ConstraintSet Tutorial**

- 31.1 Creating the Example Project in Android Studio
- 31.2 Adding Views to an Activity
- 31.3 Setting View Attributes
- 31.4 Creating View IDs
- 31.5 Configuring the Constraint Set
- 31.6 Adding the EditText View
- 31.7 Converting Density Independent Pixels (dp) to Pixels (px)
- 31.8 Summary

**32. A Guide to using Apply Changes in Android Studio**

- 32.1 Introducing Apply Changes
- 32.2 Understanding Apply Changes Options
- 32.3 Using Apply Changes
- 32.4 Configuring Apply Changes Fallback Settings
- 32.5 An Apply Changes Tutorial
- 32.6 Using Apply Code Changes
- 32.7 Using Apply Changes and Restart Activity
- 32.8 Using Run App
- 32.9 Summary

**33. An Overview and Example of Android Event Handling**

- 33.1 Understanding Android Events
- 33.2 Using the android.onClick Resource
- 33.3 Event Listeners and Callback Methods
- 33.4 An Event Handling Example
- 33.5 Designing the User Interface
- 33.6 The Event Listener and Callback Method
- 33.7 Consuming Events
- 33.8 Summary

**34. Android Touch and Multi-touch Event Handling**

- 34.1 Intercepting Touch Events
- 34.2 The MotionEvent Object
- 34.3 Understanding Touch Actions
- 34.4 Handling Multiple Touches
- 34.5 An Example Multi-Touch Application
- 34.6 Designing the Activity User Interface
- 34.7 Implementing the Touch Event Listener
- 34.8 Running the Example Application
- 34.9 Summary

**35. Detecting Common Gestures Using the Android Gesture Detector Class**

- 35.1 Implementing Common Gesture Detection
- 35.2 Creating an Example Gesture Detection Project
- 35.3 Implementing the Listener Class
- 35.4 Creating the GestureDetectorCompat Instance
- 35.5 Implementing the onTouchEvent() Method
- 35.6 Testing the Application
- 35.7 Summary

## **36. Implementing Custom Gesture and Pinch Recognition on Android**

- 36.1 The Android Gesture Builder Application
- 36.2 The GestureOverlayView Class
- 36.3 Detecting Gestures
- 36.4 Identifying Specific Gestures
- 36.5 Installing and Running the Gesture Builder Application
- 36.6 Creating a Gestures File
- 36.7 Creating the Example Project
- 36.8 Extracting the Gestures File from the SD Card
- 36.9 Adding the Gestures File to the Project
- 36.10 Designing the User Interface
- 36.11 Loading the Gestures File
- 36.12 Registering the Event Listener
- 36.13 Implementing the onGesturePerformed Method
- 36.14 Testing the Application
- 36.15 Configuring the GestureOverlayView
- 36.16 Intercepting Gestures
- 36.17 Detecting Pinch Gestures
- 36.18 A Pinch Gesture Example Project
- 36.19 Summary

## **37. An Introduction to Android Fragments**

- 37.1 What is a Fragment?
- 37.2 Creating a Fragment
- 37.3 Adding a Fragment to an Activity using the Layout XML File
- 37.4 Adding and Managing Fragments in Code
- 37.5 Handling Fragment Events
- 37.6 Implementing Fragment Communication
- 37.7 Summary

## **38. Using Fragments in Android Studio - An Example**

- 38.1 About the Example Fragment Application
- 38.2 Creating the Example Project
- 38.3 Creating the First Fragment Layout
- 38.4 Migrating a Fragment to View Binding
- 38.5 Adding the Second Fragment
- 38.6 Adding the Fragments to the Activity
- 38.7 Making the Toolbar Fragment Talk to the Activity
- 38.8 Making the Activity Talk to the Text Fragment
- 38.9 Testing the Application
- 38.10 Summary

## **39. Modern Android App Architecture with Jetpack**

- 39.1 What is Android Jetpack?
- 39.2 The “Old” Architecture
- 39.3 Modern Android Architecture
- 39.4 The ViewModel Component
- 39.5 The LiveData Component
- 39.6 ViewModel Saved State
- 39.7 LiveData and Data Binding



- 39.8 Android Lifecycles
- 39.9 Repository Modules
- 39.10 Summary

#### **40. An Android Jetpack ViewModel Tutorial**

- 40.1 About the Project
- 40.2 Creating the ViewModel Example Project
- 40.3 Reviewing the Project
  - 40.3.1 The Main Activity
  - 40.3.2 The Content Fragment
  - 40.3.3 The ViewModel
- 40.4 Designing the Fragment Layout
- 40.5 Implementing the View Model
- 40.6 Associating the Fragment with the View Model
- 40.7 Modifying the Fragment
- 40.8 Accessing the ViewModel Data
- 40.9 Testing the Project
- 40.10 Summary

#### **41. An Android Jetpack LiveData Tutorial**

- 41.1 LiveData - A Recap
- 41.2 Adding LiveData to the ViewModel
- 41.3 Implementing the Observer
- 41.4 Summary

#### **42. An Overview of Android Jetpack Data Binding**

- 42.1 An Overview of Data Binding
- 42.2 The Key Components of Data Binding
  - 42.2.1 The Project Build Configuration
  - 42.2.2 The Data Binding Layout File
  - 42.2.3 The Layout File Data Element
  - 42.2.4 The Binding Classes
  - 42.2.5 Data Binding Variable Configuration
  - 42.2.6 Binding Expressions (One-Way)
  - 42.2.7 Binding Expressions (Two-Way)
  - 42.2.8 Event and Listener Bindings
- 42.3 Summary

#### **43. An Android Jetpack Data Binding Tutorial**

- 43.1 Removing the Redundant Code
- 43.2 Enabling Data Binding
- 43.3 Adding the Layout Element
- 43.4 Adding the Data Element to Layout File
- 43.5 Working with the Binding Class
- 43.6 Assigning the ViewModel Instance to the Data Binding Variable
- 43.7 Adding Binding Expressions
- 43.8 Adding the Conversion Method
- 43.9 Adding a Listener Binding
- 43.10 Testing the App
- 43.11 Summary

#### **44. An Android ViewModel Saved State Tutorial**

## Table of Contents

- 44.1 Understanding ViewModel State Saving
- 44.2 Implementing ViewModel State Saving
- 44.3 Saving and Restoring State
- 44.4 Adding Saved State Support to the ViewModelDemo Project
- 44.5 Summary

## **45. Working with Android Lifecycle-Aware Components**

- 45.1 Lifecycle Awareness
- 45.2 Lifecycle Owners
- 45.3 Lifecycle Observers
- 45.4 Lifecycle States and Events
- 45.5 Summary

## **46. An Android Jetpack Lifecycle Awareness Tutorial**

- 46.1 Creating the Example Lifecycle Project
- 46.2 Creating a Lifecycle Observer
- 46.3 Adding the Observer
- 46.4 Testing the Observer
- 46.5 Creating a Lifecycle Owner
- 46.6 Testing the Custom Lifecycle Owner
- 46.7 Summary

## **47. An Overview of the Navigation Architecture Component**

- 47.1 Understanding Navigation
- 47.2 Declaring a Navigation Host
- 47.3 The Navigation Graph
- 47.4 Accessing the Navigation Controller
- 47.5 Triggering a Navigation Action
- 47.6 Passing Arguments
- 47.7 Summary

## **48. An Android Jetpack Navigation Component Tutorial**

- 48.1 Creating the NavigationDemo Project
- 48.2 Adding Navigation to the Build Configuration
- 48.3 Creating the Navigation Graph Resource File
- 48.4 Declaring a Navigation Host
- 48.5 Adding Navigation Destinations
- 48.6 Designing the Destination Fragment Layouts
- 48.7 Adding an Action to the Navigation Graph
- 48.8 Implement the OnFragmentInteractionListener
- 48.9 Adding View Binding Support to the Destination Fragments
- 48.10 Triggering the Action
- 48.11 Passing Data Using Safeargs
- 48.12 Summary

## **49. An Introduction to MotionLayout**

- 49.1 An Overview of MotionLayout
- 49.2 MotionLayout
- 49.3 MotionScene
- 49.4 Configuring ConstraintSets
- 49.5 Custom Attributes

- 49.6 Triggering an Animation
- 49.7 Arc Motion
- 49.8 Keyframes
  - 49.8.1 Attribute Keyframes
  - 49.8.2 Position Keyframes
- 49.9 Time Linearity
- 49.10 KeyTrigger
- 49.11 Cycle and Time Cycle Keyframes
- 49.12 Starting an Animation from Code
- 49.13 Summary

## **50. An Android MotionLayout Editor Tutorial**

- 50.1 Creating the MotionLayoutDemo Project
- 50.2 ConstraintLayout to MotionLayout Conversion
- 50.3 Configuring Start and End Constraints
- 50.4 Previewing the MotionLayout Animation
- 50.5 Adding an OnClick Gesture
- 50.6 Adding an Attribute Keyframe to the Transition
- 50.7 Adding a CustomAttribute to a Transition
- 50.8 Adding Position Keyframes
- 50.9 Summary

## **51. A MotionLayout KeyCycle Tutorial**

- 51.1 An Overview of Cycle Keyframes
- 51.2 Using the Cycle Editor
- 51.3 Creating the KeyCycleDemo Project
- 51.4 Configuring the Start and End Constraints
- 51.5 Creating the Cycles
- 51.6 Previewing the Animation
- 51.7 Adding the KeyFrameSet to the MotionScene
- 51.8 Summary

## **52. Working with the Floating Action Button and Snackbar**

- 52.1 The Material Design
- 52.2 The Design Library
- 52.3 The Floating Action Button (FAB)
- 52.4 The Snackbar
- 52.5 Creating the Example Project
- 52.6 Reviewing the Project
- 52.7 Removing Navigation Features
- 52.8 Changing the Floating Action Button
- 52.9 Adding an Action to the Snackbar
- 52.10 Summary

## **53. Creating a Tabbed Interface using the TabLayout Component**

- 53.1 An Introduction to the ViewPager2
- 53.2 An Overview of the TabLayout Component
- 53.3 Creating the TabLayoutDemo Project
- 53.4 Creating the First Fragment
- 53.5 Duplicating the Fragments
- 53.6 Adding the TabLayout and ViewPager2

## Table of Contents

- 53.7 Creating the Pager Adapter
- 53.8 Performing the Initialization Tasks
- 53.9 Testing the Application
- 53.10 Customizing the TabLayout
- 53.11 Summary

## **54. Working with the RecyclerView and CardView Widgets**

- 54.1 An Overview of the RecyclerView
- 54.2 An Overview of the CardView
- 54.3 Summary

## **55. An Android RecyclerView and CardView Tutorial**

- 55.1 Creating the CardDemo Project
- 55.2 Modifying the Basic Activity Project
- 55.3 Designing the CardView Layout
- 55.4 Adding the RecyclerView
- 55.5 Adding the Image Files
- 55.6 Creating the RecyclerView Adapter
- 55.7 Initializing the RecyclerView Component
- 55.8 Testing the Application
- 55.9 Responding to Card Selections
- 55.10 Summary

## **56. Working with the AppBar and Collapsing Toolbar Layouts**

- 56.1 The Anatomy of an AppBar
- 56.2 The Example Project
- 56.3 Coordinating the RecyclerView and Toolbar
- 56.4 Introducing the Collapsing Toolbar Layout
- 56.5 Changing the Title and Scrim Color
- 56.6 Summary

## **57. An Overview of Android Intents**

- 57.1 An Overview of Intents
- 57.2 Explicit Intents
- 57.3 Returning Data from an Activity
- 57.4 Implicit Intents
- 57.5 Using Intent Filters
- 57.6 Automatic Link Verification
- 57.7 Manually Enabling Links
- 57.8 Checking Intent Availability
- 57.9 Summary

## **58. Android Explicit Intents – A Worked Example**

- 58.1 Creating the Explicit Intent Example Application
- 58.2 Designing the User Interface Layout for MainActivity
- 58.3 Creating the Second Activity Class
- 58.4 Designing the User Interface Layout for SecondActivity
- 58.5 Reviewing the Application Manifest File
- 58.6 Creating the Intent
- 58.7 Extracting Intent Data
- 58.8 Launching SecondActivity as a Sub-Activity

- 58.9 Returning Data from a Sub-Activity
- 58.10 Testing the Application
- 58.11 Summary

## **59. Android Implicit Intents – A Worked Example**

- 59.1 Creating the Android Studio Implicit Intent Example Project
- 59.2 Designing the User Interface
- 59.3 Creating the Implicit Intent
- 59.4 Adding a Second Matching Activity
- 59.5 Adding the Web View to the UI
- 59.6 Obtaining the Intent URL
- 59.7 Modifying the MyWebView Project Manifest File
- 59.8 Installing the MyWebView Package on a Device
- 59.9 Testing the Application
- 59.10 Manually Enabling the Link
- 59.11 Automatic Link Verification
- 59.12 Summary

## **60. Android Broadcast Intents and Broadcast Receivers**

- 60.1 An Overview of Broadcast Intents
- 60.2 An Overview of Broadcast Receivers
- 60.3 Obtaining Results from a Broadcast
- 60.4 Sticky Broadcast Intents
- 60.5 The Broadcast Intent Example
- 60.6 Creating the Example Application
- 60.7 Creating and Sending the Broadcast Intent
- 60.8 Creating the Broadcast Receiver
- 60.9 Registering the Broadcast Receiver
- 60.10 Testing the Broadcast Example
- 60.11 Listening for System Broadcasts
- 60.12 Summary

## **61. An Introduction to Kotlin Coroutines**

- 61.1 What are Coroutines?
- 61.2 Threads vs Coroutines
- 61.3 Coroutine Scope
- 61.4 Suspend Functions
- 61.5 Coroutine Dispatchers
- 61.6 Coroutine Builders
- 61.7 Jobs
- 61.8 Coroutines – Suspending and Resuming
- 61.9 Returning Results from a Coroutine
- 61.10 Using withContext
- 61.11 Coroutine Channel Communication
- 61.12 Summary

## **62. An Android Kotlin Coroutines Tutorial**

- 62.1 Creating the Coroutine Example Application
- 62.2 Adding Coroutine Support to the Project
- 62.3 Designing the User Interface
- 62.4 Implementing the SeekBar

## Table of Contents

- 62.5 Adding the Suspend Function
- 62.6 Implementing the launchCoroutines Method
- 62.7 Testing the App
- 62.8 Summary

## 63. An Overview of Android Services

- 63.1 Intent Service
- 63.2 Bound Service
- 63.3 The Anatomy of a Service
- 63.4 Controlling Destroyed Service Restart Options
- 63.5 Declaring a Service in the Manifest File
- 63.6 Starting a Service Running on System Startup
- 63.7 Summary

## 64. Android Local Bound Services – A Worked Example

- 64.1 Understanding Bound Services
- 64.2 Bound Service Interaction Options
- 64.3 A Local Bound Service Example
- 64.4 Adding a Bound Service to the Project
- 64.5 Implementing the Binder
- 64.6 Binding the Client to the Service
- 64.7 Completing the Example
- 64.8 Testing the Application
- 64.9 Summary

## 65. Android Remote Bound Services – A Worked Example

- 65.1 Client to Remote Service Communication
- 65.2 Creating the Example Application
- 65.3 Designing the User Interface
- 65.4 Implementing the Remote Bound Service
- 65.5 Configuring a Remote Service in the Manifest File
- 65.6 Launching and Binding to the Remote Service
- 65.7 Sending a Message to the Remote Service
- 65.8 Summary

## 66. An Introduction to Kotlin Flow

- 66.1 Understanding Flows
- 66.2 Creating the Sample Project
- 66.3 Adding the Kotlin Lifecycle Library
- 66.4 Declaring a Flow
- 66.5 Emitting Flow Data
- 66.6 Collecting Flow Data
- 66.7 Adding a Flow Buffer
- 66.8 Transforming Data with Intermediaries
- 66.9 Terminal Flow Operators
- 66.10 Flow Flattening
- 66.11 Combining Multiple Flows
- 66.12 Hot and Cold Flows
- 66.13 StateFlow
- 66.14 SharedFlow
- 66.15 Summary

**67. An Android SharedFlow Tutorial**

- 67.1 About the Project
- 67.2 Creating the SharedFlowDemo Project
- 67.3 Designing the User Interface Layout
- 67.4 Adding the List Row Layout
- 67.5 Adding the RecyclerView Adapter
- 67.6 Completing the ViewModel
- 67.7 Modifying the Main Fragment for View Binding
- 67.8 Collecting the Flow Values
- 67.9 Testing the SharedFlowDemo App
- 67.10 Handling Flows in the Background
- 67.11 Summary

**68. An Android Notifications Tutorial**

- 68.1 An Overview of Notifications
- 68.2 Creating the NotifyDemo Project
- 68.3 Designing the User Interface
- 68.4 Creating the Second Activity
- 68.5 Creating a Notification Channel
- 68.6 Creating and Issuing a Notification
- 68.7 Launching an Activity from a Notification
- 68.8 Adding Actions to a Notification
- 68.9 Bundled Notifications
- 68.10 Summary

**69. An Android Direct Reply Notification Tutorial**

- 69.1 Creating the DirectReply Project
- 69.2 Designing the User Interface
- 69.3 Creating the Notification Channel
- 69.4 Building the RemoteInput Object
- 69.5 Creating the PendingIntent
- 69.6 Creating the Reply Action
- 69.7 Receiving Direct Reply Input
- 69.8 Updating the Notification
- 69.9 Summary

**70. An Overview of Android SQLite Databases**

- 70.1 Understanding Database Tables
- 70.2 Introducing Database Schema
- 70.3 Columns and Data Types
- 70.4 Database Rows
- 70.5 Introducing Primary Keys
- 70.6 What is SQLite?
- 70.7 Structured Query Language (SQL)
- 70.8 Trying SQLite on an Android Virtual Device (AVD)
- 70.9 The Android Room Persistence Library
- 70.10 Summary

**71. The Android Room Persistence Library**

- 71.1 Revisiting Modern App Architecture
- 71.2 Key Elements of Room Database Persistence

## Table of Contents

|  |  |
|--|--|
| 71.2.1 Repository  |  |
| 71.2.2 Room Database   |  |
| 71.2.3 Data Access Object (DAO)  |  |
| 71.2.4 Entities  |  |
| 71.2.5 SQLite Database   |  |
| 71.3 Understanding Entities  |  |
| 71.4 Data Access Objects   |  |
| 71.5 The Room Database   |  |
| 71.6 The Repository  |  |
| 71.7 In-Memory Databases   |  |
| 71.8 Database Inspector  |  |
| 71.9 Summary   |  |
| <b>72. An Android TableLayout and TableRow Tutorial</b>                              |  |
| 72.1 The TableLayout and TableRow Layout Views                                       |  |
| 72.2 Creating the Room Database Project  |  |
| 72.3 Converting to a LinearLayout  |  |
| 72.4 Adding the TableLayout to the User Interface                                    |  |
| 72.5 Configuring the TableRows   |  |
| 72.6 Adding the Button Bar to the Layout   |  |
| 72.7 Adding the RecyclerView   |  |
| 72.8 Adjusting the Layout Margins  |  |
| 72.9 Summary   |  |
| <b>73. An Android Room Database and Repository Tutorial</b>                          |  |
| 73.1 About the RoomDemo Project  |  |
| 73.2 Modifying the Build Configuration   |  |
| 73.3 Building the Entity   |  |
| 73.4 Creating the Data Access Object   |  |
| 73.5 Adding the Room Database  |  |
| 73.6 Adding the Repository   |  |
| 73.7 Modifying the ViewModel   |  |
| 73.8 Creating the Product Item Layout  |  |
| 73.9 Adding the RecyclerView Adapter   |  |
| 73.10 Preparing the Main Fragment  |  |
| 73.11 Adding the Button Listeners  |  |
| 73.12 Adding LiveData Observers  |  |
| 73.13 Initializing the RecyclerView  |  |
| 73.14 Testing the RoomDemo App   |  |
| 73.15 Using the Database Inspector   |  |
| 73.16 Summary  |  |
| <b>74. Video Playback on Android using the VideoView and MediaController Classes</b> |  |
| 74.1 Introducing the Android VideoView Class   |  |
| 74.2 Introducing the Android MediaController Class                                   |  |
| 74.3 Creating the Video Playback Example   |  |
| 74.4 Designing the VideoPlayer Layout  |  |
| 74.5 Downloading the Video File  |  |
| 74.6 Configuring the VideoView   |  |
| 74.7 Adding the MediaController to the Video View                                    |  |
| 74.8 Setting up the onPreparedListener   |  |



74.9 Summary

## **75. Android Picture-in-Picture Mode**

75.1 Picture-in-Picture Features

75.2 Enabling Picture-in-Picture Mode

75.3 Configuring Picture-in-Picture Parameters

75.4 Entering Picture-in-Picture Mode

75.5 Detecting Picture-in-Picture Mode Changes

75.6 Adding Picture-in-Picture Actions

75.7 Summary

## **76. An Android Picture-in-Picture Tutorial**

76.1 Adding Picture-in-Picture Support to the Manifest

76.2 Adding a Picture-in-Picture Button

76.3 Entering Picture-in-Picture Mode

76.4 Detecting Picture-in-Picture Mode Changes

76.5 Adding a Broadcast Receiver

76.6 Adding the PiP Action

76.7 Testing the Picture-in-Picture Action

76.8 Summary

## **77. Making Runtime Permission Requests in Android**

77.1 Understanding Normal and Dangerous Permissions

77.2 Creating the Permissions Example Project

77.3 Checking for a Permission

77.4 Requesting Permission at Runtime

77.5 Providing a Rationale for the Permission Request

77.6 Testing the Permissions App

77.7 Summary

## **78. Android Audio Recording and Playback using MediaPlayer and MediaRecorder**

78.1 Playing Audio

78.2 Recording Audio and Video using the MediaRecorder Class

78.3 About the Example Project

78.4 Creating the AudioApp Project

78.5 Designing the User Interface

78.6 Checking for Microphone Availability

78.7 Initializing the Activity

78.8 Implementing the recordAudio() Method

78.9 Implementing the stopAudio() Method

78.10 Implementing the playAudio() method

78.11 Configuring and Requesting Permissions

78.12 Testing the Application

78.13 Summary

## **79. Printing with the Android Printing Framework**

79.1 The Android Printing Architecture

79.2 The Print Service Plugins

79.3 Google Cloud Print

79.4 Printing to Google Drive

79.5 Save as PDF

## Table of Contents

- 79.6 Printing from Android Devices
- 79.7 Options for Building Print Support into Android Apps
  - 79.7.1 Image Printing
  - 79.7.2 Creating and Printing HTML Content
  - 79.7.3 Printing a Web Page
  - 79.7.4 Printing a Custom Document
- 79.8 Summary

## **80. An Android HTML and Web Content Printing Example**

- 80.1 Creating the HTML Printing Example Application
- 80.2 Printing Dynamic HTML Content
- 80.3 Creating the Web Page Printing Example
- 80.4 Removing the Floating Action Button
- 80.5 Removing Navigation Features
- 80.6 Designing the User Interface Layout
- 80.7 Accessing the WebView from the Main Activity
- 80.8 Loading the Web Page into the WebView
- 80.9 Adding the Print Menu Option
- 80.10 Summary

## **81. A Guide to Android Custom Document Printing**

- 81.1 An Overview of Android Custom Document Printing
  - 81.1.1 Custom Print Adapters
- 81.2 Preparing the Custom Document Printing Project
- 81.3 Creating the Custom Print Adapter
- 81.4 Implementing the onLayout() Callback Method
- 81.5 Implementing the onWrite() Callback Method
- 81.6 Checking a Page is in Range
- 81.7 Drawing the Content on the Page Canvas
- 81.8 Starting the Print Job
- 81.9 Testing the Application
- 81.10 Summary

## **82. An Introduction to Android App Links**

- 82.1 An Overview of Android App Links
- 82.2 App Link Intent Filters
- 82.3 Handling App Link Intents
- 82.4 Associating the App with a Website
- 82.5 Summary

## **83. An Android Studio App Links Tutorial**

- 83.1 About the Example App
- 83.2 The Database Schema
- 83.3 Loading and Running the Project
- 83.4 Adding the URL Mapping
- 83.5 Adding the Intent Filter
- 83.6 Adding Intent Handling Code
- 83.7 Testing the App
- 83.8 Creating the Digital Asset Links File
- 83.9 Testing the App Link
- 83.10 Summary

**84. An Android Biometric Authentication Tutorial**

- 84.1 An Overview of Biometric Authentication
- 84.2 Creating the Biometric Authentication Project
- 84.3 Configuring Device Fingerprint Authentication
- 84.4 Adding the Biometric Permission to the Manifest File
- 84.5 Designing the User Interface
- 84.6 Adding a Toast Convenience Method
- 84.7 Checking the Security Settings
- 84.8 Configuring the Authentication Callbacks
- 84.9 Adding the CancellationSignal
- 84.10 Starting the Biometric Prompt
- 84.11 Testing the Project
- 84.12 Summary

**85. Creating, Testing and Uploading an Android App Bundle**

- 85.1 The Release Preparation Process
- 85.2 Android App Bundles
- 85.3 Register for a Google Play Developer Console Account
- 85.4 Configuring the App in the Console
- 85.5 Enabling Google Play App Signing
- 85.6 Creating a Keystore File
- 85.7 Creating the Android App Bundle
- 85.8 Generating Test APK Files
- 85.9 Uploading the App Bundle to the Google Play Developer Console
- 85.10 Exploring the App Bundle
- 85.11 Managing Testers
- 85.12 Rolling the App Out for Testing
- 85.13 Uploading New App Bundle Revisions
- 85.14 Analyzing the App Bundle File
- 85.15 Summary

**86. An Overview of Android In-App Billing**

- 86.1 Preparing a Project for In-App Purchasing
- 86.2 Creating In-App Products and Subscriptions
- 86.3 Billing Client Initialization
- 86.4 Connecting to the Google Play Billing Library
- 86.5 Querying Available Products
- 86.6 Starting the Purchase Process
- 86.7 Completing the Purchase
- 86.8 Querying Previous Purchases
- 86.9 Summary

**87. An Android In-App Purchasing Tutorial**

- 87.1 About the In-App Purchasing Example Project
- 87.2 Creating the InAppPurchase Project
- 87.3 Adding Libraries to the Project
- 87.4 Designing the User Interface
- 87.5 Adding the App to the Google Play Store
- 87.6 Creating an In-App Product
- 87.7 Enabling License Testers

## Table of Contents

- 87.8 Initializing the Billing Client
- 87.9 Querying the Product
- 87.10 Launching the Purchase Flow
- 87.11 Handling Purchase Updates
- 87.12 Consuming the Product
- 87.13 Restoring a Previous Purchase
- 87.14 Testing the App
- 87.15 Troubleshooting
- 87.16 Summary

## **88. An Overview of Android Dynamic Feature Modules**

- 88.1 An Overview of Dynamic Feature Modules
- 88.2 Dynamic Feature Module Architecture
- 88.3 Creating a Dynamic Feature Module
- 88.4 Converting an Existing Module for Dynamic Delivery
- 88.5 Working with Dynamic Feature Modules
- 88.6 Handling Large Dynamic Feature Modules
- 88.7 Summary

## **89. An Android Studio Dynamic Feature Tutorial**

- 89.1 Creating the DynamicFeature Project
- 89.2 Adding Dynamic Feature Support to the Project
- 89.3 Designing the Base Activity User Interface
- 89.4 Adding the Dynamic Feature Module
- 89.5 Reviewing the Dynamic Feature Module
- 89.6 Adding the Dynamic Feature Activity
- 89.7 Implementing the `launchIntent()` Method
- 89.8 Uploading the App Bundle for Testing
- 89.9 Implementing the `installFeature()` Method
- 89.10 Adding the Update Listener
- 89.11 Using Deferred Installation
- 89.12 Removing a Dynamic Module
- 89.13 Summary

## **90. Working with Material Design 3 Theming**

- 90.1 Material Design 2 vs Material Design 3
- 90.2 Understanding Material Design Theming
- 90.3 Material Design 2 Theming
- 90.4 Material Design 3 Theming
- 90.5 Building a Custom Theme
- 90.6 Summary

## **91. A Material Design 3 Theming and Dynamic Color Tutorial**

- 91.1 Creating the ThemeDemo Project
- 91.2 Preparing the Project
- 91.3 Designing the User Interface
- 91.4 Building a New Theme
- 91.5 Adding the Custom Colors to the Project
- 91.6 Merging the Custom Themes
- 91.7 Enabling Dynamic Color Support
- 91.8 Summary

**92. Migrating from Material Design 2 to Material Design 3**

- 92.1 Creating the ThemeMigration Project
- 92.2 Designing the User Interface
- 92.3 Migrating to Material Design 3
- 92.4 Building a New Theme
- 92.5 Adding the Theme to the Project
- 92.6 Summary

**93. An Overview of Gradle in Android Studio**

- 93.1 An Overview of Gradle
- 93.2 Gradle and Android Studio
  - 93.2.1 Sensible Defaults
  - 93.2.2 Dependencies
  - 93.2.3 Build Variants
  - 93.2.4 Manifest Entries
  - 93.2.5 APK Signing
  - 93.2.6 ProGuard Support
- 93.3 The Property and Settings Gradle Build File
- 93.4 The Top-level Gradle Build File
- 93.5 Module Level Gradle Build Files
- 93.6 Configuring Signing Settings in the Build File
- 93.7 Running Gradle Tasks from the Command-line
- 93.8 Summary

**Index**



## 1. Introduction

Fully updated for Android Studio Dolphin, this book aims to teach you how to develop Android-based applications using the Kotlin programming language.

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

An overview of Android Studio is included covering areas such as tool windows, the code editor, and the Layout Editor tool. An introduction to the architecture of Android is followed by an in-depth look at the design of Android applications and user interfaces using the Android Studio environment.

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

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

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

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

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

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

### 1.1 Downloading the Code Samples

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

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

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

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

## 1.2 Feedback

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

## 1.3 Errata

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

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

If you find an error not listed in the errata, please let us know by emailing our technical support team at [feedback@ebookfrenzy.com](mailto:feedback@ebookfrenzy.com). They are there to help you and will work to resolve any problems you may encounter.



## 2. Setting up an Android Studio Development Environment

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

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

### 2.1 System requirements

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

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

Although Android Studio will run on computers with 8GB of RAM, performance will be greatly improved on systems containing more memory. This is particularly an issue if you plan to test your apps using the Android Virtual Device emulator (AVD).

### 2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio Dolphin 2021.3.1 using the Android API 33 SDK (Tiramisu) which, at the time of writing, are the latest versions.

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

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

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

If this page provides instructions for downloading a newer version of Android Studio it is important to note that there may be some minor differences between this book and the software. A web search for “Android Studio Dolphin” should provide the option to download the older version if these differences become a problem.

Alternatively, visit the following web page to find Android Studio Dolphin 2021.3.1 in the archives:

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

## 2.3 Installing Android Studio

Once downloaded, the exact steps to install Android Studio differ depending on the operating system on which the installation is being performed.

### 2.3.1 Installation on Windows

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

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

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

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

### 2.3.2 Installation on macOS

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

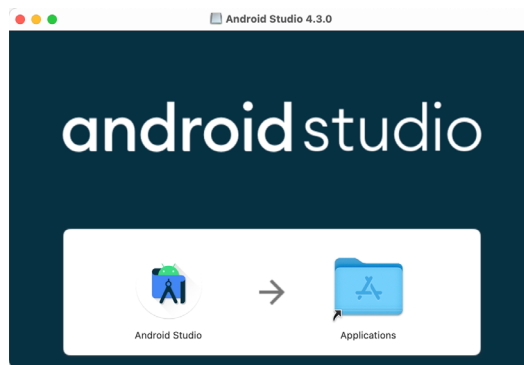


Figure 2-1

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

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

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

### 2.3.3 Installation on Linux

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

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

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

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

```
./studio.sh
```

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

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

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

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

## 2.4 The Android Studio setup wizard

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

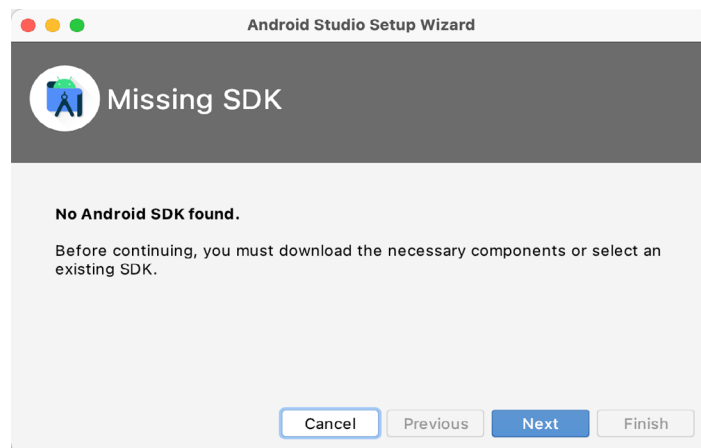


Figure 2-2

If this dialog appears, click the Next button to display the SDK Components Setup dialog (Figure 2-3). Within this dialog, make sure that the Android SDK option is selected along with the latest API package before clicking on the Next button:

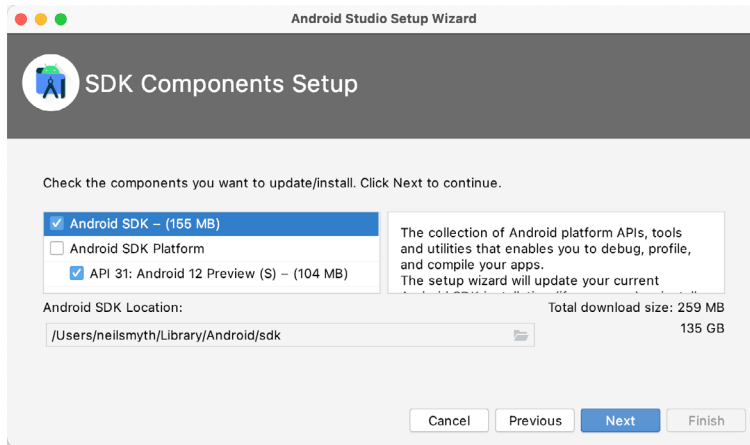


Figure 2-3

After clicking Next, Android Studio will download and install the Android SDK and tools.

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

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

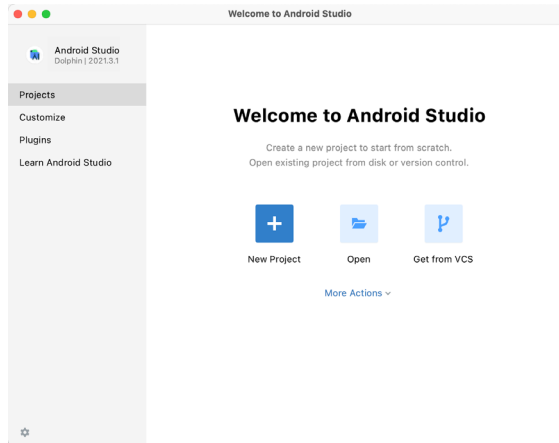


Figure 2-4

## 2.5 Installing additional Android SDK packages

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

This task can be performed by clicking on the *More Actions* link within the welcome dialog and selecting the *SDK Manager* option from the drop-down menu. Once invoked, the *Android SDK* screen of the Preferences dialog will appear as shown in Figure 2-5:

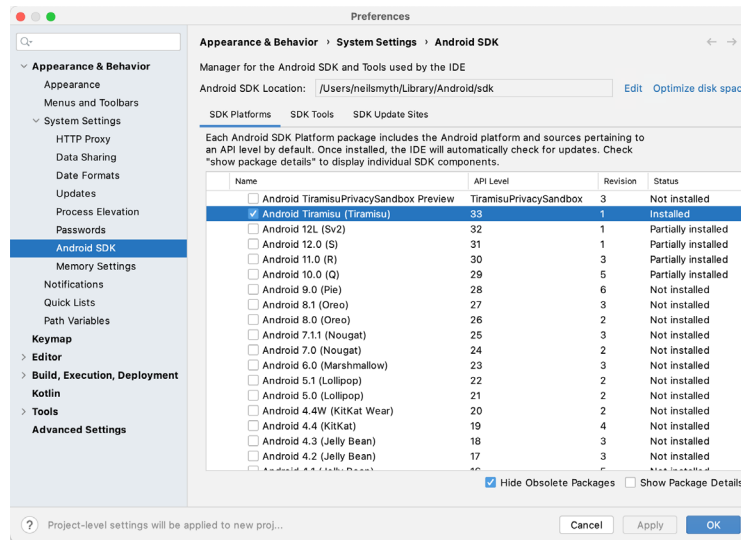


Figure 2-5

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

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

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

|                                     | Name   | API Level | Revision | Status              |
|-------------------------------------|--|-----------|----------|---------------------|
| <input type="checkbox"/>            | Android TV Intel x86 Atom System Image             | 25        | 6        | Not installed       |
| <input type="checkbox"/>            | Android Wear for China ARM EABI v7a System Image   | 25        | 3        | Not installed       |
| <input type="checkbox"/>            | Android Wear for China Intel x86 Atom System Image | 25        | 3        | Not installed       |
| <input type="checkbox"/>            | Android Wear ARM EABI v7a System Image             | 25        | 3        | Not installed       |
| <input type="checkbox"/>            | Android Wear Intel x86 Atom System Image           | 25        | 3        | Not installed       |
| <input type="checkbox"/>            | Google APIs ARM 64 v8a System Image                | 25        | 8        | Not installed       |
| <input type="checkbox"/>            | Google APIs ARM EABI v7a System Image              | 25        | 8        | Not installed       |
| <input type="checkbox"/>            | Google APIs Intel x86 Atom System Image            | 25        | 8        | Not installed       |
| <input checked="" type="checkbox"/> | Google APIs Intel x86 Atom_64 System Image         | 25        | 6        | Update Available: 8 |
| ▼ <input type="checkbox"/>          | <b>Android 7.0 (Nougat)</b>                        |           |          |                     |
| <input type="checkbox"/>            | Google APIs  | 24        | 1        | Not installed       |

Figure 2-6

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

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

## Setting up an Android Studio Development Environment

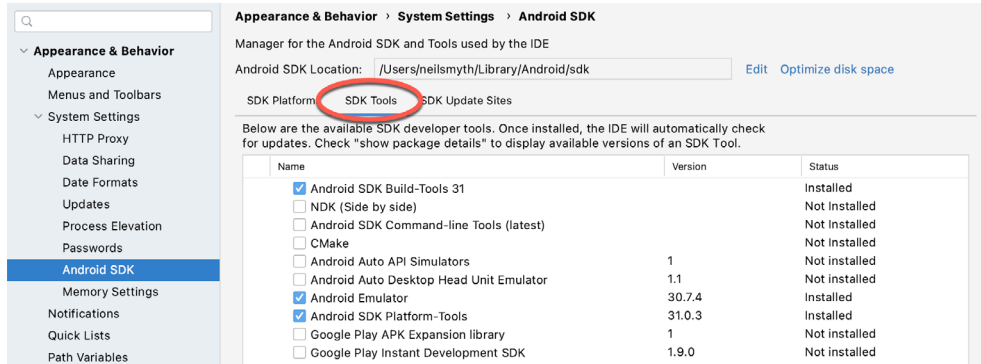


Figure 2-7

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

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

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

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

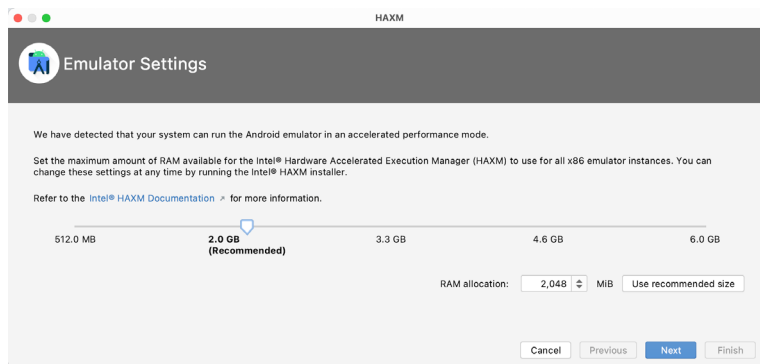


Figure 2-8

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

## 2.6 Making the Android SDK tools command-line accessible

Most of the time, the underlying tools of the Android SDK will be accessed from within the Android Studio environment. That being said, however, there will also be instances where it will be useful to be able to invoke those tools from a command prompt or terminal window. For the operating system on which you are developing to be able to find these tools, it will be necessary to add them to the system's *PATH* environment variable.

Regardless of operating system, the *PATH* variable needs to be configured to include the following paths (where *<path\_to\_android\_sdk\_installation>* represents the file system location into which the Android SDK was installed):

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

The location of the SDK on your system can be identified by launching the SDK Manager and referring to the *Android SDK Location*: field located at the top of the settings panel as highlighted in Figure 2-9:

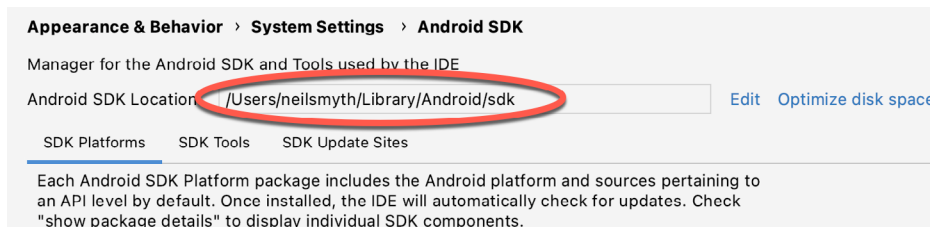


Figure 2-9

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

### 2.6.1 Windows 8.1

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

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

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

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

```
echo %Path%
```

The returned path variable value should include the paths to the Android SDK platform tools folders. Verify that

## Setting up an Android Studio Development Environment

the *platform-tools* value is correct by attempting to run the *adb* tool as follows:

```
adb
```

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

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

```
avdmanager
```

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

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

### 2.6.2 Windows 10

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

### 2.6.3 Windows 11

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

### 2.6.4 Linux

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

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

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

### 2.6.5 macOS

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

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

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

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



## 2.7 Android Studio memory management

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

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

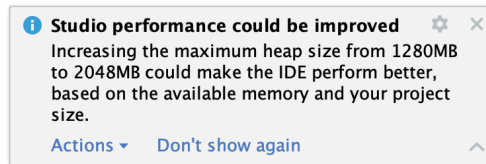


Figure 2-10

To view and modify the current memory configuration, select the *File -> Settings...* (*Android Studio -> Preferences...* on macOS) menu option and, in the resulting dialog, select the *Memory Settings* option listed under *System Settings* in the left-hand navigation panel as illustrated in Figure 2-11 below.

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

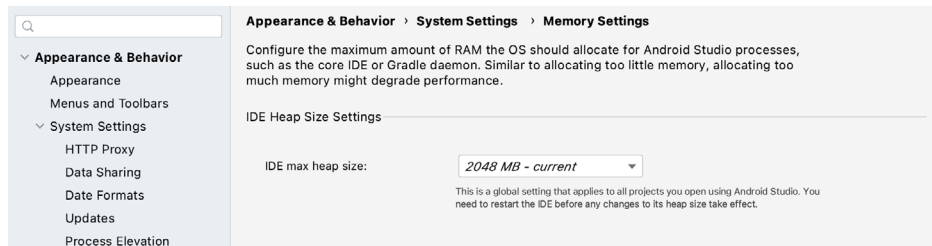


Figure 2-11

## 2.8 Updating Android Studio and the SDK

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

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

## 2.9 Summary

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



## 3. Creating an Example Android App in Android Studio

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

### 3.1 About the Project

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

### 3.2 Creating a New Android Project

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

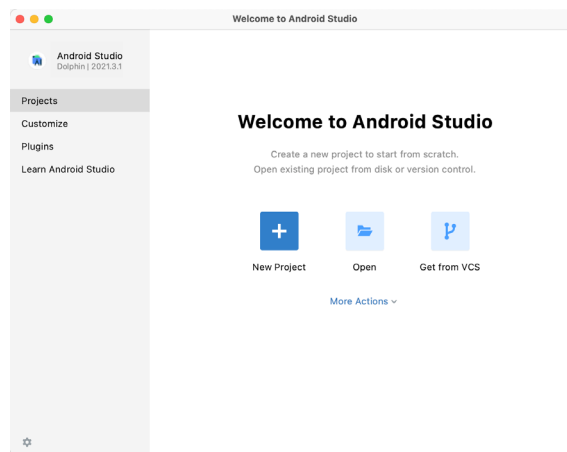


Figure 3-1

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

### 3.3 Creating an Activity

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

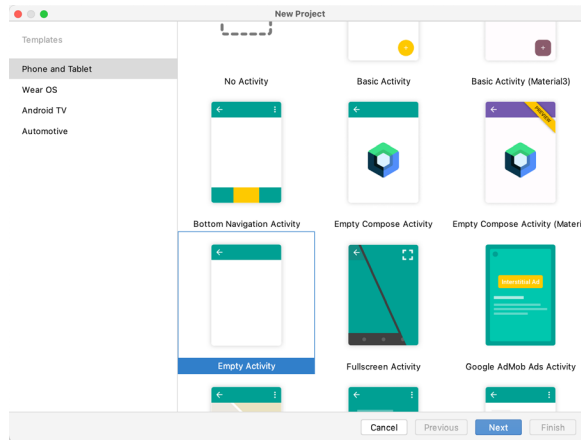


Figure 3-2

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

### 3.4 Defining the Project and SDK Settings

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

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

```
com.mycompany.androidsample
```

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

```
com.example.androidsample
```

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

Set the minimum SDK setting to API 26: Android 8.0 (Oreo). This is the minimum SDK that will be used in most of the projects created in this book unless a necessary feature is only available in a more recent version. The objective here is to build an app using the latest Android SDK, while also retaining compatibility with devices running older versions of Android (in this case as far back as Android 8.0). The text beneath the Minimum SDK

setting will outline the percentage of Android devices currently in use on which the app will run. Click on the *Help me choose* button (highlighted in Figure 3-3) to see a full breakdown of the various Android versions still in use:

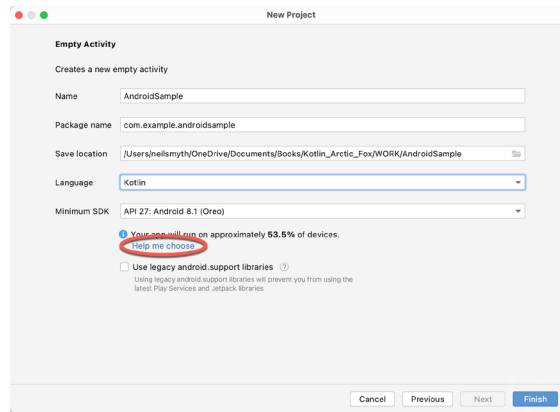


Figure 3-3

Finally, change the *Language* menu to *Kotlin* and click on *Finish* to initiate the project creation process.

### 3.5 Modifying the Example Application

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

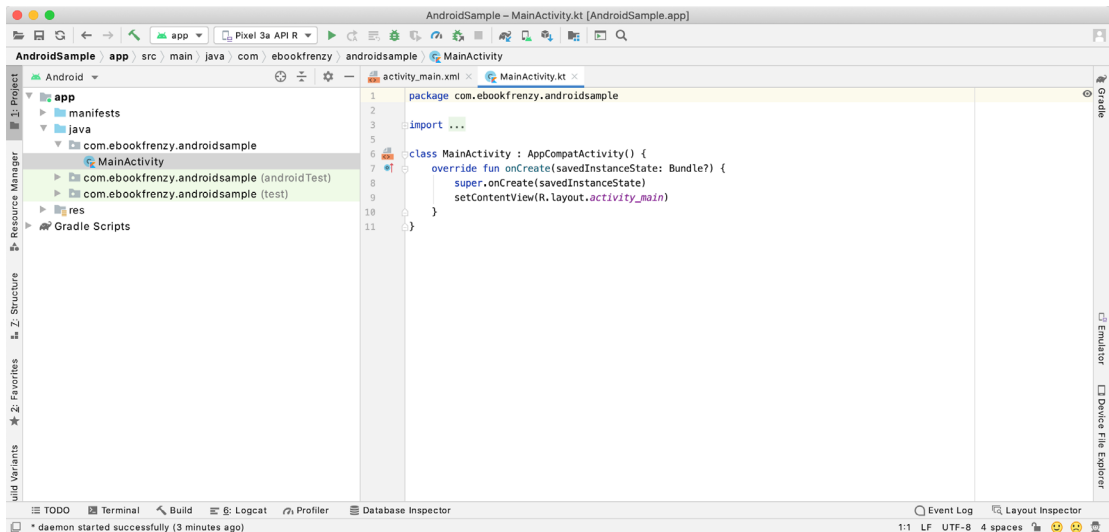


Figure 3-4

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 a number of modes in which information can be displayed. By default, this panel should be in *Android* mode. This setting is controlled by the menu at the top of the panel as highlighted in Figure 3-5. If the panel is not currently in Android mode, use the menu to switch mode:

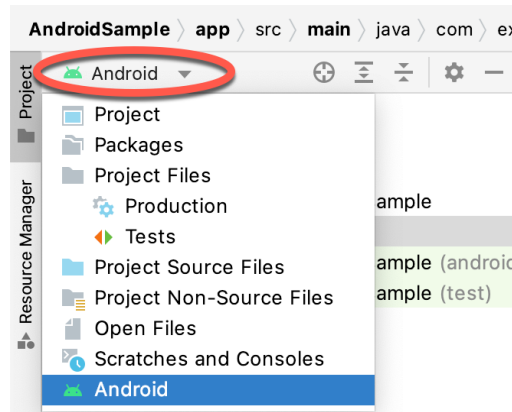


Figure 3-5

### 3.6 Modifying the User Interface

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

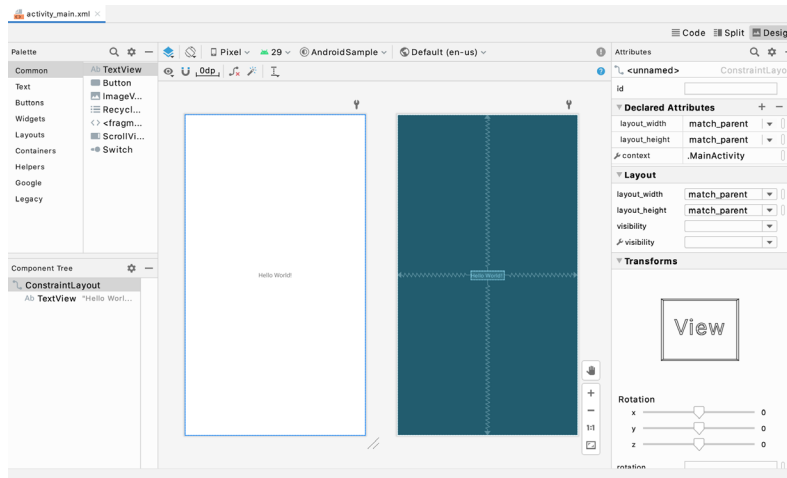




Figure 3-6

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

To change the orientation of the device representation between landscape and portrait simply use the drop down menu immediately to the left of the device selection menu showing the  icon. Use the night button () to turn Night mode on and off.

As we can see in the device screen, the content layout already includes a label that displays a “Hello World!” message. Running down the left-hand side of the panel is a palette containing different categories of user interface components that may be used to construct a user interface, such as buttons, labels and text fields. It should be noted, however, that not all user interface components are obviously visible to the user. One such category

consists of *layouts*. Android supports a variety of layouts that provide different levels of control over how visual user interface components are positioned and managed on the screen. Though it is difficult to tell from looking at the visual representation of the user interface, the current design has been created using a `ConstraintLayout`. This can be confirmed by reviewing the information in the *Component Tree* panel which, by default, is located in the lower left-hand corner of the Layout Editor panel and is shown in Figure 3-7:

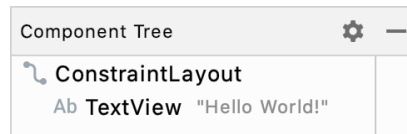


Figure 3-7

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

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



Figure 3-8

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

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

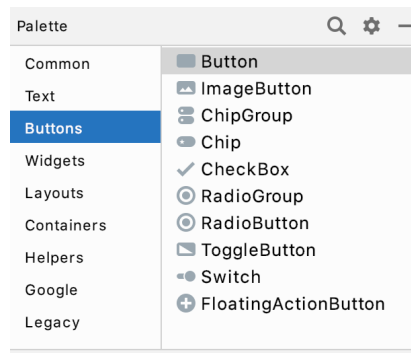


Figure 3-9

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

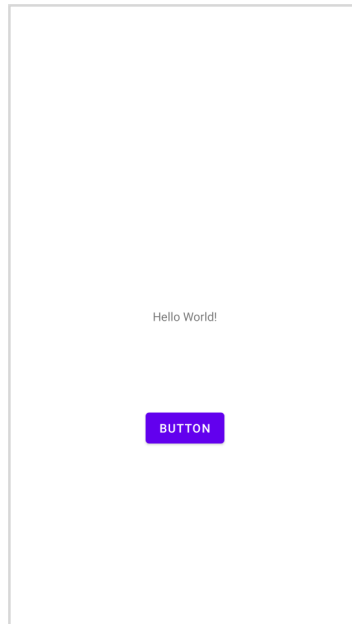


Figure 3-10

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

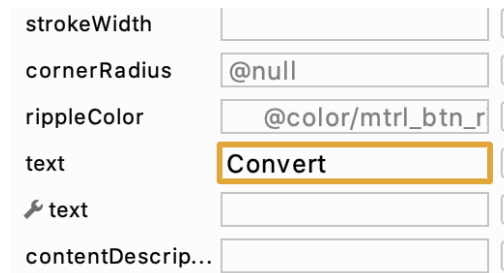


Figure 3-11

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

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

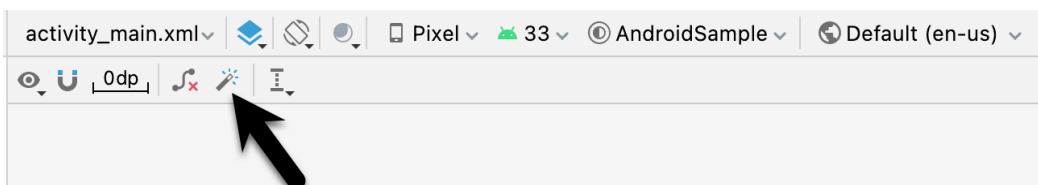


Figure 3-12



At this point it is important to explain the warning button located in the top right-hand corner of the Layout Editor tool as indicated in Figure 3-13. Obviously, this is indicating potential problems with the layout. For details on any problems, click on the button:

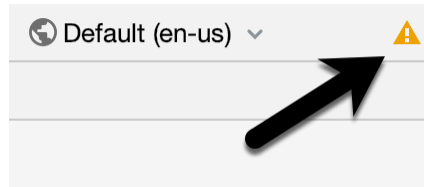


Figure 3-13

When clicked, a panel (Figure 3-14) will appear describing the nature of the problems and offering some possible corrective measures:

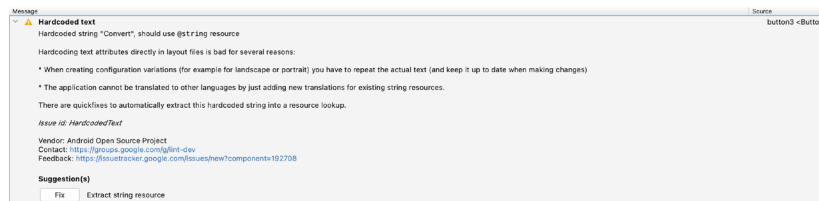


Figure 3-14

Currently, the only warning listed reads as follows:

Hardcoded string "Convert", should use @string resource

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

Click on the *Fix* button in the Issue Explanation panel to display the *Extract Resource* panel (Figure 3-15). Within this panel, change the resource name field to *convert\_string* and leave the resource value set to *Convert* before clicking on the OK button.

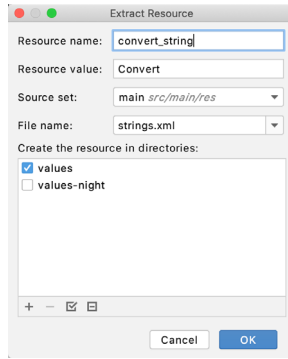


Figure 3-15

The next widget to be added is an `EditText` widget into which the user will enter the dollar amount to be converted. From the Palette panel, select the Text category and click and drag a Number (Decimal) component onto the layout so that it is centered horizontally and positioned above the existing `TextView` widget. With the widget selected, use the Attributes tools window to set the *hint* property to “dollars”. Click on the warning icon and extract the string to a resource named `dollars_hint`.

The code written later in this chapter will need to access the dollar value entered by the user into the `EditText` field. It will do this by referencing the id assigned to the widget in the user interface layout. The default id assigned to the widget by Android Studio can be viewed and changed from within the Attributes tool window when the widget is selected in the layout as shown in Figure 3-16:

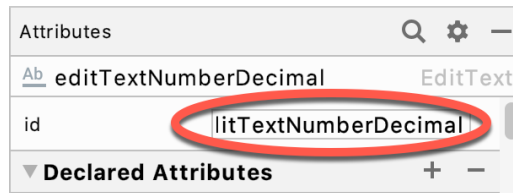


Figure 3-16

Change the id to `dollarText` and, in the Rename dialog, click on the *Refactor* button. This ensures that any references elsewhere within the project to the old id are automatically updated to use the new id:

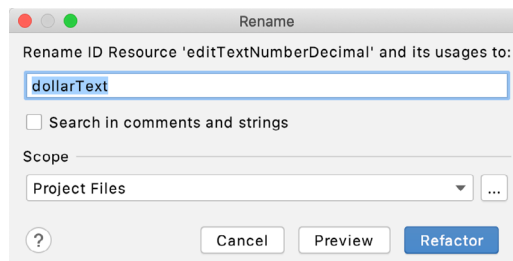


Figure 3-17

Add any missing layout constraints by clicking on the *Infer constraints* button. At this point the layout should resemble that shown in Figure 3-18:

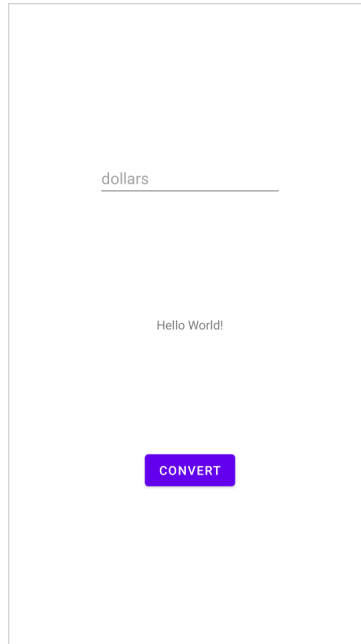


Figure 3-18

### 3.7 Reviewing the Layout and Resource Files

Before moving on to the next step, we are going to look at some of the internal aspects of user interface design and resource handling. In the previous section, we made some changes to the user interface by modifying the *activity\_main.xml* file using the Layout Editor tool. In fact, all that the Layout Editor was doing was providing a user-friendly way to edit the underlying XML content of the file. In practice, there is no reason why you cannot modify the XML directly to make user interface changes and, in some instances, this may actually be quicker than using the Layout Editor tool. In the top right-hand corner of the Layout Editor panel are three buttons as highlighted in Figure 3-19 below:

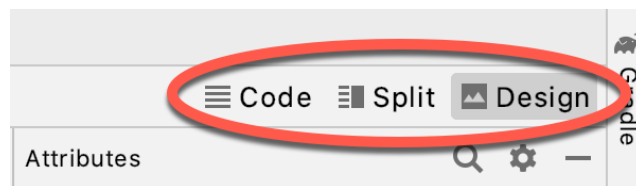


Figure 3-19

By default, the editor will be in *Design* mode whereby just the visual representation of the layout is displayed. The left-most button will switch to *Code* mode to display the XML for the layout, while the middle button enters *Split* mode where both the layout and XML are displayed, as shown in Figure 3-20:

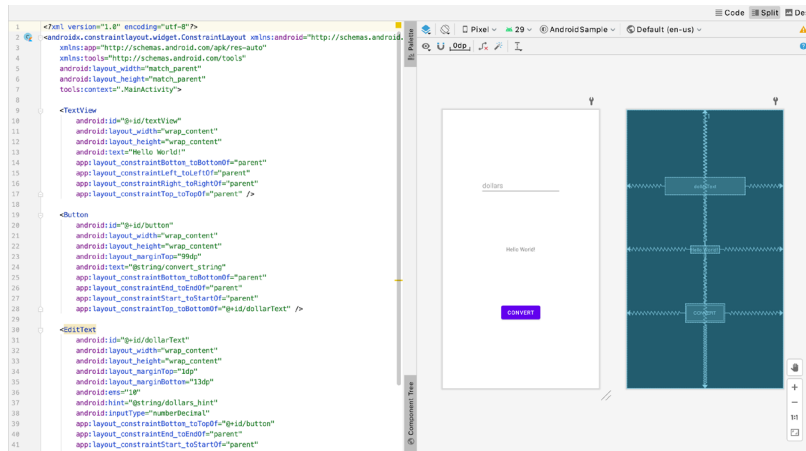


Figure 3-20

As can be seen from the structure of the XML file, the user interface consists of the `ConstraintLayout` component, which in turn, is the parent of the `TextView`, `Button` and `EditText` objects. We can also see, for example, that the `text` property of the `Button` is set to our `convert_string` resource. Although varying in complexity and content, all user interface layouts are structured in this hierarchical, XML based way.

As changes are made to the XML layout, these will be reflected in the layout canvas. The layout may also be modified visually from within the layout canvas panel with the changes appearing in the XML listing. To see this in action, switch to Split mode and modify the XML layout to change the background color of the `ConstraintLayout` to a shade of red as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.
android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity"
    android:background="#ff2438" >
    .
    .
</androidx.constraintlayout.widget.ConstraintLayout>
```

Note that the color of the layout changes in real-time to match the new setting in the XML file. Note also that a small red square appears in the left-hand margin (also referred to as the *gutter*) of the XML editor next to the line containing the color setting. This is a visual cue to the fact that the color red has been set on a property. Clicking on the red square will display a color chooser allowing a different color to be selected:

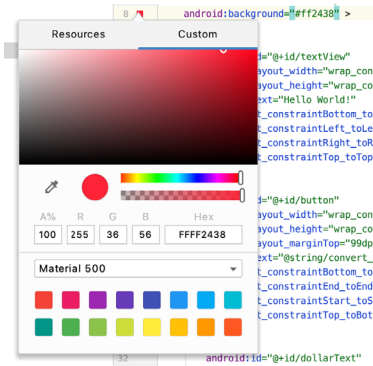


Figure 3-21

Before proceeding, delete the background property from the layout file so that the background returns to the default setting.

Finally, use the Project panel to locate the *app -> res -> values -> strings.xml* file and double-click on it to load it into the editor. Currently the XML should read as follows:

```
<resources>
    <string name="app_name">AndroidSample</string>
    <string name="convert_string">Convert</string>
    <string name="dollars_hint">dollars</string>
</resources>
```

As a demonstration of resources in action, change the string value currently assigned to the *convert\_string* resource to “Convert to Euros” and then return to the Layout Editor tool by selecting the tab for the layout file in the editor panel. Note that the layout has picked up the new resource value for the string.

There is also a quick way to access the value of a resource referenced in an XML file. With the Layout Editor tool in Split or Code mode, click on the “@string/convert\_string” property setting so that it highlights and then press Ctrl-B on the keyboard (Cmd-B on macOS). Android Studio will subsequently open the *strings.xml* file and take you to the line in that file where this resource is declared. Use this opportunity to revert the string resource back to the original “Convert” text and to add the following additional entry for a string resource that will be referenced later in the app code:

```
<resources>
.
.
    <string name="convert_string">Convert</string>
    <string name="dollars_hint">dollars</string>
    <string name="no_value_string">No Value</string>
</resources>
```

Resource strings may also be edited using the Android Studio Translations Editor. To open this editor, right-click on the *app -> res -> values -> strings.xml* file and select the *Open editor* menu option. This will display the Translation Editor in the main panel of the Android Studio window:

## Creating an Example Android App in Android Studio

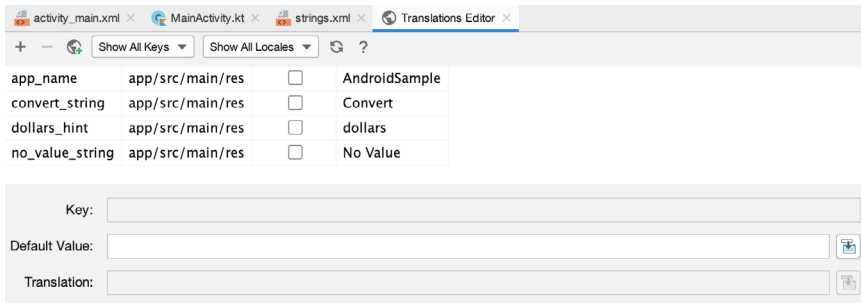


Figure 3-22

This editor allows the strings assigned to resource keys to be edited and for translations for multiple languages to be managed.

### 3.8 Adding the Kotlin Extensions Plugin

The next section will add some code to the project so that a currency conversion occurs when the button is tapped and the result displayed to the user. Before adding this code, however, we first need to add a plugin to the project build configuration which will make it easier for us to reference the user interface widgets from within the Kotlin code. To do this, begin by opening the module level *build.gradle* file located in the project tool window (*app* -> *Gradle Scripts* -> *build.gradle (Module: AndroidSample.app)*) as shown in Figure 3-23:

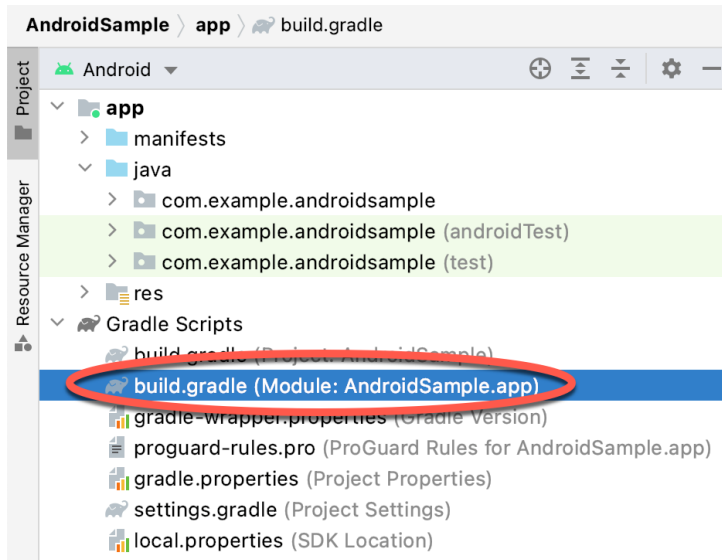


Figure 3-23

Once opened, modify the plugins section so that it reads as follows:

```
plugins {  
    id 'com.android.application'  
    id 'org.jetbrains.kotlin.android'  
    id 'kotlin-android-extensions'  
}
```

Finally, click on the *Sync Now* link highlighted in below to commit the change and update the project:

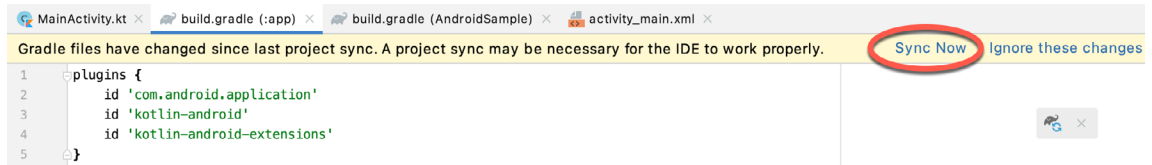


Figure 3-24

The topic of accessing widgets from within code using this technique, together with some useful alternatives, will be covered in the chapter entitled “*An Overview of Android View Binding*”.

### 3.9 Adding Interaction

The final step in this example project is to make the app interactive so that when the user enters a dollar value into the EditText field and clicks the convert button the converted euro value appears on the TextView. This involves the implementation of some event handling on the Button widget. Specifically, the Button needs to be configured so that a method in the app code is called when an *onClick* event is triggered. Event handling can be implemented in a number of different ways and is covered in detail in a later chapter entitled “*An Overview and Example of Android Event Handling*”. Return the layout editor to Design mode, select the Button widget in the layout editor, refer to the Attributes tool window and specify a method named *convertCurrency* as shown below:

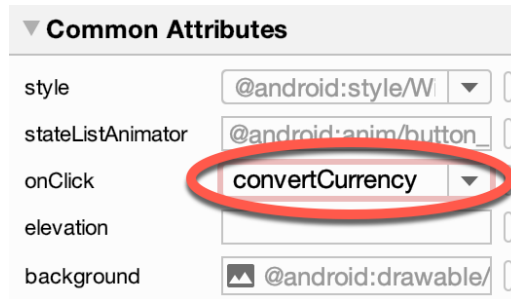


Figure 3-25

Note that the text field for the *onClick* property is now highlighted with a red border to warn us that the button has been configured to call a method which does not yet exist. To address this, double-click on the *MainActivity.kt* file in the Project tool window (*app* -> *java* -> *<package name>* -> *MainActivity*) to load it into the code editor and add the code for the *convertCurrency* method to the class file so that it reads as follows, noting that it is also necessary to import some additional Android packages:

```
package com.example.androidsample

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.View

import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
```

```

    }

    fun convertCurrency(view: View) {

        if (dollarText.text.isNotEmpty()) {

            val dollarValue = dollarText.text.toString().toFloat()

            val euroValue = dollarValue * 0.85f

            textView.text = euroValue.toString()
        } else {
            textView.text = getString(R.string.no_value_string)
        }
    }
}

```

The method begins by checking the text property of the dollarText EditText view to make sure that it is not empty (in other words that the user has entered a dollar value). If a value has not been entered, a “No Value” string is displayed on the TextView using the string resource declared earlier in the chapter. If, on the other hand, a dollar amount has been entered, it is converted into a floating point value and the equivalent euro value calculated. This floating point value is then converted into a string and displayed on the TextView. If any of this is unclear, rest assured that these concepts will be covered in greater detail in later chapters.

### 3.10 Summary

While not excessively complex, a number of steps are involved in setting up an Android development environment. Having performed those steps, it is worth working through an example to make sure the environment is correctly installed and configured. In this chapter, we have created an example application and then used the Android Studio Layout Editor tool to modify the user interface layout. In doing so, we explored the importance of using resources wherever possible, particularly in the case of string values, and briefly touched on the topic of layouts. Next we looked at the underlying XML that is used to store the user interface designs of Android applications.

Finally, an onClick event was added to a Button connected to a method that was implemented to extract the user input from the EditText component, convert from dollars to euros and then display the result on the TextView.

With the app ready for testing, the steps necessary to set up an emulator for testing purposes will be covered in detail in the next chapter.



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

Before the next chapter explores testing on physical Android devices, this chapter will take some time to provide an overview of the Android Studio AVD emulator and highlight many of the configuration features that are available to customize the environment in both standalone and tool window modes.

### 5.1 The Emulator Environment

When launched in standalone mode, the emulator displays an initial splash screen during the loading process. Once loaded, the main emulator window appears containing a representation of the chosen device type (in the case of Figure 5-1 this is a Pixel 4 device):

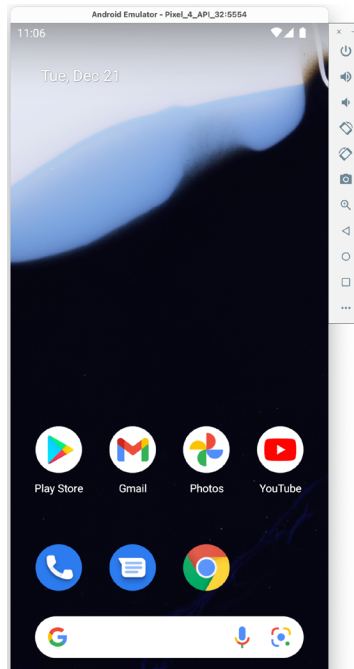


Figure 5-1

Positioned along the right-hand edge of the window is the toolbar providing quick access to the emulator controls and configuration options.

### 5.2 Emulator Toolbar Options

The emulator toolbar (Figure 5-2) provides access to a range of options relating to the appearance and behavior of the emulator environment.

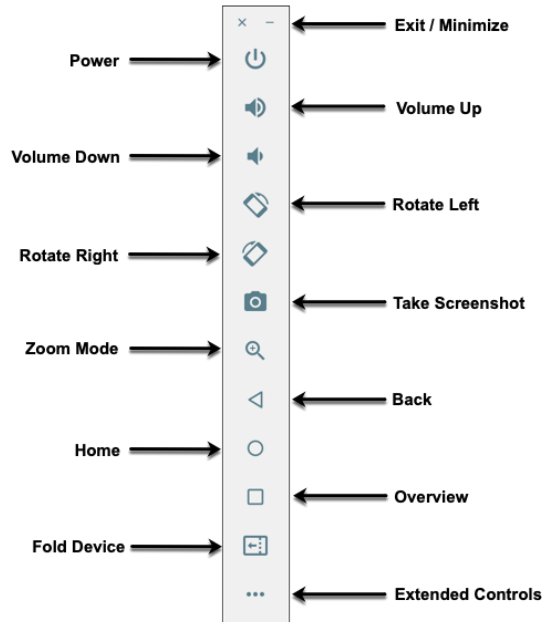


Figure 5-2

Each button in the toolbar has associated with it a keyboard accelerator which can be identified either by hovering the mouse pointer over the button and waiting for the tooltip to appear or via the help option of the extended controls panel.

Though many of the options contained within the toolbar are self-explanatory, each option will be covered for the sake of completeness:

- **Exit / Minimize** – The uppermost ‘x’ button in the toolbar exits the emulator session when selected while the ‘-’ option minimizes the entire window.
- **Power** – The Power button simulates the hardware power button on a physical Android device. Clicking and releasing this button will lock the device and turn off the screen. Clicking and holding this button will initiate the device “Power off” request sequence.
- **Volume Up / Down** – Two buttons that control the audio volume of playback within the simulator environment.
- **Rotate Left/Right** – Rotates the emulated device between portrait and landscape orientations.
- **Take Screenshot** – Takes a screenshot of the content currently displayed on the device screen. The captured image is stored at the location specified in the Settings screen of the extended controls panel as outlined later in this chapter.
- **Zoom Mode** – This button toggles in and out of zoom mode, details of which will be covered later in this chapter.
- **Back** – Performs the standard Android “Back” navigation to return to a previous screen.
- **Home** – Displays the device home screen.
- **Overview** – Simulates selection of the standard Android “Overview” navigation which displays the currently running apps on the device.

- **Fold Device** – Simulates the folding and unfolding of a foldable device. This option is only available if the emulator is running a foldable device system image.
- **Extended Controls** – Displays the extended controls panel, allowing for the configuration of options such as simulated location and telephony activity, battery strength, cellular network type, and fingerprint identification.

## 5.3 Working in Zoom Mode

The zoom button located in the emulator toolbar switches in and out of zoom mode. When zoom mode is active the toolbar button is depressed and the mouse pointer appears as a magnifying glass when hovering over the device screen. Clicking the left mouse button will cause the display to zoom in relative to the selected point on the screen, with repeated clicking increasing the zoom level. Conversely, clicking the right mouse button decreases the zoom level. Toggling the zoom button off reverts the display to the default size.

Clicking and dragging while in zoom mode will define a rectangular area into which the view will zoom when the mouse button is released.

While in zoom mode the visible area of the screen may be panned using the horizontal and vertical scrollbars located within the emulator window.

## 5.4 Resizing the Emulator Window

The size of the emulator window (and the corresponding representation of the device) can be changed at any time by clicking and dragging on any of the corners or sides of the window.

## 5.5 Extended Control Options

The extended controls toolbar button displays the panel illustrated in Figure 5-3. By default, the location settings will be displayed. Selecting a different category from the left-hand panel will display the corresponding group of controls:

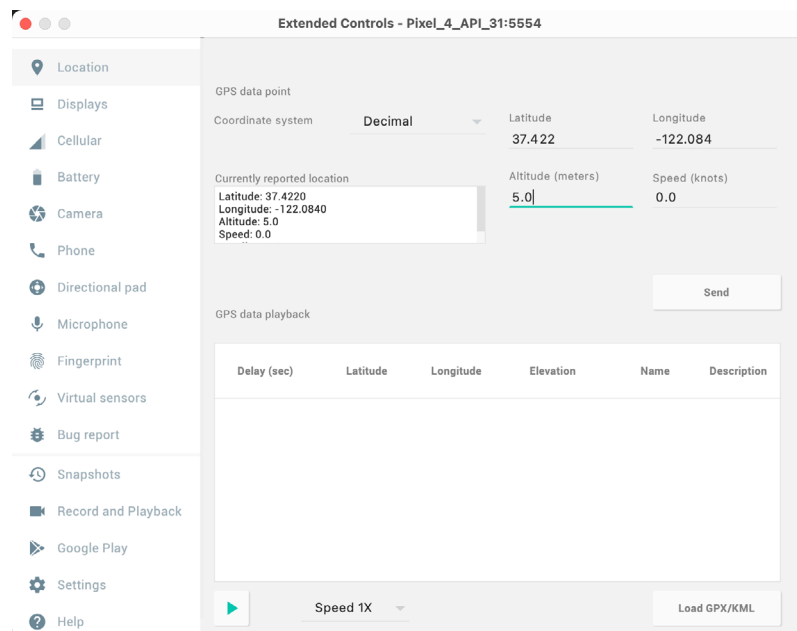


Figure 5-3

### 5.5.1 Location

The location controls allow simulated location information to be sent to the emulator in the form of decimal or sexagesimal coordinates. Location information can take the form of a single location, or a sequence of points representing movement of the device, the latter being provided via a file in either GPS Exchange (GPX) or Keyhole Markup Language (KML) format. Alternatively, the integrated Google Maps panel may be used to visually select single points or travel routes.

### 5.5.2 Displays

In addition to the main display shown within the emulator screen, the Displays option allows additional displays to be added running within the same Android instance. This can be useful for testing apps for dual-screen devices such as the Microsoft Surface Duo. These additional screens can be configured to be any required size and appear within the same emulator window as the main screen.

### 5.5.3 Cellular

The type of cellular connection being simulated can be changed within the cellular settings screen. Options are available to simulate different network types (CSM, EDGE, HSDPA, etc) in addition to a range of voice and data scenarios such as roaming and denied access.

### 5.5.4 Battery

A variety of battery state and charging conditions can be simulated on this panel of the extended controls screen, including battery charge level, battery health, and whether the AC charger is currently connected.

### 5.5.5 Camera

The emulator simulates a 3D scene when the camera is active. This takes the form of the interior of a virtual building through which you can navigate by holding down the Option key (Alt on Windows) while using the mouse pointer and keyboard keys when recording video or before taking a photo within the emulator. This extended configuration option allows different images to be uploaded for display within the virtual environment.

### 5.5.6 Phone

The phone extended controls provide two very simple but useful simulations within the emulator. The first option allows for the simulation of an incoming call from a designated phone number. This can be of particular use when testing how an app handles high-level interrupts of this nature.

The second option allows the receipt of text messages to be simulated within the emulator session. As in the real world, these messages appear within the Message app and trigger the standard notifications within the emulator.

### 5.5.7 Directional Pad

A directional pad (D-Pad) is an additional set of controls either built into an Android device or connected externally (such as a game controller) that provides directional controls (left, right, up, down). The directional pad settings allow D-Pad interaction to be simulated within the emulator.

### 5.5.8 Microphone

The microphone settings allow the microphone to be enabled and virtual headset and microphone connections to be simulated. A button is also provided to launch the Voice Assistant on the emulator.

### 5.5.9 Fingerprint

Many Android devices are now supplied with built-in fingerprint detection hardware. The AVD emulator makes it possible to test fingerprint authentication without the need to test apps on a physical device containing a fingerprint sensor. Details on how to configure fingerprint testing within the emulator will be covered in detail later in this chapter.

### 5.5.10 Virtual Sensors

The virtual sensors option allows the accelerometer and magnetometer to be simulated to emulate the effects of the physical motion of a device such as rotation, movement, and tilting through yaw, pitch and roll settings.

### 5.5.11 Snapshots

Snapshots contain the state of the currently running AVD session to be saved and rapidly restored making it easy to return the emulator to an exact state. Snapshots are covered in later in this chapter.

### 5.5.12 Record and Playback

Allows the emulator screen and audio to be recorded and saved in either WebM or animated GIF format.

### 5.5.13 Google Play

If the emulator is running a version of Android with Google Play Services installed, this option displays the current Google Play version and provides the option to update the emulator to the latest version.

### 5.5.14 Settings

The settings panel provides a small group of configuration options. Use this panel to choose a darker theme for the toolbar and extended controls panel, specify a file system location into which screenshots are to be saved, configure OpenGL support levels, and configure the emulator window to appear on top of other windows on the desktop.

### 5.5.15 Help

The Help screen contains three sub-panels containing a list of keyboard shortcuts, links to access the emulator online documentation, file bugs and send feedback, and emulator version information.

## 5.6 Working with Snapshots

When an emulator starts for the very first time it performs a *cold boot* much like a physical Android device when it is powered on. This cold boot process can take some time to complete as the operating system loads and all the background processes are started. To avoid the necessity of going through this process every time the emulator is started, the system is configured to automatically save a snapshot (referred to as a *quick-boot snapshot*) of the emulator's current state each time it exits. The next time the emulator is launched, the quick-boot snapshot is loaded into memory and execution resumes from where it left off previously, allowing the emulator to restart in a fraction of the time needed for a cold boot to complete.

The Snapshots screen of the extended controls panel can be used to store additional snapshots at any point during the execution of the emulator. This saves the exact state of the entire emulator allowing the emulator to be restored to the exact point in time that the snapshot was taken. From within the screen, snapshots can be taken using the *Take Snapshot* button (marked A in Figure 5-4). To restore an existing snapshot, select it from the list (B) and click the run button (C) located at the bottom of the screen. Options are also provided to edit (D) the snapshot name and description and to delete (E) the currently selected snapshot:

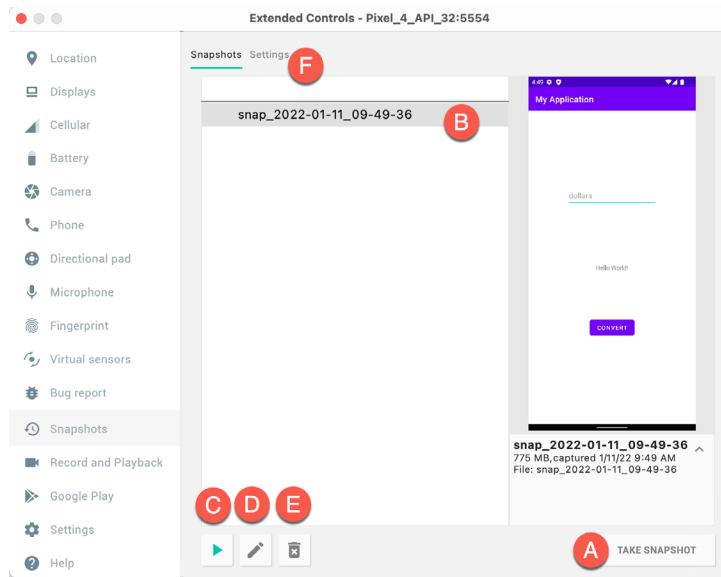


Figure 5-4

The Settings option (F) provides the option to configure the automatic saving of quick-boot snapshots (by default the emulator will ask whether to save the quick boot snapshot each time the emulator exits) and to reload the most recent snapshot. To force an emulator session to perform a cold boot instead of using a previous quick-boot snapshot, open the AVD Manager (*Tools -> AVD Manager*), click on the down arrow in the Actions column for the emulator and select the Cold Boot Now menu option.

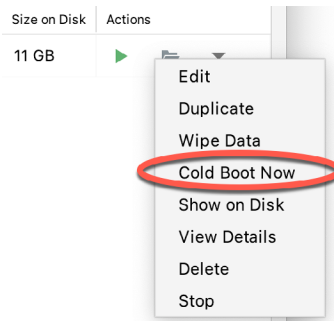


Figure 5-5

## 5.7 Configuring Fingerprint Emulation

The emulator allows up to 10 simulated fingerprints to be configured and used to test fingerprint authentication within Android apps. To configure simulated fingerprints begin by launching the emulator, opening the Settings app, and selecting the *Security & Location* option.

Within the Security settings screen, select the *Use fingerprint* option. On the resulting information screen click on the *Next* button to proceed to the Fingerprint setup screen. Before fingerprint security can be enabled a backup screen unlocking method (such as a PIN) must be configured. Click on the *Fingerprint + PIN* button and, when prompted, choose not to require the PIN on device startup. Enter and confirm a suitable PIN and complete the PIN entry process by accepting the default notifications option.

Proceed through the remaining screens until the Settings app requests a fingerprint on the sensor. At this point display the extended controls dialog, select the *Fingerprint* category in the left-hand panel and make sure that

*Finger 1* is selected in the main settings panel:

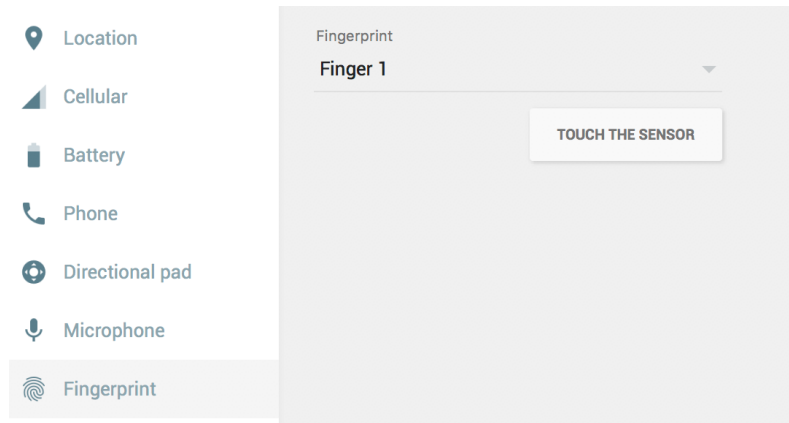


Figure 5-6

Click on the *Touch the Sensor* button to simulate Finger 1 touching the fingerprint sensor. The emulator will report the successful addition of the fingerprint:

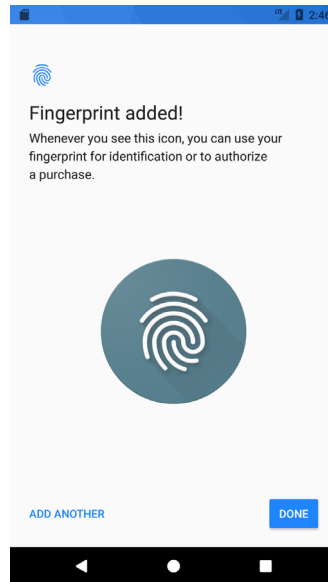


Figure 5-7

To add additional fingerprints click on the *Add Another* button and select another finger from the extended controls panel menu before clicking on the *Touch the Sensor* button once again.

## 5.8 The Emulator in Tool Window Mode

As outlined in the previous chapter (*“Creating an Android Virtual Device (AVD) in Android Studio”*), Android Studio can be configured to launch the emulator as an embedded tool window so that it does not appear in a separate window. When running in this mode, the same controls available in standalone mode are provided in the toolbar as shown in Figure 5-8:

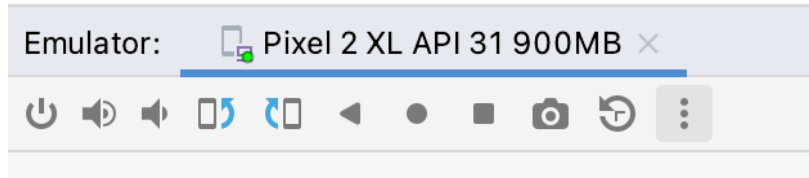


Figure 5-8

From left to right, these buttons perform the following tasks (details of which match those for standalone mode):

- Power
- Volume Up
- Volume Down
- Rotate Left
- Rotate Right
- Back
- Home
- Overview
- Screenshot
- Snapshots
- Extended Controls

## 5.9 Summary

Android Studio contains an Android Virtual Device emulator environment designed to make it easier to test applications without the need to run on a physical Android device. This chapter has provided a brief tour of the emulator and highlighted key features that are available to configure and customize the environment to simulate different testing conditions.



## 8. The Basics of the Android Studio Code Editor

Developing applications for Android involves a considerable amount of programming work which, by definition, involves typing, reviewing and modifying lines of code. It should come as no surprise that the majority of a developer's time spent using Android Studio will typically involve editing code within the editor window.

The modern code editor needs to go far beyond the original basics of typing, deleting, cutting and pasting. Today the usefulness of a code editor is generally gauged by factors such as the amount by which it reduces the typing required by the programmer, ease of navigation through large source code files and the editor's ability to detect and highlight programming errors in real-time as the code is being written. As will become evident in this chapter, these are just a few of the areas in which the Android Studio editor excels.

While not an exhaustive overview of the features of the Android Studio editor, this chapter aims to provide a guide to the key features of the tool. Experienced programmers will find that some of these features are common to most code editors available today, while a number are unique to this particular editing environment.

### 8.1 The Android Studio Editor

The Android Studio editor appears in the center of the main window when a Java, Kotlin, XML or other text based file is selected for editing. Figure 8-1, for example, shows a typical editor session with a Kotlin source code file loaded:

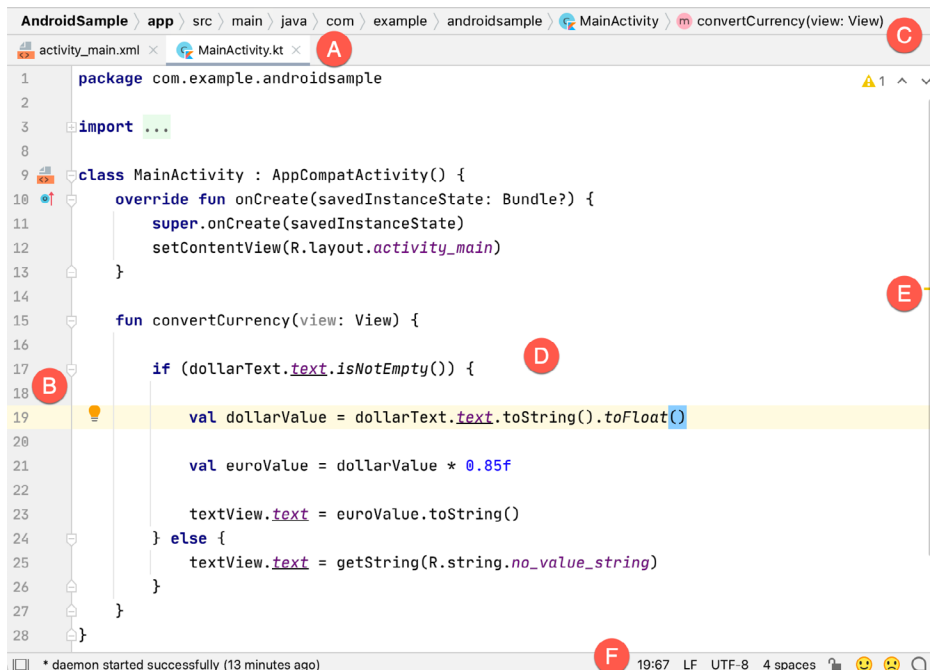


Figure 8-1

The elements that comprise the editor window can be summarized as follows:

**A – Document Tabs** – Android Studio is capable of holding multiple files open for editing at any one time. As each file is opened, it is assigned a document tab displaying the file name in the tab bar located along the top edge of the editor window. A small drop-down menu will appear in the far right-hand corner of the tab bar when there is insufficient room to display all of the tabs. Clicking on this menu will drop down a list of additional open files. A wavy red line underneath a file name in a tab indicates that the code in the file contains one or more errors that need to be addressed before the project can be compiled and run.

Switching between files is simply a matter of clicking on the corresponding tab or using the Alt-Left and Alt-Right keyboard shortcuts. Navigation between files may also be performed using the Switcher mechanism (accessible via the Ctrl-Tab keyboard shortcut).

To detach an editor panel from the Android Studio main window so that it appears in a separate window, click on the tab and drag it to an area on the desktop outside of the main window. To return the editor to the main window, click on the file tab in the separated editor window and drag and drop it onto the original editor tab bar in the main window.

**B – The Editor Gutter Area** - The gutter area is used by the editor to display informational icons and controls. Some typical items, among others, which appear in this gutter area are debugging breakpoint markers, controls to fold and unfold blocks of code, bookmarks, change markers and line numbers. Line numbers are switched on by default but may be disabled by right-clicking in the gutter and selecting the Show Line Numbers menu option.

**C – Code Structure Location** - This bar at the top of the editor displays the current position of the cursor as it relates to the overall structure of the code. In the following figure, for example, the bar indicates that the `convertCurrency` method is currently being edited, and that this method is contained within the `MainActivity` class.

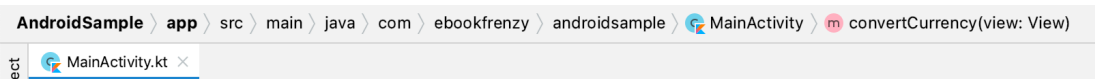


Figure 8-2

Double-clicking an element within the bar will move the cursor to the corresponding location within the code file. For example, double-clicking on the `convertCurrency` entry will move the cursor to the top of the `convertCurrency` method within the source code. Similarly clicking on the `MainActivity` entry will drop down a list of available code navigation points for selection:

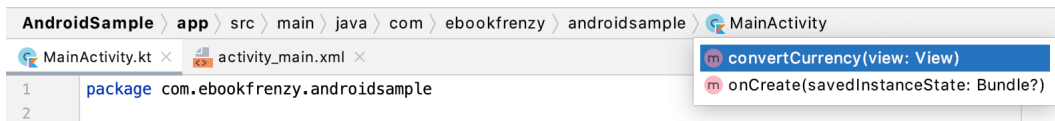


Figure 8-3

**D – The Editor Area** – This is the main area where the code is displayed, entered and edited by the user. Later sections of this chapter will cover the key features of the editing area in detail.

**E – The Validation and Marker Sidebar** – Android Studio incorporates a feature referred to as “on-the-fly code analysis”. What this essentially means is that as you are typing code, the editor is analyzing the code to check for warnings and syntax errors. The indicators at the top of the validation sidebar will update in real-time to indicate the number of errors and warnings found as code is added. Clicking on this indicator will display a popup containing a summary of the issues found with the code in the editor as illustrated in Figure 8-4:

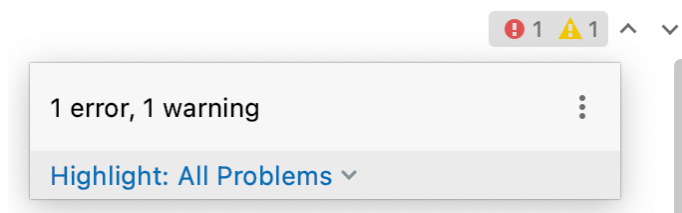


Figure 8-4

The up and down arrows may be used to move between the error locations within the code. A green check mark indicates that no warnings or errors have been detected.

The sidebar also displays markers at the locations where issues have been detected using the same color coding. Hovering the mouse pointer over a marker when the line of code is visible in the editor area will display a popup containing a description of the issue:

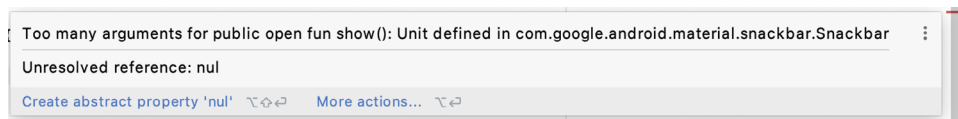


Figure 8-5

Hovering the mouse pointer over a marker for a line of code which is currently scrolled out of the viewing area of the editor will display a “lens” overlay containing the block of code where the problem is located (Figure 8-6) allowing it to be viewed without the necessity to scroll to that location in the editor:

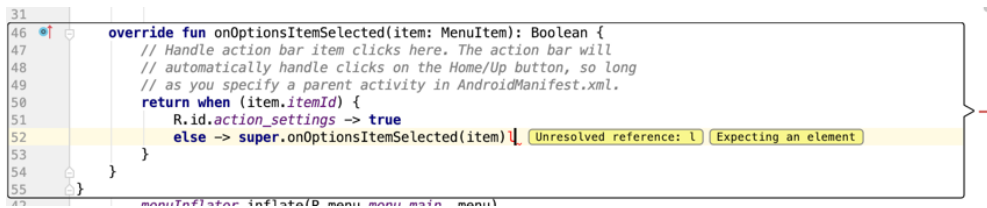


Figure 8-6

It is also worth noting that the lens overlay is not limited to warnings and errors in the sidebar. Hovering over any part of the sidebar will result in a lens appearing containing the code present at that location within the source file.

**F – The Status Bar** – Though the status bar is actually part of the main window, as opposed to the editor, it does contain some information about the currently active editing session. This information includes the current position of the cursor in terms of lines and characters and the encoding format of the file (UTF-8, ASCII etc.). Clicking on these values in the status bar allows the corresponding setting to be changed. Clicking on the line number, for example, displays the Go to Line dialog.

Having provided an overview of the elements that comprise the Android Studio editor, the remainder of this chapter will explore the key features of the editing environment in more detail.

## 8.2 Splitting the Editor Window

By default, the editor will display a single panel showing the content of the currently selected file. A particularly useful feature when working simultaneously with multiple source code files is the ability to split the editor into multiple panes. To split the editor, right-click on a file tab within the editor window and select either the Split Vertically or Split Horizontally menu option. Figure 8-7, for example, shows the splitter in action with the editor

split into three panels:

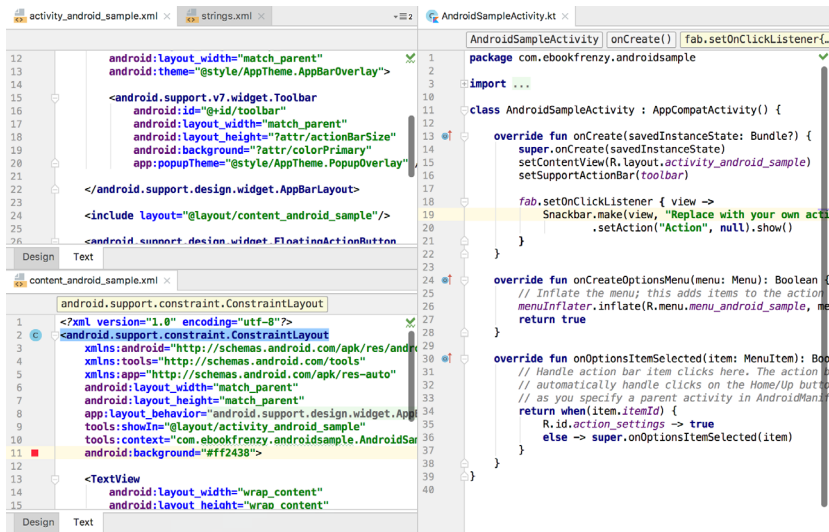


Figure 8-7

The orientation of a split panel may be changed at any time by right-clicking on the corresponding tab and selecting the Change Splitter Orientation menu option. Repeat these steps to unsplit a single panel, this time selecting the Unsplit option from the menu. All of the split panels may be removed by right-clicking on any tab and selecting the Unsplit All menu option.

Window splitting may be used to display different files, or to provide multiple windows onto the same file, allowing different areas of the same file to be viewed and edited concurrently.

## 8.3 Code Completion

The Android Studio editor has a considerable amount of built-in knowledge of Kotlin programming syntax and the classes and methods that make up the Android SDK, as well as knowledge of your own code base. As code is typed, the editor scans what is being typed and, where appropriate, makes suggestions with regard to what might be needed to complete a statement or reference. When a completion suggestion is detected by the editor, a panel will appear containing a list of suggestions. In Figure 8-8, for example, the editor is suggesting possibilities for the beginning of a String declaration:

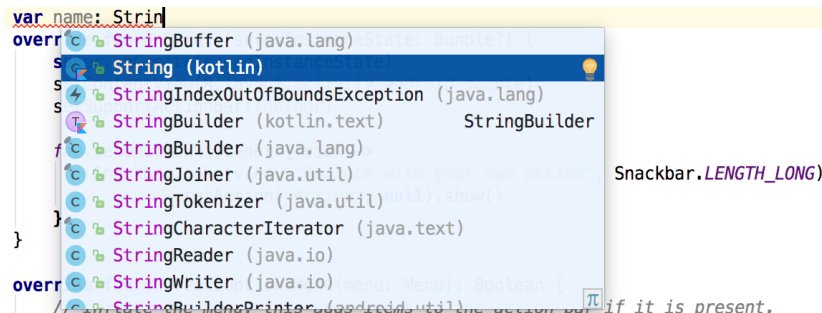


Figure 8-8

If none of the auto completion suggestions are correct, simply keep typing and the editor will continue to refine the suggestions where appropriate. To accept the top most suggestion, simply press the Enter or Tab key on the keyboard. To select a different suggestion, use the arrow keys to move up and down the list, once again using the

Enter or Tab key to select the highlighted item.

Completion suggestions can be manually invoked using the Ctrl-Space keyboard sequence. This can be useful when changing a word or declaration in the editor. When the cursor is positioned over a word in the editor, that word will automatically highlight. Pressing Ctrl-Space will display a list of alternate suggestions. To replace the current word with the currently highlighted item in the suggestion list, simply press the Tab key.

In addition to the real-time auto completion feature, the Android Studio editor also offers a system referred to as Smart Completion. Smart completion is invoked using the Shift-Ctrl-Space keyboard sequence and, when selected, will provide more detailed suggestions based on the current context of the code. Pressing the Shift-Ctrl-Space shortcut sequence a second time will provide more suggestions from a wider range of possibilities.

Code completion can be a matter of personal preference for many programmers. In recognition of this fact, Android Studio provides a high level of control over the auto completion settings. These can be viewed and modified by selecting the *File -> Settings...* menu option (or *Android Studio -> Preferences...* on macOS) and choosing *Editor -> General -> Code Completion* from the settings panel as shown in Figure 8-9:

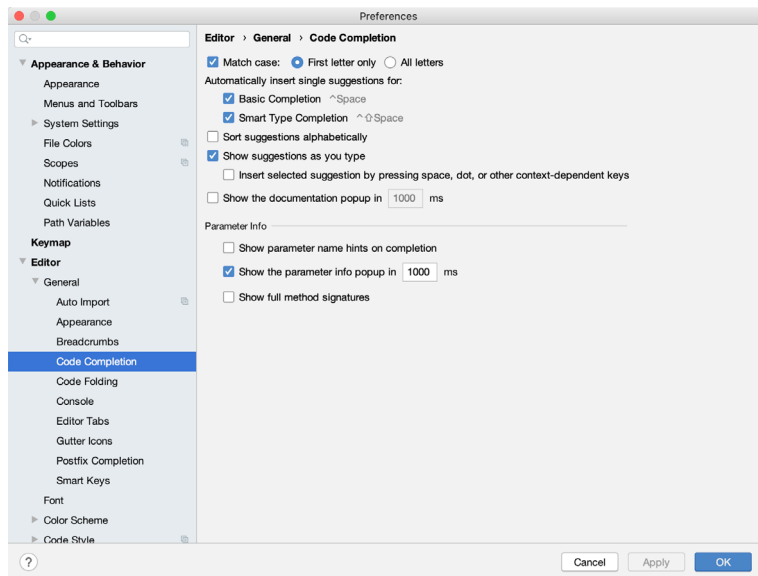


Figure 8-9

## 8.4 Statement Completion

Another form of auto completion provided by the Android Studio editor is statement completion. This can be used to automatically fill out the parentheses and braces for items such as methods and loop statements. Statement completion is invoked using the *Shift-Ctrl-Enter* (*Shift-Cmd-Enter* on macOS) keyboard sequence. Consider for example the following code:

```
fun myMethod()
```

Having typed this code into the editor, triggering statement completion will cause the editor to automatically add the braces to the method:

```
fun myMethod() {  
  
}
```

## 8.5 Parameter Information

It is also possible to ask the editor to provide information about the argument parameters accepted by a method. With the cursor positioned between the brackets of a method call, the Ctrl-P (Cmd-P on macOS) keyboard sequence will display the parameters known to be accepted by that method, with the most likely suggestion highlighted in bold:

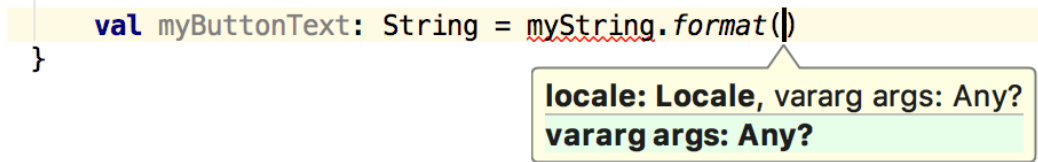


Figure 8-10

## 8.6 Parameter Name Hints

The code editor may be configured to display parameter name hints within method calls. Figure 8-11, for example, highlights the parameter name hints within the calls to the `make()` and `setAction()` methods of the `Snackbar` class:



Figure 8-11

The settings for this mode may be configured by selecting the *File -> Settings* menu (*Android Studio -> Preferences* on macOS) option followed by *Editor -> Inlay Hints -> Kotlin* in the left-hand panel. On the resulting screen, select the *Parameter Hints* item from the list and enable or disable the *Show parameter hints* option. To adjust the hint settings, click on the *Exclude list...* link and make any necessary adjustments.

## 8.7 Code Generation

In addition to completing code as it is typed the editor can, under certain conditions, also generate code for you. The list of available code generation options shown in Figure 8-12 can be accessed using the Alt-Insert (Cmd-N on macOS) keyboard shortcut when the cursor is at the location in the file where the code is to be generated.

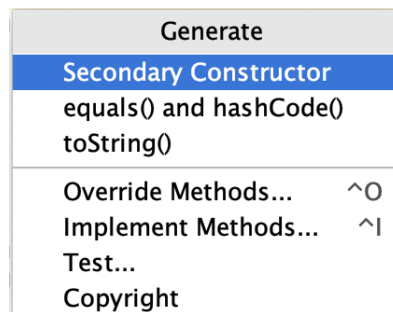


Figure 8-12

For the purposes of an example, consider a situation where we want to be notified when an Activity in our project is about to be destroyed by the operating system. As will be outlined in a later chapter of this book, this can be achieved by overriding the `onStop()` lifecycle method of the Activity superclass. To have Android Studio

generate a stub method for this, simply select the *Override Methods...* option from the code generation list and select the *onStop()* method from the resulting list of available methods:

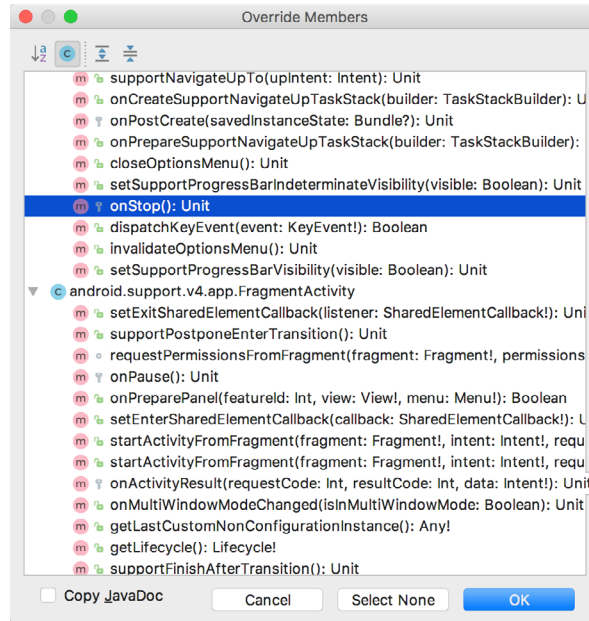


Figure 8-13

Having selected the method to override, clicking on OK will generate the stub method at the current cursor location in the Kotlin source file as follows:

```
override fun onStop() {
    super.onStop()
}
```

## 8.8 Code Folding

Once a source code file reaches a certain size, even the most carefully formatted and well organized code can become overwhelming and difficult to navigate. Android Studio takes the view that it is not always necessary to have the content of every code block visible at all times. Code navigation can be made easier through the use of the code folding feature of the Android Studio editor. Code folding is controlled using markers appearing in the editor gutter at the beginning and end of each block of code in a source file. Figure 8-14, for example, highlights the start and end markers for a method declaration which is not currently folded:

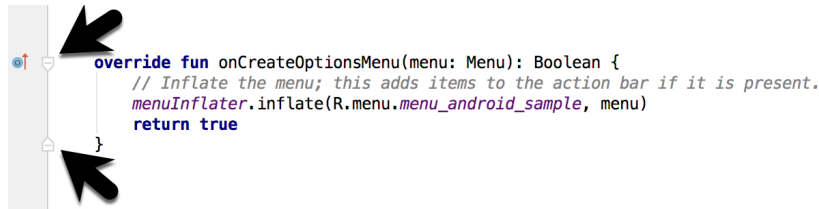


Figure 8-14

Clicking on either of these markers will fold the statement such that only the signature line is visible as shown in Figure 8-15:

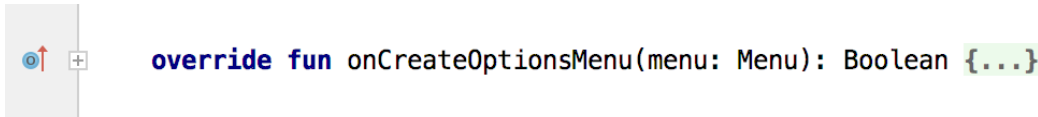


Figure 8-15

To unfold a collapsed section of code, simply click on the '+' marker in the editor gutter. To see the hidden code without unfolding it, hover the mouse pointer over the "{...}" indicator as shown in Figure 8-16. The editor will then display the lens overlay containing the folded code block:

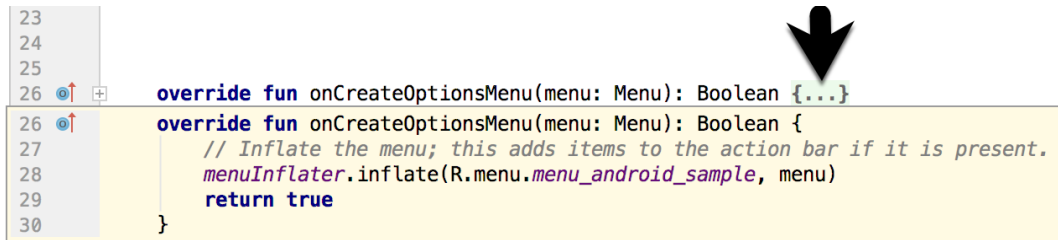


Figure 8-16

All of the code blocks in a file may be folded or unfolded using the Ctrl-Shift-Plus and Ctrl-Shift-Minus keyboard sequences.

By default, the Android Studio editor will automatically fold some code when a source file is opened. To configure the conditions under which this happens, select File -> Settings... (Android Studio -> Preferences... on macOS) and choose the Editor -> General -> Code Folding entry in the resulting settings panel (Figure 8-17):

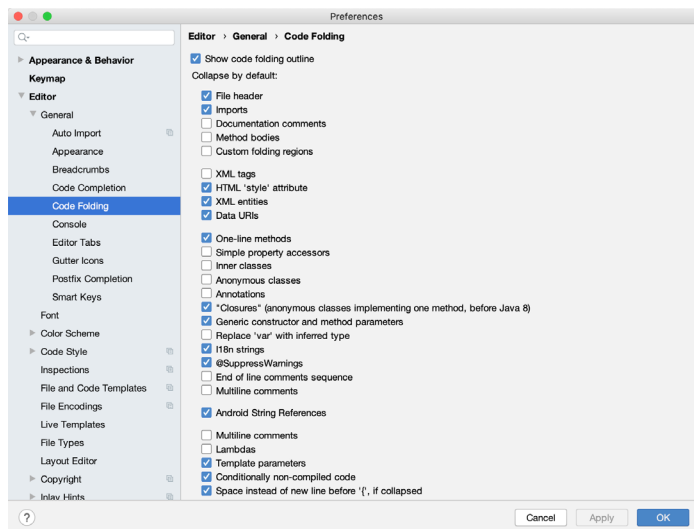


Figure 8-17

## 8.9 Quick Documentation Lookup

Context sensitive Kotlin and Android documentation can be accessed by placing the cursor over the declaration for which documentation is required and pressing the Ctrl-Q keyboard shortcut (Ctrl-J on macOS). This will display a popup containing the relevant reference documentation for the item. Figure 8-18, for example, shows the documentation for the Android Menu class.



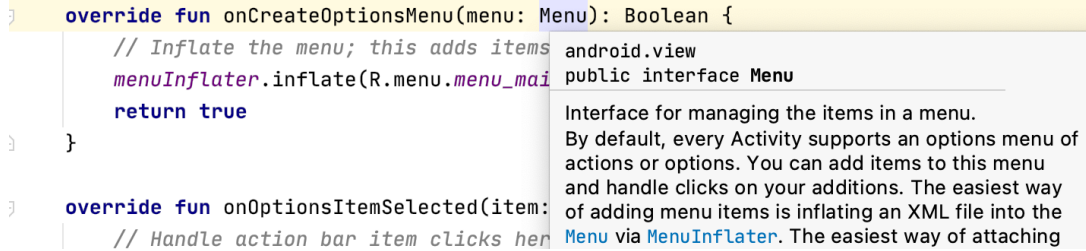


Figure 8-18

Once displayed, the documentation popup can be moved around the screen as needed.

## 8.10 Code Reformatting

In general, the Android Studio editor will automatically format code in terms of indenting, spacing and nesting of statements and code blocks as they are added. In situations where lines of code need to be reformatted (a common occurrence, for example, when cutting and pasting sample code from a website), the editor provides a source code reformatting feature which, when selected, will automatically reformat code to match the prevailing code style.

To reformat source code, press the Ctrl-Alt-L (Cmd-Opt-L on macOS) keyboard shortcut sequence. To display the Reformat Code dialog (Figure 8-19) use the Ctrl-Alt-Shift-L (Cmd-Opt-Shift-L on macOS). This dialog provides the option to reformat only the currently selected code, the entire source file currently active in the editor or only code that has changed as the result of a source code control update.

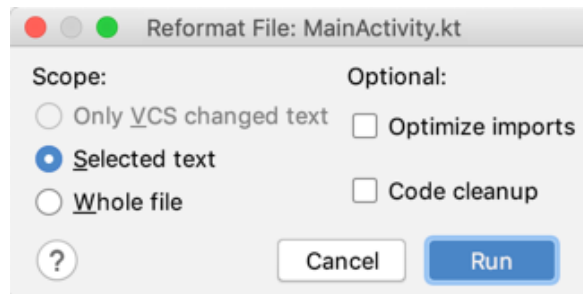


Figure 8-19

The full range of code style preferences can be changed from within the project settings dialog. Select the *File* -> *Settings* menu option (*Android Studio* -> *Preferences...* on macOS) and choose Code Style in the left-hand panel to access a list of supported programming and markup languages. Selecting a language will provide access to a vast array of formatting style options, all of which may be modified from the Android Studio default to match your preferred code style. To configure the settings for the Rearrange code option in the above dialog, for example, unfold the Code Style section, select Kotlin and, from the Kotlin settings, select the Arrangement tab.

## 8.11 Finding Sample Code

The Android Studio editor provides a way to access sample code relating to the currently highlighted entry within the code listing. This feature can be useful for learning how a particular Android class or method is used. To find sample code, highlight a method or class name in the editor, right-click on it and select the *Find Sample Code* menu option. The Find Sample Code panel (Figure 8-20) will appear beneath the editor with a list of matching samples. Selecting a sample from the list will load the corresponding code into the right-hand panel:

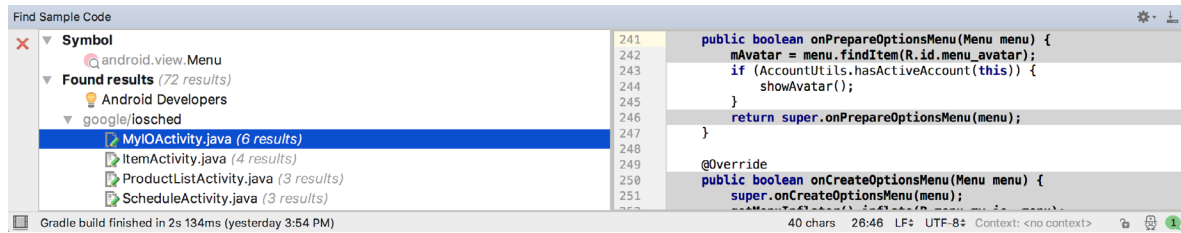


Figure 8-20

## 8.12 Live Templates

As you write Android code you will find that there are common constructs that are used frequently. For example, a common requirement is to display a popup message to the user using the Android Toast class. Live templates are a collection of common code constructs that can be entered into the editor by typing the initial characters followed by a special key (set to the Tab key by default) to insert template code. To experience this in action, type toast in the code editor followed by the Tab key and Android Studio will insert the following code at the cursor position ready for editing:

```
Toast.makeText(, "", Toast.LENGTH_SHORT).show()
```

To list and edit existing templates, change the special key, or add your own templates, open the Preferences dialog and select Live Templates from the Editor section of the left-hand navigation panel:

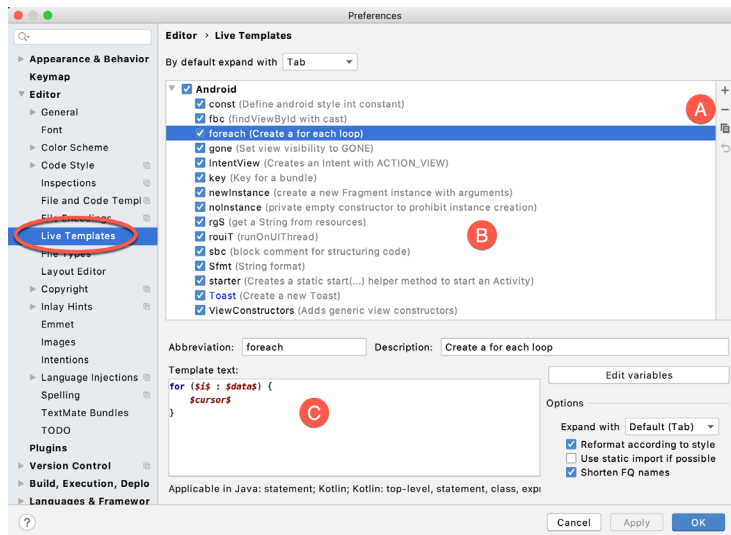


Figure 8-21

Add, remove, duplicate or reset templates using the buttons marked A in Figure 8-21 above. To modify a template, select it from the list (B) and change the settings in the panel marked C.

## 8.13 Summary

The Android Studio editor goes to great length to reduce the amount of typing needed to write code and to make that code easier to read and navigate. In this chapter we have covered a number of the key editor features including code completion, code generation, editor window splitting, code folding, reformatting, documentation lookup and live templates.

## 11. An Introduction to Kotlin

Android development is performed primarily using Android Studio which is, in turn, based on the IntelliJ IDEA development environment created by a company named JetBrains. Prior to the release of Android Studio 3.0, all Android apps were written using Android Studio and the Java programming language (with some occasional C++ code when needed).

Since the introduction of Android Studio 3.0, however, developers now have the option of creating Android apps using another programming language called Kotlin. Although detailed coverage of all features of this language is beyond the scope of this book (entire books can and have been written covering solely Kotlin), the objective of this and the following six chapters is to provide enough information to begin programming in Kotlin and quickly get up to speed developing Android apps using this programming language.

### 11.1 What is Kotlin?

Named after an island located in the Baltic Sea, Kotlin is a programming language created by JetBrains and follows Java in the tradition of naming programming languages after islands. Kotlin code is intended to be easier to understand and write and also safer than many other programming languages. The language, compiler and related tools are all open source and available for free under the Apache 2 license.

The primary goals of the Kotlin language are to make code both concise and safe. Code is generally considered concise when it can be easily read and understood. Conciseness also plays a role when writing code, allowing code to be written more quickly and with greater efficiency. In terms of safety, Kotlin includes a number of features that improve the chances that potential problems will be identified when the code is being written instead of causing runtime crashes.

A third objective in the design and implementation of Kotlin involves interoperability with Java.

### 11.2 Kotlin and Java

Originally introduced by Sun Microsystems in 1995 Java is still by far the most popular programming language in use today. Until the introduction of Kotlin, it is quite likely that every Android app available on the market was written in Java. Since acquiring the Android operating system, Google has invested heavily in tuning and optimizing compilation and runtime environments for running Java-based code on Android devices.

Rather than try to re-invent the wheel, Kotlin is designed to both integrate with and work alongside Java. When Kotlin code is compiled it generates the same bytecode as that generated by the Java compiler enabling projects to be built using a combination of Java and Kotlin code. This compatibility also allows existing Java frameworks and libraries to be used seamlessly from within Kotlin code and also for Kotlin code to be called from within Java.

Kotlin's creators also acknowledged that while there were ways to improve on existing languages, there are many features of Java that did not need to be changed. Consequently, those familiar with programming in Java will find many of these skills to be transferable to Kotlin-based development. Programmers with Swift programming experience will also find much that is familiar when learning Kotlin.

### 11.3 Converting from Java to Kotlin

Given the high level of interoperability between Kotlin and Java it is not essential to convert existing Java code to Kotlin since these two languages will comfortably co-exist within the same project. That being said, Java code

can be converted to Kotlin from within Android Studio using a built-in Java to Kotlin converter. To convert an entire Java source file to Kotlin, load the file into the Android Studio code editor and select the *Code -> Convert Java File to Kotlin File* menu option. Alternatively, blocks of Java code may be converted to Kotlin by cutting the code and pasting it into an existing Kotlin file within the Android Studio code editor. Note when performing Java to Kotlin conversions that the Java code will not always convert to the best possible Kotlin code and that time should be taken to review and tidy up the code after conversion.

### 11.4 Kotlin and Android Studio

Support for Kotlin is provided within Android Studio via the Kotlin Plug-in which is integrated by default into Android Studio 3.0 or later.

### 11.5 Experimenting with Kotlin

When learning a new programming language, it is often useful to be able to enter and execute snippets of code. One of the best ways to do this with Kotlin is to use the Kotlin Playground (Figure 11-1) located at <https://play.kotlinlang.org>:

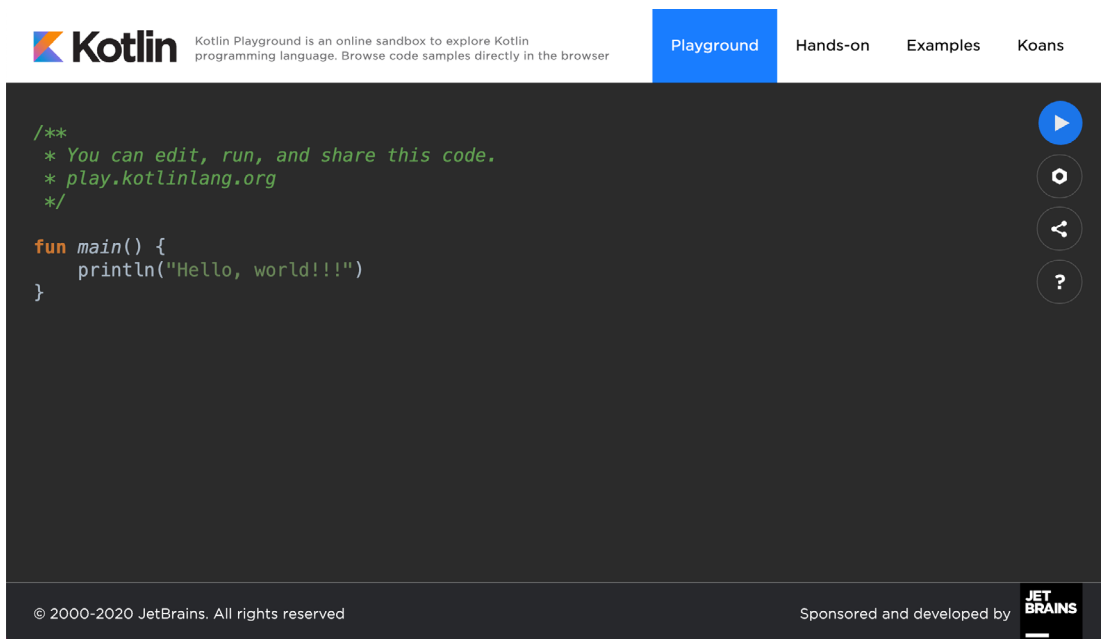


Figure 11-1

In addition to providing an environment in which Kotlin code may be quickly entered and executed, the playground also includes a set of examples and tutorials demonstrating key Kotlin features in action.

Try out some Kotlin code by opening a browser window, navigating to the playground and entering the following into the main code panel:

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

    println("Welcome to Kotlin")

    for (i in 1..8) {
        println("i = $i")
    }
}
```

```
}
```

After entering the code, click on the Run button and note the output in the console panel:

```
Welcome to Kotlin
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
```

Figure 11-2

## 11.6 Semi-colons in Kotlin

Unlike programming languages such as Java and C++, Kotlin does not require semi-colons at the end of each statement or expression line. The following, therefore, is valid Kotlin code:

```
val mynumber = 10
println(mynumber)
```

Semi-colons are only required when multiple statements appear on the same line:

```
val mynumber = 10; println(mynumber)
```

## 11.7 Summary

For the first time since the Android operating system was introduced, developers now have an alternative to writing apps in Java code. Kotlin is a programming language developed by JetBrains, the company that created the development environment on which Android Studio is based. Kotlin is intended to make code safer and easier to understand and write. Kotlin is also highly compatible with Java, allowing Java and Kotlin code to co-exist within the same projects. This interoperability ensures that most of the standard Java and Java-based Android libraries and frameworks are available for use when developing using Kotlin.

Kotlin support for Android Studio is provided via a plug-in bundled with Android Studio 3.0 or later. This plug-in also provides a converter to translate Java code to Kotlin.

When learning Kotlin, the online playground provides a useful environment for quickly trying out Kotlin code.



## 15. An Overview of Kotlin Functions and Lambdas

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

### 15.1 What is a Function?

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

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

### 15.2 How to Declare a Kotlin Function

A Kotlin function is declared using the following syntax:

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

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

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

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

```
fun sayHello() {
```

## An Overview of Kotlin Functions and Lambdas

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

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

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

### 15.3 Calling a Kotlin Function

Once declared, functions are called using the following syntax:

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

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

```
sayHello()
```

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

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

### 15.4 Single Expression Functions

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

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

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

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

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

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

### 15.5 Local Functions

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

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

    val name = "John"
    val count = 5

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



```
        displayString()
    }
}
```

## 15.6 Handling Return Values

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

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

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

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

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

## 15.7 Declaring Default Function Parameters

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

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

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

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

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

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

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

## 15.8 Variable Number of Function Parameters

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

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

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

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

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

## 15.9 Lambda Expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Similarly, the following lambda simply returns a string literal:

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

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

## 15.10 Higher-order Functions

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

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

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

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

In order to accept a function as a parameter, the receiving function simply declares the type of the function it is able to accept.

For the purposes of an example, we will begin by declaring two unit conversion functions:

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

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

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

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

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

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

Functions can also be returned as a data type simply by declaring the type of the function as the return type.

## An Overview of Kotlin Functions and Lambdas

The following function is configured to return either our `inchesToFeet` or `inchesToYards` function type (in other words a function which accepts and returns a `Double` value) based on the value of a `Boolean` parameter:

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

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

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

### 15.11 Summary

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

## 19. Understanding Android Application and Activity Lifecycles

In earlier chapters we have learned that Android applications run within processes and that they are comprised of multiple components in the form of activities, services and broadcast receivers. The goal of this chapter is to expand on this knowledge by looking at the lifecycle of applications and activities within the Android runtime system.

Regardless of the fanfare about how much memory and computing power resides in the mobile devices of today compared to the desktop systems of yesterday, it is important to keep in mind that these devices are still considered to be “resource constrained” by the standards of modern desktop and laptop based systems, particularly in terms of memory. As such, a key responsibility of the Android system is to ensure that these limited resources are managed effectively and that both the operating system and the applications running on it remain responsive to the user at all times. To achieve this, Android is given full control over the lifecycle and state of both the processes in which the applications run, and the individual components that comprise those applications.

An important factor in developing Android applications, therefore, is to gain an understanding of both the application and activity lifecycle management models of Android, and the ways in which an application can react to the state changes that are likely to be imposed upon it during its execution lifetime.

### 19.1 Android Applications and Resource Management

Each running Android application is viewed by the operating system as a separate process. If the system identifies that resources on the device are reaching capacity it will take steps to terminate processes to free up memory.

When making a determination as to which process to terminate to free up memory, the system takes into consideration both the *priority* and *state* of all currently running processes, combining these factors to create what is referred to by Google as an *importance hierarchy*. Processes are then terminated starting with the lowest priority and working up the hierarchy until sufficient resources have been liberated for the system to function.

### 19.2 Android Process States

Processes host applications and applications are made up of components. Within an Android system, the current state of a process is defined by the highest-ranking active component within the application that it hosts. As outlined in Figure 19-1, a process can be in one of the following five states at any given time:

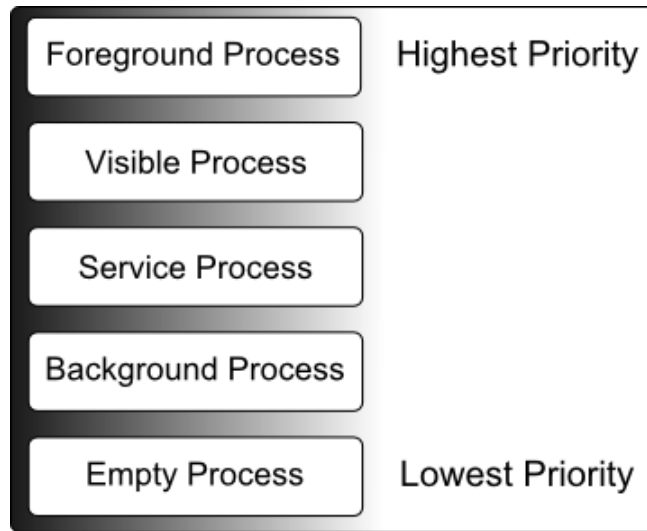


Figure 19-1

### 19.2.1 Foreground Process

These processes are assigned the highest level of priority. At any one time, there are unlikely to be more than one or two foreground processes active and these are usually the last to be terminated by the system. A process must meet one or more of the following criteria to qualify for foreground status:

- Hosts an activity with which the user is currently interacting.
- Hosts a Service connected to the activity with which the user is interacting.
- Hosts a Service that has indicated, via a call to *startForeground()*, that termination would be disruptive to the user experience.
- Hosts a Service executing either its *onCreate()*, *onResume()* or *onStart()* callbacks.
- Hosts a Broadcast Receiver that is currently executing its *onReceive()* method.

### 19.2.2 Visible Process

A process containing an activity that is visible to the user but is not the activity with which the user is interacting is classified as a “visible process”. This is typically the case when an activity in the process is visible to the user, but another activity, such as a partial screen or dialog, is in the foreground. A process is also eligible for visible status if it hosts a Service that is, itself, bound to a visible or foreground activity.

### 19.2.3 Service Process

Processes that contain a Service that has already been started and is currently executing.

### 19.2.4 Background Process

A process that contains one or more activities that are not currently visible to the user, and does not host a Service that qualifies for *Service Process* status. Processes that fall into this category are at high risk of termination if additional memory needs to be freed for higher priority processes. Android maintains a dynamic list of background processes, terminating processes in chronological order such that processes that were the least recently in the foreground are killed first.

### 19.2.5 Empty Process

Empty processes no longer contain any active applications and are held in memory ready to serve as hosts for newly launched applications. This is somewhat analogous to keeping the doors open and the engine running on a bus in anticipation of passengers arriving. Such processes are, obviously, considered the lowest priority and are the first to be killed to free up resources.

### 19.3 Inter-Process Dependencies

The situation with regard to determining the highest priority process is slightly more complex than outlined in the preceding section for the simple reason that processes can often be inter-dependent. As such, when making a determination as to the priority of a process, the Android system will also take into consideration whether the process is in some way serving another process of higher priority (for example, a service process acting as the content provider for a foreground process). As a basic rule, the Android documentation states that a process can never be ranked lower than another process that it is currently serving.

### 19.4 The Activity Lifecycle

As we have previously determined, the state of an Android process is determined largely by the status of the activities and components that make up the application that it hosts. It is important to understand, therefore, that these activities also transition through different states during the execution lifetime of an application. The current state of an activity is determined, in part, by its position in something called the *Activity Stack*.

### 19.5 The Activity Stack

For each application that is running on an Android device, the runtime system maintains an *Activity Stack*. When an application is launched, the first of the application's activities to be started is placed onto the stack. When a second activity is started, it is placed on the top of the stack and the previous activity is *pushed* down. The activity at the top of the stack is referred to as the *active* (or *running*) activity. When the active activity exits, it is *popped* off the stack by the runtime and the activity located immediately beneath it in the stack becomes the current active activity. The activity at the top of the stack might, for example, simply exit because the task for which it is responsible has been completed. Alternatively, the user may have selected a "Back" button on the screen to return to the previous activity, causing the current activity to be popped off the stack by the runtime system and therefore destroyed. A visual representation of the Android Activity Stack is illustrated in Figure 19-2.

As shown in the diagram, new activities are pushed on to the top of the stack when they are started. The current active activity is located at the top of the stack until it is either pushed down the stack by a new activity, or popped off the stack when it exits or the user navigates to the previous activity. If resources become constrained, the runtime will kill activities, starting with those at the bottom of the stack.

The Activity Stack is what is referred to in programming terminology as a Last-In-First-Out (LIFO) stack in that the last item to be pushed onto the stack is the first to be popped off.

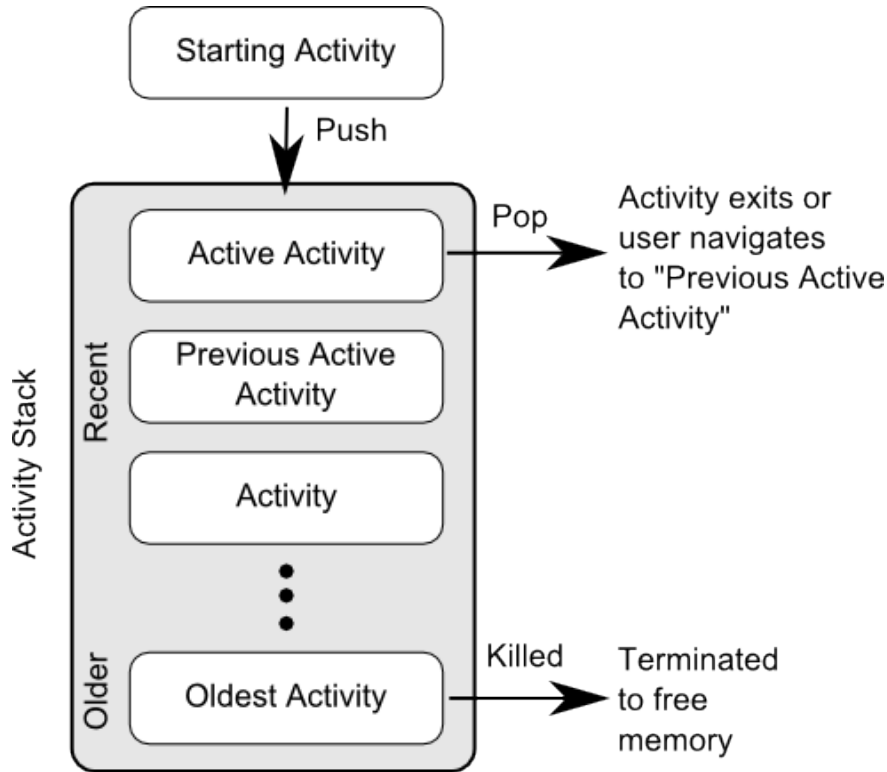


Figure 19-2

## 19.6 Activity States

An activity can be in one of a number of different states during the course of its execution within an application:

- **Active / Running** – The activity is at the top of the Activity Stack, is the foreground task visible on the device screen, has focus and is currently interacting with the user. This is the least likely activity to be terminated in the event of a resource shortage.
- **Paused** – The activity is visible to the user but does not currently have focus (typically because this activity is partially obscured by the current *active* activity). Paused activities are held in memory, remain attached to the window manager, retain all state information and can quickly be restored to active status when moved to the top of the Activity Stack.
- **Stopped** – The activity is currently not visible to the user (in other words it is totally obscured on the device display by other activities). As with paused activities, it retains all state and member information, but is at higher risk of termination in low memory situations.
- **Killed** – The activity has been terminated by the runtime system in order to free up memory and is no longer present on the Activity Stack. Such activities must be restarted if required by the application.

## 19.7 Configuration Changes

So far in this chapter, we have looked at two of the causes for the change in state of an Android activity, namely the movement of an activity between the foreground and background, and termination of an activity by the runtime system to free up memory. In fact, there is a third scenario in which the state of an activity can dramatically change and this involves a change to the device configuration.



By default, any configuration change that impacts the appearance of an activity (such as rotating the orientation of the device between portrait and landscape, or changing a system font setting) will cause the activity to be destroyed and recreated. The reasoning behind this is that such changes affect resources such as the layout of the user interface and simply destroying and recreating impacted activities is the quickest way for an activity to respond to the configuration change. It is, however, possible to configure an activity so that it is not restarted by the system in response to specific configuration changes.

## 19.8 Handling State Change

If nothing else, it should be clear from this chapter that an application and, by definition, the components contained therein will transition through many states during the course of its lifespan. Of particular importance is the fact that these state changes (up to and including complete termination) are imposed upon the application by the Android runtime subject to the actions of the user and the availability of resources on the device.

In practice, however, these state changes are not imposed entirely without notice and an application will, in most circumstances, be notified by the runtime system of the changes and given the opportunity to react accordingly. This will typically involve saving or restoring both internal data structures and user interface state, thereby allowing the user to switch seamlessly between applications and providing at least the appearance of multiple, concurrently running applications.

Android provides two ways to handle the changes to the lifecycle states of the objects within in app. One approach involves responding to state change method calls from the operating system and is covered in detail in the next chapter entitled “*Handling Android Activity State Changes*”.

A new approach, and one that is recommended by Google, involves the lifecycle classes included with the Jetpack Android Architecture components, introduced in “*Modern Android App Architecture with Jetpack*” and explained in more detail in the chapter entitled “*Working with Android Lifecycle-Aware Components*”.

## 19.9 Summary

Mobile devices are typically considered to be resource constrained, particularly in terms of on-board memory capacity. Consequently, a prime responsibility of the Android operating system is to ensure that applications, and the operating system in general, remain responsive to the user.

Applications are hosted on Android within processes. Each application, in turn, is made up of components in the form of activities and Services.

The Android runtime system has the power to terminate both processes and individual activities to free up memory. Process state is taken into consideration by the runtime system when deciding whether a process is a suitable candidate for termination. The state of a process is largely dependent upon the status of the activities hosted by that process.

The key message of this chapter is that an application moves through a variety of states during its execution lifespan and has very little control over its destiny within the Android runtime environment. Those processes and activities that are not directly interacting with the user run a higher risk of termination by the runtime system. An essential element of Android application development, therefore, involves the ability of an application to respond to state change notifications from the operating system.



## 26. A Guide to Using ConstraintLayout in Android Studio

As mentioned more than once in previous chapters, Google has made significant changes to the Android Studio Layout Editor tool, many of which were made solely to support user interface layout design using ConstraintLayout. Now that the basic concepts of ConstraintLayout have been outlined in the previous chapter, this chapter will explore these concepts in more detail while also outlining the ways in which the Layout Editor tool allows ConstraintLayout-based user interfaces to be designed and implemented.

### 26.1 Design and Layout Views

The chapter entitled “A Guide to the Android Studio Layout Editor Tool” explained that the Android Studio Layout Editor tool provides two ways to view the user interface layout of an activity in the form of Design and Layout (also known as blueprint) views. These views of the layout may be displayed individually or, as in Figure 26-1, side-by-side:

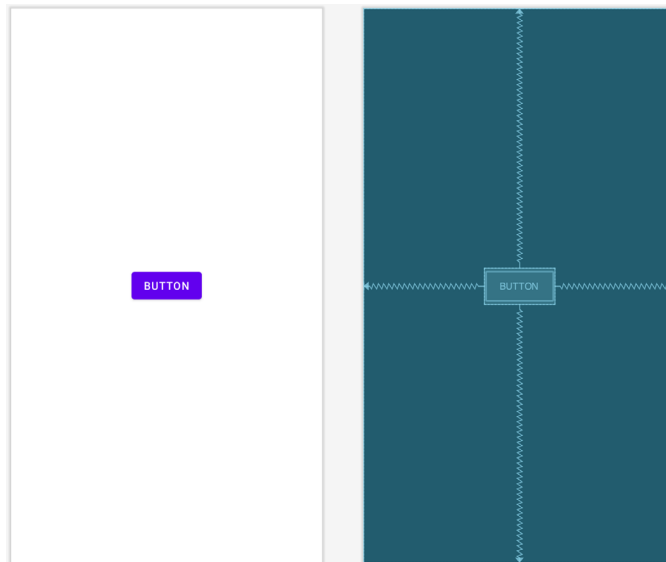


Figure 26-1

The Design view (positioned on the left in the above figure) presents a “what you see is what you get” representation of the layout, wherein the layout appears as it will within the running app. The Layout view, on the other hand, displays a blueprint style of view where the widgets are represented by shaded outlines. As can be seen in Figure 26-1 above, Layout view also displays the constraint connections (in this case opposing constraints used to center a button within the layout). These constraints are also overlaid onto the Design view when a specific widget in the layout is selected or when the mouse pointer hovers over the design area as illustrated in Figure 26-2:

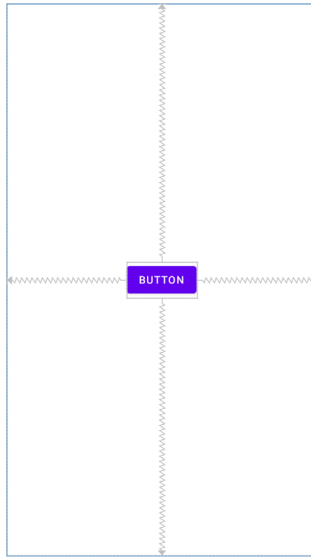


Figure 26-2

The appearance of constraint connections in both views can be changed using the View Options menu shown in Figure 26-3:

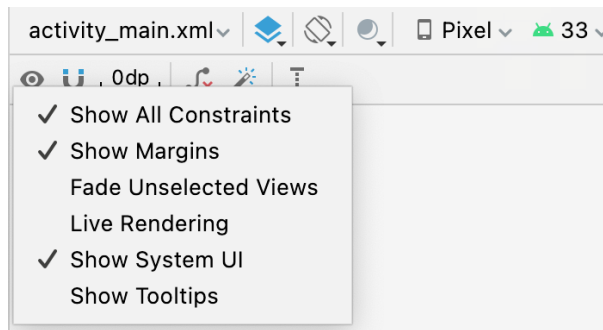


Figure 26-3

In addition to the two modes of displaying the user interface layout, the Layout Editor tool also provides three different ways of establishing the constraints required for a specific layout design.

## 26.2 Autoconnect Mode

Autoconnect, as the name suggests, automatically establishes constraint connections as items are added to the layout. Autoconnect mode may be enabled and disabled using the toolbar button indicated in Figure 26-4:

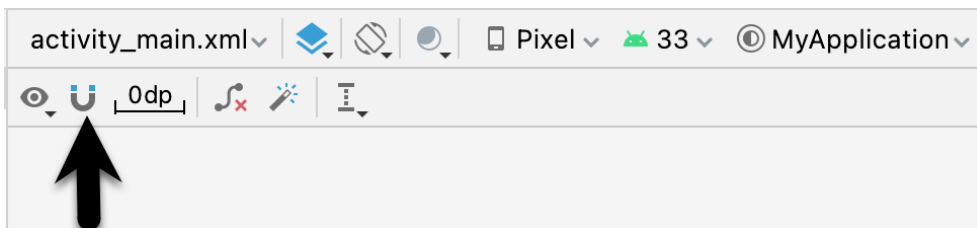


Figure 26-4

Autoconnect mode uses algorithms to decide the best constraints to establish based on the position of the widget and the widget's proximity to both the sides of the parent layout and other elements in the layout. If any of the automatic constraint connections fail to provide the desired behavior, these may be changed manually as outlined later in this chapter.

## 26.3 Inference Mode

Inference mode uses a heuristic approach involving algorithms and probabilities to automatically implement constraint connections after widgets have already been added to the layout. This mode is usually used when the Autoconnect feature has been turned off and objects have been added to the layout without any constraint connections. This allows the layout to be designed simply by dragging and dropping objects from the palette onto the layout canvas and making size and positioning changes until the layout appears as required. In essence this involves “painting” the layout without worrying about constraints. Inference mode may also be used at any time during the design process to fill in missing constraints within a layout.

Constraints are automatically added to a layout when the *Infer constraints* button (Figure 26-5) is clicked:

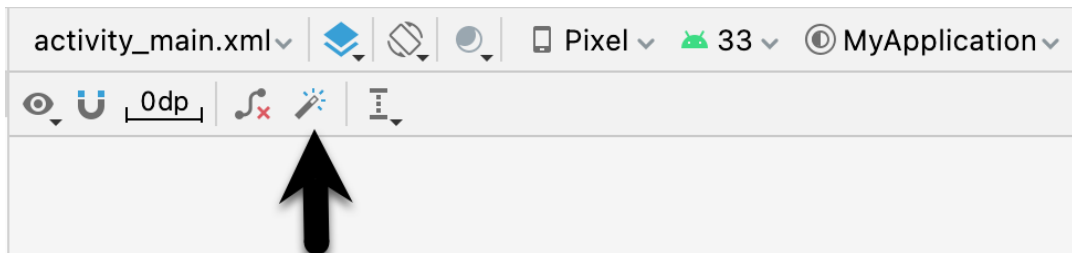


Figure 26-5

As with Autoconnect mode, there is always the possibility that the Layout Editor tool will infer incorrect constraints, though these may be modified and corrected manually.

## 26.4 Manipulating Constraints Manually

The third option for implementing constraint connections is to do so manually. When doing so, it will be helpful to understand the various handles that appear around a widget within the Layout Editor tool. Consider, for example, the widget shown in Figure 26-6:

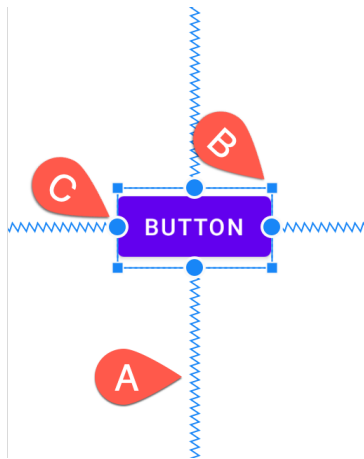


Figure 26-6

Clearly the spring-like lines (A) represent established constraint connections leading from the sides of the widget

to the targets. The small square markers (B) in each corner of the object are resize handles which, when clicked and dragged, serve to resize the widget. The small circle handles (C) located on each side of the widget are the side constraint anchors. To create a constraint connection, click on the handle and drag the resulting line to the element to which the constraint is to be connected (such as a guideline or the side of either the parent layout or another widget) as outlined in Figure 26-7. When connecting to the side of another widget, simply drag the line to the side constraint handle of that widget and release the line when the widget and handle highlight.

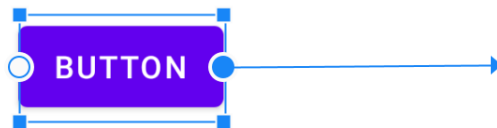


Figure 26-7

If the constraint line is dragged to a widget and released, but not attached to a constraint handle, the layout editor will display a menu containing a list of the sides to which the constraint may be attached. In Figure 26-8, for example, the constraint can be attached to the top or bottom edge of the destination button widget:

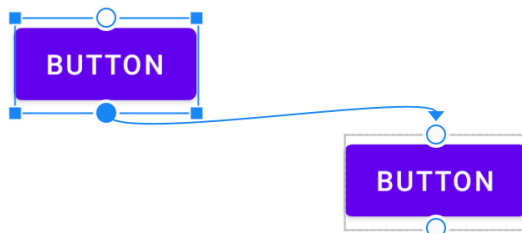


Figure 26-8

An additional marker indicates the anchor point for baseline constraints whereby the content within the widget (as opposed to outside edges) is used as the alignment point. To display this marker, simply right-click on the widget and select the *Show Baseline* menu option. To establish a constraint connection from a baseline constraint handle, simply hover the mouse pointer over the handle until it highlights before clicking and dragging to the target (such as the baseline anchor of another widget as shown in Figure 26-9).

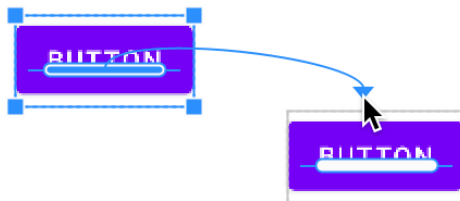


Figure 26-9

To hide the baseline anchors, right click on the widget a second time and select the *Hide Baseline* menu option.

## 26.5 Adding Constraints in the Inspector

Constraints may also be added to a view within the Android Studio Layout Editor tool using the *Inspector* panel located in the Attributes tool window as shown in Figure 26-10. The square in the center represents the currently selected view and the areas around the square the constraints, if any, applied to the corresponding sides of the view:

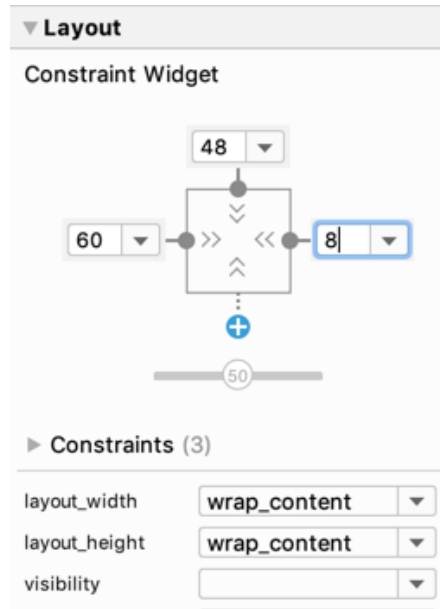


Figure 26-10

The absence of a constraint on a side of the view is represented by a dotted line leading to a blue circle containing a plus sign (as is the case with the bottom edge of the view in the above figure). To add a constraint, simply click on this blue circle and the layout editor will add a constraint connected to what it considers to be the most appropriate target within the layout.

## 26.6 Viewing Constraints in the Attributes Window

A list of constraints configured on the currently select widget can be viewing by displaying the Constraints section of the Attributes tool window as shown in Figure 26-11 below:

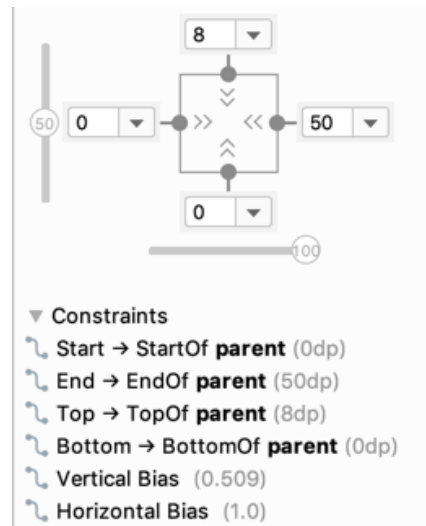


Figure 26-11

Clicking on a constraint in the list will select that constraint within the design layout.

## 26.7 Deleting Constraints

To delete an individual constraint, simply select the constraint either within the design layout or the Attributes tool window so that it highlights (in Figure 26-12, for example, the right-most constraint has been selected) and tap the keyboard delete key. The constraint will then be removed from the layout.

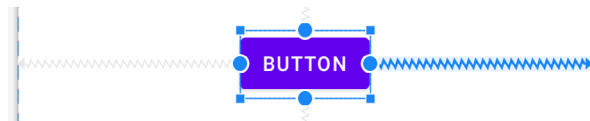


Figure 26-12

Another option is to hover the mouse pointer over the constraint anchor while holding down the Ctrl (Cmd on macOS) key and clicking on the anchor after it turns red:

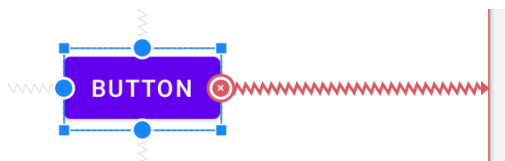


Figure 26-13

Alternatively, remove all of the constraints on a widget by right-clicking on it selecting the *Clear Constraints of Selection* menu option.

To remove all of the constraints from every widget in a layout, use the toolbar button highlighted in Figure 26-14:

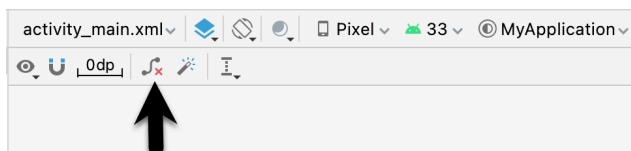


Figure 26-14

## 26.8 Adjusting Constraint Bias

In the previous chapter, the concept of using bias settings to favor one opposing constraint over another was outlined. Bias within the Android Studio Layout Editor tool is adjusted using the *Inspector* located in the Attributes tool window and shown in Figure 26-15. The two sliders indicated by the arrows in the figure are used to control the bias of the vertical and horizontal opposing constraints of the currently selected widget.

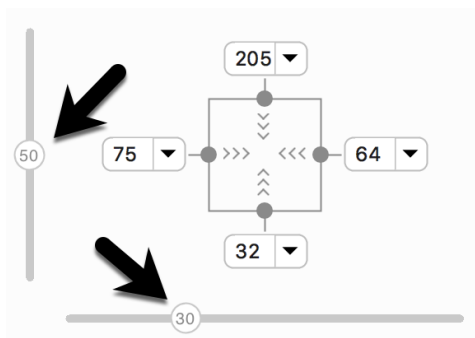


Figure 26-15



## 26.9 Understanding ConstraintLayout Margins

Constraints can be used in conjunction with margins to implement fixed gaps between a widget and another element (such as another widget, a guideline or the side of the parent layout). Consider, for example, the horizontal constraints applied to the Button object in Figure 26-16:

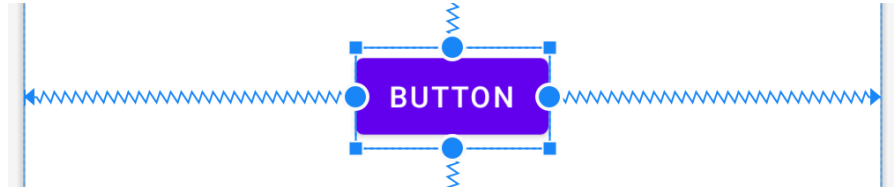


Figure 26-16

As currently configured, horizontal constraints run to the left and right edges of the parent ConstraintLayout. As such, the widget has opposing horizontal constraints indicating that the ConstraintLayout layout engine has some discretion in terms of the actual positioning of the widget at runtime. This allows the layout some flexibility to accommodate different screen sizes and device orientation. The horizontal bias setting is also able to control the position of the widget right up to the right-hand side of the layout. Figure 26-17, for example, shows the same button with 100% horizontal bias applied:



Figure 26-17

ConstraintLayout margins can appear at the end of constraint connections and represent a fixed gap into which the widget cannot be moved even when adjusting bias or in response to layout changes elsewhere in the activity. In Figure 26-18, the right-hand constraint now includes a 50dp margin into which the widget cannot be moved even though the bias is still set at 100%.



Figure 26-18

Existing margin values on a widget can be modified from within the Inspector. As can be seen in Figure 26-19, a drop-down menu is being used to change the right-hand margin on the currently selected widget to 16dp. Alternatively, clicking on the current value also allows a number to be typed into the field.

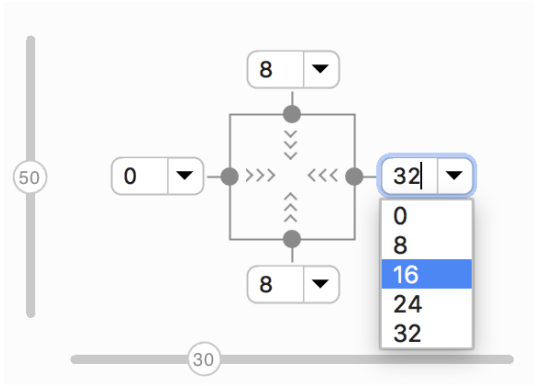


Figure 26-19

The default margin for new constraints can be changed at any time using the option in the toolbar highlighted in Figure 26-20:

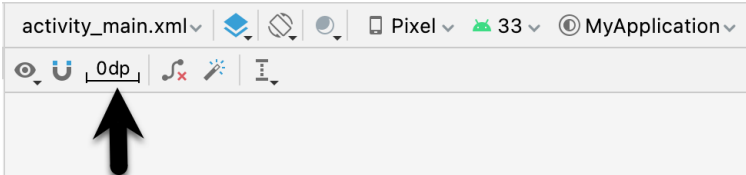


Figure 26-20

### 26.10 The Importance of Opposing Constraints and Bias

As discussed in the previous chapter, opposing constraints, margins and bias form the cornerstone of responsive layout design in Android when using the ConstraintLayout. When a widget is constrained without opposing constraint connections, those constraints are essentially margin constraints. This is indicated visually within the Layout Editor tool by solid straight lines accompanied by margin measurements as shown in Figure 26-21.

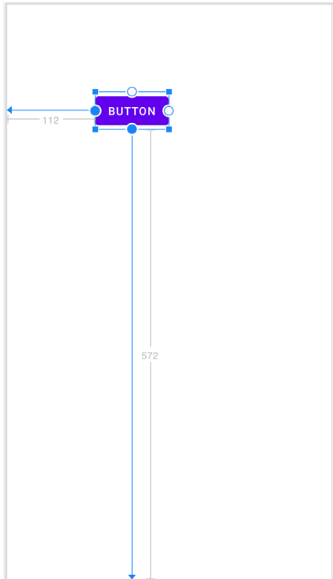


Figure 26-21

The above constraints essentially fix the widget at that position. The result of this is that if the device is rotated to landscape orientation, the widget will no longer be visible since the vertical constraint pushes it beyond the top edge of the device screen (as is the case in Figure 26-22). A similar problem will arise if the app is run on a device with a smaller screen than that used during the design process.



Figure 26-22

When opposing constraints are implemented, the constraint connection is represented by the spring-like jagged line (the spring metaphor is intended to indicate that the position of the widget is not fixed to absolute X and Y coordinates):

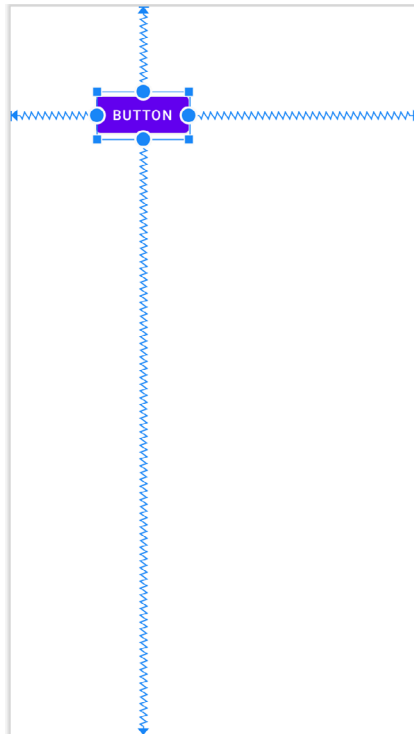


Figure 26-23

In the above layout, vertical and horizontal bias settings have been configured such that the widget will always be positioned 90% of the distance from the bottom and 35% from the left-hand edge of the parent layout. When

rotated, therefore, the widget is still visible and positioned in the same location relative to the dimensions of the screen:

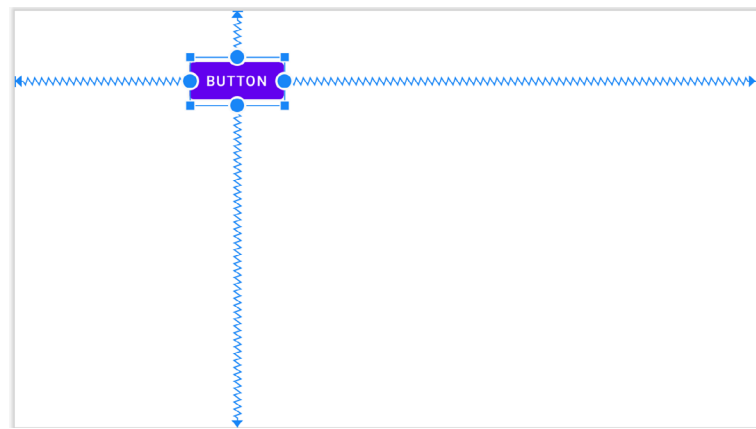


Figure 26-24

When designing a responsive and adaptable user interface layout, it is important to take into consideration both bias and opposing constraints when manually designing a user interface layout and making corrections to automatically created constraints.

### 26.11 Configuring Widget Dimensions

The inner dimensions of a widget within a ConstraintLayout can also be configured using the Inspector. As outlined in the previous chapter, widget dimensions can be set to wrap content, fixed or match constraint modes. The prevailing settings for each dimension on the currently selected widget are shown within the square representing the widget in the Inspector as illustrated in Figure 26-25:

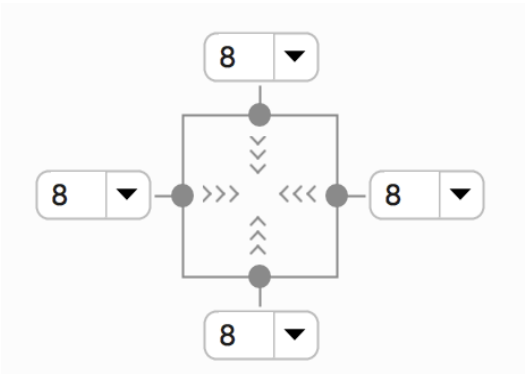


Figure 26-25

In the above figure, both the horizontal and vertical dimensions are set to wrap content mode (indicated by the inward pointing chevrons). The inspector uses the following visual indicators to represent the three dimension modes:

|                  |  |
|------------------|--|
| Fixed Size       |  |
| Match Constraint |  |

|              |     |
|--------------|-----|
| Wrap Content | >>> |
|--------------|-----|

Table 26-1

To change the current setting, simply click on the indicator to cycle through the three different settings. When the dimension of a view within the layout editor is set to match constraint mode, the corresponding sides of the view are drawn with the spring-like line instead of the usual straight lines. In Figure 26-26, for example, only the width of the view has been set to match constraint:

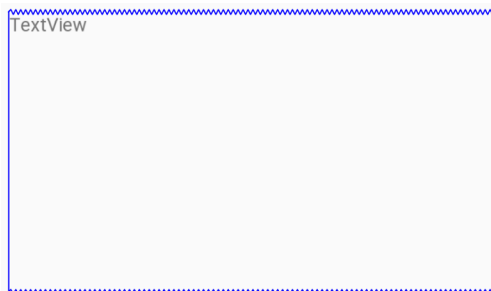


Figure 26-26

In addition, the size of a widget can be expanded either horizontally or vertically to the maximum amount allowed by the constraints and other widgets in the layout using the *Expand horizontally* and *Expand vertically* options. These are accessible by right clicking on a widget within the layout and selecting the *Organize* option from the resulting menu (Figure 26-27). When used, the currently selected widget will increase in size horizontally or vertically to fill the available space around it.

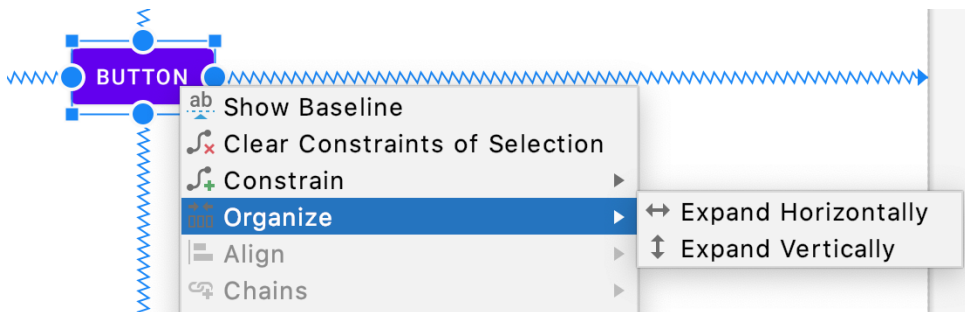


Figure 26-27

## 26.12 Design Time Tools Positioning

The chapter entitled “*A Guide to the Android Studio Layout Editor Tool*” introduced the concept of the *tools* namespace and explained how it can be used to set visibility attributes which only take effect within the layout editor. Behind the scenes, Android Studio also uses tools attributes to hold widgets in position when they are placed on the layout without constraints. Imagine, for example, a Button placed onto the layout while autoconnect mode is disabled. While the widget will appear to be in the correct position within the preview canvas, when the app is run it will appear in the top left-hand corner of the screen. This is because the widget has no constraints to tell the ConstraintLayout parent where to position it.

The reason that the widget appears to be in the correct location in the layout editor is because Android Studio has set absolute X and Y positioning tools attributes to keep it in the correct location until constraints can be added. Within the XML layout file, this might read as follows:

<Button

```
    android:id="@+id/button4"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Button"
    tools:layout_editor_absoluteX="111dp"
    tools:layout_editor_absoluteY="88dp" />
```

Once adequate constraints have been added to the widget, these tools attributes will be removed by the layout editor. A useful technique to quickly identify which widgets lack constraints without waiting until the app runs is to click on the button highlighted in Figure 26-28 to toggle tools position visibility. Any widgets that jump to the top left-hand corner are not fully constrained and are being held in place by temporary tools absolute X and Y positioning attributes.

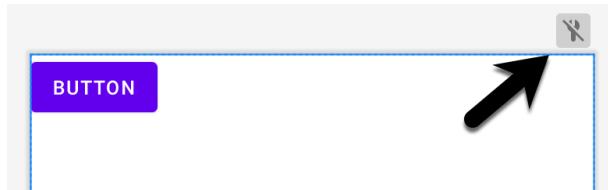


Figure 26-28

### 26.13 Adding Guidelines

Guidelines provide additional elements to which constraints may be anchored. Guidelines are added by right-clicking on the layout and selecting either the *Vertical Guideline* or *Horizontal Guideline* menu option or using the toolbar menu options as shown in Figure 26-29:

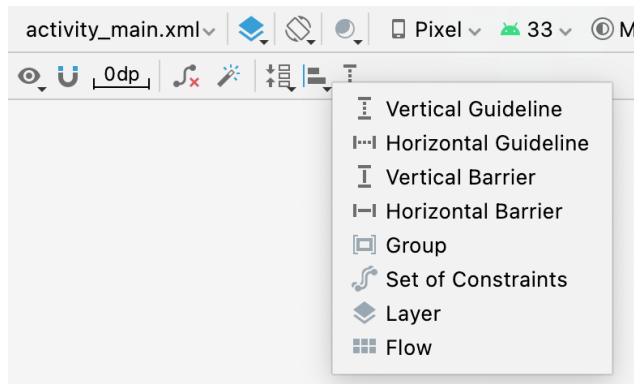


Figure 26-29

Alternatively, horizontal and vertical Guidelines may be dragged from the Helpers section of the Palette and dropped either onto the layout canvas or Component Tree panel:

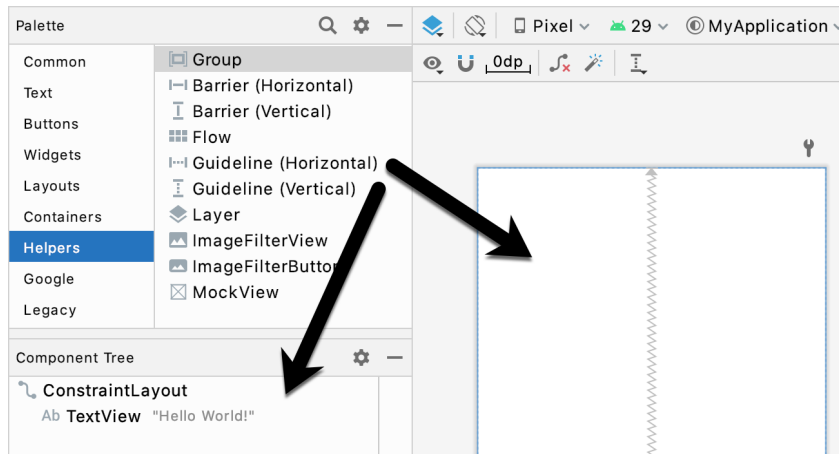


Figure 26-30

Once added, a guideline will appear as a dashed line in the layout and may be moved simply by clicking and dragging the line. To establish a constraint connection to a guideline, click in the constraint handler of a widget and drag to the guideline before releasing. In Figure 26-31, the left sides of two Buttons are connected by constraints to a vertical guideline.

The position of a vertical guideline can be specified as an absolute distance from either the left or the right of the parent layout (or the top or bottom for a horizontal guideline). The vertical guideline in the above figure, for example, is positioned 96dp from the left-hand edge of the parent.

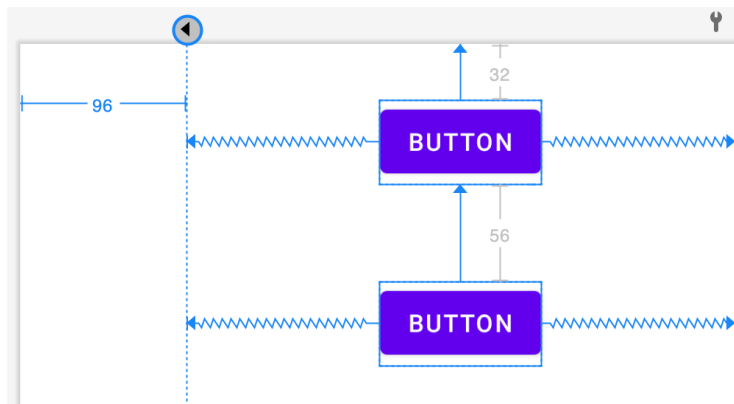


Figure 26-31

Alternatively, the guideline may be positioned as a percentage of the overall width or height of the parent layout. To switch between these three modes, select the guideline and click on the circle at the top or start of the guideline (depending on whether the guideline is vertical or horizontal). Figure 26-32, for example, shows a guideline positioned based on percentage:

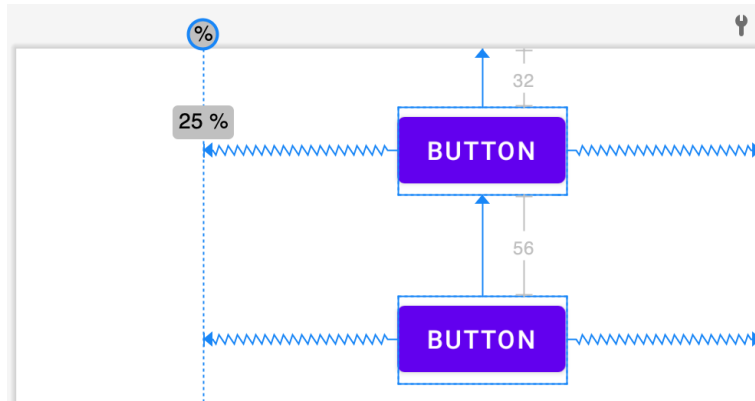


Figure 26-32

## 26.14 Adding Barriers

Barriers are added by right-clicking on the layout and selecting either the *Vertical Barrier* or *Horizontal Barrier* option from the *Add helpers...* menu, or using the toolbar menu options as shown previously in Figure 26-29. Alternatively, locate the Barrier types in the Helpers section of the Palette and drag and drop them either onto the layout canvas or Component Tree panel.

Once a barrier has been added to the layout, it will appear as an entry in the Component Tree panel:

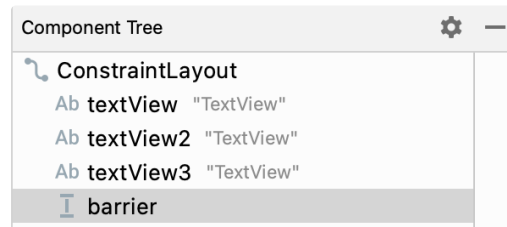


Figure 26-33

To add views as reference views (in other words, the views that control the position of the barrier), simply drag the widgets from within the Component Tree onto the barrier entry. In Figure 26-34, for example, widgets named `textView1` and `textView2` have been assigned as the reference widgets for `barrier1`:

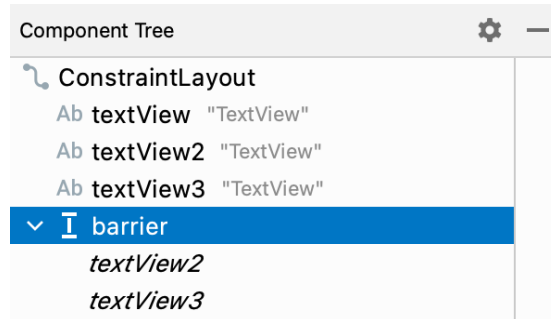


Figure 26-34

After the reference views have been added, the barrier needs to be configured to specify the direction of the barrier in relation those views. This is the *barrier direction* setting and is defined within the Attributes tool window when the barrier is selected in the Component Tree panel:



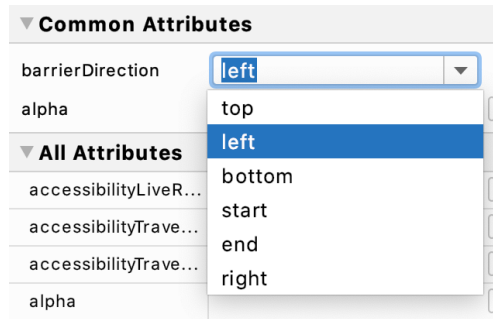


Figure 26-35

The following figure shows a layout containing a barrier declared with `textView1` and `textView2` acting as the reference views and `textView3` as the constrained view. Since the barrier is pushing from the end of the reference views towards the constrained view, the barrier direction has been set to `end`:

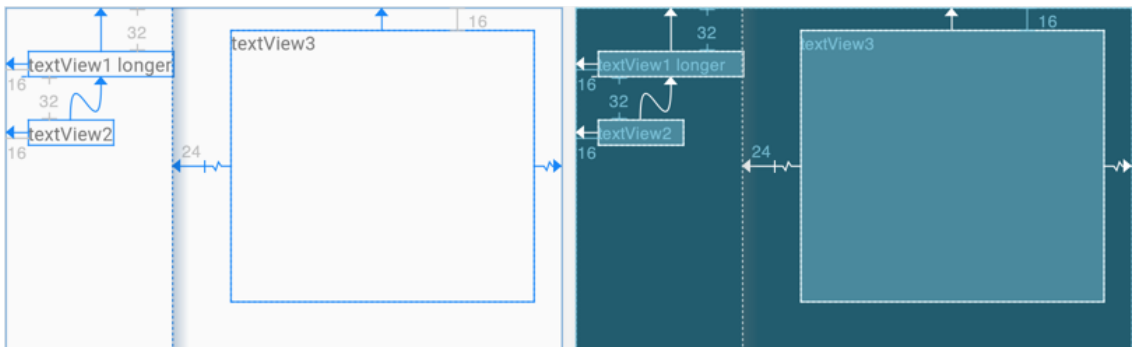


Figure 26-36

## 26.15 Adding a Group

To add a Group to a layout, right-click on the layout and select the *Group* option from the *Add helpers..* menu, or use the toolbar menu options as shown previously in Figure 26-29. Alternatively, locate the Group item in the Helpers section of the Palette and drag and drop it either onto the layout canvas or Component Tree panel.

To add widgets to the group, select them in the Component Tree and drag them onto the Group entry. Figure 26-37 for example, shows three selected widgets being added to a group:

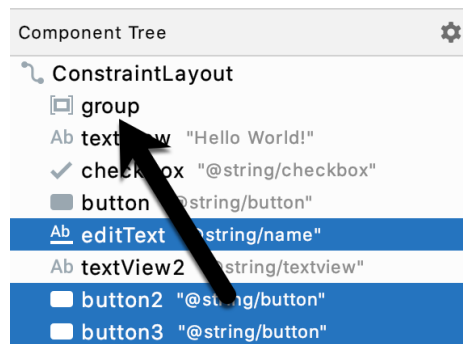


Figure 26-37

Any widgets referenced by the group will appear italicized beneath the group entry in the Component Tree as

shown in Figure 26-38. To remove a widget from the group, simply select it and tap the keyboard delete key:

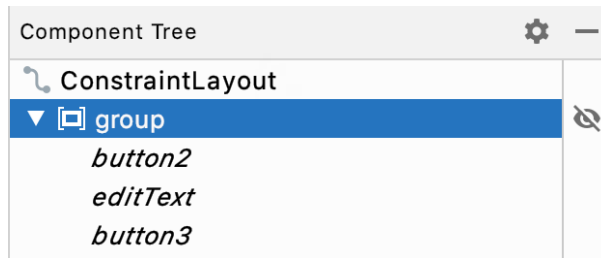


Figure 26-38

Once widgets have been assigned to the group, use the Constraints section of the Attributes tool window to modify the visibility setting:



Figure 26-39

## 26.16 Working with the Flow Helper

Flow helpers may be added using either the menu or Palette as outlined previously for the other helpers. As with the Group helper (Figure 26-37), widgets are added to a Flow instance by dragging them within the Component Tree onto the Flow entry. Having added a Flow helper and assigned widgets to it, select it in the Component Tree and use the Common Attributes section of the Attribute tool window to configure the flow layout behavior:

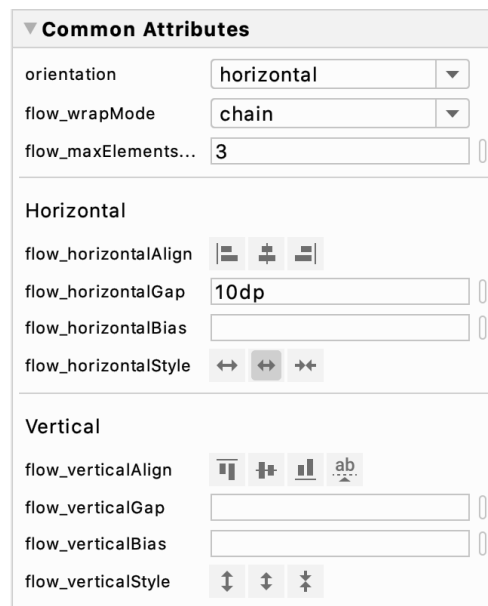


Figure 26-40

## 26.17 Widget Group Alignment and Distribution

The Android Studio Layout Editor tool provides a range of alignment and distribution actions that can be performed when two or more widgets are selected in the layout. Simply shift-click on each of the widgets to be included in the action, right-click on the layout and make a selection from the many options displayed in the Align menu:

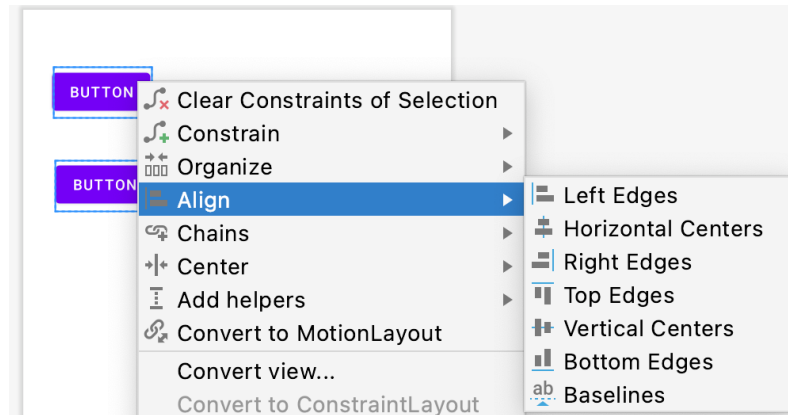


Figure 26-41

As shown in Figure 26-42 below, these options are also accessible via the Align button located in the Layout Editor toolbar:

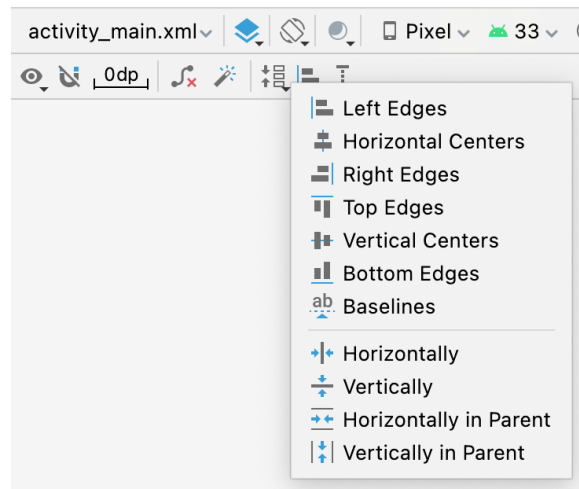


Figure 26-42

Similarly, the Pack menu (Figure 26-43) can be used to collectively reposition the selected widgets so that they are packed tightly together either vertically or horizontally. It achieves this by changing the absolute x and y coordinates of the widgets but does not apply any constraints. The two distribution options in the Pack menu, on the other hand, move the selected widgets so that they are spaced evenly apart in either vertical or horizontal axis and applies constraints between the views to maintain this spacing.

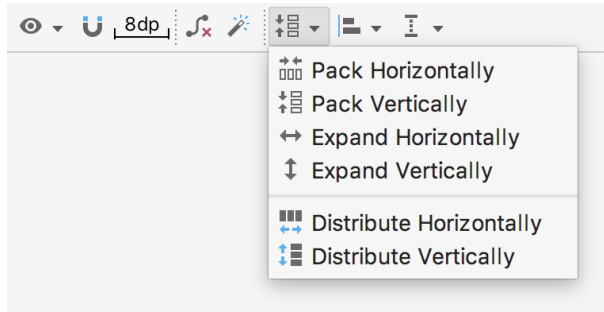


Figure 26-43

## 26.18 Converting other Layouts to ConstraintLayout

For existing user interface layouts that make use of one or more of the other Android layout classes (such as `RelativeLayout` or `LinearLayout`), the Layout Editor tool provides an option to convert the user interface to use the `ConstraintLayout`.

When the Layout Editor tool is open and in Design mode, the Component Tree panel is displayed beneath the Palette. To convert a layout to `ConstraintLayout`, locate it within the Component Tree, right-click on it and select the *Convert <current layout> to Constraint Layout* menu option:

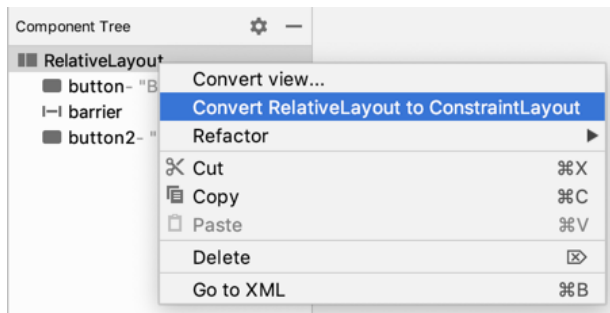


Figure 26-44

When this menu option is selected, Android Studio will convert the selected layout to a `ConstraintLayout` and use inference to establish constraints designed to match the layout behavior of the original layout type.

## 26.19 Summary

A redesigned Layout Editor tool combined with `ConstraintLayout` makes designing complex user interface layouts with Android Studio a relatively fast and intuitive process. This chapter has covered the concepts of constraints, margins and bias in more detail while also exploring the ways in which `ConstraintLayout`-based design has been integrated into the Layout Editor tool.

## 33. An Overview and Example of Android Event Handling

Much has been covered in the previous chapters relating to the design of user interfaces for Android applications. An area that has yet to be covered, however, involves the way in which a user's interaction with the user interface triggers the underlying activity to perform a task. In other words, we know from the previous chapters how to create a user interface containing a button view, but not how to make something happen within the application when it is touched by the user.

The primary objective of this chapter, therefore, is to provide an overview of event handling in Android applications together with an Android Studio based example project.

### 33.1 Understanding Android Events

Events in Android can take a variety of different forms, but are usually generated in response to an external action. The most common form of events, particularly for devices such as tablets and smartphones, involve some form of interaction with the touch screen. Such events fall into the category of *input events*.

The Android framework maintains an *event queue* into which events are placed as they occur. Events are then removed from the queue on a first-in, first-out (FIFO) basis. In the case of an input event such as a touch on the screen, the event is passed to the view positioned at the location on the screen where the touch took place. In addition to the event notification, the view is also passed a range of information (depending on the event type) about the nature of the event such as the coordinates of the point of contact between the user's fingertip and the screen.

To be able to handle the event that it has been passed, the view must have in place an *event listener*. The Android View class, from which all user interface components are derived, contains a range of event listener interfaces, each of which contains an abstract declaration for a callback method. To be able to respond to an event of a particular type, a view must register the appropriate event listener and implement the corresponding callback. For example, if a button is to respond to a *click* event (the equivalent to the user touching and releasing the button view as though clicking on a physical button) it must both register the *View.OnClickListener* event listener (via a call to the target view's *setOnClickListener()* method) and implement the corresponding *onClick()* callback method. If a "click" event is detected on the screen at the location of the button view, the Android framework will call the *onClick()* method of that view when that event is removed from the event queue. It is, of course, within the implementation of the *onClick()* callback method that any tasks should be performed or other methods called in response to the button click.

### 33.2 Using the android:onClick Resource

Before exploring event listeners in more detail it is worth noting that a shortcut is available when all that is required is for a callback method to be called when a user "clicks" on a button view in the user interface. Consider a user interface layout containing a button view named *button1* with the requirement that when the user touches the button, a method called *buttonClick()* declared in the activity class is called. All that is required to implement this behavior is to write the *buttonClick()* method (which takes as an argument a reference to the view that triggered the click event) and add a single line to the declaration of the button view in the XML file. For example:

```
<Button
```

## An Overview and Example of Android Event Handling

```
android:id="@+id/button1"  
android:layout_width="wrap_content"  
android:layout_height="wrap_content"  
android:onClick="buttonClick"  
android:text="Click me" />
```

This provides a simple way to capture click events. It does not, however, provide the range of options offered by event handlers, which are the topic of the rest of this chapter. As will be outlined in later chapters, the `onClick` property also has limitations in layouts involving fragments. When working within Android Studio Layout Editor, the `onClick` property can be found and configured in the Attributes panel when a suitable view type is selected in the device screen layout.

### 33.3 Event Listeners and Callback Methods

In the example activity outlined later in this chapter the steps involved in registering an event listener and implementing the callback method will be covered in detail. Before doing so, however, it is worth taking some time to outline the event listeners that are available in the Android framework and the callback methods associated with each one.

- **onClickListener** – Used to detect click style events whereby the user touches and then releases an area of the device display occupied by a view. Corresponds to the `onClick()` callback method which is passed a reference to the view that received the event as an argument.
- **onLongClickListener** – Used to detect when the user maintains the touch over a view for an extended period. Corresponds to the `onLongClick()` callback method which is passed as an argument the view that received the event.
- **onTouchListener** – Used to detect any form of contact with the touch screen including individual or multiple touches and gesture motions. Corresponding with the `onTouch()` callback, this topic will be covered in greater detail in the chapter entitled “*Android Touch and Multi-touch Event Handling*”. The callback method is passed as arguments the view that received the event and a `MotionEvent` object.
- **onCreateContextMenuListener** – Listens for the creation of a context menu as the result of a long click. Corresponds to the `onCreateContextMenu()` callback method. The callback is passed the menu, the view that received the event and a menu context object.
- **onFocusChangeListener** – Detects when focus moves away from the current view as the result of interaction with a track-ball or navigation key. Corresponds to the `onFocusChange()` callback method which is passed the view that received the event and a Boolean value to indicate whether focus was gained or lost.
- **onKeyListener** – Used to detect when a key on a device is pressed while a view has focus. Corresponds to the `onKey()` callback method. Passed as arguments are the view that received the event, the `KeyCode` of the physical key that was pressed and a `KeyEvent` object.

### 33.4 An Event Handling Example

In the remainder of this chapter, we will work through the creation of an Android Studio project designed to demonstrate the implementation of an event listener and corresponding callback method to detect when the user has clicked on a button. The code within the callback method will update a text view to indicate that the event has been processed.

Select the *New Project* option from the welcome screen and, within the resulting new project dialog, choose the Empty Activity template before clicking on the Next button.

Enter *EventExample* into the Name field and specify *com.ebookfrenzy.eventexample* as the package name. Before

clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Kotlin. Using the steps outlined in section 18.8 *Migrating a Project to View Binding*, convert the project to use view binding.

### 33.5 Designing the User Interface

The user interface layout for the *MainActivity* class in this example is to consist of a *ConstraintLayout*, a *Button* and a *TextView* as illustrated in Figure 33-1.

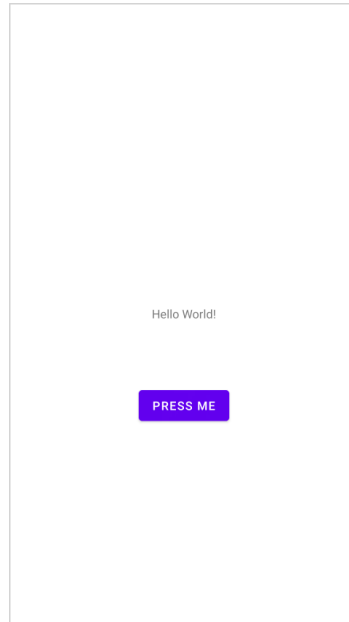


Figure 33-1

Locate and select the *activity\_main.xml* file created by Android Studio (located in the Project tool window under *app -> res -> layouts*) and double-click on it to load it into the Layout Editor tool.

Make sure that Autoconnect is enabled, then drag a *Button* widget from the palette and move it so that it is positioned in the horizontal center of the layout and beneath the existing *TextView* widget. When correctly positioned, drop the widget into place so that appropriate constraints are added by the autoconnect system.

Select the “Hello World!” *TextView* widget and use the Attributes panel to set the ID to *statusText*. Repeat this step to change the ID of the *Button* widget to *myButton*.

Add any missing constraints by clicking on the *Infer Constraints* button in the layout editor toolbar.

With the *Button* widget selected, use the Attributes panel to set the text property to Press Me. Using the yellow warning button located in the top right-hand corner of the Layout Editor (Figure 33-2), display the warnings list and click on the *Fix* button to extract the text string on the button to a resource named *press\_me*:

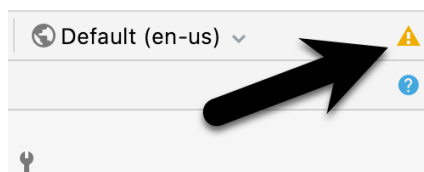


Figure 33-2

With the user interface layout now completed, the next step is to register the event listener and callback method.

### 33.6 The Event Listener and Callback Method

For the purposes of this example, an *onClickListener* needs to be registered for the *myButton* view. This is achieved by making a call to the *setOnClickListener()* method of the button view, passing through a new *onClickListener* object as an argument and implementing the *onClick()* callback method. Since this is a task that only needs to be performed when the activity is created, a good location is the *onCreate()* method of the *MainActivity* class.

If the *MainActivity.kt* file is already open within an editor session, select it by clicking on the tab in the editor panel. Alternatively locate it within the Project tool window by navigating to (*app -> java -> com -> ebookfrenzy -> eventexample -> MainActivity*) and double-click on it to load it into the code editor. Once loaded, locate the template *onCreate()* method and modify it to obtain a reference to the button view, register the event listener and implement the *onClick()* callback method:

```
package com.ebookfrenzy.eventexample

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.View

import com.ebookfrenzy.eventexample.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.myButton.setOnClickListener(object : View.OnClickListener {
            override fun onClick(v: View?) {

            }
        })
    }
}
```

The above code has now registered the event listener on the button and implemented the *onClick()* method. In fact, the code to configure the listener can be made more efficient by using a lambda as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    binding.myButton.setOnClickListener(object : View.OnClickListener {
        override fun onClick(v: View?) {
```



```

—
—————}
—————}}
}

    binding.myButton.setOnClickListener {
    }
}

```

If the application were to be run at this point, however, there would be no indication that the event listener installed on the button was working since there is, as yet, no code implemented within the body of the lambda. The goal for the example is to have a message appear on the `TextView` when the button is clicked, so some further code changes need to be made:

```

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

    binding.myButton.setOnClickListener {
        binding.statusText.text = "Button clicked"
    }
}

```

Complete this phase of the tutorial by compiling and running the application on either an AVD emulator or physical Android device. On touching and releasing the button view (otherwise known as “clicking”) the text view should change to display the “Button clicked” text.

### 33.7 Consuming Events

The detection of standard clicks (as opposed to long clicks) on views is a very simple case of event handling. The example will now be extended to include the detection of long click events which occur when the user clicks and holds a view on the screen and, in doing so, cover the topic of event consumption.

Consider the code for the *onClick* listener code in the above section of this chapter. The lambda code assigned to the listener does not return any value and is not required to do so.

The code assigned to the *onLongClickListener*, on the other hand, is required to return a Boolean value to the Android framework. The purpose of this return value is to indicate to the Android runtime whether or not the callback has *consumed* the event. If the callback returns a *true* value, the event is discarded by the framework. If, on the other hand, the callback returns a *false* value the Android framework will consider the event still to be active and will consequently pass it along to the next matching event listener that is registered on the same view.

As with many programming concepts this is, perhaps, best demonstrated with an example. The first step is to add an event listener for long clicks to the button view in the example activity:

```

override fun onCreate(savedInstanceState: Bundle?) {
    .
    .

    binding.myButton.setOnClickListener {
        binding.statusText.text = "Button clicked"
    }

    binding.myButton.setOnLongClickListener {

```

```
        binding.statusText.text = "Long button click"
        true
    }
}
```

Clearly, when a long click is detected, the lambda code will display “Long button click” on the text view. Note, however, that the callback method also returns a value of *true* to indicate that it has consumed the event. Run the application and press and hold the Button view until the “Long button click” text appears in the text view. On releasing the button, the text view continues to display the “Long button click” text indicating that the *onClick* listener code was not called.

Next, modify the code so that the *onLongClick* listener now returns a *false* value:

```
binding.myButton.setOnLongClickListener {
    statusText.text = "Long button click"
    false
}
```

Once again, compile and run the application and perform a long click on the button until the long click message appears. Upon releasing the button this time, however, note that the *onClick* listener is also triggered and the text changes to “Button clicked”. This is because the *false* value returned by the *onLongClick* listener code indicated to the Android framework that the event was not consumed by the method and was eligible to be passed on to the next registered listener on the view. In this case, the runtime ascertained that the *onClick* listener on the button was also interested in events of this type and subsequently called the *onClick* listener code.

### 33.8 Summary

A user interface is of little practical use if the views it contains do not do anything in response to user interaction. Android bridges the gap between the user interface and the back end code of the application through the concepts of event listeners and callback methods. The Android View class defines a set of event listeners, which can be registered on view objects. Each event listener also has associated with it a callback method.

When an event takes place on a view in a user interface, that event is placed into an event queue and handled on a first in, first out basis by the Android runtime. If the view on which the event took place has registered a listener that matches the type of event, the corresponding callback method or lambda expression is called. This code then performs any tasks required by the activity before returning. Some callback methods are required to return a Boolean value to indicate whether the event needs to be passed on to any other event listeners registered on the view or discarded by the system.

Having covered the basics of event handling, the next chapter will explore in some depth the topic of touch events with a particular emphasis on handling multiple touches.

## 39. Modern Android App Architecture with Jetpack

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

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

### 39.1 What is Android Jetpack?

Android Jetpack consists of Android Studio, the Android Architecture Components and Android Support Library together with a set of guidelines that recommend how an Android App should be structured. The Android Architecture Components are 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 all of the Android Architecture Components will be covered in this book, the objective of this chapter is to introduce the key architectural guidelines together with the ViewModel, LiveData, Lifecycle components while also introducing Data Binding and the use of Repositories.

Before moving on, it is important to understand the Jetpack approach to app development is not mandatory. While highlighting some of the shortcoming of other techniques that have gained popularity of the years, Google stopped short of completely condemning those approaches to app development. Google appears to be taking the position that while there is no right or wrong way to develop an app, there is a recommended way.

### 39.2 The “Old” Architecture

In the chapter entitled *“Creating an Example Android App in Android Studio”*, an Android project was created consisting of a single activity which 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.

### 39.3 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 Google refers to as “separation of concerns”). One of the keys to this

approach is the ViewModel component.

### 39.4 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 actually 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.

In effect, the ViewModel only knows about the data model and corresponding logic. It knows nothing about the user interface and makes no attempt to directly access or respond to events relating to views within the user interface. When a UI controller needs data to display, it simply asks the ViewModel to provide it. Similarly, when the user enters data into a view within the user interface, the UI controller passes it to the ViewModel for handling.

This separation of responsibility addresses the issues relating to the lifecycle of UI controllers. Regardless of how many times a UI controller 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 completely finishes which, in the single activity app, is not until the app exits.

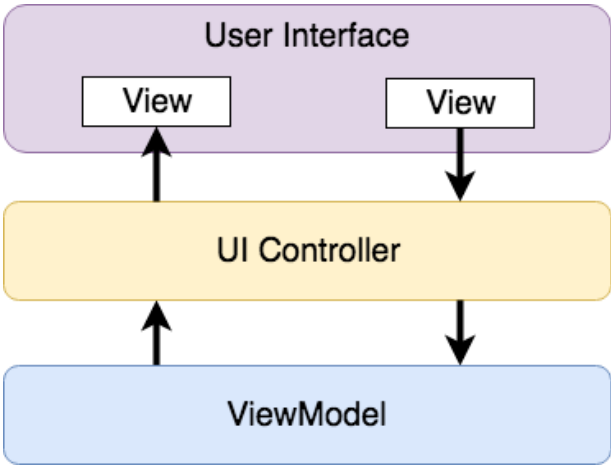


Figure 39-1

### 39.5 The LiveData Component

Consider an app that displays real-time data such as the current price of a financial stock. The app would probably use some form of stock price web service to continuously update the data model within the ViewModel with the latest information. Obviously, this real-time data is of little use unless it is displayed to the user in a timely manner. There are only two ways that the UI controller can ensure that the latest data is displayed in the user interface. One option is for the controller to continuously check with the ViewModel to find out if the data has changed since it was last displayed. The problem with this approach, however, is that it is inefficient. To maintain the real-time nature of the data feed, the UI controller would have to run on a loop, continuously checking for the data to change.

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

This means, for example, that a UI controller that is interested in a ViewModel value can set up an *observer* which will, in turn, be notified when that value changes. In our hypothetical application, for example, the stock price would be wrapped in a LiveData object within the ViewModel and the UI controller would assign an observer to the value, declaring a method to be called when the value changes. This method will, when triggered by data change, read the updated value from the ViewModel and use it to update the user interface.

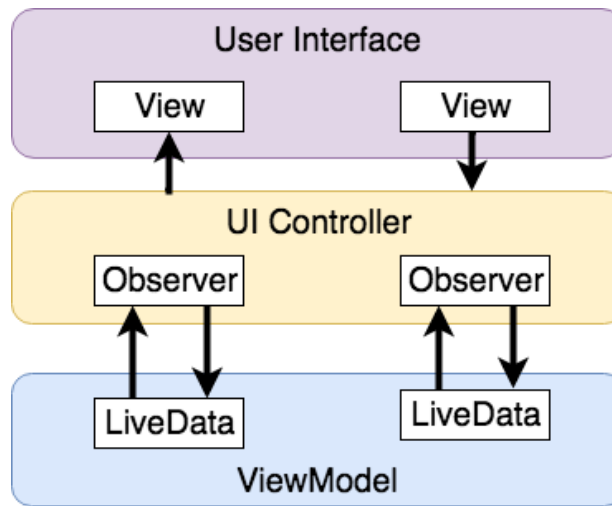


Figure 39-2

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

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

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

## 39.6 ViewModel Saved State

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

### 39.7 LiveData and Data Binding

Android Jetpack includes the Data Binding Library which allows data in a ViewModel to be mapped directly to specific views within the XML user interface layout file. In the AndroidSample project created earlier, code had to be written both to obtain references to the EditText and TextView views and to set and get the text properties to reflect data changes. Data binding allows the LiveData value stored in the ViewModel to be referenced directly within the XML layout file avoiding the need to write code to keep the layout views updated.

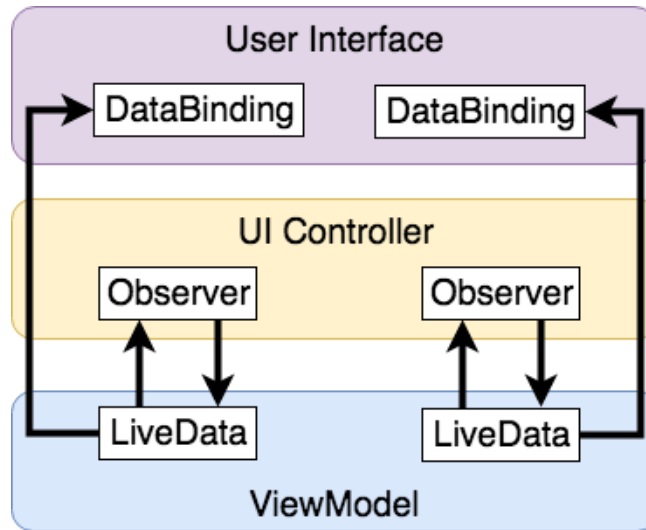


Figure 39-3

Data binding will be covered in greater detail starting with the chapter entitled “*An Overview of Android Jetpack Data Binding*”.

### 39.8 Android Lifecycles

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

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

Objects that are able to detect and react to lifecycle state changes in other objects are said to be *lifecycle-aware*, while objects that provide access to their lifecycle state are called *lifecycle-owners*. Lifecycles will be covered in greater detail in the chapter entitled “*Working with Android Lifecycle-Aware Components*”.

### 39.9 Repository Modules

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

Google's architecture guidelines recommend placing this code in a separate *Repository* module.

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

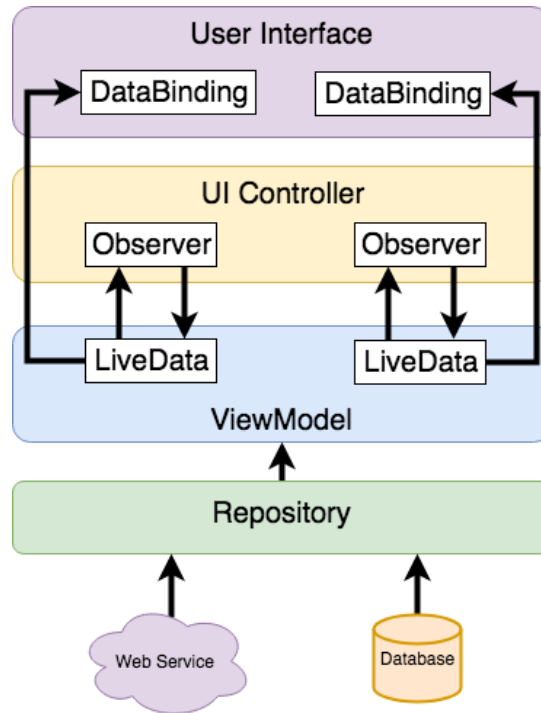


Figure 39-4

### 39.10 Summary

Until recently, Google has tended not to recommend any particular approach to structuring an Android app. That has now changed with the introduction of Android Jetpack which consists of a set of tools, components, libraries and architecture guidelines. Google now recommends 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. In addition, the code responsible for gathering data from data sources such as web services or databases should be built into a separate repository module instead of being bundled with the view model.

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





## 43. An Android Jetpack Data Binding Tutorial

So far in this book we have covered the basic concepts of modern Android app architecture and looked in more detail at the ViewModel and LiveData components. The concept of data binding was also covered in the previous chapter and will now be used in this chapter to further modify the ViewModelDemo app.

### 43.1 Removing the Redundant Code

If you have not already done so, copy the ViewModelDemo project folder and save it as ViewModelDemo\_LiveData so that it can be used again in the next chapter. Once copied, open the original ViewModelDemo project ready to implement data binding.

Before implementing data binding within the ViewModelDemo app, the power of data binding will be demonstrated by deleting all of the code within the project that will no longer be needed by the end of this chapter.

Launch Android Studio, open the ViewModelDemo project, edit the *MainFragment.kt* file and modify the code as follows:

```
package com.ebookfrenzy.viewmodeldemo.ui.main
.
.
import androidx.lifecycle.Observer

class MainFragment : Fragment() {
.
.
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        val resultObserver = Observer<Float>{
            result -> binding.resultText.text = result.toString()
        }

        viewModel.getResult().observe(viewLifecycleOwner, resultObserver)

        binding.convertButton.setOnClickListener {
            if (binding.dollarText.text.isNotEmpty()) {
                viewModel.setAmount(binding.dollarText.text.toString())
            } else {
                binding.resultText.text = "No Value"
            }
    }
}
```

```
    }  
}  
}
```

Next, edit the *MainViewModel.kt* file and continue deleting code as follows (note also the conversion of the *dollarText* variable to *LiveData*):

```
package com.ebookfrenzy.viewmodeldemo.ui.main  
  
import androidx.lifecycle.ViewModel  
import androidx.lifecycle.MutableLiveData  
  
class MainViewModel : ViewModel() {  
  
    private val rate = 0.74f  
    private var dollarText = ""  
    var dollarValue: MutableLiveData<String> = MutableLiveData()  
    private var result: MutableLiveData<Float> = MutableLiveData()  
  
    fun setAmount(value: String) {  
            this.dollarText = value  
            result.setValue(value.toFloat() * rate)  
    }  
  
    fun getResult(): MutableLiveData<Float> {  
            return result  
    }  
}
```

Though we'll be adding a few additional lines of code in the course of implementing data binding, clearly data binding has significantly reduced the amount of code that needed to be written.

### 43.2 Enabling Data Binding

The first step in using data binding is to enable it within the Android Studio project. This involves adding a new property to the *Gradle Scripts* -> *build.gradle* (Module: *ViewModelDemo.app*) file.

Within the *build.gradle* file, add the element shown below to enable data binding within the project and to apply the Kotlin *kapt* plugin. This plugin is required to process the data binding annotations that will be added to the fragment XML layout file later in the chapter:

```
plugins {  
    id 'com.android.application'  
    id 'org.jetbrains.kotlin.android'  
    id 'kotlin-kapt'  
}  
  
android {  
  
    buildFeatures {
```

```

        viewBinding true
        dataBinding true
    }
    .
    .
}

```

Once the entry has been added, a bar will appear across the top of the editor screen containing a *Sync Now* link. Click this to resynchronize the project with the new build configuration settings.

### 43.3 Adding the Layout Element

As described in “*An Overview of Android Jetpack Data Binding*”, to be able to use data binding, the layout hierarchy must have a *layout* component as the root view. This requires that the following changes be made to the *fragment\_main.xml* layout file (*app* -> *res* -> *layout* -> *fragment\_main.xml*). Open this file in the layout editor tool, switch to Code mode and make these changes:

```

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:android="http://schemas.android.com/apk/res/android">

    <androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:id="@+id/main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ui.main.MainFragment">

        .
        .

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>

```

Once these changes have been made, switch back to Design mode and note that the new root view, though invisible in the layout canvas, is now listed in the component tree as shown in Figure 43-1:

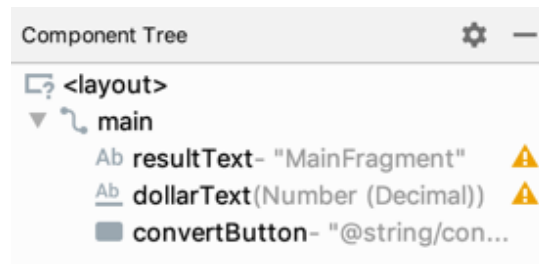


Figure 43-1

Build and run the app to verify that the addition of the layout element has not changed the user interface appearance in any way.

## 43.4 Adding the Data Element to Layout File

The next step in converting the layout file to a data binding layout file is to add the *data* element. For this example, the layout will be bound to `MainViewModel` so edit the `fragment_main.xml` file to add the data element as follows:

```
<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <data>
        <variable
            name="myViewModel"
            type="com.ebookfrenzy.viewmodeldemo.ui.main.MainViewModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ui.main.MainFragment">

        .
        .
    </layout>
```

Build and run the app once again to make sure that these changes take effect.

## 43.5 Working with the Binding Class

The next step is to modify the code within the `MainFragment.kt` file to inflate the data binding. This is best achieved by rewriting the `onCreateView()` method:

```
.
.
import androidx.databinding.DataBindingUtil

import com.ebookfrenzy.viewmodeldemo.R
.
.
class MainFragment : Fragment() {

    private var _binding: FragmentMainBinding? = null
    private val binding get() = _binding!!

    companion object {
        fun newInstance() = MainFragment()
    }
}
```

```

private lateinit var viewModel: MainViewModel
lateinit var binding: FragmentMainBinding

override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?): View {

    _binding = FragmentMainBinding.inflate(inflater, container, false)
    binding = DataBindingUtil.inflate(
        inflater, R.layout.fragment_main, container, false)

    binding.setLifecycleOwner(this)
    return binding.root
}

override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
.
.

```

The old code simply inflated the *fragment\_main.xml* layout file (in other words created the layout containing all of the view objects) and returned a reference to the root view (the top level layout container). The Data Binding Library contains a utility class which provides a special inflation method which, in addition to constructing the UI, also initializes and returns an instance of the layout's data binding class. The new code calls this method and stores a reference to the binding class instance in a variable:

```

binding = DataBindingUtil.inflate(
    inflater, R.layout.fragment_main, container, false)

```

The binding object will only need to remain in memory for as long as the fragment is present. To ensure that the instance is destroyed when the fragment goes away, the current fragment is declared as the lifecycle owner for the binding object.

```

binding.setLifecycleOwner(this)
return binding.getRoot()

```

## 43.6 Assigning the ViewModel Instance to the Data Binding Variable

At this point, the data binding knows that it will be binding to an instance of a class of type `MainViewModel` but has not yet been connected to an actual `MainViewModel` object. This requires the additional step of assigning the `MainViewModel` instance used within the app to the `viewModel` variable declared in the layout file. Add this code to the *onViewCreated()* method in the `MainFragment.kt` file as follows:

```

.
.
import com.ebookfrenzy.viewmodeldemo.BR.myViewModel
.
.
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

```

```

        binding.setVariable(myViewModel, viewModel)
    }
    .
    .

```

If Android Studio reports `myViewModel` as undefined, rebuild the project using the *Build -> Make Project* menu option to force the class to be generated. With these changes made, the next step is to begin inserting some binding expressions into the view elements of the data binding layout file.

## 43.7 Adding Binding Expressions

The first binding expression will bind the `resultText` `TextView` to the `result` value within the model view. Edit the *fragment\_main.xml* file, locate the `resultText` element and modify the `text` property so that the element reads as follows:

```

<TextView
    android:id="@+id/resultText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="MainFragment"
    android:text='@{safeUnbox(myViewModel.result) == 0.0 ? "Enter value" :
String.valueOf(safeUnbox(myViewModel.result)) + " euros"}'
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

```

The expression begins by checking if the `result` value is currently zero and, if it is, displays a message instructing the user to enter a value. If the `result` is not zero, however, the value is converted to a string and concatenated with the word “euros” before being displayed to the user.

The `result` value only requires a one-way binding in that the layout does not ever need to update the value stored in the `ViewModel`. The *dollarValue* `EditText` view, on the other hand, needs to use two-way binding so that the data model can be updated with the latest value entered by the user, and to allow the current value to be redisplayed in the view in the event of a lifecycle event such as that triggered by a device rotation. The *dollarText* element should now be declared as follows:

```

<EditText
    android:id="@+id/dollarText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="96dp"
    android:ems="10"
    android:importantForAutofill="no"
    android:inputType="numberDecimal"
    android:text="@={myViewModel.dollarValue}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.502"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

```

Now that these initial binding expressions have been added a method now needs to be written to perform the

conversion when the user clicks on the Button widget.

## 43.8 Adding the Conversion Method

When the Convert button is clicked, it is going to call a method on the ViewModel to perform the conversion calculation and place the euro value in the *result* LiveData variable. Add this method now within the *MainViewModel.kt* file:

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

    private val rate = 0.74f
    var dollarValue: MutableLiveData<String> = MutableLiveData()
    var result: MutableLiveData<Float> = MutableLiveData()

    fun convertValue() {
        dollarValue.let {
            if (!it.value.equals("")) {
                result.value = it.value?.toFloat()?.times(rate)
            } else {
                result.value = 0f
            }
        }
    }
}
```

Note that in the absence of a valid dollar value, a zero value is assigned to the *result* LiveData variable. This ensures that the binding expression assigned to the *resultText* TextView displays the “Enter value” message if no value has been entered by the user.

## 43.9 Adding a Listener Binding

The final step before testing the project is to add a listener binding expression to the Button element within the layout file to call the *convertValue()* method when the button is clicked. Edit the *fragment\_main.xml* file in Code mode once again, locate the *convertButton* element and add an *onClick* entry as follows:

```
<Button
    android:id="@+id/convertButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="77dp"
    android:onClick="@{ () -> myViewModel.convertValue() }"
    android:text="Convert"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/resultText" />
```

## 43.10 Testing the App

Compile and run the app and test that entering a value into the dollar field and clicking on the Convert button displays the correct result on the TextView (together with the “euros” suffix) and that the “Enter value” prompt

appears if a conversion is attempted while the dollar field is empty. Also, verify that information displayed in the user interface is retained through a device rotation.

### 43.11 Summary

The primary goal of this chapter has been to work through the steps involved in setting up a project to use data binding and to demonstrate the use of one-way, two-way and listener binding expressions. The chapter also provided a practical example of how much code writing is saved by using data binding in conjunction with LiveData to connect the user interface views with the back-end data and logic of the app.



## Index

### Symbols

?. 93  
 <application> 500  
 <fragment> 287  
 <fragment> element 287  
 <receiver> 478  
 <service> 500, 506, 513  
 Code Reformatting 73  
 :: operator 95  
 .well-known folder 451, 474, 670

### A

AbsoluteLayout 168  
 ACCESS\_COARSE\_LOCATION permission 620  
 ACCESS\_FINE\_LOCATION permission 620  
 acknowledgePurchase() method 709  
 ACTION\_DOWN 264  
 ACTION\_MOVE 264  
 ACTION\_POINTER\_DOWN 264  
 ACTION\_POINTER\_UP 264  
 ACTION\_UP 264  
 ACTION\_VIEW 469  
 Active / Running state 144  
 Activity 79, 147  
   adding to a project 225  
   adding views in Java code 245  
   class 147  
   creation 14  
   Entire Lifetime 151  
   Foreground Lifetime 151  
   lifecycle methods 149  
   lifecycles 141  
   returning data from 448  
   state change example 155  
   state changes 147

states 144  
   Visible Lifetime 151  
 ActivityCompat class 625  
 Activity Lifecycle 143  
 Activity Manager 78  
 ActivityResultLauncher 449  
 Activity Stack 143  
 Actual screen pixels 236  
 adb  
   command-line tool 57  
   connection testing 63  
   device pairing 61  
   enabling on Android devices 57  
   Linux configuration 60  
   list devices 57  
   macOS configuration 58  
   overview 57  
   restart server 58  
   testing connection 63  
   WiFi debugging 61  
   Windows configuration 59  
   Wireless debugging 61  
   Wireless pairing 61  
 addCategory() method 477  
 addView() method 240  
 ADD\_VOICEMAIL permission 620  
 android  
   command-line tool 35  
   exported 501  
   gestureColor 280  
   layout\_behavior property 441  
   onClick 289  
   process 501, 513  
   uncertainGestureColor 280  
 Android  
   Activity 79  
   architecture 75  
   events 257  
   intents 80

## Index

- onClick Resource 257
- runtime 76
- SDK Packages 6
- android.app 76
- Android Architecture Components 303
- android.content 76
- android.content.Intent 447
- android.database 76
- Android Debug Bridge. *See* ADB
- Android Design Support Library 411
- Android Development
  - System Requirements 3
- Android Devices
  - designing for different 167
- android.graphics 76
- android.hardware 76
- android.intent.action 483
- android.intent.action.BOOT\_COMPLETED 502
- android.intent.action.MAIN 469
- android.intent.category.LAUNCHER 469
- Android Libraries 76
- AndroidManifest.xml file 226
- android.media 77
- Android Monitor tool window 31
- Android Native Development Kit 77
- android.net 77
- android.opengl 77
- android.os 77
- android.permission.RECORD\_AUDIO 629
- android.print 77
- Android Project
  - create new 13
- android.provider 77
- Android SDK Location
  - identifying 9
- Android SDK Manager 8, 10
- Android SDK Packages
  - version requirements 8
- Android SDK Tools
  - command-line access 9
  - Linux 10
  - macOS 10
  - Windows 7 9
  - Windows 8 9
- Android Software Stack 75
- Android Studio
  - changing theme 54
  - downloading 3
  - Editor Window 49
  - installation 4
  - Linux installation 5
  - macOS installation 4
  - Main Window 48
  - Menu Bar 48
  - Navigation Bar 48
  - Project tool window 49
  - setup wizard 5
  - Status Bar 49
  - Toolbar 48
  - Tool window bars 50
  - tool windows 49
  - updating 11
  - Welcome Screen 47
  - Windows installation 4
- android.text 77
- android.util 77
- android.view 77
- android.view.View 170
- android.view.ViewGroup 167, 170
- Android Virtual Device. *See* AVD
  - overview 27
- Android Virtual Device Manager 27
- android.webkit 77
- android.widget 77
- AndroidX libraries 768
- APK analyzer 702
- APK file 696
  - split 724
- APK File
  - analyzing 702
- APK Signing 768
- APK Wizard dialog 694
- App Architecture
  - modern 303

- AppBar
    - anatomy of 439
  - appbar\_scrolling\_view\_behavior 441
  - App Bundles 691
    - creating 696
    - overview 691
    - revisions 701
    - uploading 698
  - AppCompatActivity class 148
  - App Inspector 51
  - Application
    - stopping 31
  - Application Context 81
  - Application Framework 77
  - Application Manifest 81
  - Application Resources 81
  - App Link
    - Adding Intent Filter 678
    - Assistant 673
    - Digital Asset Links file 670, 451
    - Intent Filter Handling 678
    - Intent Filters 669
    - Intent Handling 670
    - Testing 682
    - tutorial 673
    - URL Mapping 675
  - App Link Assistant 673
  - App Links 669
    - auto verification 450
    - autoVerify 451
    - manually enabling 453
    - overview 669
  - Apply Changes 253
    - Apply Changes and Restart Activity 253
    - Apply Code Changes 253
    - fallback settings 255
    - options 253
    - Run App 253
    - tutorial 255
  - applyToActivitiesIfAvailable() method 759
  - Architecture Components 303
  - ART 76
    - as 95
    - as? 95
  - asFlow() builder 518
  - assetlinks.json , 670, 451
  - asSharedFlow() 528
  - asStateFlow() 527
  - async 487
  - Attribute Keyframes 378
  - Audio
    - supported formats 627
  - Audio Playback 627
  - Audio Recording 627
  - Autoconnect Mode 200
  - Automatic Link Verification 450, 473
  - autoVerify 451, 678
  - AVD
    - command-line creation 27, 35
    - configuration files 37
    - creation 27
    - device frame 34
    - launch in tool window 34
    - overview 27
    - renaming 37
    - running an application 29
    - standalone 33
    - starting 28
    - Startup size and orientation 29
- ## B
- Background Process 142
  - Barriers 194
    - adding 212
    - constrained views 194
  - Base APK file 724
  - Baseline Alignment 193
  - beginTransaction() method 288
  - BillingClient 710
    - acknowledgePurchase() method 709
    - consumeAsync() method 709
    - getPurchaseState() method 708
    - initialization 706, 714
    - launchBillingFlow() method 708

## Index

- queryProductDetailsAsync() method 707
- queryPurchasesAsync() method 710
- startConnection() method 707
- BillingResult 721
  - getDebugMessage() 721
- Binding Expressions 325
  - one-way 325
  - two-way 326
- BIND\_JOB\_SERVICE permission 501
- bindService() method 499, 503, 507
- Biometric Authentication 683
  - callbacks 687
  - overview 683
  - tutorial 683
- Biometric Prompt 688
- Bitwise AND 101
- Bitwise Inversion 100
- Bitwise Left Shift 102
- Bitwise OR 101
- Bitwise Right Shift 102
- Bitwise XOR 101
- black activity 14
- Blank template 171
- Blueprint view 199
- BODY\_SENSORS permission 620
- Boolean 88
- Bound Service 499, 503
  - adding to a project 504
  - Implementing the Binder 504
  - Interaction options 503
- BoundService class 505
- Broadcast Intent 477
  - example 480
  - overview 80, 477
  - sending 480
  - Sticky 479
- Broadcast Receiver 477
  - adding to manifest file 482
  - creation 481
  - overview 80, 478
- BroadcastReceiver class 478
- BroadcastReceiver superclass 481

- buffer() operator 521
- Build tool window 51
- Build Variants 51, 768
  - tool window 51
- Bundle class 164
- Bundled Notifications 550

## C

- Calendar permissions 620
- CALL\_PHONE permission 620
- CAMERA permission 620
- Camera permissions 620
- cancelAndJoin() 488
- cancelChildren() 487
- CancellationSignal 688
- Canvas class 664
- CardView
  - example 431
  - layout file 429
  - responding to selection of 437
- CardView class 429
- C/C++ Libraries 77
- Chain bias 220
- chain head 192
- chains 192
- Chains
  - creation of 217
- Chain style
  - changing 219
- chain styles 192
- Char 88
- CharSequence 165
- CheckBox 167
- checkSelfPermission() method 624
- Code completion 68
- Code Editor
  - basics 65
  - Code completion 68
  - Code Generation 70
  - Code Reformatting 73
  - Document Tabs 66
  - Editing area 66

- Gutter Area 66
- Live Templates 74
- Splitting 67
- Statement Completion 69
- Status Bar 67
- Code Generation 70
- code samples
  - download 1
- Cold flows 526
- CollapsingToolbarLayout
  - example 442
  - introduction 442
  - parallax mode 442
  - pin mode 442
  - setting scrim color 445
  - setting title 445
  - with image 442
- collectLatest() operator 520
- collect() operator 519
- Color class 665
- COLOR\_MODE\_COLOR 640, 660
- COLOR\_MODE\_MONOCHROME 640, 660
- com.android.application 727
- com.android.dynamic-feature 727
- combine() operator 525
- Common Gestures 269
  - detection 269
- Communicating Sequential Processes 485
- Companion Objects 125
- Component tree 17
- Configuration APK file 724
- conflate() operator 520
- Constraint Bias 191
  - adjusting 204
- ConstraintLayout
  - advantages of 197
  - Availability 198
  - Barriers 194
  - Baseline Alignment 193
  - chain bias 220
  - chain head 192
  - chains 192
  - chain styles 192
  - Constraint Bias 191
  - Constraints 189
  - conversion to 216
  - convert to MotionLayout 385
  - deleting constraints 204
  - guidelines 210
  - Guidelines 194
  - manual constraint manipulation 201
  - Margins 190, 205
  - Opposing Constraints 190, 206
  - overview of 189
  - Packed chain 193, 220
  - ratios 197, 221
  - Spread chain 192
  - Spread inside 219
  - Spread inside chain 192
  - tutorial 225
  - using in Android Studio 199
  - Weighted chain 192, 220
  - Widget Dimensions 193, 208
  - Widget Group Alignment 215
- ConstraintLayout chains
  - creation of 217
  - in layout editor 217
- ConstraintLayout Chain style
  - changing 219
- Constraints
  - deleting 204
- ConstraintSet
  - addToHorizontalChain() method 242
  - addToVerticalChain() method 242
  - alignment constraints 241
  - apply to layout 240
  - applyTo() method 240
  - centerHorizontally() method 241
  - centerVertically() method 241
  - chains 241
  - clear() method 242
  - clone() method 241
  - connect() method 240
  - connect to parent 240

## Index

- constraint bias 241
- copying constraints 241
- create 240
- create connection 240
- createHorizontalChain() method 241
- createVerticalChain() method 241
- guidelines 242
- removeFromHorizontalChain() method 242
- removeFromVerticalChain() method 242
- removing constraints 242
- rotation 243
- scaling 242
- setGuidelineBegin() method 242
- setGuidelineEnd() method 242
- setGuidelinePercent() method 242
- setHorizontalBias() method 241
- setRotationX() method 243
- setRotationY() method 243
- setScaleX() method 242
- setScaleY() method 242
- setTransformPivot() method 243
- setTransformPivotX() method 243
- setTransformPivotY() method 243
- setVerticalBias() method 241
- sizing constraints 241
- tutorial 245
- view IDs 247
- ConstraintSet class 239, 240
- ConstraintSet.PARENT\_ID 240
- Constraint Sets 240
- ConstraintSets
  - configuring 374
- consumeAsync() method 709
- ConsumeParams 719
- Contacts permissions 620
- container view 167
- Content Provider 78
  - overview 81
- Context class 81
- CoordinatorLayout 168, 439, 441
- Coroutine Builders 487
  - async 487
  - coroutineScope 487
  - launch 487
  - runBlocking 487
  - supervisorScope 487
  - withContext 487
- Coroutine Dispatchers 486
- Coroutines 485, 517
  - adding libraries 493
  - channel communication 491
  - GlobalScope 486
  - returning results 489
  - Suspend Functions 486
  - suspending 488
  - tutorial 493
  - ViewModelScope 486
  - vs. Threads 485
- coroutineScope 487
- Coroutine Scope 486
- createPrintDocumentAdapter() method 655
- Custom Accessors 123
- Custom Attribute 375
- Custom Document Printing 643, 655
- Custom Gesture
  - recognition 275
- Custom Print Adapter
  - implementation 657
- Custom Print Adapters 655
- Custom Theme
  - building 752
- Cycle Editor 403
- Cycle Keyframe 383
- Cycle Keyframes
  - overview 399

## D

- dangerous permissions 619
  - list of 620
- Dark Theme 32
  - enable on device 32
- Data Access Object (DAO) 568
- Database Inspector 574, 598
  - live updates 598

- SQL query 598
- Database Rows 562
- Database Schema 561
- Database Tables 561
- Data binding
  - binding expressions 325
- Data Binding 306
  - binding classes 324
  - enabling 330
  - event and listener binding 326
  - key components 321
  - overview 321
  - tutorial 329
  - variables 324
  - with LiveData 306
- DDMS 31
- Debugging
  - enabling on device 57
- debug.keystore file 451, 473
- Default Function Parameters 115
- DefaultLifecycleObserver 344, 347
- deltaRelative 379
- Density-independent pixels 235
- Density Independent Pixels
  - converting to pixels 250
- Device Definition
  - custom 184
- Device File Explorer 51
- device frame 34
- Device Manager 51
- device pairing 61
- Digital Asset Links file 451, 670, 451
- Direct Reply Input 558
- Direct Reply Notification 553
- Dispatchers.Default 487
- Dispatchers.IO 487
- Dispatchers.Main 486
- dp 235
- DROP\_LATEST 528
- DROP\_OLDEST 528
- Dynamic Colors
  - applyToActivitiesIfAvailable() method 759

- enabling 758
- enabling in Android 758
- Dynamic Delivery 726
- Dynamic Feature APK 724
- Dynamic Feature Module
  - architecture 723
  - overview 723
  - removal 747
  - tutorial 733
- Dynamic Feature Modules
  - deferred installation 729
  - handling of large 730
- Dynamic Feature Support
  - adding to project 733
- Dynamic State 149
  - saving 163

## E

- Elvis Operator 95
- Empty Process 143
- Empty template 171
- Emulator 51
  - battery 42
  - cellular configuration 42
  - configuring fingerprints 44
  - directional pad 42
  - extended control options 41
  - Extended controls 41
  - fingerprint 42
  - location configuration 42
  - phone settings 42
  - resize 41
  - rotate 40
  - Screen Record 43
  - Snapshots 43
  - starting 28
  - take screenshot 40
  - toolbar 39
  - toolbar options 39
  - tool window mode 45
  - Virtual Sensors 43
  - zoom 40

## Index

- enablePendingPurchases() method 709
- enabling ADB support 57
- Escape Sequences 89
- ettings.gradle file 768
- Event Handling 257
  - example 258
- Event Listener 260
- Event Listeners 258
- Event Log 51
- Events
  - consuming 261
- explicit
  - intent 80
- explicit intent 447
- Explicit Intent 447
- Extended Control
  - options 41

## F

- Favorites
  - tool window 51
- Files
  - switching between 66
- filter() operator 522
- findPointerIndex() method 264
- findViewById() 135
- Fingerprint
  - emulation 44
- Fingerprint authentication
  - device configuration 684
  - permission 684
  - steps to implement 683
- Fingerprint Authentication
  - overview 683
  - tutorial 683
- FLAG\_INCLUDE\_STOPPED\_PACKAGES 477
- flatMapConcat() operator 525
- flatMapMerge() operator 525
- flexible space area 439
- Float 88
- floating action button 14, 172, 411
  - changing appearance of 414

- margins 412
- overview of 411
- removing 173
- sizes 412
- Flow 517
  - asFlow() builder 518
  - asSharedFlow() 528
  - asStateFlow() 527
  - backgroundn handling 537
  - buffering 520
  - buffer() operator 521
  - builder 518
  - cold 526
  - collect() 519
  - collecting data 519
  - collectLatest() operator 520
  - combine() operator 525
  - conflate() operator 520
  - declaring 518
  - emit() 519
  - emitting data 519
  - filter() operator 522
  - flatMapConcat() operator 525
  - flatMapMerge() operator 525
  - flattening 524
  - flowOf() builder 518
  - flow of flows 524
  - fold() operator 524
  - hot 526
  - intermediate operators 522
  - library requirements 518
  - map() operator 522
  - MutableSharedFlow 528
  - MutableStateFlow 527
  - onEach() operator 526
  - reduce() operator 523, 524
  - repeatOnLifecycle 538
  - SharedFlow 528
  - single() operator 520
  - StateFlow 527
  - terminal flow operators 523
  - transform() operator 523



- try/finally 520
- zip() operator 525
- flow builder 518
- flowOf() builder 518
- flow of flows 524
- Flow operators 522
- Flows
  - combining 525
  - Introduction to 517
- Foldable Devices 152
  - multi-resume 152
- Foreground Process 142
- Fragment
  - creation 285
  - event handling 289
  - XML file 285, 286
- FragmentActivity class 148
- Fragment Communication 289
- FragmentPagerAdapter class 423
- Fragments 285
  - adding in code 288
  - duplicating 420
  - example 293
  - overview 285
- FrameLayout 168
- Function Parameters
  - variable number of 115
- Functions 113
- G**
- Gesture Builder Application 275
  - building and running 276
- Gesture Detector class 269
- GestureDetectorCompat 272
  - instance creation 272
- GestureDetectorCompat class 269
- GestureDetector.OnDoubleTapListener 269, 270
- GestureDetector.OnGestureListener 270
- GestureLibrary 275
- GestureLibrary class 275
- GestureOverlayView 275
  - configuring color 280
  - configuring multiple strokes 280
- GestureOverlayView class 275
- GesturePerformedListener 275
- Gestures
  - interception of 281
- Gestures File
  - creation 276
  - extract from SD card 276
  - loading into application 278
- GET\_ACCOUNTS permission 620
- getAction() method 483
- getDebugMessage() 721
- getId() method 240
- getIntent() method 448
- getPointerCount() method 264
- getPointerId() method 264
- getPurchaseState() method 708
- getService() method 507
- GlobalScope 486
- GNU/Linux 76
- Google Cloud Print 638
- Google Drive
  - printing to 638
- Google Play Billing Library 705
- Google Play Console 712
  - Creating an in-app product 712
  - License Testers 713
- Google Play Developer Console 692
- Gradle
  - APK signing settings 772
  - Build Variants 768
  - command line tasks 773
  - dependencies 767
  - Manifest Entries 768
  - overview 767
  - sensible defaults 767
  - tool window 51
- Gradle Build File
  - top level 769
- Gradle Build Files
  - module level 770
- gradle.properties file 768

## Index

GridLayout 168

LayoutManager 427

## H

Handler class 512

Higher-order Functions 117

Hot flows 526

HP Print Services Plugin 637

HTML printing 641

HTML Printing

    example 645

## I

IBinder 499, 505

IBinder object 503, 511, 512

Image Printing 640

Immutable Variables 90

implicit

    intent 80

implicit intent 447

Implicit Intent 449

Implicit Intents

    example 465

in 235

INAPP 710

In-App Products 705

In-App Purchasing 711

    acknowledgePurchase() method 709

    BillingClient 706

    BillingResult 721

    consumeAsync() method 709

    ConsumeParams 719

    Consuming purchases 718

    enablePendingPurchases() method 709

    getPurchaseState() method 708

    Google Play Billing Library 705

    launchBillingFlow() method 708

    Libraries 711

    newBuilder() method 706

    onBillingServiceDisconnected() callback 715

    onBillingServiceDisconnected() method 707

    onBillingSetupFinished() listener 715

    onProductDetailsResponse() callback 716

Overview 705

ProductDetail 708

ProductDetails 716

products 705

ProductType 710

Purchase Flow 717

PurchaseResponseListener 710

PurchasesUpdatedListener 708

PurchaseUpdatedListener 717

purchase updates 717

queryProductDetailsAsync() 716

queryProductDetailsAsync() method 707

queryPurchasesAsync() 719

queryPurchasesAsync() method 710

runOnUiThread() 717

startConnection() method 707

subscriptions 705

tutorial 711

Initializer Blocks 123

In-Memory Database 574

Inner Classes 124

Instant Dynamic Feature Module 724

IntelliJ IDEA 83

Intent 80

    explicit 80

    implicit 80

Intent Availability

    checking for 454

intent filters 447

Intent Filters 450

    App Link 669

intent resolution 450

Intents 447

    ActivityResultLauncher 449

    overview 447

    registerForActivityResult() 449, 462

Intent Service 499

IntentService class 499, 502

Intent URL 468

intermediate flow operators 522

is 95

isInitialized property 95

## J

Java

convert to Kotlin 83

Java Native Interface 77

JetBrains 83

Jetpack 303

overview 303

JobIntentService 499

BIND\_JOB\_SERVICE permission 501

onHandleWork() method 499

join() 488

## K

KeyAttribute 378

Keyboard Shortcuts 52

KeyCycle 383, 399

Cycle Editor 403

tutorial 399

Keyframe 391

Keyframes 378

KeyFrameSet 408

KeyPosition 379

deltaRelative 379

parentRelative 379

pathRelative 380

Keystore File

creation 694

KeyTimeCycle 383, 399

keytool 451

KeyTrigger 382

Killed state 144

Kotlin

accessing class properties 123

and Java 83

arithmetic operators 97

assignment operator 97

augmented assignment operators 98

bitwise operators 100

Boolean 88

break 108

breaking from loops 107

calling class methods 123

Char 88

class declaration 119

class initialization 120

class properties 120

Companion Objects 125

conditional control flow 109

continue labels 108

continue statement 108

control flow 105

convert from Java 83

Custom Accessors 123

data types 87

decrement operator 98

Default Function Parameters 115

defining class methods 120

do ... while loop 107

Elvis Operator 95

equality operators 99

Escape Sequences 89

expression syntax 97

Float 88

Flow 517

for-in statement 105

function calling 114

Functions 113

Higher-order Functions 117

if ... else ... expressions 110

if expressions 109

Immutable Variables 90

increment operator 98

inheritance 129

Initializer Blocks 123

Inner Classes 124

introduction 83

Lambda Expressions 116

let Function 93

Local Functions 114

logical operators 99

looping 105

Mutable Variables 90

## Index

- Not-Null Assertion 93
  - Nullable Type 92
  - Overriding inherited methods 132
  - playground 84
  - Primary Constructor 120
  - properties 123
  - range operator 100
  - Safe Call Operator 92
  - Secondary Constructors 120
  - Single Expression Functions 114
  - String 88
  - subclassing 129
  - Type Annotations 91
  - Type Casting 95
  - Type Checking 95
  - Type Inference 91
  - variable parameters 115
  - when statement 110
  - while loop 106
- ## L
- Lambda Expressions 116
  - lateinit 94
  - Late Initialization 94
  - launch 487
  - launchBillingFlow() method 708
  - launcher activity 226
  - layout\_collapseMode
    - parallax 444
    - pin 444
  - layout\_constraintDimentionRatio 222
  - layout\_constraintHorizontal\_bias 220
  - layout\_constraintVertical\_bias 220
  - layout editor
    - ConstraintLayout chains 217
  - Layout Editor 16, 225
    - Autoconnect Mode 200
    - code mode 178
    - Component Tree 175
    - design mode 175
    - device screen 175
    - example project 225
    - Inference Mode 201
    - palette 175
    - properties panel 176
    - Sample Data 184
    - Setting Properties 179
    - toolbar 176
    - user interface design 227
    - view conversion 183
  - Layout Editor Tool
    - changing orientation 16
    - overview 175
  - Layout Inspector 52
  - Layout Managers 167
  - LayoutResultCallback object 661
  - Layouts 167
  - layout\_scrollFlags
    - enterAlwaysCollapsed mode 441
    - enterAlways mode 441
    - exitUntilCollapsed mode 441
    - scroll mode 441
  - Layout Validation 186
  - let Function 93
  - libc 77
  - License Testers 713
  - Lifecycle
    - awareness 343
    - components 306
    - observers 344
    - owners 343
    - states and events 344
    - tutorial 347
  - Lifecycle-Aware Components 343
  - Lifecycle library 518
  - Lifecycle Methods 149
  - Lifecycle Observer 347
    - creating a 347
  - Lifecycle Owner
    - creating a 349
  - Lifecycles
    - modern 306
  - Lifecycle.State.CREATED 538
  - Lifecycle.State.DESTROYED 538

- Lifecycle.State.INITIALIZED 538
- Lifecycle.State.RESUMED 538
- Lifecycle.State.STARTED 538
- LinearLayout 168
- LinearLayoutManager 427
- LinearLayoutManager layout 435
- Linux Kernel 76
- list devices 57
- LiveData 304, 317
  - adding to ViewModel 317
  - observer 319
  - tutorial 317
- Live Templates 74
- Local Bound Service 503
  - example 503
- Local Functions 114
- Location Manager 78
- Location permission 620
- Logcat
  - tool window 52
- LogCat
  - enabling 158

## M

- MANAGE\_EXTERNAL\_STORAGE 621
  - adb enabling 621
  - testing 621
- Manifest File
  - permissions 469
- map() operator 522
- match\_parent properties 235
- Material design 411
- Material Design 2 749
- Material Design 2 Theming 749
- Material Design 3 749
- Material Design 3 Theming 751
- Material Theme Builder 752
- Material You 752
- measureTimeMillis() function 521
- MediaController
  - adding to VideoView instance 605
- MediaController class 602

- methods 602
- MediaPlayer class 627
  - methods 627
- MediaRecorder class 627
  - methods 628
  - recording audio 628
- Messenger object 512
- Microphone
  - checking for availability 630
- Microphone permissions 620
- mm 235
- MotionEvent 263, 264, 283
  - getActionMasked() 264
- MotionLayout 373
  - arc motion 378
  - Attribute Keyframes 378
  - ConstraintSets 374
  - Custom Attribute 394
  - Custom Attributes 375
  - Cycle Editor 403
  - Cycle Keyframes 383
  - Editor 385
  - KeyAttribute 378
  - KeyCycle 399
  - Keyframes 378
  - KeyFrameSet 408
  - KeyPosition 379
  - KeyTimeCycle 399
  - KeyTrigger 382
  - OnClick 377, 390
  - OnSwipe 377
  - overview 373
  - Position Keyframes 379
  - previewing animation 389
  - starting animation 376
  - Trigger Keyframe 382
  - Tutorial 385
- MotionScene
  - ConstraintSets 374
  - Custom Attributes 375
  - file 374
  - overview 373

## Index

- transition 374
- multiple devices
  - testing app on 31
- Multiple Touches
  - handling 264
- multi-resume 152
- Multi-Touch
  - example 265
- Multi-touch Event Handling 263
- multi-window support 152
- MutableSharedFlow 528
- MutableStateFlow 527
- Mutable Variables 90

## N

- Navigation 353
  - adding destinations 362
  - overview 353
  - pass data with safeargs 369
  - passing arguments 358
  - safeargs 358
  - stack 353
  - tutorial 359
- Navigation Action
  - triggering 357
- Navigation Architecture Component 353
- Navigation Component
  - tutorial 359
- Navigation Controller
  - accessing 357
- Navigation Graph 356, 360
  - adding actions 365
  - creating a 360
- Navigation Host 354
  - declaring 361
- newBuilder() method 706
- normal permissions 619
- Notification
  - adding actions 550
  - direct reply 553
  - Direct Reply Input 558
  - issuing a basic 546

- launch activity from a 548
- PendingIntent 555
- Reply Action 556
- updating direct reply 558
- Notifications 541
  - bundled 550
  - overview 541
- Notifications Manager 78
- Not-Null Assertion 93
- Nullable Type 92

## O

- Observer
  - implementing a LiveData 319
- onAttach() method 290
- onBillingServiceDisconnected() callback 715
- onBillingServiceDisconnected() method 707
- onBillingSetupFinished() listener 715
- onBind() method 500, 503, 511
- onBindViewHolder() method 435
- OnClick 377
- onClickListener 258, 260, 262
- onClick() method 257
- onCreateContextMenuListener 258
- onCreate() method 142, 149, 500
- onCreateView() method 150
- on-demand modules 723
- onDestroy() method 150, 500
- onDoubleTap() method 269
- onDown() method 269
- onEach() operator 526
- onFling() method 269
- onFocusChangeListener 258
- OnFragmentInteractionListener
  - implementation 366
- onGesturePerformed() method 275
- onHandleWork() method 499, 500
- onKeyListener 258
- onLayoutFailed() method 661
- onLayoutFinished() method 661
- onLongClickListener 258, 261
- onLongPress() method 269

- onPageFinished() callback 646
- onPause() method 150
- onProductDetailsResponse() callback 716
- onReceive() method 142, 478, 479, 481
- onRequestPermissionsResult() method 623, 634
- onRestart() method 150
- onRestoreInstanceState() method 150
- onResume() method 142, 150
- onSaveInstanceState() method 150
- onScaleBegin() method 281
- onScaleEnd() method 281
- onScale() method 281
- onScroll() method 269
- OnSeekBarChangeListener 300
- onServiceConnected() method 503, 506, 513
- onServiceDisconnected() method 503, 506, 513
- onShowPress() method 269
- onSingleTapUp() method 269
- onStartCommand() method 500
- onStart() method 150
- onStop() method 150
- onTouchEvent() method 269, 281
- onTouchListener 258, 263
- onTouch() method 263, 264
- onViewCreated() method 150
- onViewStatusRestored() method 150
- OpenJDK 3

## P

- Package Explorer 15
- Package Manager 78
- PackageManager class 630
- PackageManager.FEATURE\_MICROPHONE 630
- PackageManager.PERMISSION\_DENIED 621
- PackageManager.PERMISSION\_GRANTED 621
- Package Name 14
- Packed chain 193, 220
- PageRange 662, 663
- Paint class 665
- parentRelative 379
- parent view 169
- pathRelative 380
- Paused state 144
- PdfDocument 643
- PdfDocument.Page 655, 662
- PendingIntent class 555
- Permission
  - checking for 621
- permissions
  - dangerous 619
  - normal 619
- Persistent State 149
- Phone permissions 620
- Pinch Gesture
  - detection 281
  - example 281
- Pinch Gesture Recognition 275
- Play Core Library 729, 733
- Position Keyframes 379
- Primary Constructor 120
- PrintAttributes 660
- PrintDocumentAdapter 643, 655
- PrintDocumentInfo 660
- Printing
  - color 640
  - monochrome 640
- Printing framework
  - architecture 637
- Printing Framework 637
- Print Job
  - starting 666
- Print Manager 637
- PrintManager service 647
- Problems
  - tool window 52
- PROCESS\_OUTGOING\_CALLS permission 620
- Process States 141
- ProductDetail 708
- ProductDetails 716
- ProductType 710
- Profiler
  - tool window 52
- ProgressBar 167
- proguard-rules.pro file 772

## Index

ProGuard Support 768

Project

tool window 52

Project Name 14

Project tool window 15, 52

pt 235

PurchaseResponseListener 710

PurchasesUpdatedListener 708

PurchaseUpdatedListener 717

putExtra() method 447, 477

px 236

## Q

queryProductDetailsAsync() 716

queryProductDetailsAsync() method 707

queryPurchaseHistoryAsync() method 710

queryPurchasesAsync() 719

queryPurchasesAsync() method 710

Quick Documentation 72

## R

RadioButton 167

Range Operator 100

ratios 221

READ\_CALENDAR permission 620

READ\_CALL\_LOG permission 620

READ\_CONTACTS permission 620

READ\_EXTERNAL\_STORAGE permission 621

READ\_PHONE\_STATE permission 620

READ\_SMS permission 620

RECEIVE\_MMS permission 620

RECEIVE\_SMS permission 620

RECEIVE\_WAP\_PUSH permission 620

Recent Files Navigation 53

RECORD\_AUDIO permission 620

Recording Audio

permission 629

RecyclerView 427

adding to layout file 428

example 431

LayoutManager 427

initializing 435

LayoutManager 427

StaggeredLayoutManager 427

RecyclerView Adapter

creation of 433

RecyclerView.Adapter 428, 433

getItemCount() method 428

onBindViewHolder() method 428

onCreateViewHolder() method 428

RecyclerView.ViewHolder

getAdapterPosition() method 438

reduce() operator 523, 524

registerForActivityResult() 449

registerForActivityResult() method 448, 462

registerReceiver() method 479

RelativeLayout 168

release mode 691

Release Preparation 691

Remote Bound Service 511

client communication 511

implementation 512

manifest file declaration 513

RemoteInput.Builder() method 555

RemoteInput Object 555

Remote Service

launching and binding 513

sending a message 515

repeatOnLifecycle 538

Repository

tutorial 585

Repository Modules 306

requestPermissions() method 623

Resource

string creation 19

Resource File 21

Resource Management 141

Resource Manager 52, 78

result receiver 479

Room

Data Access Object (DAO) 568

entities 568, 569

In-Memory Database 574

Repository 568



- Room Database 568
  - tutorial 585
- Room Database Persistence 567
- Room Persistence Library 565, 567
- root element 167
- root view 169
- Run
  - tool window 52
- runBlocking 487
- runOnUiThread() 717
- Runtime Permission Requests 619

## S

- safeargs 358, 369
- Safe Call Operator 92
- Sample Data 184
- Saved State 305, 337
  - library dependencies 339
- SavedStateHandle 338
  - contains() method 339
  - keys() method 339
  - remove() method 339
- Saved State module 337
- SavedStateViewModelFactory 338
- ScaleGestureDetector class 281
- Scale-independent 236
- SDK Packages 6
- Secondary Constructors 120
- Secure Sockets Layer (SSL) 77
- SeekBar 293
- sendBroadcast() method 477, 479
- sendOrderedBroadcast() method 477, 479
- SEND\_SMS permission 620
- sendStickyBroadcast() method 477
- Sensor permissions 620
- Service
  - anatomy 500
  - launch at system start 502
  - manifest file entry 500
  - overview 80
  - run in separate process 501
- ServiceConnection class 513
- Service Process 142
- Service Restart Options 500
- setAudioEncoder() method 628
- setAudioSource() method 628
- setBackgroundColor() 240
- setContentView() method 239, 245
- setId() method 240
- setOnClickListener() method 257, 260
- setOnDoubleTapListener() method 269, 272
- setOutputFile() method 628
- setOutputFormat() method 628
- setResult() method 449
- setText() method 166
- setTransition() 383
- setVideoSource() method 628
- SHA-256 certificate fingerprint 451
- SharedFlow 528, 531
  - backgroundn handling 537
  - DROP\_LATEST 528
  - DROP\_OLDEST 528
  - in ViewModel 533
  - repeatOnLifecycle 538
  - SUSPEND 529
  - tutorial 531
- shouldOverrideUrlLoading() method 646
- shouldShowRequestPermissionRationale() method 625
- SimpleOnScaleGestureListener 281
- SimpleOnScaleGestureListener class 283
- single() operator 520
- SMS permissions 620
- Snackbar 411, 412, 413
  - overview of 412
- Snapshots
  - emulator 43
- sp 236
- Space class 168
- split APK files 724
- SplitCompatApplication 728
- SplitInstallManager 729
- Spread chain 192
- Spread inside 219
- Spread inside chain 192

## Index

- SQL 562
  - SQLite 561
    - AVD command-line use 563
    - Columns and Data Types 561
    - overview 562
    - Primary keys 562
  - StaggeredGridLayoutManager 427
  - startActivity() method 447
  - startConnection() method 707
  - startForeground() method 142
  - START\_NOT\_STICKY 500
  - START\_REDELIVER\_INTENT 500
  - START\_STICKY 500
  - State
    - restoring 166
  - State Change
    - handling 145
  - StateFlow 527
  - Statement Completion 69
  - status bar 439
  - Sticky Broadcast Intents 479
  - Stopped state 144
  - Storage permissions 621
  - String 88
  - strings.xml file 23
  - Structure
    - tool window 52
  - Structured Query Language 562
  - Structure tool window 52
  - SUBS 710
  - subscriptions 705
  - supervisorScope 487
  - SUSPEND 529
  - Suspend Functions 486
  - Switcher 53
  - synthetic properties 135
  - System Broadcasts 483
  - system requirements 3
  - T**
  - tab bar 439
  - TabLayout 417
    - adding to layout 421
  - app
    - tabGravity property 426
    - tabMode property 426
  - example 418
  - fixed mode 425
  - getCount() method 417
  - getItem() method 417
  - overview 417
  - scrollable mode 425
- TableLayout 168, 577
  - TableRow 577
  - Telephony Manager 78
  - Templates
    - blank vs. empty 171
  - Terminal
    - tool window 52
  - terminal flow operators 523
  - Theme
    - building a custom 752
  - Theming 749
    - Material Theme Builder 752
    - tutorial 755, 761
  - Time Cycle Keyframes 383
  - TODO
    - tool window 52
  - toolbar 439
  - ToolBarListener 290
  - tools
    - layout 287
  - Tool window bars 50
  - Tool windows 49
  - Touch Actions 264
  - Touch Event Listener
    - implementation 265
  - Touch Events
    - intercepting 263
  - Touch handling 263
  - transform() operator 523
  - try/finally 520
  - Type Annotations 91
  - Type Casting 95

Type Checking 95

Type Inference 91

## U

unbindService() method 499

unregisterReceiver() method 479

URL Mapping 675

USB connection issues

    resolving 60

USE\_BIOMETRIC 684

user interface state 149

USE\_SIP permission 620

## V

Video Playback 601

VideoView class 601

    methods 601

    supported formats 601

view bindings 135

    enabling 136

    using 137

View class

    setting properties 246

view conversion 183

ViewGroup 167

View Groups 167

View Hierarchy 169

ViewHolder class 428

    sample implementation 434

ViewModel

    adding LiveData 317

    data access 314

    fragment association 313

    overview 304

    saved state 337

    Saved State 305, 337

    tutorial 309

ViewModelProvider 313

ViewModel Saved State 337

ViewModelScope 486

ViewPager 417, 422

    adapter 422

    adding to layout 421

    example 418

Views 167

    Java creation 239

View System 78

Virtual Device Configuration dialog 28

Virtual Sensors 43

Visible Process 142

## W

WebViewClient 641, 646

WebView view 467

Weighted chain 192, 220

Welcome screen 47

while Loop 106

Widget Dimensions 193

Widget Group Alignment 215

Widgets palette 228

WiFi debugging 61

Wireless debugging 61

Wireless pairing 61

withContext 487, 489

wrap\_content properties 237

WRITE\_CALENDAR permission 620

WRITE\_CALL\_LOG permission 620

WRITE\_CONTACTS permission 620

WRITE\_EXTERNAL\_STORAGE permission 621

## X

XML Layout File

    manual creation 235

    vs. Java Code 239

## Z

zip() operator 525

