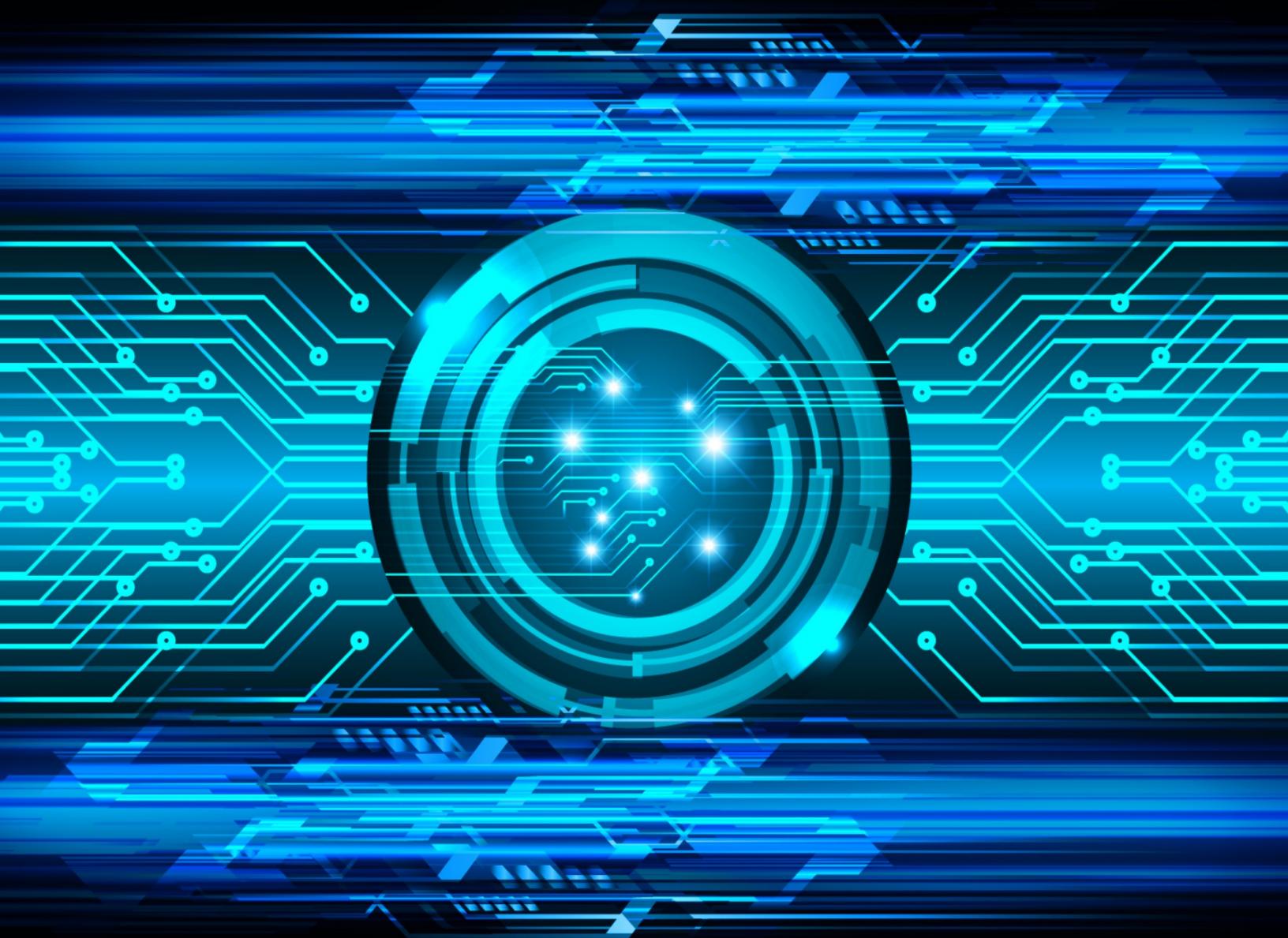


Android Studio 3.0 Development Essentials



Android 8 Edition

Android Studio 3.0

Development Essentials

Android 8 Edition

Android Studio 3.0 Development Essentials – Android 8 Edition

© 2017 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 Firebase Essentials Book Now Available](#)
- [1.3 Feedback](#)
- [1.4 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 7](#)
 - [2.6.2 Windows 8.1](#)
 - [2.6.3 Windows 10](#)
 - [2.6.4 Linux](#)
 - [2.6.5 macOS](#)
- [2.7 Updating Android Studio and the SDK](#)
- [2.8 Summary](#)

3. Creating an Example Android App in Android Studio

- [3.1 Creating a New Android Project](#)
- [3.2 Defining the Project and SDK Settings](#)
- [3.3 Creating an Activity](#)
- [3.4 Modifying the Example Application](#)
- [3.5 Reviewing the Layout and Resource Files](#)
- [3.6 Summary](#)

4. A Tour of the Android Studio User Interface

- [4.1 The Welcome Screen](#)
- [4.2 The Main Window](#)

[4.3 The Tool Windows](#)

[4.4 Android Studio Keyboard Shortcuts](#)

[4.5 Switcher and Recent Files Navigation](#)

[4.6 Changing the Android Studio Theme](#)

[4.7 Summary](#)

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

[5.1 About Android Virtual Devices](#)

[5.2 Creating a New AVD](#)

[5.3 Starting the Emulator](#)

[5.4 Running the Application in the AVD](#)

[5.5 Run/Debug Configurations](#)

[5.6 Stopping a Running Application](#)

[5.7 AVD Command-line Creation](#)

[5.8 Android Virtual Device Configuration Files](#)

[5.9 Moving and Renaming an Android Virtual Device](#)

[5.10 Summary](#)

[6. Using and Configuring the Android Studio AVD Emulator](#)

[6.1 The Emulator Environment](#)

[6.2 The Emulator Toolbar Options](#)

[6.3 Working in Zoom Mode](#)

[6.4 Resizing the Emulator Window](#)

[6.5 Extended Control Options](#)

[6.5.1 Location](#)

[6.5.2 Cellular](#)

[6.5.3 Battery](#)

[6.5.4 Phone](#)

[6.5.5 Directional Pad](#)

[6.5.6 Microphone](#)

[6.5.7 Fingerprint](#)

[6.5.8 Virtual Sensors](#)

[6.5.9 Settings](#)

[6.5.10 Help](#)

[6.6 Drag and Drop Support](#)

[6.7 Configuring Fingerprint Emulation](#)

[6.8 Summary](#)

7. Testing Android Studio Apps on a Physical Android Device

[7.1 An Overview of the Android Debug Bridge \(ADB\)](#)

[7.2 Enabling ADB on Android based Devices](#)

[7.2.1 macOS ADB Configuration](#)

[7.2.2 Windows ADB Configuration](#)

[7.2.3 Linux adb Configuration](#)

[7.3 Testing the adb Connection](#)

[7.4 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 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 Intents](#)

[10.3 Broadcast Intents](#)

- [10.4 Broadcast Receivers](#)
- [10.5 Android Services](#)
- [10.6 Content Providers](#)
- [10.7 The Application Manifest](#)
- [10.8 Application Resources](#)
- [10.9 Application Context](#)
- [10.10 Summary](#)

11. Understanding Android Application and Activity Lifecycles

- [11.1 Android Applications and Resource Management](#)
- [11.2 Android Process States](#)
 - [11.2.1 Foreground Process](#)
 - [11.2.2 Visible Process](#)
 - [11.2.3 Service Process](#)
 - [11.2.4 Background Process](#)
 - [11.2.5 Empty Process](#)
- [11.3 Inter-Process Dependencies](#)
- [11.4 The Activity Lifecycle](#)
- [11.5 The Activity Stack](#)
- [11.6 Activity States](#)
- [11.7 Configuration Changes](#)
- [11.8 Handling State Change](#)
- [11.9 Summary](#)

12. Handling Android Activity State Changes

- [12.1 The Activity Class](#)
- [12.2 Dynamic State vs. Persistent State](#)
- [12.3 The Android Activity Lifecycle Methods](#)
- [12.4 Activity Lifetimes](#)
- [12.5 Disabling Configuration Change Restarts](#)
- [12.6 Summary](#)

13. Android Activity State Changes by Example

- [13.1 Creating the State Change Example Project](#)
- [13.2 Designing the User Interface](#)
- [13.3 Overriding the Activity Lifecycle Methods](#)
- [13.4 Filtering the Logcat Panel](#)
- [13.5 Running the Application](#)

[13.6 Experimenting with the Activity](#)

[13.7 Summary](#)

[14. Saving and Restoring the State of an Android Activity](#)

[14.1 Saving Dynamic State](#)

[14.2 Default Saving of User Interface State](#)

[14.3 The Bundle Class](#)

[14.4 Saving the State](#)

[14.5 Restoring the State](#)

[14.6 Testing the Application](#)

[14.7 Summary](#)

[15. Understanding Android Views, View Groups and Layouts](#)

[15.1 Designing for Different Android Devices](#)

[15.2 Views and View Groups](#)

[15.3 Android Layout Managers](#)

[15.4 The View Hierarchy](#)

[15.5 Creating User Interfaces](#)

[15.6 Summary](#)

[16. A Guide to the Android Studio Layout Editor Tool](#)

[16.1 Basic vs. Empty Activity Templates](#)

[16.2 The Android Studio Layout Editor](#)

[16.3 Design Mode](#)

[16.4 The Palette](#)

[16.5 Pan and Zoom](#)

[16.6 Design and Layout Views](#)

[16.7 Text Mode](#)

[16.8 Setting Attributes](#)

[16.9 Configuring Favorite Attributes](#)

[16.10 Creating a Custom Device Definition](#)

[16.11 Changing the Current Device](#)

[16.12 Summary](#)

[17. A Guide to the Android ConstraintLayout](#)

[17.1 How ConstraintLayout Works](#)

[17.1.1 Constraints](#)

[17.1.2 Margins](#)

[17.1.3 Opposing Constraints](#)
[17.1.4 Constraint Bias](#)
[17.1.5 Chains](#)
[17.1.6 Chain Styles](#)
[17.2 Baseline Alignment](#)
[17.3 Working with Guidelines](#)
[17.4 Configuring Widget Dimensions](#)
[17.5 Working with Barriers](#)
[17.6 Ratios](#)
[17.7 ConstraintLayout Advantages](#)
[17.8 ConstraintLayout Availability](#)
[17.9 Summary](#)

[18. A Guide to using ConstraintLayout in Android Studio](#)

[18.1 Design and Layout Views](#)
[18.2 Autoconnect Mode](#)
[18.3 Inference Mode](#)
[18.4 Manipulating Constraints Manually](#)
[18.5 Adding Constraints in the Inspector](#)
[18.6 Deleting Constraints](#)
[18.7 Adjusting Constraint Bias](#)
[18.8 Understanding ConstraintLayout Margins](#)
[18.9 The Importance of Opposing Constraints and Bias](#)
[18.10 Configuring Widget Dimensions](#)
[18.11 Adding Guidelines](#)
[18.12 Adding Barriers](#)
[18.13 Widget Group Alignment](#)
[18.14 Converting other Layouts to ConstraintLayout](#)
[18.15 Summary](#)

[19. Working with ConstraintLayout Chains and Ratios in Android Studio](#)

[19.1 Creating a Chain](#)
[19.2 Changing the Chain Style](#)
[19.3 Spread Inside Chain Style](#)
[19.4 Packed Chain Style](#)
[19.5 Packed Chain Style with Bias](#)
[19.6 Weighted Chain](#)

[19.7 Working with Ratios](#)

[19.8 Summary](#)

[20. An Android Studio Layout Editor ConstraintLayout Tutorial](#)

[20.1 An Android Studio Layout Editor Tool Example](#)

[20.2 Creating a New Activity](#)

[20.3 Preparing the Layout Editor Environment](#)

[20.4 Adding the Widgets to the User Interface](#)

[20.5 Adding the Constraints](#)

[20.6 Testing the Layout](#)

[20.7 Using the Layout Inspector](#)

[20.8 Summary](#)

[21. Manual XML Layout Design in Android Studio](#)

[21.1 Manually Creating an XML Layout](#)

[21.2 Manual XML vs. Visual Layout Design](#)

[21.3 Summary](#)

[22. Managing Constraints using Constraint Sets](#)

[22.1 Java Code vs. XML Layout Files](#)

[22.2 Creating Views](#)

[22.3 View Attributes](#)

[22.4 Constraint Sets](#)

[22.4.1 Establishing Connections](#)

[22.4.2 Applying Constraints to a Layout](#)

[22.4.3 Parent Constraint Connections](#)

[22.4.4 Sizing Constraints](#)

[22.4.5 Constraint Bias](#)

[22.4.6 Alignment Constraints](#)

[22.4.7 Copying and Applying Constraint Sets](#)

[22.4.8 ConstraintLayout Chains](#)

[22.4.9 Guidelines](#)

[22.4.10 Removing Constraints](#)

[22.4.11 Scaling](#)

[22.4.12 Rotation](#)

[22.5 Summary](#)

[23. An Android ConstraintSet Tutorial](#)

- [23.1 Creating the Example Project in Android Studio](#)
- [23.2 Adding Views to an Activity](#)
- [23.3 Setting View Attributes](#)
- [23.4 Creating View IDs](#)
- [23.5 Configuring the Constraint Set](#)
- [23.6 Adding the EditText View](#)
- [23.7 Converting Density Independent Pixels \(dp\) to Pixels \(px\)](#)
- [23.8 Summary](#)

24. A Guide to using Instant Run in Android Studio

- [24.1 Introducing Instant Run](#)
- [24.2 Understanding Instant Run Swapping Levels](#)
- [24.3 Enabling and Disabling Instant Run](#)
- [24.4 Using Instant Run](#)
- [24.5 An Instant Run Tutorial](#)
- [24.6 Triggering an Instant Run Hot Swap](#)
- [24.7 Triggering an Instant Run Warm Swap](#)
- [24.8 Triggering an Instant Run Cold Swap](#)
- [24.9 The Run Button](#)
- [24.10 Summary](#)

25. An Overview and Example of Android Event Handling

- [25.1 Understanding Android Events](#)
- [25.2 Using the android:onClick Resource](#)
- [25.3 Event Listeners and Callback Methods](#)
- [25.4 An Event Handling Example](#)
- [25.5 Designing the User Interface](#)
- [25.6 The Event Listener and Callback Method](#)
- [25.7 Consuming Events](#)
- [25.8 Summary](#)

26. Android Touch and Multi-touch Event Handling

- [26.1 Intercepting Touch Events](#)
- [26.2 The MotionEvent Object](#)
- [26.3 Understanding Touch Actions](#)
- [26.4 Handling Multiple Touches](#)
- [26.5 An Example Multi-Touch Application](#)
- [26.6 Designing the Activity User Interface](#)

[26.7 Implementing the Touch Event Listener](#)

[26.8 Running the Example Application](#)

[26.9 Summary](#)

[27. Detecting Common Gestures using the Android Gesture Detector Class](#)

[27.1 Implementing Common Gesture Detection](#)

[27.2 Creating an Example Gesture Detection Project](#)

[27.3 Implementing the Listener Class](#)

[27.4 Creating the GestureDetectorCompat Instance](#)

[27.5 Implementing the onTouchEvent\(\) Method](#)

[27.6 Testing the Application](#)

[27.7 Summary](#)

[28. Implementing Custom Gesture and Pinch Recognition on Android](#)

[28.1 The Android Gesture Builder Application](#)

[28.2 The GestureOverlayView Class](#)

[28.3 Detecting Gestures](#)

[28.4 Identifying Specific Gestures](#)

[28.5 Building and Running the Gesture Builder Application](#)

[28.6 Creating a Gestures File](#)

[28.7 Creating the Example Project](#)

[28.8 Extracting the Gestures File from the SD Card](#)

[28.9 Adding the Gestures File to the Project](#)

[28.10 Designing the User Interface](#)

[28.11 Loading the Gestures File](#)

[28.12 Registering the Event Listener](#)

[28.13 Implementing the onGesturePerformed Method](#)

[28.14 Testing the Application](#)

[28.15 Configuring the GestureOverlayView](#)

[28.16 Intercepting Gestures](#)

[28.17 Detecting Pinch Gestures](#)

[28.18 A Pinch Gesture Example Project](#)

[28.19 Summary](#)

[29. An Introduction to Android Fragments](#)

[29.1 What is a Fragment?](#)

[29.2 Creating a Fragment](#)

[29.3 Adding a Fragment to an Activity using the Layout XML File](#)

[29.4 Adding and Managing Fragments in Code](#)

[29.5 Handling Fragment Events](#)

[29.6 Implementing Fragment Communication](#)

[29.7 Summary](#)

[30. Using Fragments in Android Studio - An Example](#)

[30.1 About the Example Fragment Application](#)

[30.2 Creating the Example Project](#)

[30.3 Creating the First Fragment Layout](#)

[30.4 Creating the First Fragment Class](#)

[30.5 Creating the Second Fragment Layout](#)

[30.6 Adding the Fragments to the Activity](#)

[30.7 Making the Toolbar Fragment Talk to the Activity](#)

[30.8 Making the Activity Talk to the Text Fragment](#)

[30.9 Testing the Application](#)

[30.10 Summary](#)

[31. Creating and Managing Overflow Menus on Android](#)

[31.1 The Overflow Menu](#)

[31.2 Creating an Overflow Menu](#)

[31.3 Displaying an Overflow Menu](#)

[31.4 Responding to Menu Item Selections](#)

[31.5 Creating Checkable Item Groups](#)

[31.6 Menus and the Android Studio Menu Editor](#)

[31.7 Creating the Example Project](#)

[31.8 Designing the Menu](#)

[31.9 Modifying the onOptionsItemSelected\(\) Method](#)

[31.10 Testing the Application](#)

[31.11 Summary](#)

[32. Animating User Interfaces with the Android Transitions Framework](#)

[32.1 Introducing Android Transitions and Scenes](#)

[32.2 Using Interpolators with Transitions](#)

[32.3 Working with Scene Transitions](#)

[32.4 Custom Transitions and TransitionSets in Code](#)

[32.5 Custom Transitions and TransitionSets in XML](#)

[32.6 Working with Interpolators](#)

[32.7 Creating a Custom Interpolator](#)

[32.8 Using the beginDelayedTransition Method](#)

[32.9 Summary](#)

[**33. An Android Transition Tutorial using beginDelayedTransition**](#)

[33.1 Creating the Android Studio TransitionDemo Project](#)

[33.2 Preparing the Project Files](#)

[33.3 Implementing beginDelayedTransition Animation](#)

[33.4 Customizing the Transition](#)

[33.5 Summary](#)

[**34. Implementing Android Scene Transitions – A Tutorial**](#)

[34.1 An Overview of the Scene Transition Project](#)

[34.2 Creating the Android Studio SceneTransitions Project](#)

[34.3 Identifying and Preparing the Root Container](#)

[34.4 Designing the First Scene](#)

[34.5 Designing the Second Scene](#)

[34.6 Entering the First Scene](#)

[34.7 Loading Scene 2](#)

[34.8 Implementing the Transitions](#)

[34.9 Adding the Transition File](#)

[34.10 Loading and Using the Transition Set](#)

[34.11 Configuring Additional Transitions](#)

[34.12 Summary](#)

[**35. Working with the Floating Action Button and Snackbar**](#)

[35.1 The Material Design](#)

[35.2 The Design Library](#)

[35.3 The Floating Action Button \(FAB\)](#)

[35.4 The Snackbar](#)

[35.5 Creating the Example Project](#)

[35.6 Reviewing the Project](#)

[35.7 Changing the Floating Action Button](#)

[35.8 Adding the ListView to the Content Layout](#)

[35.9 Adding Items to the ListView](#)

[35.10 Adding an Action to the Snackbar](#)

[35.11 Summary](#)

36. Creating a Tabbed Interface using the TabLayout Component

- [36.1 An Introduction to the ViewPager](#)
- [36.2 An Overview of the TabLayout Component](#)
- [36.3 Creating the TabLayoutDemo Project](#)
- [36.4 Creating the First Fragment](#)
- [36.5 Duplicating the Fragments](#)
- [36.6 Adding the TabLayout and ViewPager](#)
- [36.7 Creating the Pager Adapter](#)
- [36.8 Performing the Initialization Tasks](#)
- [36.9 Testing the Application](#)
- [36.10 Customizing the TabLayout](#)
- [36.11 Displaying Icon Tab Items](#)
- [36.12 Summary](#)

37. Working with the RecyclerView and CardView Widgets

- [37.1 An Overview of the RecyclerView](#)
- [37.2 An Overview of the CardView](#)
- [37.3 Adding the Libraries to the Project](#)
- [37.4 Summary](#)

38. An Android RecyclerView and CardView Tutorial

- [38.1 Creating the CardDemo Project](#)
- [38.2 Removing the Floating Action Button](#)
- [38.3 Adding the RecyclerView and CardView Libraries](#)
- [38.4 Designing the CardView Layout](#)
- [38.5 Adding the RecyclerView](#)
- [38.6 Creating the RecyclerView Adapter](#)
- [38.7 Adding the Image Files](#)
- [38.8 Initializing the RecyclerView Component](#)
- [38.9 Testing the Application](#)
- [38.10 Responding to Card Selections](#)
- [38.11 Summary](#)

39. Working with the AppBar and Collapsing Toolbar Layouts

- [39.1 The Anatomy of an AppBar](#)
- [39.2 The Example Project](#)
- [39.3 Coordinating the RecyclerView and Toolbar](#)

[39.4 Introducing the Collapsing Toolbar Layout](#)

[39.5 Changing the Title and Scrim Color](#)

[39.6 Summary](#)

[40. Implementing an Android Navigation Drawer](#)

[40.1 An Overview of the Navigation Drawer](#)

[40.2 Opening and Closing the Drawer](#)

[40.3 Responding to Drawer Item Selections](#)

[40.4 Using the Navigation Drawer Activity Template](#)

[40.5 Creating the Navigation Drawer Template Project](#)

[40.6 The Template Layout Resource Files](#)

[40.7 The Header Coloring Resource File](#)

[40.8 The Template Menu Resource File](#)

[40.9 The Template Code](#)

[40.10 Running the App](#)

[40.11 Summary](#)

[41. An Android Studio Master/Detail Flow Tutorial](#)

[41.1 The Master/Detail Flow](#)

[41.2 Creating a Master/Detail Flow Activity](#)

[41.3 The Anatomy of the Master/Detail Flow Template](#)

[41.4 Modifying the Master/Detail Flow Template](#)

[41.5 Changing the Content Model](#)

[41.6 Changing the Detail Pane](#)

[41.7 Modifying the WebsiteDetailFragment Class](#)

[41.8 Modifying the WebsiteListActivity Class](#)

[41.9 Adding Manifest Permissions](#)

[41.10 Running the Application](#)

[41.11 Summary](#)

[42. An Overview of Android Intents](#)

[42.1 An Overview of Intents](#)

[42.2 Explicit Intents](#)

[42.3 Returning Data from an Activity](#)

[42.4 Implicit Intents](#)

[42.5 Using Intent Filters](#)

[42.6 Checking Intent Availability](#)

[42.7 Summary](#)

43. Android Explicit Intents – A Worked Example

- [43.1 Creating the Explicit Intent Example Application](#)
- [43.2 Designing the User Interface Layout for ActivityA](#)
- [43.3 Creating the Second Activity Class](#)
- [43.4 Designing the User Interface Layout for ActivityB](#)
- [43.5 Reviewing the Application Manifest File](#)
- [43.6 Creating the Intent](#)
- [43.7 Extracting Intent Data](#)
- [43.8 Launching ActivityB as a Sub-Activity](#)
- [43.9 Returning Data from a Sub-Activity](#)
- [43.10 Testing the Application](#)
- [43.11 Summary](#)

44. Android Implicit Intents – A Worked Example

- [44.1 Creating the Android Studio Implicit Intent Example Project](#)
- [44.2 Designing the User Interface](#)
- [44.3 Creating the Implicit Intent](#)
- [44.4 Adding a Second Matching Activity](#)
- [44.5 Adding the Web View to the UI](#)
- [44.6 Obtaining the Intent URL](#)
- [44.7 Modifying the MyWebView Project Manifest File](#)
- [44.8 Installing the MyWebView Package on a Device](#)
- [44.9 Testing the Application](#)
- [44.10 Summary](#)

45. Android Broadcast Intents and Broadcast Receivers

- [45.1 An Overview of Broadcast Intents](#)
- [45.2 An Overview of Broadcast Receivers](#)
- [45.3 Obtaining Results from a Broadcast](#)
- [45.4 Sticky Broadcast Intents](#)
- [45.5 The Broadcast Intent Example](#)
- [45.6 Creating the Example Application](#)
- [45.7 Creating and Sending the Broadcast Intent](#)
- [45.8 Creating the Broadcast Receiver](#)
- [45.9 Registering the Broadcast Receiver](#)
- [45.10 Testing the Broadcast Example](#)
- [45.11 Listening for System Broadcasts](#)

45.12 Summary

46. A Basic Overview of Threads and AsyncTasks

[46.1 An Overview of Threads](#)

[46.2 The Application Main Thread](#)

[46.3 Thread Handlers](#)

[46.4 A Basic AsyncTask Example](#)

[46.5 Subclassing AsyncTask](#)

[46.6 Testing the App](#)

[46.7 Canceling a Task](#)

[46.8 Summary](#)

47. An Overview of Android Started and Bound Services

[47.1 Started Services](#)

[47.2 Intent Service](#)

[47.3 Bound Service](#)

[47.4 The Anatomy of a Service](#)

[47.5 Controlling Destroyed Service Restart Options](#)

[47.6 Declaring a Service in the Manifest File](#)

[47.7 Starting a Service Running on System Startup](#)

[47.8 Summary](#)

48. Implementing an Android Started Service – A Worked Example

[48.1 Creating the Example Project](#)

[48.2 Creating the Service Class](#)

[48.3 Adding the Service to the Manifest File](#)

[48.4 Starting the Service](#)

[48.5 Testing the IntentService Example](#)

[48.6 Using the Service Class](#)

[48.7 Creating the New Service](#)

[48.8 Modifying the User Interface](#)

[48.9 Running the Application](#)

[48.10 Creating an AsyncTask for Service Tasks](#)

[48.11 Summary](#)

49. Android Local Bound Services – A Worked Example

[49.1 Understanding Bound Services](#)

[49.2 Bound Service Interaction Options](#)

[49.3 An Android Studio Local Bound Service Example](#)

[49.4 Adding a Bound Service to the Project](#)

[49.5 Implementing the Binder](#)

[49.6 Binding the Client to the Service](#)

[49.7 Completing the Example](#)

[49.8 Testing the Application](#)

[49.9 Summary](#)

[50. Android Remote Bound Services – A Worked Example](#)

[50.1 Client to Remote Service Communication](#)

[50.2 Creating the Example Application](#)

[50.3 Designing the User Interface](#)

[50.4 Implementing the Remote Bound Service](#)

[50.5 Configuring a Remote Service in the Manifest File](#)

[50.6 Launching and Binding to the Remote Service](#)

[50.7 Sending a Message to the Remote Service](#)

[50.8 Summary](#)

[51. An Android 8 Notifications Tutorial](#)

[51.1 An Overview of Notifications](#)

[51.2 Creating the NotifyDemo Project](#)

[51.3 Designing the User Interface](#)

[51.4 Creating the Second Activity](#)

[51.5 Creating a Notification Channel](#)

[51.6 Creating and Issuing a Basic Notification](#)

[51.7 Launching an Activity from a Notification](#)

[51.8 Adding Actions to a Notification](#)

[51.9 Bundled Notifications](#)

[51.10 Summary](#)

[52. An Android 8 Direct Reply Notification Tutorial](#)

[52.1 Creating the DirectReply Project](#)

[52.2 Designing the User Interface](#)

[52.3 Creating the Notification Channel](#)

[52.4 Building the RemoteInput Object](#)

[52.5 Creating the PendingIntent](#)

[52.6 Creating the Reply Action](#)

[52.7 Receiving Direct Reply Input](#)

[52.8 Updating the Notification](#)

[52.9 Summary](#)

[53. An Introduction to Android Multi-Window Support](#)

[53.1 Split-Screen, Freeform and Picture-in-Picture Modes](#)

[53.2 Entering Multi-Window Mode](#)

[53.3 Enabling Freeform Support](#)

[53.4 Checking for Freeform Support](#)

[53.5 Enabling Multi-Window Support in an App](#)

[53.6 Specifying Multi-Window Attributes](#)

[53.7 Detecting Multi-Window Mode in an Activity](#)

[53.8 Receiving Multi-Window Notifications](#)

[53.9 Launching an Activity in Multi-Window Mode](#)

[53.10 Configuring Freeform Activity Size and Position](#)

[53.11 Summary](#)

[54. An Android Studio Multi-Window Split-Screen and Freeform Tutorial](#)

[54.1 Creating the Multi-Window Project](#)

[54.2 Designing the FirstActivity User Interface](#)

[54.3 Adding the Second Activity](#)

[54.4 Launching the Second Activity](#)

[54.5 Enabling Multi-Window Mode](#)

[54.6 Testing Multi-Window Support](#)

[54.7 Launching the Second Activity in a Different Window](#)

[54.8 Summary](#)

[55. An Overview of Android SQLite Databases](#)

[55.1 Understanding Database Tables](#)

[55.2 Introducing Database Schema](#)

[55.3 Columns and Data Types](#)

[55.4 Database Rows](#)

[55.5 Introducing Primary Keys](#)

[55.6 What is SQLite?](#)

[55.7 Structured Query Language \(SQL\)](#)

[55.8 Trying SQLite on an Android Virtual Device \(AVD\)](#)

[55.9 Android SQLite Classes](#)

[55.9.1 Cursor](#)

[55.9.2 SQLiteDatabase](#)

[55.9.3 SQLiteOpenHelper](#)

[55.9.4 ContentValues](#)

[55.10 Summary](#)

[56. An Android TableLayout and TableRow Tutorial](#)

[56.1 The TableLayout and TableRow Layout Views](#)

[56.2 Creating the Database Project](#)

[56.3 Adding the TableLayout to the User Interface](#)

[56.4 Configuring the TableRows](#)

[56.5 Adding the Button Bar to the Layout](#)

[56.6 Adjusting the Layout Margins](#)

[56.7 Summary](#)

[57. An Android SQLite Database Tutorial](#)

[57.1 About the Database Example](#)

[57.2 Creating the Data Model](#)

[57.3 Implementing the Data Handler](#)

[57.3.1 The Add Handler Method](#)

[57.3.2 The Query Handler Method](#)

[57.3.3 The Delete Handler Method](#)

[57.4 Implementing the Activity Event Methods](#)

[57.5 Testing the Application](#)

[57.6 Summary](#)

[58. Understanding Android Content Providers](#)

[58.1 What is a Content Provider?](#)

[58.2 The Content Provider](#)

[58.2.1 onCreate\(\)](#)

[58.2.2 query\(\)](#)

[58.2.3 insert\(\)](#)

[58.2.4 update\(\)](#)

[58.2.5 delete\(\)](#)

[58.2.6 getType\(\)](#)

[58.3 The Content URI](#)

[58.4 The Content Resolver](#)

[58.5 The <provider> Manifest Element](#)

[58.6 Summary](#)

59. Implementing an Android Content Provider in Android Studio

- [59.1 Copying the Database Project](#)
- [59.2 Adding the Content Provider Package](#)
- [59.3 Creating the Content Provider Class](#)
- [59.4 Constructing the Authority and Content URI](#)
- [59.5 Implementing URI Matching in the Content Provider](#)
- [59.6 Implementing the Content Provider onCreate\(\) Method](#)
- [59.7 Implementing the Content Provider insert\(\) Method](#)
- [59.8 Implementing the Content Provider query\(\) Method](#)
- [59.9 Implementing the Content Provider update\(\) Method](#)
- [59.10 Implementing the Content Provider delete\(\) Method](#)
- [59.11 Declaring the Content Provider in the Manifest File](#)
- [59.12 Modifying the Database Handler](#)
- [59.13 Summary](#)

60. Accessing Cloud Storage using the Android Storage Access Framework

- [60.1 The Storage Access Framework](#)
- [60.2 Working with the Storage Access Framework](#)
- [60.3 Filtering Picker File Listings](#)
- [60.4 Handling Intent Results](#)
- [60.5 Reading the Content of a File](#)
- [60.6 Writing Content to a File](#)
- [60.7 Deleting a File](#)
- [60.8 Gaining Persistent Access to a File](#)
- [60.9 Summary](#)

61. An Android Storage Access Framework Example

- [61.1 About the Storage Access Framework Example](#)
- [61.2 Creating the Storage Access Framework Example](#)
- [61.3 Designing the User Interface](#)
- [61.4 Declaring Request Codes](#)
- [61.5 Creating a New Storage File](#)
- [61.6 The onActivityResult\(\) Method](#)
- [61.7 Saving to a Storage File](#)
- [61.8 Opening and Reading a Storage File](#)
- [61.9 Testing the Storage Access Application](#)
- [61.10 Summary](#)

62. Implementing Video Playback on Android using the VideoView and MediaController Classes

- [62.1 Introducing the Android VideoView Class](#)
- [62.2 Introducing the Android MediaController Class](#)
- [62.3 Creating the Video Playback Example](#)
- [62.4 Designing the VideoPlayer Layout](#)
- [62.5 Configuring the VideoView](#)
- [62.6 Adding Internet Permission](#)
- [62.7 Adding the MediaController to the Video View](#)
- [62.8 Setting up the onPreparedListener](#)
- [62.9 Summary](#)

63. Android Picture-in-Picture Mode

- [63.1 Picture-in-Picture Features](#)
- [63.2 Enabling Picture-in-Picture Mode](#)
- [63.3 Configuring Picture-in-Picture Parameters](#)
- [63.4 Entering Picture-in-Picture Mode](#)
- [63.5 Detecting Picture-in-Picture Mode Changes](#)
- [63.6 Adding Picture-in-Picture Actions](#)
- [63.7 Summary](#)

64. An Android Picture-in-Picture Tutorial

- [64.1 Changing the Minimum SDK Setting](#)
- [64.2 Adding Picture-in-Picture Support to the Manifest](#)
- [64.3 Adding a Picture-in-Picture Button](#)
- [64.4 Entering Picture-in-Picture Mode](#)
- [64.5 Detecting Picture-in-Picture Mode Changes](#)
- [64.6 Adding a Broadcast Receiver](#)
- [64.7 Adding the PiP Action](#)
- [64.8 Testing the Picture-in-Picture Action](#)
- [64.9 Summary](#)

65. Video Recording and Image Capture on Android using Camera Intents

- [65.1 Checking for Camera Support](#)
- [65.2 Calling the Video Capture Intent](#)
- [65.3 Calling the Image Capture Intent](#)
- [65.4 Creating an Android Studio Video Recording Project](#)

[65.5 Designing the User Interface Layout](#)
[65.6 Checking for the Camera](#)
[65.7 Launching the Video Capture Intent](#)
[65.8 Handling the Intent Return](#)
[65.9 Testing the Application](#)
[65.10 Summary](#)

[66. Making Runtime Permission Requests in Android](#)

[66.1 Understanding Normal and Dangerous Permissions](#)
[66.2 Creating the Permissions Example Project](#)
[66.3 Checking for a Permission](#)
[66.4 Requesting Permission at Runtime](#)
[66.5 Providing a Rationale for the Permission Request](#)
[66.6 Testing the Permissions App](#)
[66.7 Summary](#)

[67. Android Audio Recording and Playback using MediaPlayer and MediaRecorder](#)

[67.1 Playing Audio](#)
[67.2 Recording Audio and Video using the MediaRecorder Class](#)
[67.3 About the Example Project](#)
[67.4 Creating the AudioApp Project](#)
[67.5 Designing the User Interface](#)
[67.6 Checking for Microphone Availability](#)
[67.7 Performing the Activity Initialization](#)
[67.8 Implementing the recordAudio\(\) Method](#)
[67.9 Implementing the stopAudio\(\) Method](#)
[67.10 Implementing the playAudio\(\) method](#)
[67.11 Configuring and Requesting Permissions](#)
[67.12 Testing the Application](#)
[67.13 Summary](#)

[68. Working with the Google Maps Android API in Android Studio](#)

[68.1 The Elements of the Google Maps Android API](#)
[68.2 Creating the Google Maps Project](#)
[68.3 Obtaining Your Developer Signature](#)
[68.4 Testing the Application](#)
[68.5 Understanding Geocoding and Reverse Geocoding](#)

[68.6 Adding a Map to an Application](#)
[68.7 Requesting Current Location Permission](#)
[68.8 Displaying the User's Current Location](#)
[68.9 Changing the Map Type](#)
[68.10 Displaying Map Controls to the User](#)
[68.11 Handling Map Gesture Interaction](#)
 [68.11.1 Map Zooming Gestures](#)
 [68.11.2 Map Scrolling/Panning Gestures](#)
 [68.11.3 Map Tilt Gestures](#)
 [68.11.4 Map Rotation Gestures](#)
[68.12 Creating Map Markers](#)
[68.13 Controlling the Map Camera](#)
[68.14 Summary](#)

[69. Printing with the Android Printing Framework](#)

[69.1 The Android Printing Architecture](#)
[69.2 The Print Service Plugins](#)
[69.3 Google Cloud Print](#)
[69.4 Printing to Google Drive](#)
[69.5 Save as PDF](#)
[69.6 Printing from Android Devices](#)
[69.7 Options for Building Print Support into Android Apps](#)
 [69.7.1 Image Printing](#)
 [69.7.2 Creating and Printing HTML Content](#)
 [69.7.3 Printing a Web Page](#)
 [69.7.4 Printing a Custom Document](#)
[69.8 Summary](#)

[70. An Android HTML and Web Content Printing Example](#)

[70.1 Creating the HTML Printing Example Application](#)
[70.2 Printing Dynamic HTML Content](#)
[70.3 Creating the Web Page Printing Example](#)
[70.4 Removing the Floating Action Button](#)
[70.5 Designing the User Interface Layout](#)
[70.6 Loading the Web Page into the WebView](#)
[70.7 Adding the Print Menu Option](#)
[70.8 Summary](#)

71. A Guide to Android Custom Document Printing

- [71.1 An Overview of Android Custom Document Printing](#)
- [71.1.1 Custom Print Adapters](#)
- [71.2 Preparing the Custom Document Printing Project](#)
- [71.3 Creating the Custom Print Adapter](#)
- [71.4 Implementing the onLayout\(\) Callback Method](#)
- [71.5 Implementing the onWrite\(\) Callback Method](#)
- [71.6 Checking a Page is in Range](#)
- [71.7 Drawing the Content on the Page Canvas](#)
- [71.8 Starting the Print Job](#)
- [71.9 Testing the Application](#)
- [71.10 Summary](#)

72. An Introduction to Android App Links

- [72.1 An Overview of Android App Links](#)
- [72.2 App Link Intent Filters](#)
- [72.3 Handling App Link Intents](#)
- [72.4 Associating the App with a Website](#)
- [72.5 Summary](#)

73. An Android Studio App Links Tutorial

- [73.1 About the Example App](#)
- [73.2 The Database Schema](#)
- [73.3 Loading and Running the Project](#)
- [73.4 Adding the URL Mapping](#)
- [73.5 Adding the Intent Filter](#)
- [73.6 Adding Intent Handling Code](#)
- [73.7 Testing the App Link](#)
- [73.8 Associating an App Link with a Web Site](#)
- [73.9 Summary](#)

74. An Introduction to Android Instant Apps

- [74.1 An Overview of Android Instant Apps](#)
- [74.2 Instant App Feature Modules](#)
- [74.3 Instant App Project Structure](#)
- [74.4 The Application and Feature Build Plugins](#)
- [74.5 Installing the Instant Apps Development SDK](#)

74.6 Summary

75. An Android Instant App Tutorial

- 75.1 Creating the Instant App Project
- 75.2 Reviewing the Project
- 75.3 Testing the Installable App
- 75.4 Testing the Instant App
- 75.5 Reviewing the Instant App APK Files
- 75.6 Summary

76. Adapting an Android Studio Project for Instant Apps

- 76.1 Getting Started
- 76.2 Adding the Application APK Module
- 76.3 Adding an Instant App Module
- 76.4 Testing the Instant App
- 76.5 Summary

77. Creating Multi-Feature Instant Apps

- 77.1 Adding the Second App Link
- 77.2 Creating a Second Feature Module
- 77.3 Moving the Landmark Activity to the New Feature Module
- 77.4 Testing the Multi-Feature Project
- 77.5 Summary

78. A Guide to the Android Studio Profiler

- 78.1 Accessing the Android Profiler
- 78.2 Enabling Advanced Profiling
- 78.3 The Android Profiler Tool Window
- 78.4 The CPU Profiler
- 78.5 Memory Profiler
- 78.6 Network Profiler
- 78.7 Summary

79. An Android Fingerprint Authentication Tutorial

- 79.1 An Overview of Fingerprint Authentication
- 79.2 Creating the Fingerprint Authentication Project
- 79.3 Configuring Device Fingerprint Authentication
- 79.4 Adding the Fingerprint Permission to the Manifest File
- 79.5 Adding the Fingerprint Icon

- [79.6 Designing the User Interface](#)
- [79.7 Accessing the Keyguard and Fingerprint Manager Services](#)
- [79.8 Checking the Security Settings](#)
- [79.9 Accessing the Android Keystore and KeyGenerator](#)
- [79.10 Generating the Key](#)
- [79.11 Initializing the Cipher](#)
- [79.12 Creating the CryptoObject Instance](#)
- [79.13 Implementing the Fingerprint Authentication Handler Class](#)
- [79.14 Testing the Project](#)
- [79.15 Summary](#)

80. Handling Different Android Devices and Displays

- [80.1 Handling Different Device Displays](#)
- [80.2 Creating a Layout for each Display Size](#)
- [80.3 Creating Layout Variants in Android Studio](#)
- [80.4 Providing Different Images](#)
- [80.5 Checking for Hardware Support](#)
- [80.6 Providing Device Specific Application Binaries](#)
- [80.7 Summary](#)

81. Signing and Preparing an Android Application for Release

- [81.1 The Release Preparation Process](#)
- [81.2 Register for a Google Play Developer Console Account](#)
- [81.3 Configuring the App in the Console](#)
- [81.4 Enabling Google Play App Signing](#)
- [81.5 Changing the Build Variant](#)
- [81.6 Enabling ProGuard](#)
- [81.7 Creating a Keystore File](#)
- [81.8 Creating the Application APK File](#)
- [81.9 Uploading New APK Versions to the Google Play Developer Console](#)
- [81.10 Managing Testers](#)
- [81.11 Uploading Instant App APK Files](#)
- [81.12 Uploading New APK Revisions](#)
- [81.13 Analyzing the APK File](#)
- [81.14 Enabling Google Play Signing for an Existing App](#)
- [81.15 Summary](#)

82. An Overview of Gradle in Android Studio

[82.1 An Overview of Gradle](#)

[82.2 Gradle and Android Studio](#)

[82.2.1 Sensible Defaults](#)

[82.2.2 Dependencies](#)

[82.2.3 Build Variants](#)

[82.2.4 Manifest Entries](#)

[82.2.5 APK Signing](#)

[82.2.6 ProGuard Support](#)

[82.3 The Top-level Gradle Build File](#)

[82.4 Module Level Gradle Build Files](#)

[82.5 Configuring Signing Settings in the Build File](#)

[82.6 Running Gradle Tasks from the Command-line](#)

[82.7 Summary](#)

1. Introduction

Fully updated for Android Studio 3.0 and Android 8, the goal of this book is to teach the skills necessary to develop Android based applications using the Android Studio Integrated Development Environment (IDE), the Android 8 Software Development Kit (SDK) and the Java programming language.

Beginning with the basics, this book provides an outline of the steps necessary to set up an Android development and testing environment. 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. More advanced topics such as database management, content providers and intents are also covered, as are touch screen handling, gesture recognition, camera access and the playback and recording of both video and audio. This edition of the book also covers printing, transitions and cloud-based file storage.

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.

In addition to covering general Android development techniques, the book also includes Google Play specific topics such as implementing maps using the Google Maps Android API, and submitting apps to the Google Play Developer Console.

Other key features of Android Studio 3 and Android 8 are also covered in detail including the Layout Editor, the ConstraintLayout and ConstraintSet classes, constraint chains and barriers, direct reply notifications and multi-window support.

Chapters also cover advanced features of Android Studio such as App Links, Instant Apps, the Android Studio Profiler and Gradle build configuration.

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

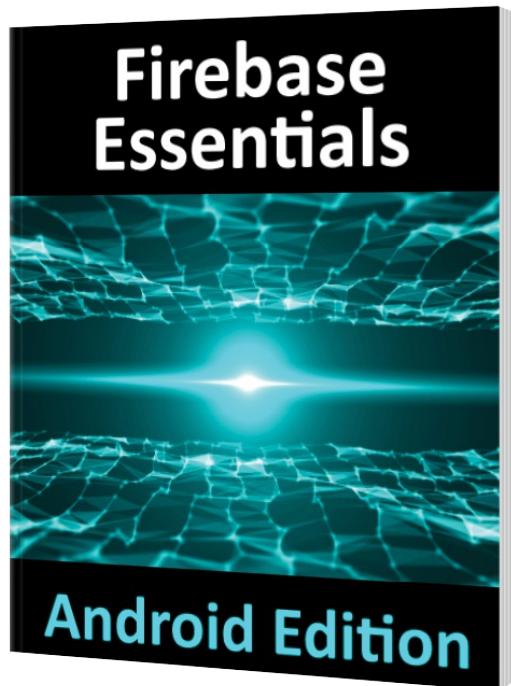
<http://www.ebookfrenzy.com/retail/androidstudio30/index.php>

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

1. From the Welcome to Android Studio dialog, select the Open an existing Android Studio project 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 Firebase Essentials Book Now Available

Firebase Essentials - Android Edition, a companion book to *Android Studio Development Essentials* provides everything you need to successfully integrate Firebase cloud features into your Android apps.



The *Firebase Essentials* book covers the key features of Android app development using Firebase including integration with Android Studio, User Authentication (including email, Twitter, Facebook and phone number sign-in), Realtime Database, Cloud Storage, Firebase Cloud Messaging (both upstream and downstream), Dynamic Links, Invites, App Indexing, Test Lab, Remote Configuration, Cloud Functions, Analytics and Performance

Monitoring.

Find out more at <https://goo.gl/5F381e>.

1.3 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.

1.4 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:

<http://www.ebookfrenzy.com/errata/androidstudio30.html>

In the event that you find an error not listed in the errata, please let us know by emailing our technical support team at 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 a number of 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 7/8/10 (32-bit or 64-bit)
- macOS 10.10 or later (Intel based systems only)
- Linux systems with version 2.19 or later of GNU C Library (glibc)
- Minimum of 3GB of RAM (8GB is preferred)
- Approximately 4GB of available disk space
- 1280 x 800 minimum screen resolution

2.2 Downloading the Android Studio Package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio version 3.0 which, at the time writing is the current version.

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 3.0 should provide the option to download the older version in the event that these differences become a problem.

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-bundle-<version>.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*, *Android SDK* 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 task bar for easy access by navigating to the *Android Studio\bin* directory, right-clicking on the executable and selecting the *Pin to Taskbar* menu option. Note that the executable is provided in 32-bit (*studio*) and 64-bit (*studio64*) executable versions. If you are running a 32-bit system be sure to use the *studio* executable.

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-ide-<version>.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 which will typically take a few minutes 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 sub-directory 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 will 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:i386
```

On RedHat 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

The first time that Android Studio is launched after being installed, a dialog will 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.

Next, the setup wizard may appear as shown in [Figure 2-2](#) though this dialog does not appear on all platforms:

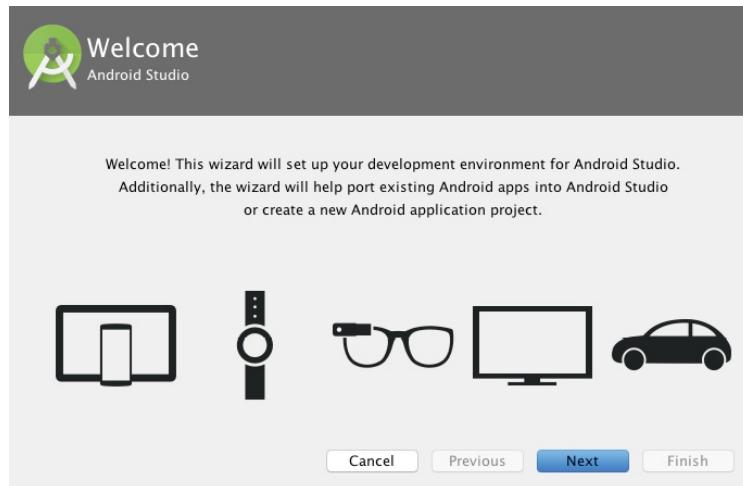


Figure 2-2

If the wizard appears, click on the Next button, choose the Standard installation option and click on Next once again.

Android Studio will proceed to download and configure the latest Android SDK and some additional components and packages. Once this process has completed, click on the *Finish* button in the *Downloading Components* dialog

at which point the Welcome to Android Studio screen should then appear:

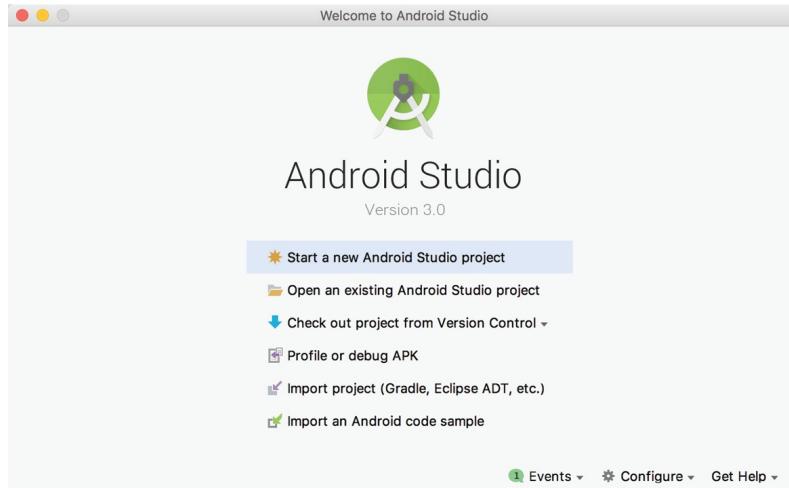


Figure 2-3

2.5 Installing Additional Android SDK Packages

The steps performed so far have installed Java, 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 using the *Android SDK Settings* screen, which may be launched from within the Android Studio tool by selecting the *Configure -> SDK Manager* option from within the Android Studio welcome dialog. Once invoked, the *Android SDK* screen of the default settings dialog will appear as shown in [Figure 2-4](#):

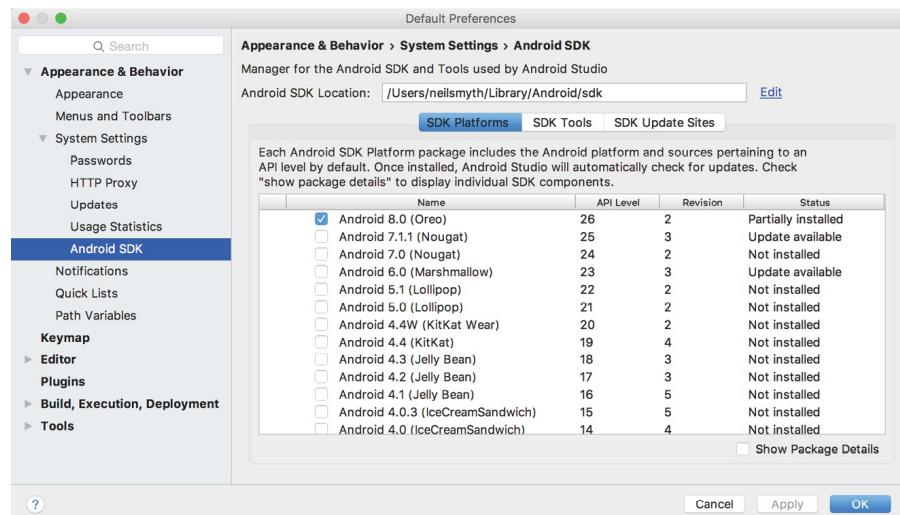


Figure 2-4

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.

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 available for update, 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 2-5](#):

Name	API Level	Revision	Status
Android TV Intel x86 Atom System Image	25	6	Not installed
Android Wear for China ARM EABI v7a System Image	25	3	Not installed
Android Wear for China Intel x86 Atom System Image	25	3	Not installed
Android Wear ARM EABI v7a System Image	25	3	Not installed
Android Wear Intel x86 Atom System Image	25	3	Not installed
Google APIs ARM 64 v8a System Image	25	8	Not installed
Google APIs ARM EABI v7a System Image	25	8	Not installed
Google APIs Intel x86 Atom System Image	25	8	Not installed
Google APIs Intel x86 Atom_64 System Image	25	6	Update Available: 8
Android 7.0 (Nougat)			
Google APIs	24	1	Not installed

Figure 2-5

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, a number of 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 2-6](#):

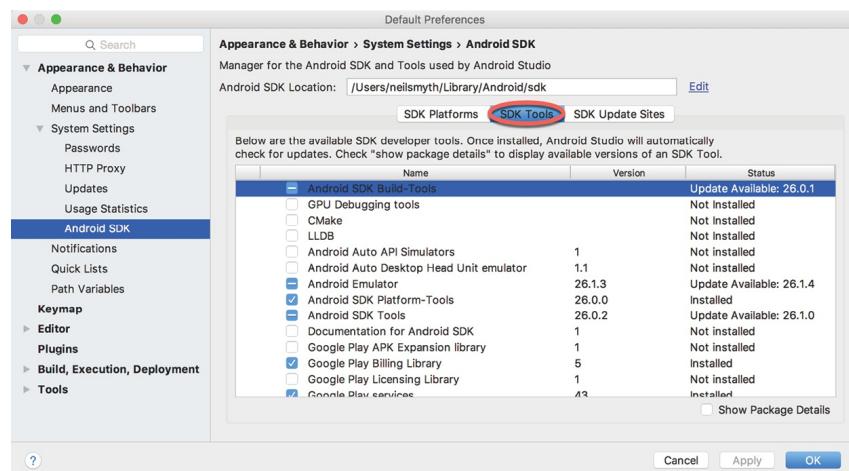


Figure 2-6

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
- Android SDK Tools
- Google Play Services
- Instant Apps Development SDK
- Intel x86 Emulator Accelerator (HAXM installer)
- ConstraintLayout for Android
- Solver for ConstraintLayout
- Android Support Repository
- Google Repository
- Google USB Driver (Windows only)

In the event that 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.

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. In order 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 2-7](#):

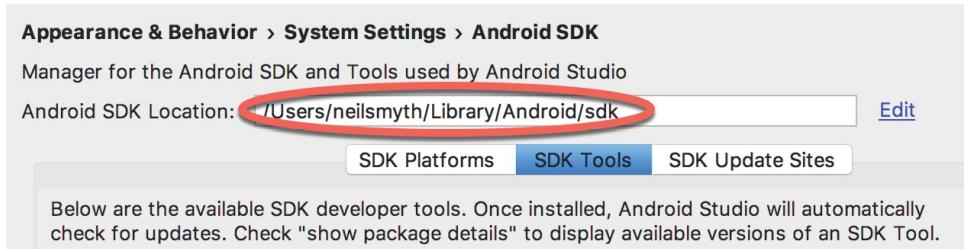


Figure 2-7

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 7

1. Right-click on Computer in the desktop start menu and select Properties from the resulting menu.
2. In the properties panel, select the Advanced System Settings link and, in the resulting dialog, click on the Environment Variables... button.
3. In the Environment Variables dialog, locate the Path variable in the System variables list, select it and click on *Edit*.... Locate the end of the current variable value string and append the path to the Android platform tools to the end, using a semicolon to separate the path from the preceding values. For example, assuming the Android SDK was installed into C:\Users\demo\AppData\Local\Android\sdk, the following would be appended to the end of the current Path value:

```
;C:\Users\demo\AppData\Local\Android\sdk\platform-tools; C:\Users\demo\AppData\Local\Android\sdk\tools; C:\Users\demo\A
```

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

Once the above steps are complete, verify that the path is correctly set by

opening a *Command Prompt* window (*Start -> All Programs -> Accessories -> Command Prompt*) and at the prompt enter:

```
echo %Path%
```

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

```
adb
```

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

Similarly, check the *tools* path setting by attempting to launch the AVD Manager command line tool:

```
avdmanager
```

In the event that 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 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. Follow the steps outlined for Windows 7 starting from step 2 through to step 4.

Open the command prompt window (move the mouse to the bottom right-hand corner of the screen, select the Search option and enter *cmd* into the search box). Select *Command Prompt* from the search results.

Within the Command Prompt window, enter:

```
echo %Path%
```

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

```
adb
```

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

Similarly, check the *tools* path setting by attempting to run the AVD Manager command line tool:

```
avdmanager
```

In the event that 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.
```

2.6.3 Windows 10

Right-click on the Start menu, select *System* from the resulting menu and click on the *Advanced system settings* option in the System window. Follow the steps outlined for Windows 7 starting from step 2 through to step 4.

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:/l  
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

A number of 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 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, click on the *Configure -> Check for Update* menu option within the Android Studio welcome screen, or use the *Help -> Check for Update* menu option accessible from within the Android Studio main window.

2.8 Summary

Prior to beginning the development of Android based applications, the first step is to set up a suitable development environment. This consists of the Java Development Kit (JDK), Android SDKs, and Android Studio IDE. 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 a simple 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 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](#):

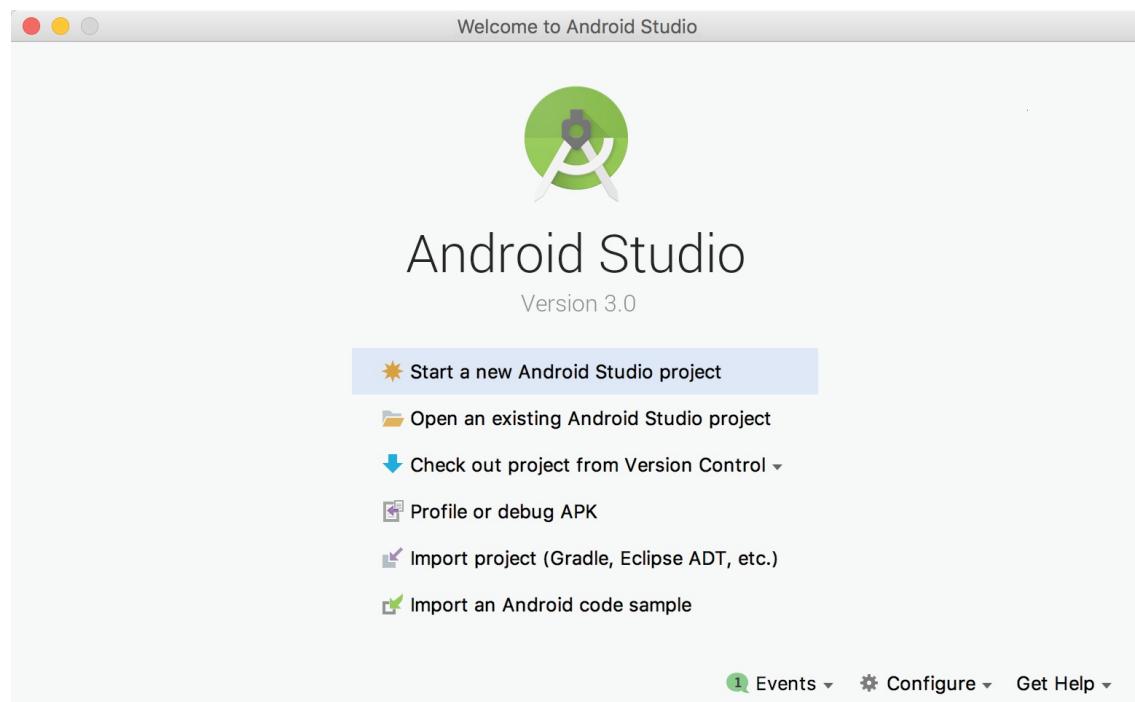


Figure 3-1

Once this window appears, Android Studio is ready for a new project to be created. To create the new project, simply click on the *Start a new Android Studio project* option to display the first screen of the *New Project* wizard as shown in [Figure 3-2](#):

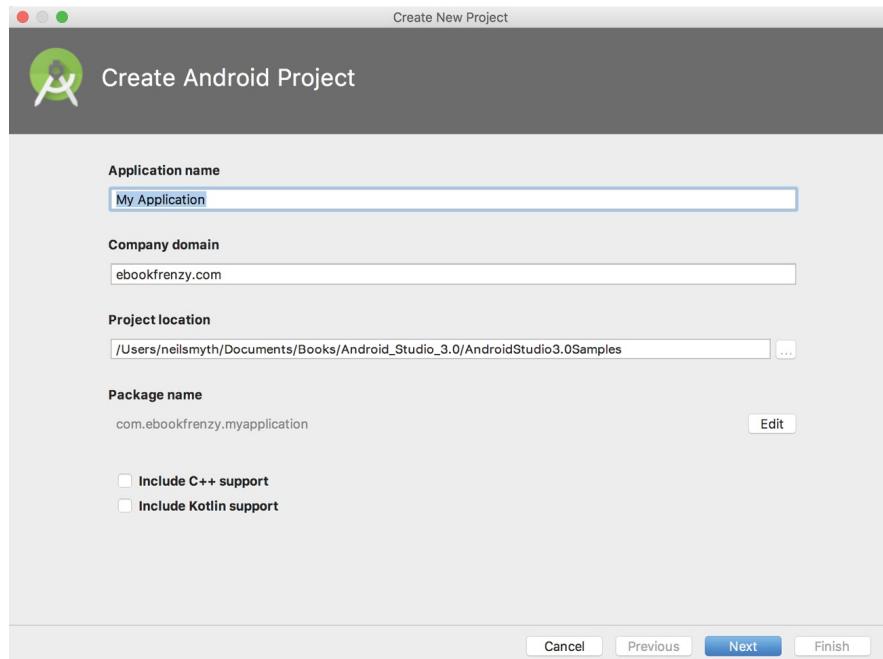


Figure 3-2

3.2 Defining the Project and SDK Settings

In the *New Project* window, set the *Application 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 will be used when the completed application goes 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 *Project 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 button to the right of the text field containing the current path setting.

Click *Next* to proceed. On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). The reason for selecting an older SDK release is that this ensures that the finished application will be able to run on the widest possible range of Android devices. The higher the minimum SDK selection, the more the application will be restricted to newer Android devices. A useful chart ([Figure 3-3](#)) can be viewed by clicking on the *Help me choose* link. This outlines the various SDK versions and API levels available for use and the percentage of Android devices in the marketplace on which the application will run if that SDK is used as the minimum level. In general it should only be necessary to select a more recent SDK when that release contains a specific feature that is required for your application.

To help in the decision process, selecting an API level from the chart will display the features that are supported at that level.

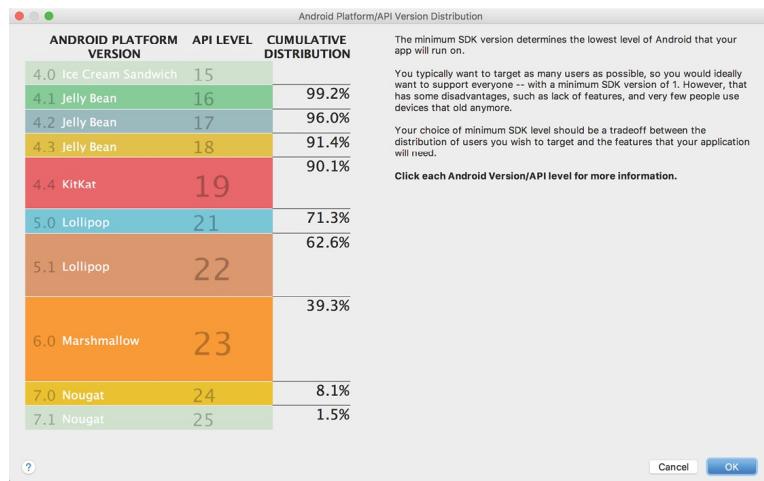


Figure 3-3

Since the project is not intended for Google TV, Android Auto or wearable devices, leave the remaining options disabled before clicking *Next*. Instant Apps will not be covered until later in this book so make sure that the *Include Android Instant App support* option is disabled.

3.3 Creating an Activity

The next step is to define the type of initial activity that is to be created for the application. A range of different activity types is available when developing Android applications. The *Empty*, *Master/Detail Flow*, *Google Maps* and *Navigation Drawer* options will be covered extensively in later chapters. For the purposes of this example, however, simply select the option to create a *Basic Activity*. The Basic Activity option creates a template user interface consisting of an app bar, menu, content area and a single floating action button.

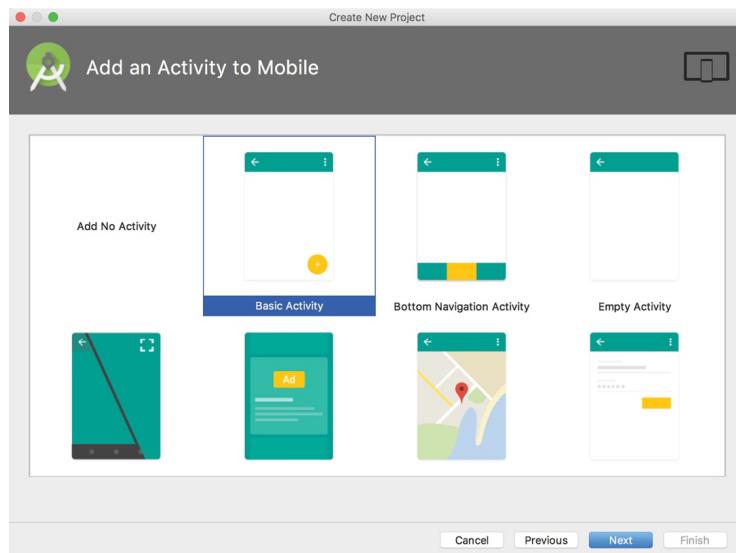


Figure 3-4

With the Basic Activity option selected, click *Next*. On the final screen ([Figure 3-5](#)) name the activity and title *AndroidSampleActivity*. The activity will consist of a single user interface screen layout which, for the purposes of this example, should be named *activity_android_sample*. Finally, enter *My Android App* into the title field as shown in [Figure 3-5](#):

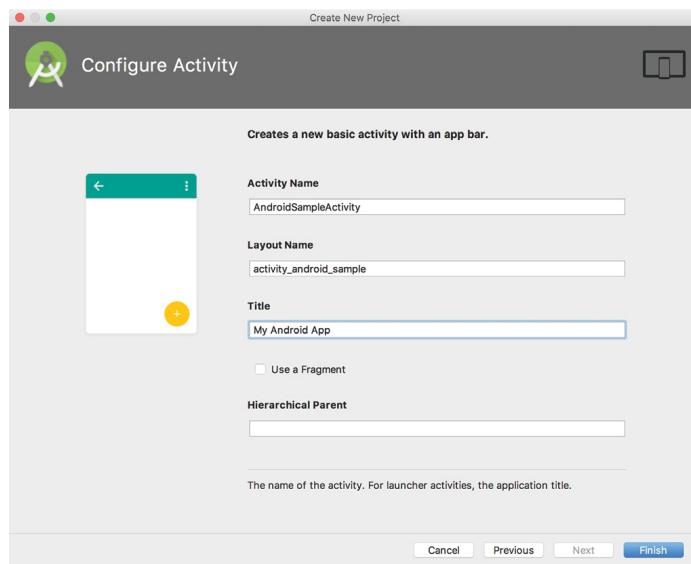


Figure 3-5

Since the `AndroidSampleActivity` is essentially the top level activity for the project and has no parent activity, there is no need to specify an activity for the Hierarchical parent (in other words `AndroidSampleActivity` does not need an “Up” button to return to another activity).

Click on *Finish* to initiate the project creation process.

3.4 Modifying the Example Application

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

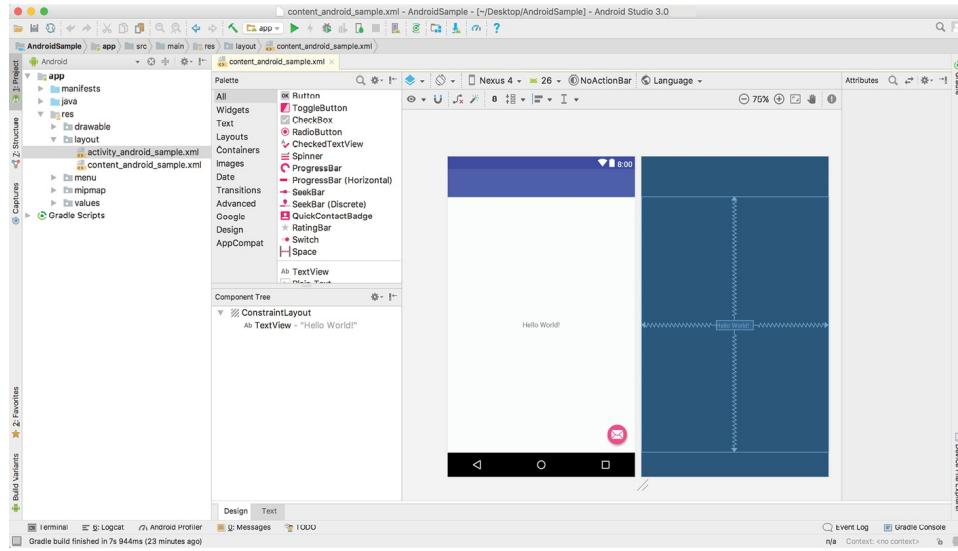


Figure 3-6

The newly created project and references to associated files are listed in the

Project tool window located on the left-hand side of the main project window. The Project tool window has a number of modes in which information can be displayed. By default, this panel will be in *Android* mode. This setting is controlled by the menu at the top of the panel as highlighted in [Figure 3-7](#). If the panel is not currently in *Android* mode, use the menu to switch mode:

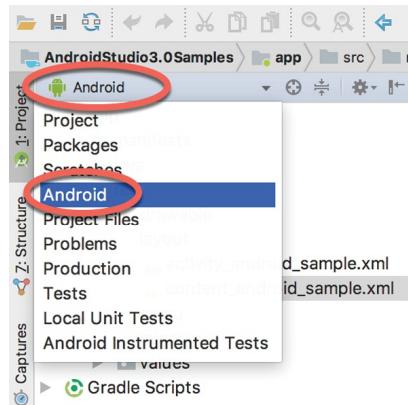


Figure 3-7

The example project created for us when we selected the option to create an activity consists of a user interface containing a label that will read “Hello World!” when the application is executed.

The next step in this tutorial is to modify the user interface of our application so that it displays a larger text view object with a different message to the one provided for us by Android Studio.

The user interface design for our activity is stored in a file named *activity_android_sample.xml* which, in turn, is located under *app -> res -> layout* in the project file hierarchy. This layout file includes the app bar (also known as an action bar) that appears across the top of the device screen (marked A in [Figure 3-8](#)) and the floating action button (the email button marked B). In addition to these items, the *activity_android_sample.xml* layout file contains a reference to a second file containing the content layout (marked C):



Figure 3-8

By default, the content layout is contained within a file named *content_android_sample.xml* and it is within this file that changes to the layout of the activity are made. Using the Project tool window, locate this file as illustrated in [Figure 3-9](#):

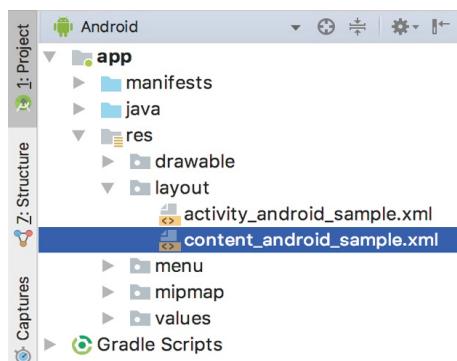


Figure 3-9

Once located, 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:

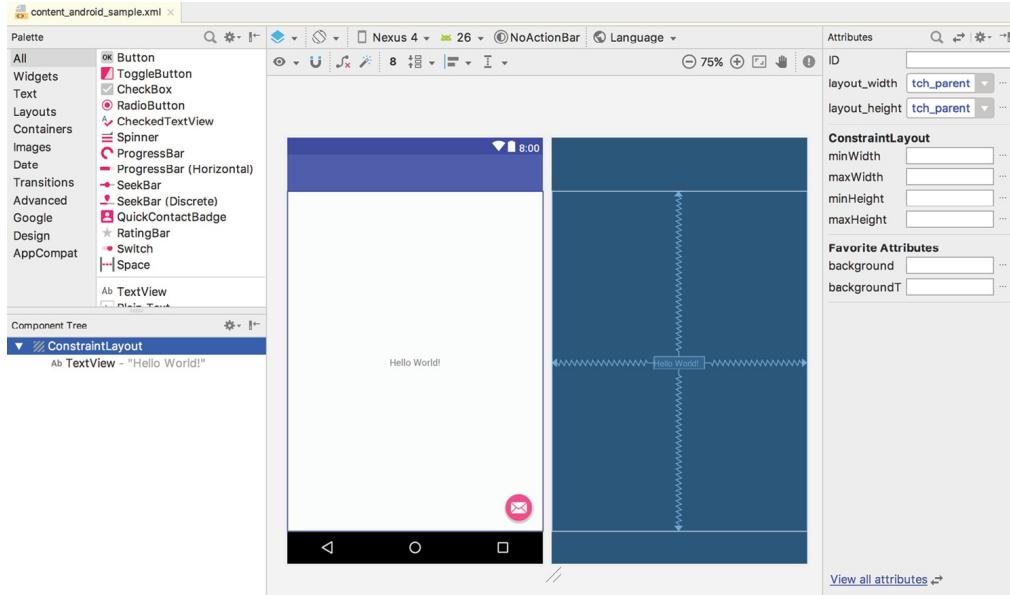


Figure 3-10

In the toolbar across the top of the Layout Editor window is a menu (currently set to *Nexus 4* 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.

As can be seen 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 3-11](#):

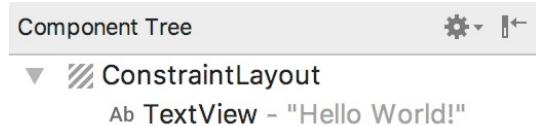


Figure 3-11

As we can see from the component tree hierarchy, the user interface layout consists of a ConstraintLayout parent with a single child in the form of a TextView object.

Before proceeding, check that the Layout Editor's Autoconnect mode is enabled. This means that as components are added to the layout, the Layout Editor will automatically add constraints to 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 3-12](#)). If necessary, re-enable Autoconnect mode by clicking on this button.

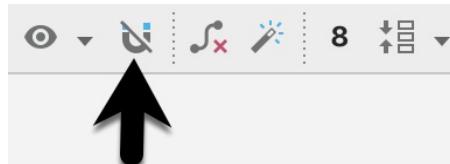


Figure 3-12

The next step in modifying the application is to delete the TextView component from the design. Begin by clicking on the TextView object within the user interface view so that it appears with a blue border around it. Once selected, press the Delete key on the keyboard to remove the object from the layout.

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 3-13](#), for example, the Button view is currently selected within the Widgets category:

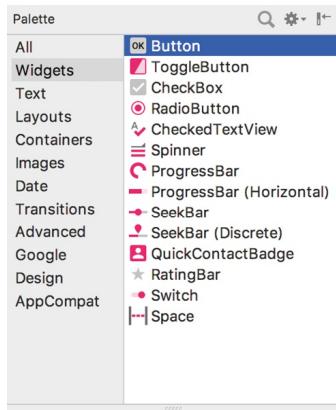


Figure 3-13

Click and drag the *Button* object from the Widgets list and drop it in the center of the user interface design when the marker lines appear indicating the center of the display:

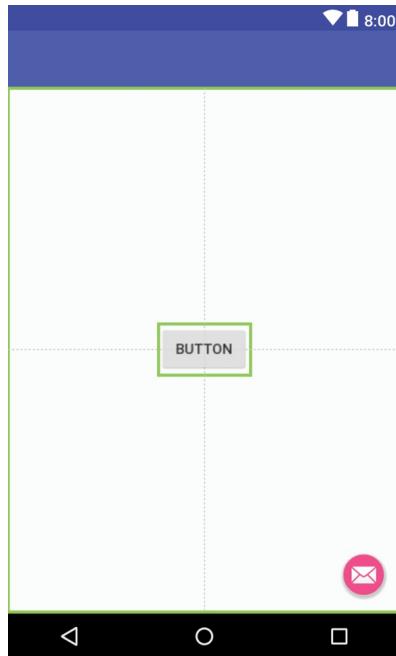


Figure 3-14

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 and change the current value from “Button” to “Demo” as shown in [Figure 3-15](#):

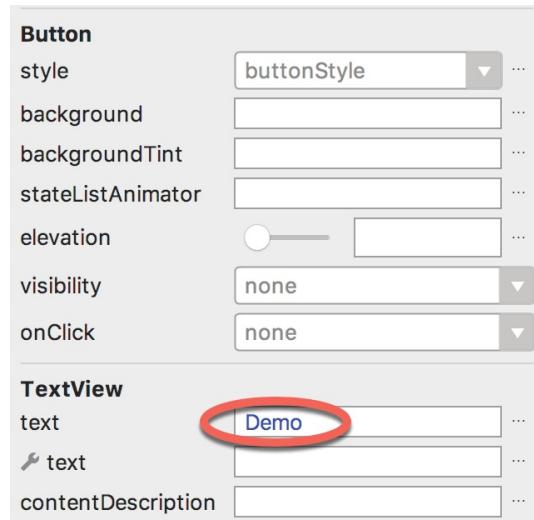


Figure 3-15

A useful shortcut to changing the text property of a component is to double-click on it in the layout. This will automatically locate the attribute in the attributes panel and select it ready for editing.

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.

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 3-16](#). Obviously, this is indicating potential problems with the layout. For details on any problems, click on the button:

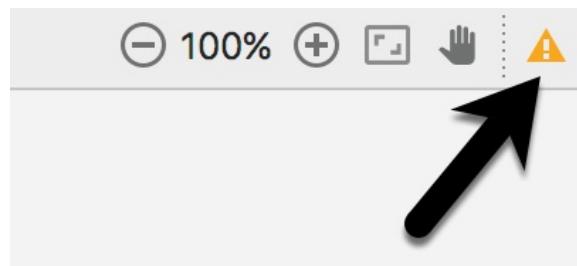


Figure 3-16

When clicked, a panel ([Figure 3-17](#)) will appear describing the nature of the problems and offering some possible corrective measures:

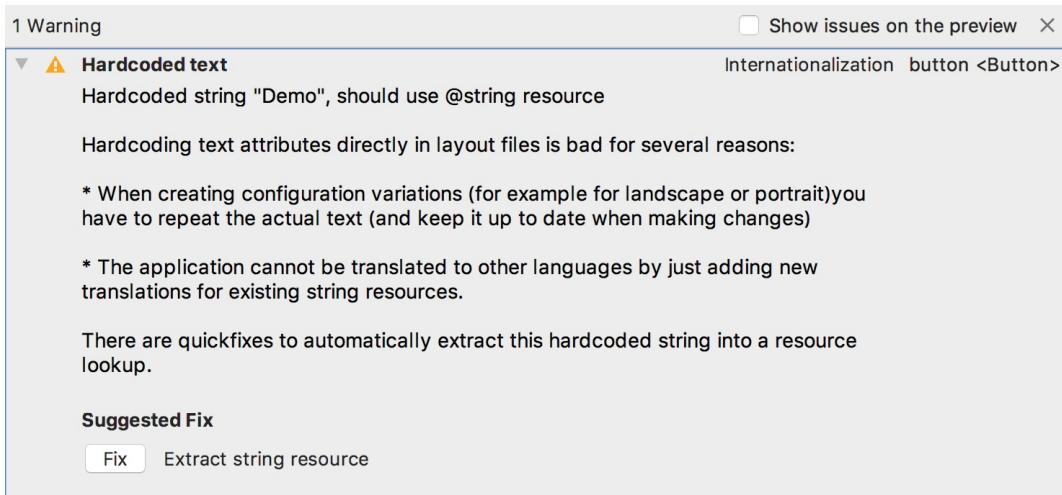


Figure 3-17

Currently, the only warning listed reads as follows:

```
Hardcoded string "Demo", 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 *demostring* and assign to it the string "Demo".

Click on the *Fix* button in the Issue Explanation panel to display the *Extract Resource* panel ([Figure 3-18](#)). Within this panel, change the resource name field to *demostring* and leave the resource value set to *Demo* before clicking on the OK button.

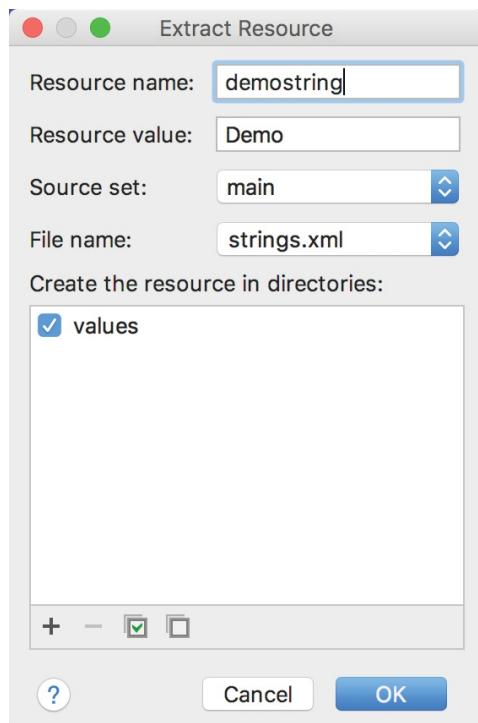


Figure 3-18

It is also worth noting that the string could also have been assigned to a resource when it was entered into the Attributes panel. This involves clicking on the button displaying three dots to the right of the property field in the Attributes panel and selecting the *Add new resource -> New String Value...* menu option from the resulting Resources dialog. In practice, however, it is often quicker to simply set values directly into the Attributes panel fields for any widgets in the layout, then work sequentially through the list in the warnings dialog to extract any necessary resources when the layout is complete.

3.5 Reviewing the Layout and Resource Files

Before moving on to the next chapter, 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 *content_android_sample.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 in order to make user interface changes and, in some instances, this may actually be quicker than using the Layout Editor

tool. At the bottom of the Layout Editor panel are two tabs labeled *Design* and *Text* respectively. To switch to the XML view simply select the *Text* tab as shown in [Figure 3-19](#):

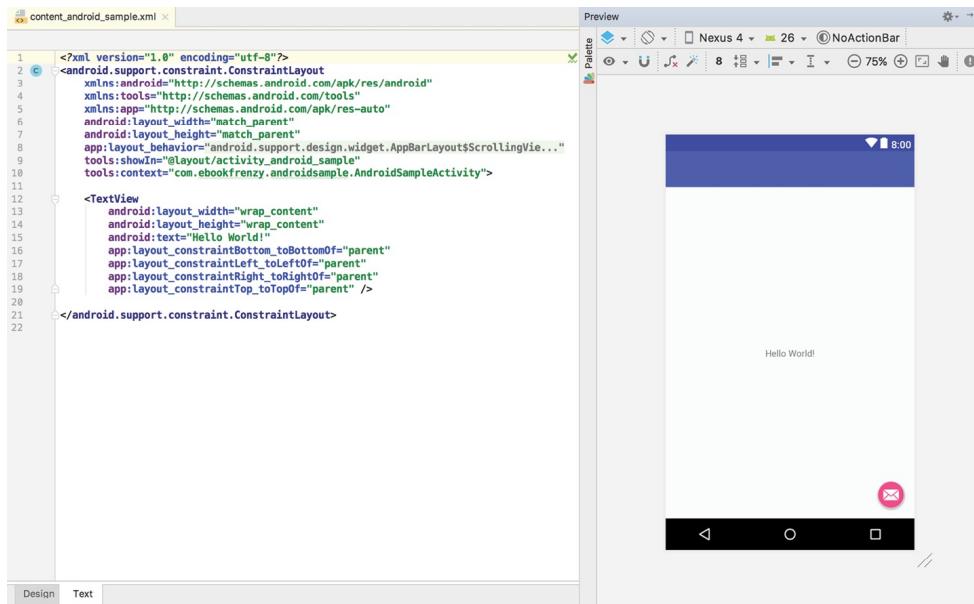


Figure 3-19

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 `Button` object. We can also see that the `text` property of the `Button` is set to our `demostring` resource. Although varying in complexity and content, all user interface layouts are structured in this hierarchical, XML based way.

One of the more powerful features of Android Studio can be found to the right-hand side of the XML editing panel. If the panel is not visible, display it by selecting the `Preview` button located along the right-hand edge of the Android Studio window. This is the Preview panel and shows the current visual state of the layout. As changes are made to the XML layout, these will be reflected in the preview panel. The layout may also be modified visually from within the Preview panel with the changes appearing in the XML listing. To see this in action, 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"?>
<android.support.constraint.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">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"/>

```

```
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="com.ebookfrenzy.myapplication.AndroidSampleActivity"
    tools:showIn="@layout/activity_android_sample"
    android:background="#ff2438" >
    .
    .
</android.support.constraint.ConstraintLayout>
```

Note that the color of the preview 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. Change the color value to #a0ff28 and note that both the small square in the margin and the preview change to green.

Finally, use the Project view 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="action_settings">Settings</string>
    <string name="demostring">Demo</string>
</resources>
```

As a demonstration of resources in action, change the string value currently assigned to the *demostring* resource to “Hello” 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 Text mode, click on the “@string/demostring” 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 “Demo” text.

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:

The screenshot shows the 'Translations Editor' window in Android Studio. The title bar includes tabs for 'content_android_sample.xml', 'strings.xml', and 'Translations Editor'. Below the tabs are buttons for '+', 'Show All Keys', 'Show All Locales', and a question mark icon. To the right is a link 'Order a translation...'. The main area is a table with four columns: 'Key', 'Resource Folder', 'Untranslated...', and 'Default Value'. The data rows are:

Key	Resource Folder	Untranslated...	Default Value
app_name	app/src/main/res	<input type="checkbox"/>	AndroidSample
action_settings	app/src/main/res	<input type="checkbox"/>	Settings
demostring	app/src/main/res	<input type="checkbox"/>	Demo

Figure 3-20

This editor allows the strings assigned to resource keys to be edited and for translations for multiple languages to be managed. The *Order a translation...* link may also be used to order a translation of the strings contained within the application to other languages. The cost of the translations will vary depending on the number of strings involved.

3.6 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 a simple example to make sure the environment is correctly installed and configured. In this chapter, we have created a simple 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. Finally, we looked at the underlying XML that is used to store the user interface designs of Android applications.

While it is useful to be able to preview a layout from within the Android Studio Layout Editor tool, there is no substitute for testing an application by compiling and running it. In a later chapter, the steps necessary to set up an emulator for testing purposes will be covered in detail. Before running the application, however, the next chapter will take a small detour to provide a guided tour of the Android Studio user interface.

4. A Tour of the Android Studio User Interface

While it is tempting to plunge into running the example application created in the previous chapter, doing so involves using aspects of the Android Studio user interface which are best described in advance.

Android Studio is a powerful and feature rich development environment that is, to a large extent, intuitive to use. That being said, taking the time now to gain familiarity with the layout and organization of the Android Studio user interface will considerably shorten the learning curve in later chapters of the book. With this in mind, this chapter will provide an initial overview of the various areas and components that make up the Android Studio environment.

4.1 The Welcome Screen

The welcome screen ([Figure 4-1](#)) is displayed any time that Android Studio is running with no projects currently open (open projects can be closed at any time by selecting the *File -> Close Project* menu option). If Android Studio was previously exited while a project was still open, the tool will by-pass the welcome screen next time it is launched, automatically opening the previously active project.

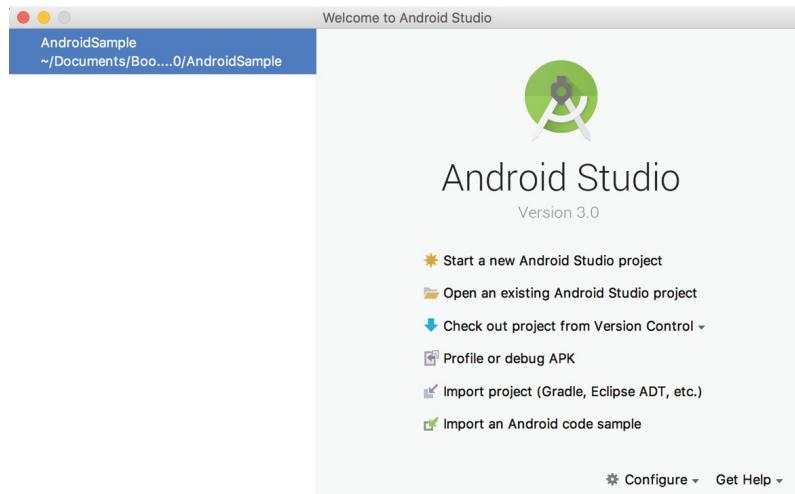


Figure 4-1

In addition to a list of recent projects, the Quick Start menu provides a range of options for performing tasks such as opening, creating and importing

projects along with access to projects currently under version control. In addition, the *Configure* menu at the bottom of the window provides access to the SDK Manager along with a vast array of settings and configuration options. A review of these options will quickly reveal that there is almost no aspect of Android Studio that cannot be configured and tailored to your specific needs.

The Configure menu also includes an option to check if updates to Android Studio are available for download.

4.2 The Main Window

When a new project is created, or an existing one opened, the Android Studio *main window* will appear. When multiple projects are open simultaneously, each will be assigned its own main window. The precise configuration of the window will vary depending on which tools and panels were displayed the last time the project was open, but will typically resemble that of [Figure 4-2](#).

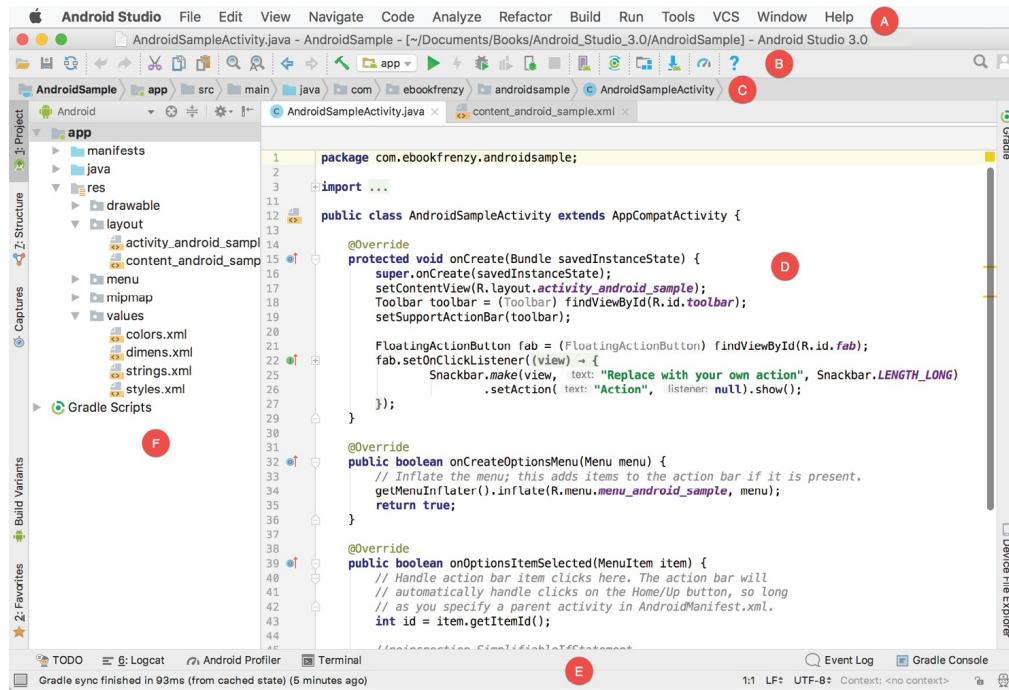


Figure 4-2

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

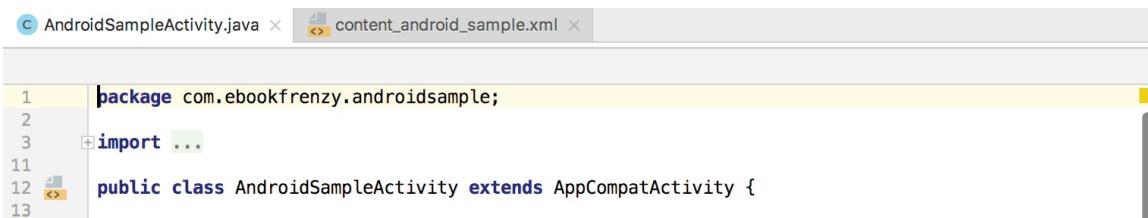
A – Menu Bar – Contains a range of menus for performing tasks within the Android Studio environment.

B – Toolbar – A selection of shortcuts to frequently performed actions. The toolbar buttons provide quicker access to a select group of menu bar actions.

The toolbar can be customized by right-clicking on the bar and selecting the *Customize Menus and Toolbars...* menu option.

C – Navigation Bar – The navigation bar provides a convenient way to move around the files and folders that make up the project. Clicking on an element in the navigation bar will drop down a menu listing the subfolders and files at that location ready for selection. This provides an alternative to the Project tool window.

D – Editor Window – The editor window displays the content of the file on which the developer is currently working. What gets displayed in this location, however, is subject to context. When editing code, for example, the code editor will appear. When working on a user interface layout file, on the other hand, the user interface Layout Editor tool will appear. When multiple files are open, each file is represented by a tab located along the top edge of the editor as shown in [Figure 4-3](#).



```
1 package com.ebookfrenzy.androidsample;
2
3 import ...
4
5 public class AndroidSampleActivity extends AppCompatActivity {
```

Figure 4-3

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

F – Project Tool Window – The project tool window provides a hierarchical overview of the project file structure allowing navigation to specific files and folders to be performed. The toolbar can be used to display the project in a number of different ways. The default setting is the *Android* view which is the mode primarily used in the remainder of this book.

The project tool window is just one of a number of tool windows available within the Android Studio environment.

4.3 The Tool Windows

In addition to the project view tool window, Android Studio also includes a

number of other windows which, when enabled, are displayed along the bottom and sides of the main window. The tool window quick access menu can be accessed by hovering the mouse pointer over the button located in the far left-hand corner of the status bar ([Figure 4-4](#)) without clicking the mouse button.

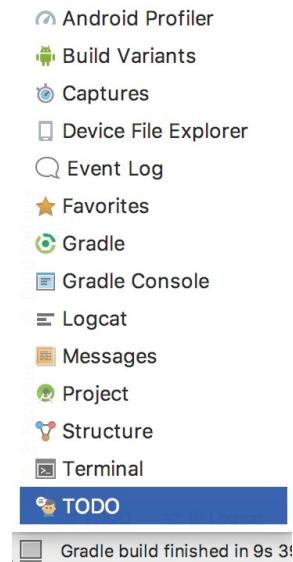


Figure 4-4

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

Alternatively, a set of *tool window bars* can be displayed by clicking on the quick access menu icon in the status bar. These bars appear along the left, right and bottom edges of the main window (as indicated by the arrows in [Figure 4-5](#)) and contain buttons for showing and hiding each of the tool windows. When the tool window bars are displayed, a second click on the button in the status bar will hide them.

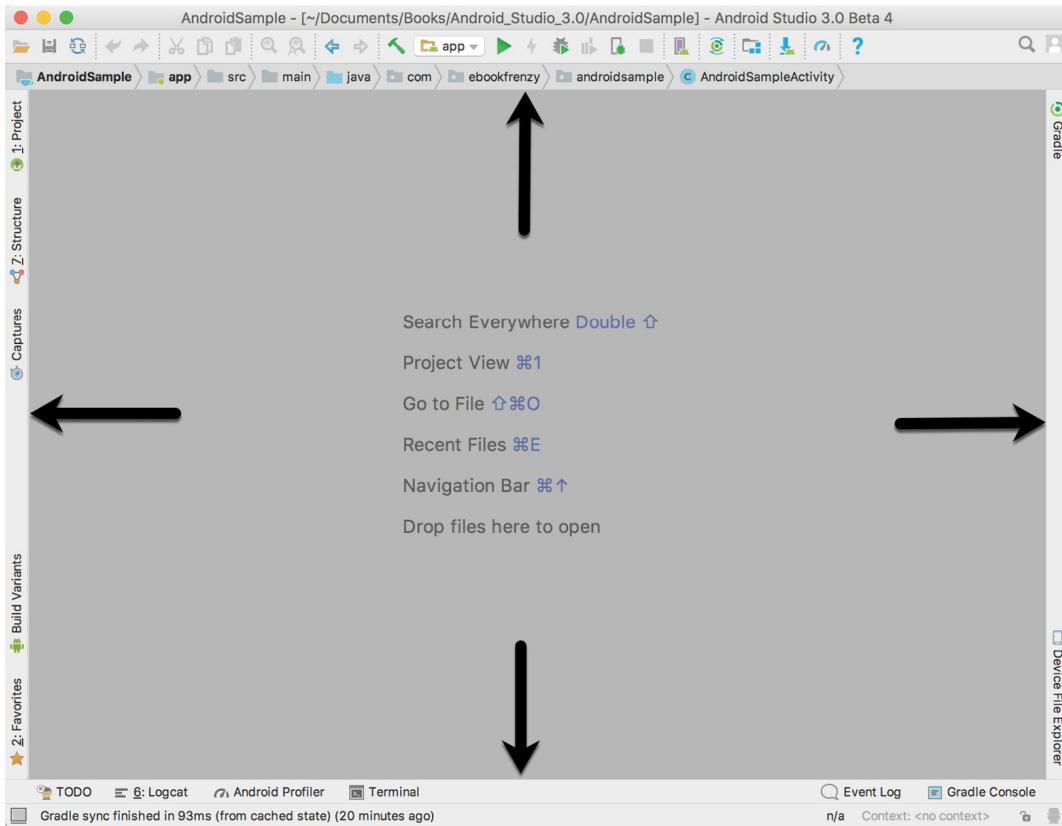


Figure 4-5

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

The location of a button in a tool window bar indicates the side of the window against which the window will appear when displayed. These positions can be changed by clicking and dragging the buttons to different locations in other window tool bars.

Each tool window has its own toolbar along the top edge. The buttons within these toolbars vary from one tool to the next, though all tool windows contain a settings option, represented by the cog icon, which allows various aspects of the window to be changed. [Figure 4-6](#) shows the settings menu for the project view tool window. Options are available, for example, to undock a window and to allow it to float outside of the boundaries of the Android Studio main window and to move and resize the tool panel.

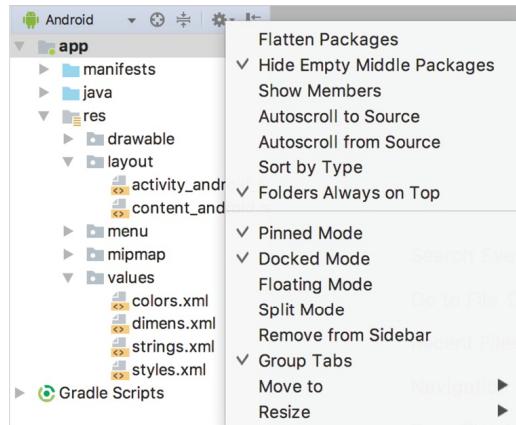


Figure 4-6

All of the windows also include a far right button on the toolbar providing an additional way to hide the tool window from view. A search of the items within a tool window can be performed simply by giving that window focus by clicking in it and then typing the search term (for example the name of a file in the Project tool window). A search box will appear in the window's toolbar and items matching the search highlighted.

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

Project – The project view provides an overview of the file structure that makes up the project allowing for quick navigation between files. Generally, double-clicking on a file in the project view will cause that file to be loaded into the appropriate editing tool.

Structure – The structure tool provides a high level view of the structure of the source file currently displayed in the editor. This information includes a list of items such as classes, methods and variables in the file. Selecting an item from the structure list will take you to that location in the source file in the editor window.

Captures – The captures tool window provides access to performance data files that have been generated by the monitoring tools contained within Android Studio.

Favorites – A variety of project items can be added to the favorites list. Right-clicking on a file in the project view, for example, provides access to an *Add to Favorites* menu option. Similarly, a method in a source file can be added as a favorite by right-clicking on it in the Structure tool window. Anything added

to a Favorites list can be accessed through this Favorites tool window.

Build Variants – The build variants tool window provides a quick way to configure different build targets for the current application project (for example different builds for debugging and release versions of the application, or multiple builds to target different device categories).

TODO – As the name suggests, this tool provides a place to review items that have yet to be completed on the project. Android Studio compiles this list by scanning the source files that make up the project to look for comments that match specified TODO patterns. These patterns can be reviewed and changed by selecting the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS) and navigating to the *TODO* page listed under *Editor*.

Messages – The messages tool window records output from the Gradle build system (Gradle is the underlying system used by Android Studio for building the various parts of projects into runnable applications) and can be useful for identifying the causes of build problems when compiling application projects.

Logcat – The Logcat tool window provides access to the monitoring log output from a running application in addition to options for taking screenshots and videos of the application and stopping and restarting a process.

Terminal – Provides access to a terminal window on the system on which Android Studio is running. On Windows systems this is the Command Prompt interface, while on Linux and macOS systems this takes the form of a Terminal prompt.

Run – The run tool window becomes available when an application is currently running and provides a view of the results of the run together with options to stop or restart a running process. If an application is failing to install and run on a device or emulator, this window will typically provide diagnostic information relating to the problem.

Event Log – The event log window displays messages relating to events and activities performed within Android Studio. The successful build of a project, for example, or the fact that an application is now running will be reported within this tool window.

Gradle Console – The Gradle console is used to display all output from the

Gradle system as projects are built from within Android Studio. This will include information about the success or otherwise of the build process together with details of any errors or warnings.

Gradle – The Gradle tool window provides a view onto the Gradle tasks that make up the project build configuration. The window lists the tasks that are involved in compiling the various elements of the project into an executable application. Right-click on a top level Gradle task and select the *Open Gradle Config* menu option to load the Gradle build file for the current project into the editor. Gradle will be covered in greater detail later in this book.

Android Profiler – The Android Profiler tool window provides realtime monitoring and analysis tools for identifying performance issues within running apps, including CPU, memory and network usage.

Device File Explorer – The Device File Explorer tool window provides direct access to the filesystem of the currently connected Android device or emulator allowing the filesystem to be browsed and files copied to the local filesystem.

4.4 Android Studio Keyboard Shortcuts

Android Studio includes an abundance of keyboard shortcuts designed to save time when performing common tasks. A full keyboard shortcut keymap listing can be viewed and printed from within the Android Studio project window by selecting the *Help -> Keymap Reference* menu option.

4.5 Switcher and Recent Files Navigation

Another useful mechanism for navigating within the Android Studio main window involves the use of the *Switcher*. Accessed via the *Ctrl-Tab* keyboard shortcut, the switcher appears as a panel listing both the tool windows and currently open files ([Figure 4-7](#)).

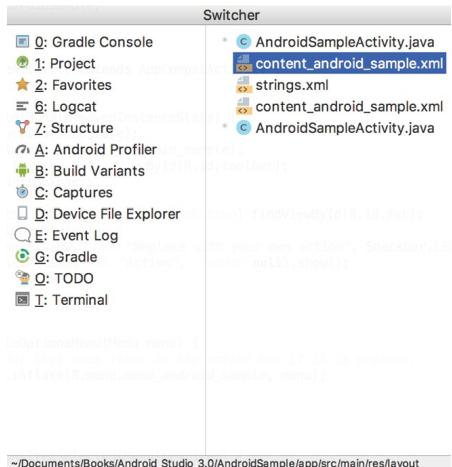


Figure 4-7

Once displayed, the switcher will remain visible for as long as the Ctrl key remains depressed. Repeatedly tapping the Tab key while holding down the Ctrl key will cycle through the various selection options, while releasing the Ctrl key causes the currently highlighted item to be selected and displayed within the main window.

In addition to the switcher, navigation to recently opened files is provided by the Recent Files panel ([Figure 4-8](#)). This can be accessed using the Ctrl-E keyboard shortcut (Cmd-E on macOS). Once displayed, either the mouse pointer can be used to select an option or, alternatively, the keyboard arrow keys used to scroll through the file name and tool window options. Pressing the Enter key will select the currently highlighted item.

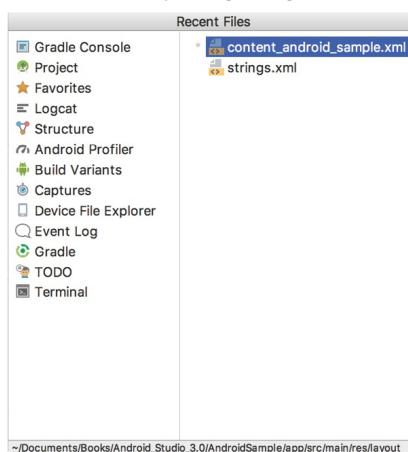


Figure 4-8

4.6 Changing the Android Studio Theme

The overall theme of the Android Studio environment may be changed either

from the welcome screen using the *Configure -> Settings* option, or via the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS) of the main window.

Once the settings dialog is displayed, select the *Appearance* option in the left-hand panel and then change the setting of the *Theme* menu before clicking on the *Apply* button. The themes available will depend on the platform but usually include options such as IntelliJ, Windows, Default and Darcula. [Figure 4-9](#) shows an example of the main window with the Darcula theme selected:

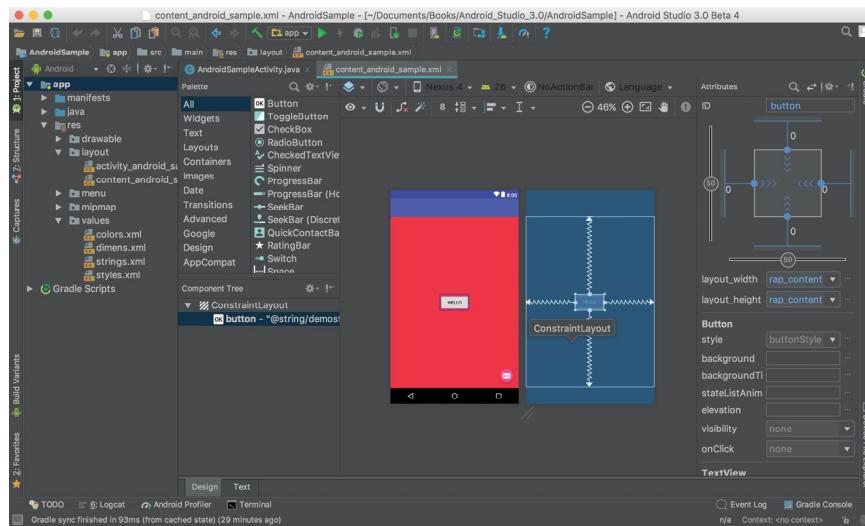


Figure 4-9

4.7 Summary

The primary elements of the Android Studio environment consist of the welcome screen and main window. Each open project is assigned its own main window which, in turn, consists of a menu bar, toolbar, editing and design area, status bar and a collection of tool windows. Tool windows appear on the sides and bottom edges of the main window and can be accessed either using the quick access menu located in the status bar, or via the optional tool window bars.

There are very few actions within Android Studio which cannot be triggered via a keyboard shortcut. A keymap of default keyboard shortcuts can be accessed at any time from within the Android Studio main window.

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

In the course of developing Android apps in Android Studio it will be necessary to compile and run an application multiple times. An Android application may be tested by installing and running it either on a physical device or in an *Android Virtual Device (AVD)* emulator environment. Before an AVD can be used, it must first be created and configured to match the specifications of a particular device model. The goal of this chapter, therefore, is to work through the steps involved in creating such a virtual device using the Nexus 5X phone as a reference example.

5.1 About Android Virtual Devices

AVDs are essentially emulators that allow Android applications to be tested without the necessity to install the application on a physical Android based device. An AVD may be configured to emulate a variety of hardware features including options such as screen size, memory capacity and the presence or otherwise of features such as a camera, GPS navigation support or an accelerometer. As part of the standard Android Studio installation, a number of emulator templates are installed allowing AVDs to be configured for a range of different devices. Additional templates may be loaded or custom configurations created to match any physical Android device by specifying properties such as processor type, memory capacity and the size and pixel density of the screen. Check the online developer documentation for your device to find out if emulator definitions are available for download and installation into the AVD environment.

When launched, an AVD will appear as a window containing an emulated Android device environment. [Figure 5-1](#), for example, shows an AVD session configured to emulate the Google Nexus 5X model.

New AVDs are created and managed using the Android Virtual Device Manager, which may be used either in command-line mode or with a more user-friendly graphical user interface.



Figure 5-1

5.2 Creating a New AVD

In order to test the behavior of an application in the absence of a physical device, it will be necessary to create an AVD for a specific Android device configuration.

To create a new AVD, the first step is to launch the AVD Manager. This can be achieved from within the Android Studio environment by selecting the *Tools -> Android -> AVD Manager* menu option from within the main window.

Once launched, the tool will appear as outlined in [Figure 5-2](#) if existing AVD instances have been created:



Figure 5-2

To add an additional AVD, begin by clicking on the *Create Virtual Device*

button in order to invoke the *Virtual Device Configuration* dialog:

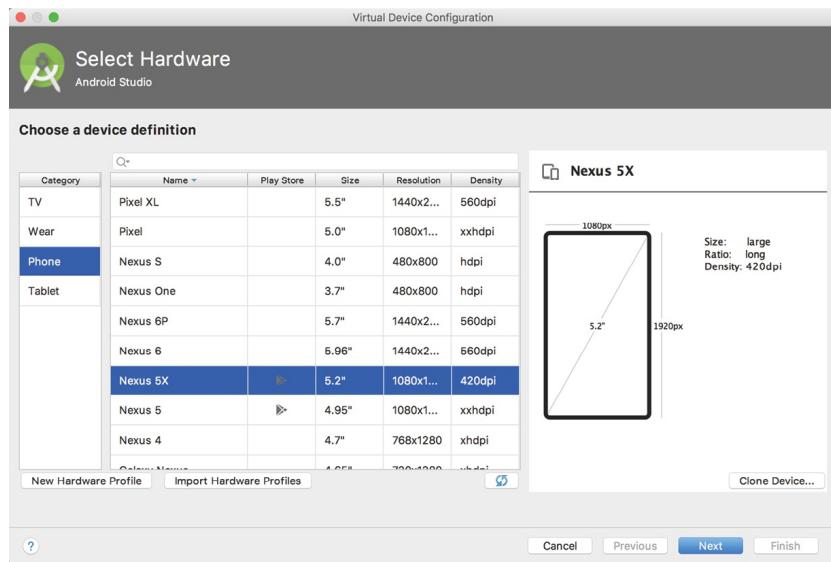


Figure 5-3

Within the dialog, perform the following steps to create a Nexus 5X compatible emulator:

1. From the *Category* panel, select the *Phone* option to display the list of available Android tablet AVD templates.
2. Select the *Nexus 5X* device option and click *Next*.
3. On the System Image screen, select the latest version of Android (at time of writing this is Oreo, API level 26, Android 8.0 with Google Play) for the *x86* ABI. Note that if the system image has not yet been installed a *Download* link will be provided next to the Release Name. Click this link to download and install the system image before selecting it. If the image you need is not listed, click on the *x86 images* and *Other images* tabs to view alternative lists.
4. Click *Next* to proceed and enter a descriptive name (for example *Nexus 5X API 26*) into the name field or simply accept the default name.
5. Click *Finish* to create the AVD.
6. With the AVD created, the AVD Manager may now be closed. If future modifications to the AVD are necessary, simply re-open the AVD Manager, select the AVD from the list and click on the pencil icon in the *Actions* column of the device row in the AVD Manager.

5.3 Starting the Emulator

To perform a test run of the newly created AVD emulator, simply select the emulator from the AVD Manager and click on the launch button (the green triangle in the Actions column). The emulator will appear in a new window and begin the startup process. The amount of time it takes for the emulator to start will depend on the configuration of both the AVD and the system on which it is running. In the event that the startup time on your system is considerable, do not hesitate to leave the emulator running. The system will detect that it is already running and attach to it when applications are launched, thereby saving considerable amounts of startup time.

The emulator probably defaulted to appearing in portrait orientation. It is useful to be aware that this and other default options can be changed. Within the AVD Manager, select the new Nexus 5X entry and click on the pencil icon in the *Actions* column of the device row. In the configuration screen locate the *Startup and orientation* section and change the orientation setting. Exit and restart the emulator session to see this change take effect. More details on the emulator are covered in the next chapter ([“Using and Configuring the Android Studio AVD Emulator”](#)).

To save time in the next section of this chapter, leave the emulator running before proceeding.

5.4 Running the Application in the AVD

With an AVD emulator configured, the example AndroidSample application created in the earlier chapter now can be compiled and run. With the AndroidSample project loaded into Android Studio, simply click on the run button represented by a green triangle located in the Android Studio toolbar as shown in [Figure 5-4](#) below, select the *Run -> Run ‘app’* menu option or use the Ctrl-R keyboard shortcut:

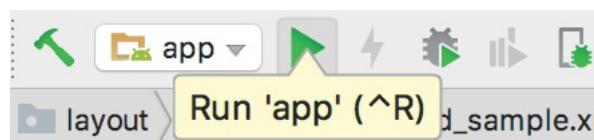


Figure 5-4

By default, Android Studio will respond to the run request by displaying the *Select Deployment Target* dialog. This provides the option to execute the application on an AVD instance that is already running, or to launch a new AVD session specifically for this application. [Figure 5-5](#) lists the previously

created Nexus 5X AVD as a running device as a result of the steps performed in the preceding section. With this device selected in the dialog, click on *OK* to install and run the application on the emulator.

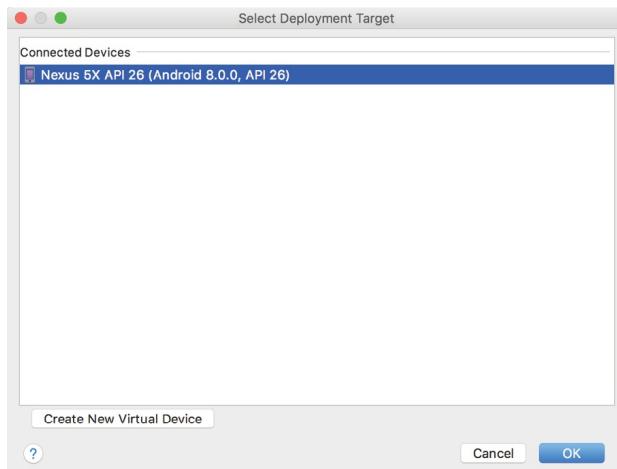


Figure 5-5

Once the application is installed and running, the user interface for the `AndroidSampleActivity` class will appear within the emulator:



Figure 5-6

In the event that the activity does not automatically launch, check to see if the launch icon has appeared among the apps on the emulator. If it has, simply click on it to launch the application. Once the run process begins, the Run and Logcat tool windows will become available. The Run tool window will display diagnostic information as the application package is installed and launched. [Figure 5-7](#) shows the Run tool window output from a successful application launch:

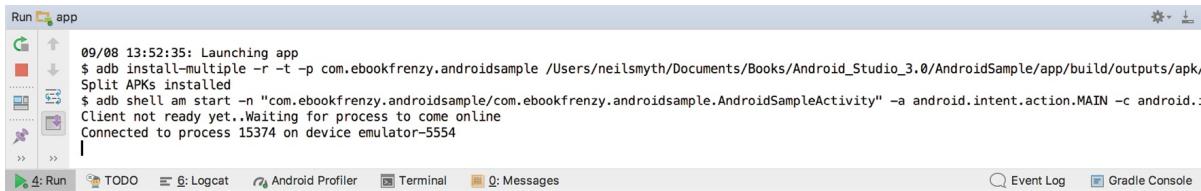


Figure 5-7

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

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

5.5 Run/Debug Configurations

A particular project can be configured such that a specific device or emulator is used automatically each time it is run from within Android Studio. This avoids the necessity to make a selection from the device chooser each time the application is executed. To review and modify the Run/Debug configuration, click on the button to the left of the run button in the Android Studio toolbar and select the *Edit Configurations...* option from the resulting menu:

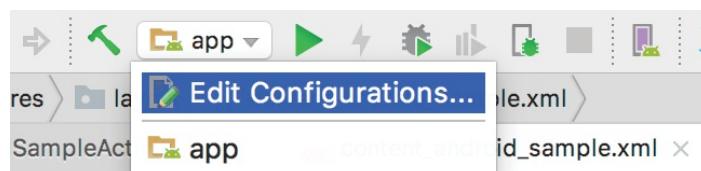


Figure 5-8

In the *Run/Debug Configurations* dialog, the application may be configured to always use a preferred emulator by selecting *Emulator* from the *Target* menu located in the *Deployment Target Options* section and selecting the emulator from the drop down menu. [Figure 5-9](#), for example, shows the *AndroidSample* application configured to run by default on the previously created Nexus 5X emulator:

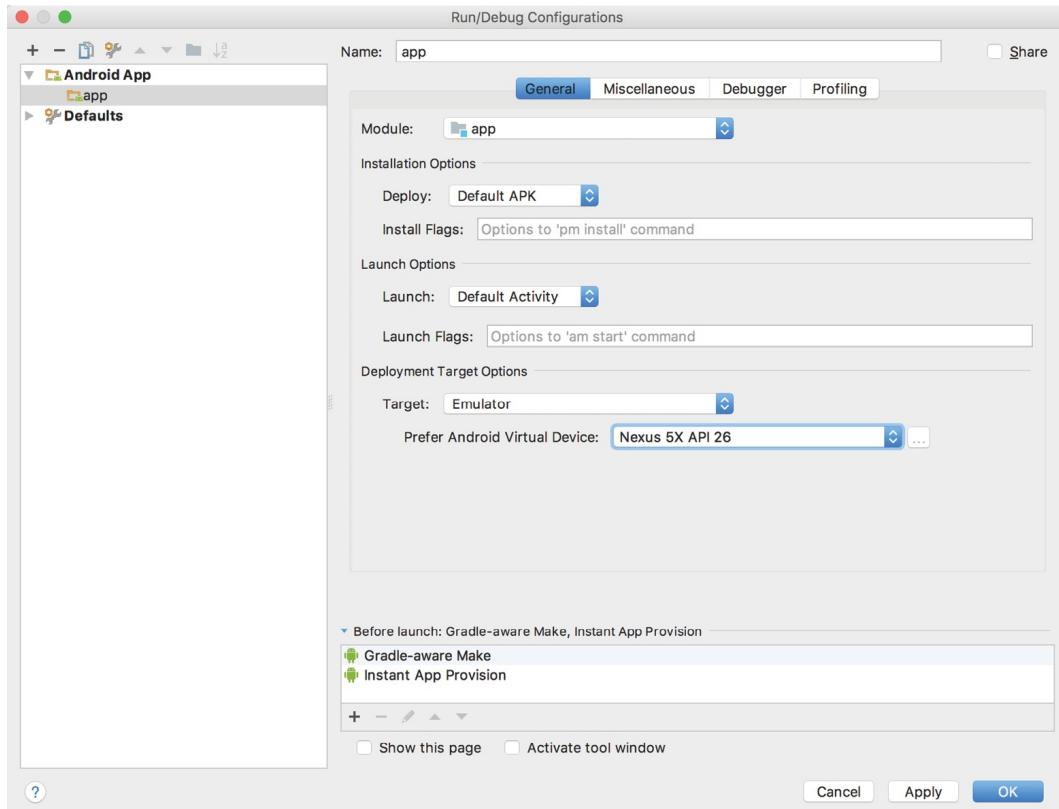


Figure 5-9

Be sure to switch the Target menu setting back to “Open Select Deployment Target Dialog” mode before moving on to the next chapter of the book.

5.6 Stopping a Running Application

To stop a running application, simply click on the stop button located in the main toolbar as shown in [Figure 5-10](#):

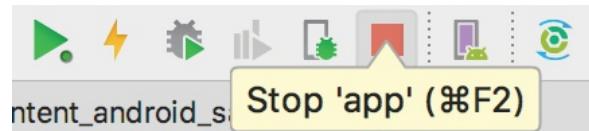


Figure 5-10

An app may also be terminated using the Logcat tool window. Begin by displaying the *Logcat* tool window either using the window bar button, or via the quick access menu (invoked by moving the mouse pointer over the button in the left-hand corner of the status bar as shown in [Figure 5-11](#)).

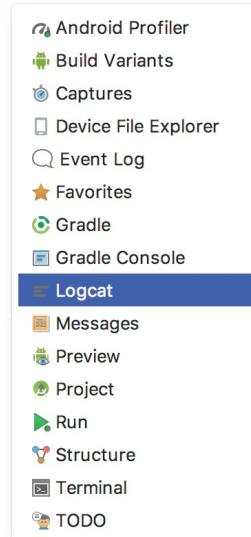


Figure 5-11

Once the Logcat tool window appears, select the *androidsample* app menu highlighted in [Figure 5-12](#) below:



Figure 5-12

With the process selected, stop it by clicking on the red *Terminate Application* button in the toolbar to the left of the process list indicated by the arrow in the above figure.

An alternative to using the Android tool window is to open the Android Device Monitor. This can be launched via the *Tools -> Android -> Android Device Monitor* menu option. Once launched, the process may be selected from the list ([Figure 5-13](#)) and terminated by clicking on the red *Stop* button located in the toolbar above the list.

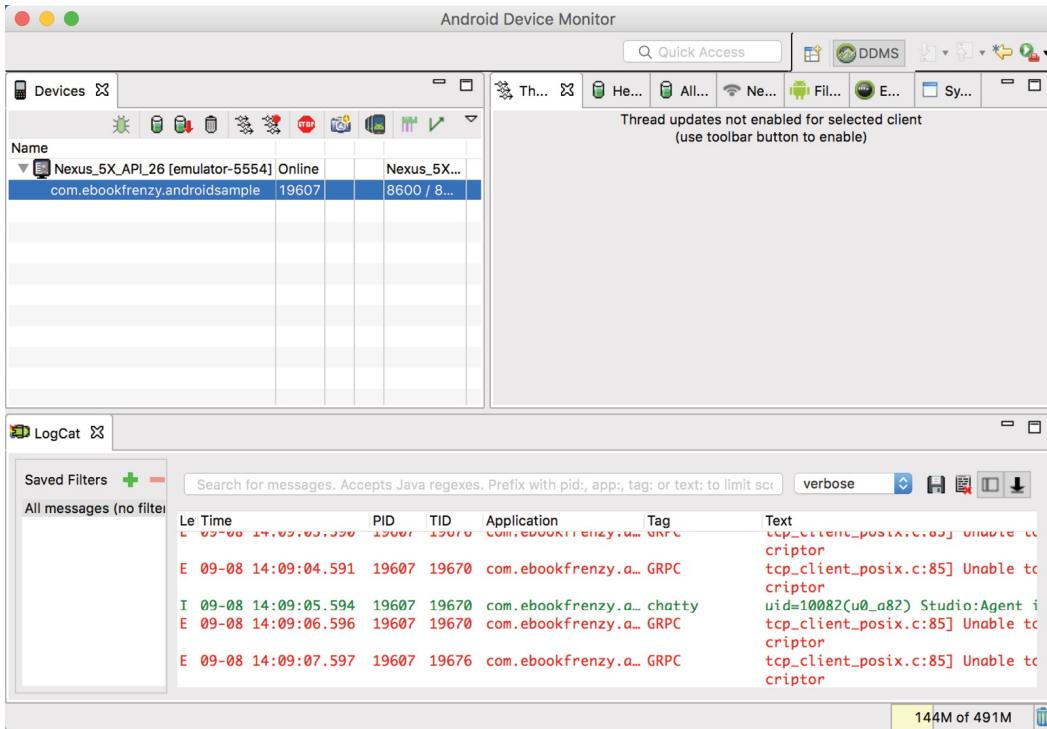


Figure 5-13

5.7 AVD Command-line Creation

As previously discussed, in addition to the graphical user interface it is also possible to create a new AVD directly from the command-line. This is achieved using the *avdmanager* tool in conjunction with some command-line options. Once initiated, the tool will prompt for additional information before creating the new AVD.

Assuming that the system has been configured such that the Android SDK *tools* directory is included in the PATH environment variable, a list of available targets for the new AVD may be obtained by issuing the following command in a terminal or command window:

```
avdmanager list targets
```

The resulting output from the above command will contain a list of Android SDK versions that are available on the system. For example:

```
Available Android targets:
```

```
-----
```

```
id: 1 or "android-25"
```

```
Name: Android API 25
```

```
Type: Platform
```

```
API level: 25
```

```
Revision: 3
-----
id: 2 or "android-26"
Name: Android API 26
Type: Platform
API level: 26
Revision: 1
```

The avdmanager tool also allows new AVD instances to be created from the command line. For example, to create a new AVD named *Nexus9* using the target ID for the Android API level 26 device using the x86 ABI, the following command may be used:

```
avdmanager create avd -n Nexus9 -k "system-images;android-26;google_apis;x86"
```

The android tool will create the new AVD to the specifications required for a basic Android 8 device, also providing the option to create a custom configuration to match the specification of a specific device if required. Once a new AVD has been created from the command line, it may not show up in the Android Device Manager tool until the *Refresh* button is clicked.

In addition to the creation of new AVDs, a number of other tasks may be performed from the command line. For example, a list of currently available AVDs may be obtained using the *list avd* command line arguments:

```
avdmanager list avd
```

Available Android Virtual Devices:

```
Name: Nexus_5X_API_26
Device: Nexus 5X (Google)
Path: /Users/neilsmyth/.android/avd/Nexus_5X_API_26.avd
Target: Google Play (Google Inc.)
Based on: Android 8.0 (Oreo) Tag/ABI: google_apis_playstore,
Skin: nexus_5x
Sdcard: 100M
```

Similarly, to delete an existing AVD, simply use the *delete* option as follows:

```
avdmanager delete avd -n <avd name>
```

5.8 Android Virtual Device Configuration Files

By default, the files associated with an AVD are stored in the *.android/avd* sub-directory of the user's home directory, the structure of which is as follows (where *<avd name>* is replaced by the name assigned to the AVD):

```
<avd name>.avd/config.ini
```

```
<avd name>.avd/userdata.img  
<avd name>.ini
```

The *config.ini* file contains the device configuration settings such as display dimensions and memory specified during the AVD creation process. These settings may be changed directly within the configuration file and will be adopted by the AVD when it is next invoked.

The *<avd name>.ini* file contains a reference to the target Android SDK and the path to the AVD files. Note that a change to the *image.sysdir* value in the *config.ini* file will also need to be reflected in the *target* value of this file.

5.9 Moving and Renaming an Android Virtual Device

The current name or the location of the AVD files may be altered from the command line using the *avdmanager* tool's *move avd* argument. For example, to rename an AVD named Nexus9 to Nexus9B, the following command may be executed:

```
avdmanager move avd -n Nexus9 -r Nexus9B
```

To physically relocate the files associated with the AVD, the following command syntax should be used:

```
avdmanager move avd -n <avd name> -p <path to new location>
```

For example, to move an AVD from its current file system location to /tmp/Nexus9Test:

```
avdmanager move avd -n Nexus9 -p /tmp/Nexus9Test
```

Note that the destination directory must not already exist prior to executing the command to move an AVD.

5.10 Summary

A typical application development process follows a cycle of coding, compiling and running in a test environment. Android applications may be tested on either a physical Android device or using an Android Virtual Device (AVD) emulator. AVDs are created and managed using the Android AVD Manager tool which may be used either as a command line tool or using a graphical user interface. When creating an AVD to simulate a specific Android device model it is important that the virtual device be configured with a hardware specification that matches that of the physical device.

6. Using and Configuring the Android Studio AVD Emulator

The Android Virtual Device (AVD) emulator environment bundled with Android Studio 1.x was an uncharacteristically weak point in an otherwise reputable application development environment. Regarded by many developers as slow, inflexible and unreliable, the emulator was long overdue for an overhaul. Fortunately, Android Studio 2 introduced an enhanced emulator environment providing significant improvements in terms of configuration flexibility and overall performance and further enhancements have been made for Android Studio 3.

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.

6.1 The Emulator Environment

When launched, 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 6-1](#) this is a Nexus 5X device):



Figure 6-1

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

6.2 The Emulator Toolbar Options

The emulator toolbar ([Figure 6-2](#)) provides access to a range of options relating to the appearance and behavior of the emulator environment.

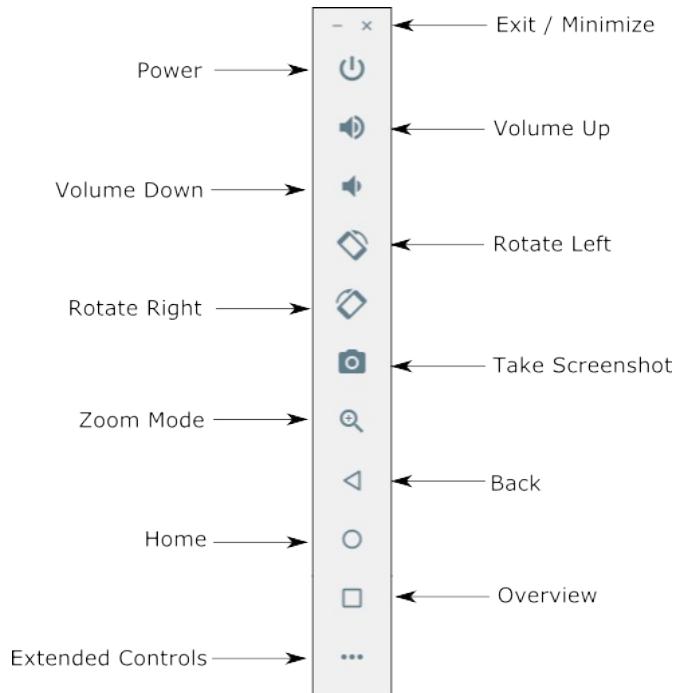


Figure 6-2

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.
- **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** – Simulates selection of the standard Android “Back” button. As with the Home and Overview buttons outlined below, the same results can be achieved by selecting the actual buttons on the emulator screen.
- **Home** – Simulates selection of the standard Android “Home” button.
- **Overview** – Simulates selection of the standard Android “Overview” button which displays the currently running apps on the device.
- **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.

6.3 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.

6.4 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.

6.5 Extended Control Options

The extended controls toolbar button displays the panel illustrated in [Figure 6-3](#). By default, the location settings will be displayed. Selecting a different category from the left-hand panel will display the corresponding group of controls:

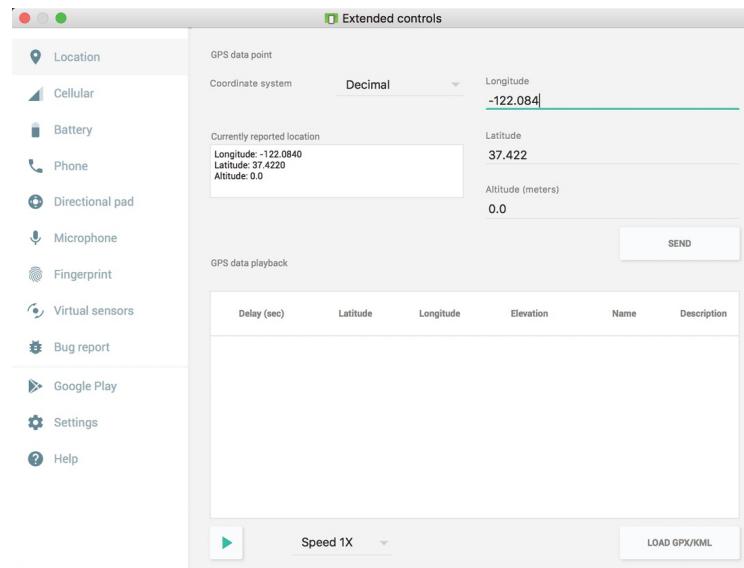


Figure 6-3

6.5.1 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.

A single location is transmitted to the emulator when the *Send* button is clicked. The transmission of GPS data points begins once the “play” button located beneath the data table is selected. The speed at which the GPS data points are fed to the emulator can be controlled using the speed menu adjacent to the play button.

6.5.2 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.

6.5.3 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.

6.5.4 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 the way in which 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.

6.5.5 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.

6.5.6 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.

6.5.7 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.

6.5.8 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.

6.5.9 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 to configure the emulator window to appear on top of other windows on the desktop.

6.5.10 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.

6.6 Drag and Drop Support

An Android application is packaged into an APK file when it is built. When Android Studio built and ran the `AndroidSample` app created earlier in this book, for example, the application was compiled and packaged into an APK file. That APK file was then transferred to the emulator and launched.

The Android Studio emulator also supports installation of apps by dragging and dropping the corresponding APK file onto the emulator window. To experience this in action, start the emulator, open Settings and select the *Apps & notifications* option followed by the *App Info* option on the subsequent screen. Within the list of installed apps, locate and select the `AndroidSample` app and, in the app detail screen, uninstall the app from the emulator.

Open the file system navigation tool for your operating system (e.g. Windows Explorer for Windows or Finder for macOS) and navigate to the folder containing the `AndroidSample` project. Within this folder locate the `app/build/outputs/apk/debug` subfolder. This folder should contain an APK file named `app-debug.apk`. Drag this file and drop it onto the emulator window. The dialog shown in ([Figure 6-4](#)) will subsequently appear as the APK file is installed.

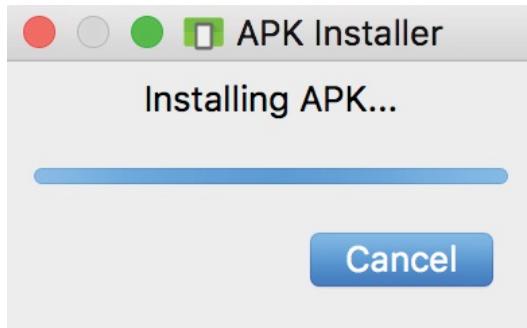


Figure 6-4

Once the APK file installation has completed, locate the app on the device and click on it to launch it.

In addition to APK files, any other type of file such as image, video or data files can be installed onto the emulator using this drag and drop feature. Such files are added to the SD card storage area of the emulator where they may subsequently be accessed from within app code.

6.7 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 number) 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 number 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:

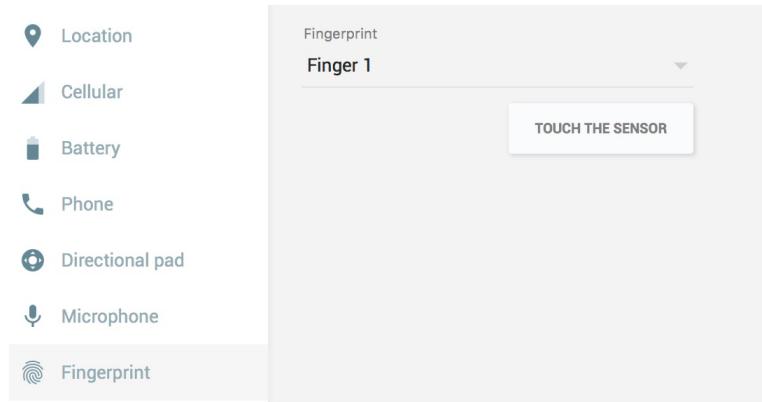


Figure 6-5

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:

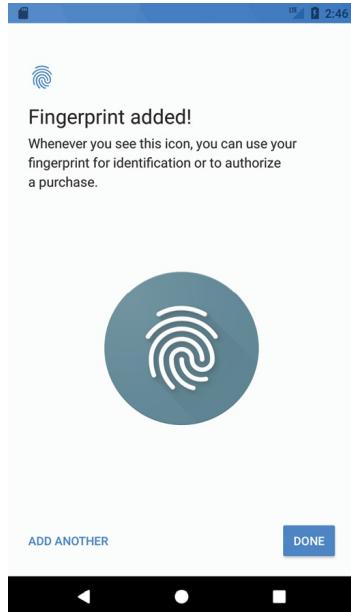


Figure 6-6

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. The topic of building fingerprint authentication into an Android app is covered in detail in the chapter entitled "[An Android Fingerprint Authentication Tutorial](#)".

6.8 Summary

Android Studio 3 contains a new and improved 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

7. Testing Android Studio Apps on a Physical Android Device

While much can be achieved by testing applications using an Android Virtual Device (AVD), there is no substitute for performing real world application testing on a physical Android device and there are a number of Android features that are only available on physical Android devices.

Communication with both AVD instances and connected Android devices is handled by the *Android Debug Bridge (ADB)*. In this chapter we will work through the steps to configure the adb environment to enable application testing on a physical Android device with macOS, Windows and Linux based systems.

7.1 An Overview of the Android Debug Bridge (ADB)

The primary purpose of the ADB is to facilitate interaction between a development system, in this case Android Studio, and both AVD emulators and physical Android devices for the purposes of running and debugging applications.

The ADB consists of a client, a server process running in the background on the development system and a daemon background process running in either AVDs or real Android devices such as phones and tablets.

The ADB client can take a variety of forms. For example, a client is provided in the form of a command-line tool named *adb* located in the Android SDK *platform-tools* sub-directory. Similarly, Android Studio also has a built-in client.

A variety of tasks may be performed using the *adb* command-line tool. For example, a listing of currently active virtual or physical devices may be obtained using the *devices* command-line argument. The following command output indicates the presence of an AVD on the system but no physical devices:

```
$ adb devices
List of devices attached
emulator-5554    device
```

7.2 Enabling ADB on Android based Devices

Before ADB can connect to an Android device, that device must first be configured to allow the connection. On phone and tablet devices running Android 6.0 or later, the steps to achieve this are as follows:

1. Open the Settings app on the device and select the *About tablet* or *About phone* option.
2. On the *About* screen, scroll down to the *Build number* field ([Figure 7-1](#)) and tap on it seven times until a message appears indicating that developer mode has been enabled.

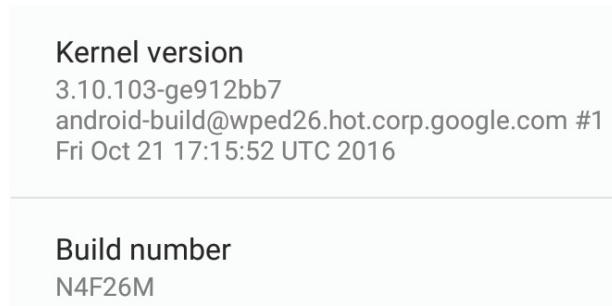


Figure 7-1

3. Return to the main Settings screen and note the appearance of a new option titled Developer options. Select this option and locate the setting on the developer screen entitled USB debugging. Enable the switch next to this item as illustrated in [Figure 7-2](#):



Figure 7-2

4. Swipe downward from the top of the screen to display the notifications panel ([Figure 7-3](#)) and note that the device is currently connected for debugging.

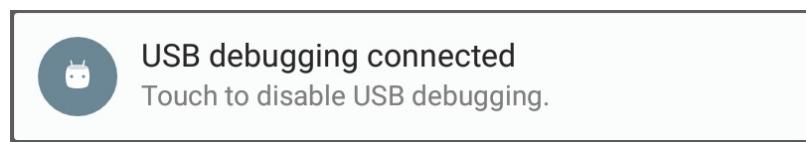


Figure 7-3

At this point, the device is now configured to accept debugging connections from adb on the development system. All that remains is to configure the development system to detect the device when it is attached. While this is a

relatively straightforward process, the steps involved differ depending on whether the development system is running Windows, macOS or Linux. Note that the following steps assume that the Android SDK *platform-tools* directory is included in the operating system PATH environment variable as described in the chapter entitled “[Setting up an Android Studio Development Environment](#)”.

7.2.1 macOS ADB Configuration

In order to configure the ADB environment on a macOS system, connect the device to the computer system using a USB cable, open a terminal window and execute the following command to restart the adb server:

```
$ adb kill-server  
$ adb start-server  
* daemon not running. starting it now on port 5037 *  
* daemon started successfully *
```

Once the server is successfully running, execute the following command to verify that the device has been detected:

```
$ adb devices  
List of devices attached  
74CE000600000001      offline
```

If the device is listed as *offline*, go to the Android device and check for the presence of the dialog shown in [Figure 7-4](#) seeking permission to *Allow USB debugging*. Enable the checkbox next to the option that reads *Always allow from this computer*, before clicking on *OK*. Repeating the *adb devices* command should now list the device as being available:

```
List of devices attached  
015d41d4454bf80c      device
```

In the event that the device is not listed, try logging out and then back in to the macOS desktop and, if the problem persists, rebooting the system.

7.2.2 Windows ADB Configuration

The first step in configuring a Windows based development system to connect to an Android device using ADB is to install the appropriate USB drivers on the system. The USB drivers to install will depend on the model of Android Device. If you have a Google Nexus device, then it will be necessary to install and configure the Google USB Driver package on your Windows system. Detailed steps to achieve this are outlined on the following web page:

<http://developer.android.com/sdk/win-usb.html>

For Android devices not supported by the Google USB driver, it will be necessary to download the drivers provided by the device manufacturer. A listing of drivers together with download and installation information can be obtained online at:

<http://developer.android.com/tools/extras/oem-usb.html>

With the drivers installed and the device now being recognized as the correct device type, open a Command Prompt window and execute the following command:

```
adb devices
```

This command should output information about the connected device similar to the following:

```
List of devices attached  
HT4CTJT01906      offline
```

If the device is listed as *offline* or *unauthorized*, go to the device display and check for the dialog shown in [Figure 7-4](#) seeking permission to *Allow USB debugging*.

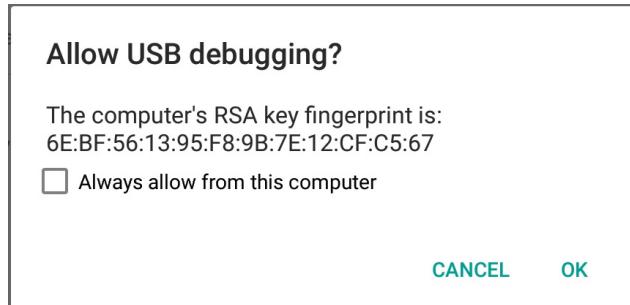


Figure 7-4

Enable the checkbox next to the option that reads *Always allow from this computer*, before clicking on OK. Repeating the *adb devices* command should now list the device as being ready:

```
List of devices attached  
HT4CTJT01906      device
```

In the event that the device is not listed, execute the following commands to restart the ADB server:

```
adb kill-server  
adb start-server
```

If the device is still not listed, try executing the following command:

```
android update adb
```

Note that it may also be necessary to reboot the system.

7.2.3 Linux adb Configuration

For the purposes of this chapter, we will once again use Ubuntu Linux as a reference example in terms of configuring adb on Linux to connect to a physical Android device for application testing.

Physical device testing on Ubuntu Linux requires the installation of a package named *android-tools-adb* which, in turn, requires that the Android Studio user be a member of the *plugdev* group. This is the default for user accounts on most Ubuntu versions and can be verified by running the *id* command. If *plugdev* group is not listed, run the following command to add your account to the group:

```
sudo usermod -aG plugdev $LOGNAME
```

After the group membership requirement has been met, the *android-tools-adb* package can be installed by executing the following command:

```
sudo apt-get install android-tools-adb
```

Once the above changes have been made, reboot the Ubuntu system. Once the system has restarted, open a Terminal window, start the adb server and check the list of attached devices:

```
$ adb start-server
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
$ adb devices
List of devices attached
015d41d4454bf80c    offline
```

If the device is listed as *offline* or *unauthorized*, go to the Android device and check for the dialog shown in [Figure 7-4](#) seeking permission to *Allow USB debugging*.

7.3 Testing the adb Connection

Assuming that the adb configuration has been successful on your chosen development platform, the next step is to try running the test application created in the chapter entitled [“Creating an Example Android App in Android Studio”](#) on the device.

Launch Android Studio, open the AndroidSample project and, once the project has loaded, click on the run button located in the Android Studio

toolbar ([Figure 7-5](#)).

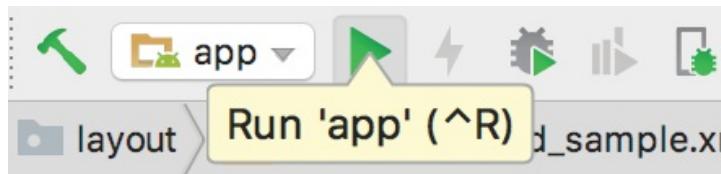


Figure 7-5

Assuming that the project has not previously been configured to run automatically in an emulator environment, the deployment target selection dialog will appear with the connected Android device listed as a currently running device. [Figure 7-6](#), for example, lists a Nexus 9 device as a suitable target for installing and executing the application.

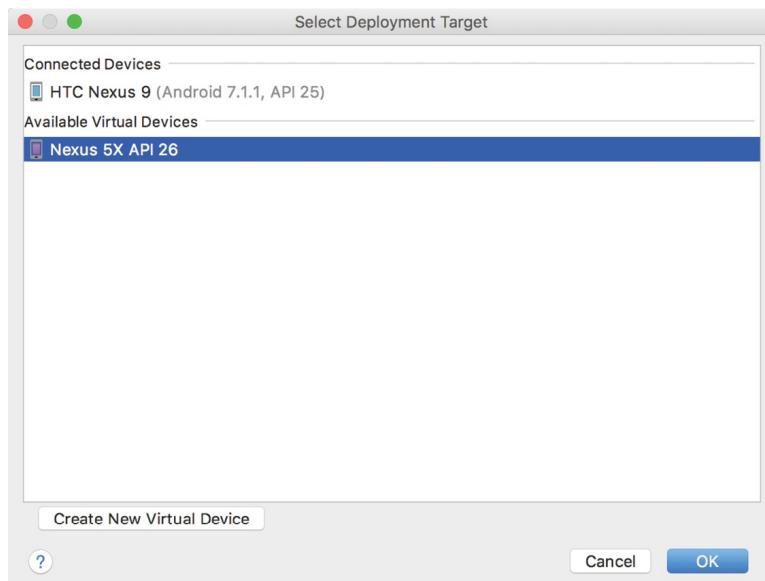


Figure 7-6

To make this the default device for testing, enable the *Use same device for future launches* option. With the device selected, click on the OK button to install and run the application on the device. As with the emulator environment, diagnostic output relating to the installation and launch of the application on the device will be logged in the Run tool window.

7.4 Summary

While the Android Virtual Device emulator provides an excellent testing environment, it is important to keep in mind that there is no real substitute for making sure an application functions correctly on a physical Android device. This, after all, is where the application will be used in the real world.

By default, however, the Android Studio environment is not configured to detect Android devices as a target testing device. It is necessary, therefore, to perform some steps in order to be able to load applications directly onto an Android device from within the Android Studio development environment. The exact steps to achieve this goal differ depending on the development platform being used. In this chapter, we have covered those steps for Linux, macOS and Windows based platforms.

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 source code file loaded:

```

c AndroidSampleActivity.java x content_android_sample.xml x
A
1 package com.ebookfrenzy.androidsample;
2
3 import ...
11
12 public class AndroidSampleActivity extends AppCompatActivity {
13
14     @Override
15     protected void onCreate(Bundle savedInstanceState) {
16         super.onCreate(savedInstanceState);
17         setContentView(R.layout.activity_android_sample);
18         Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
19         setSupportActionBar(toolbar);
20
21         FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
22         fab.setOnClickListener((view) -> {
23             Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
24                 .setAction("Action", null).show();
25         });
26
27     }
28
29     @Override
30     public boolean onCreateOptionsMenu(Menu menu) {
31         // Inflate the menu; this adds items to the action bar if it is present.
32         getMenuInflater().inflate(R.menu.menu_android_sample, menu);
33         return true;
34     }
35
36     @Override
37     public boolean onOptionsItemSelected(MenuItem item) {
38
39     }

```

Emulator: Process finished with exit code 0 (6 minutes ago)

18:28 LF+ UTF-8+ Context: <no context>

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 dropdown 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 – 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.

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 indicator at the top of the validation sidebar will change from a green check mark (no warnings or errors detected) to a yellow square (warnings detected) or red alert icon (errors have been detected). 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 8-2](#):



Figure 8-2

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 8-3](#)):

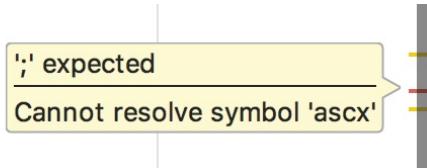


Figure 8-3

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 8-4](#)) allowing it to be viewed without the necessity to scroll to that location in the editor:



Figure 8-4

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.

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 8-5](#), for example, shows the splitter in action with the editor split into three panels:

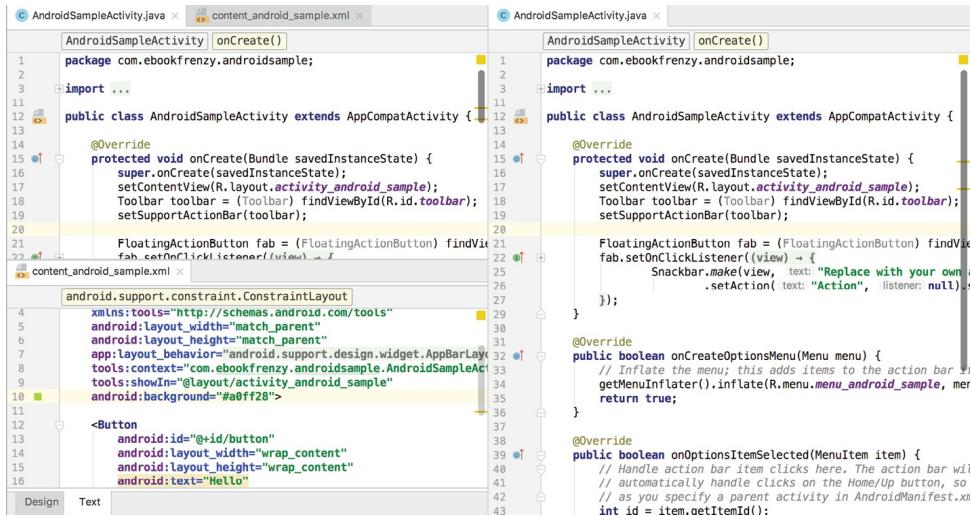


Figure 8-5

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 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 8-6](#), for example, the editor is suggesting possibilities for the beginning of a String declaration:

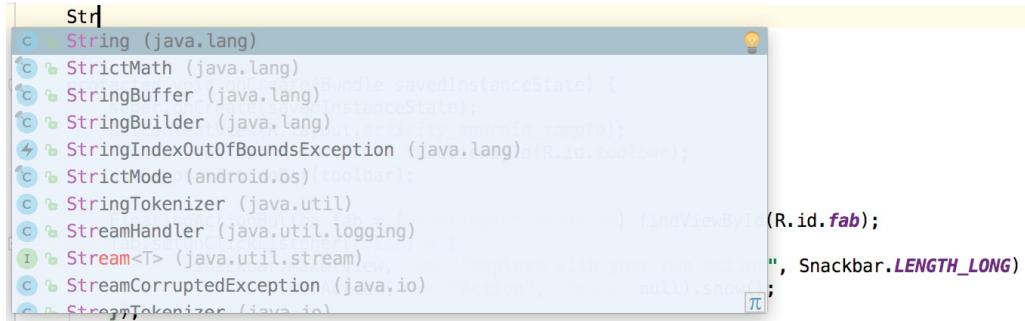


Figure 8-6

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 8-7](#):

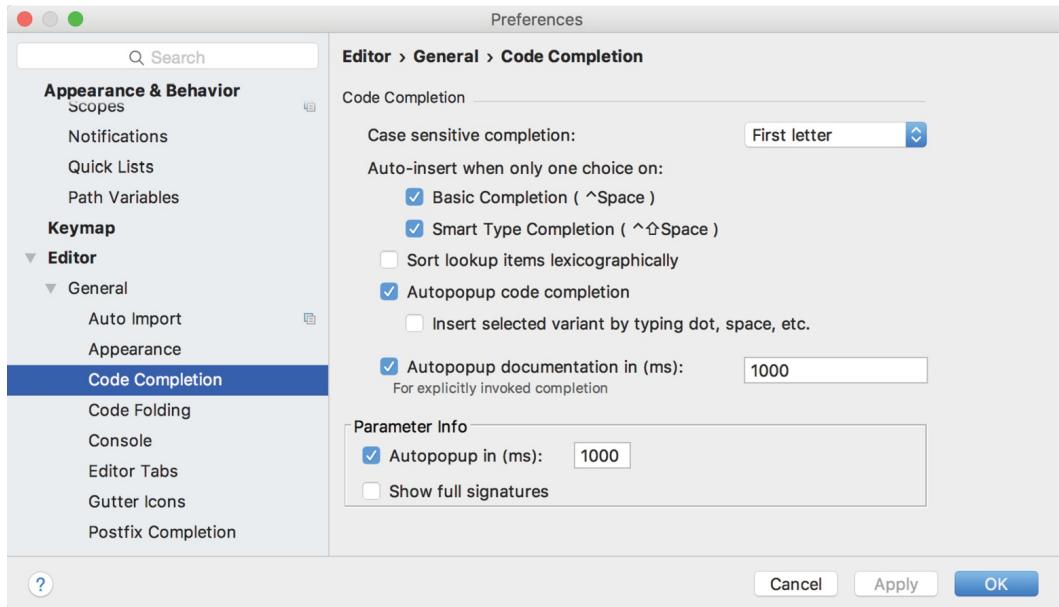


Figure 8-7

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:

```
protected void myMethod()
```

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

```
protected void 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:

```

String myButtonText = mystring.replaceAll();

```

@NonNull String regex, @NonNull String replacement

Figure 8-8

8.6 Parameter Name Hints

The code editor may be configured to display parameter name hints within method calls. [Figure 8-9](#), for example, highlights the parameter name hints within the calls to the *make()* and *setAction()* methods of the *Snackbar* class:

```

FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener(view) -> {
    Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
        .setAction("Action", null).show();
}

```

Figure 8-9

The settings for this mode may be configured by selecting the *File -> Settings* (*Android Studio -> Preferences* on macOS) menu option followed by *Editor -> Appearance* in the left-hand panel. On the Appearance screen, enable or disable the *Show parameter name hints* option. To adjust the hint settings, click on the *Configure...* button, select the programming language 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 8-10](#) 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.

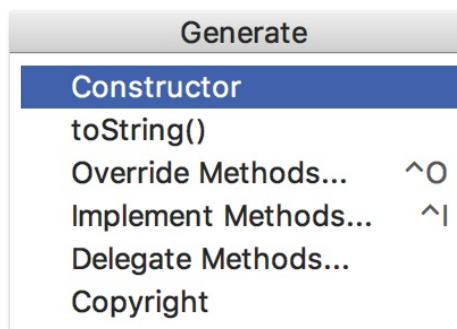


Figure 8-10

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:

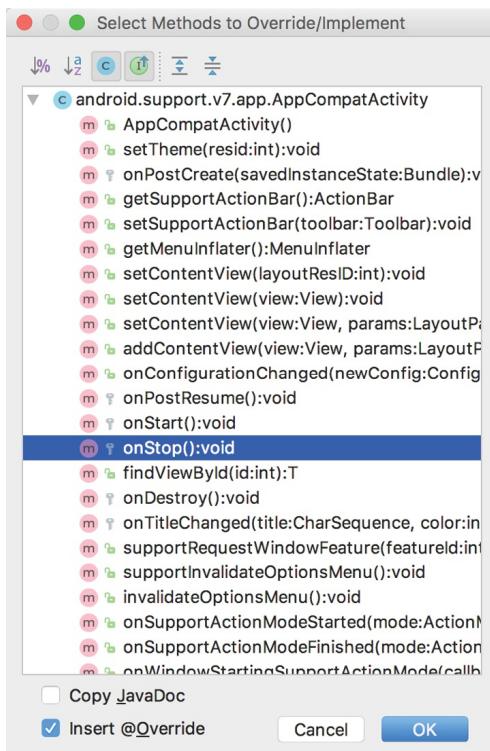


Figure 8-11

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

```
@Override
protected void 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 8-12](#), for example, highlights the start and end markers for a method declaration which is not currently folded:

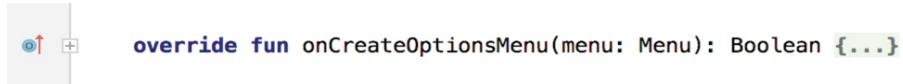


Figure 8-12

Clicking on either of these markers will fold the statement such that only the signature line is visible as shown in [Figure 8-13](#):

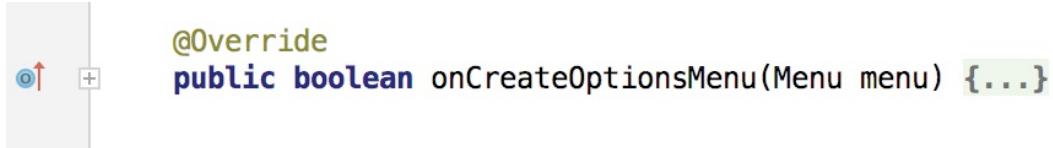


Figure 8-13

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 8-14](#). The editor will then display the lens overlay containing the folded code block:

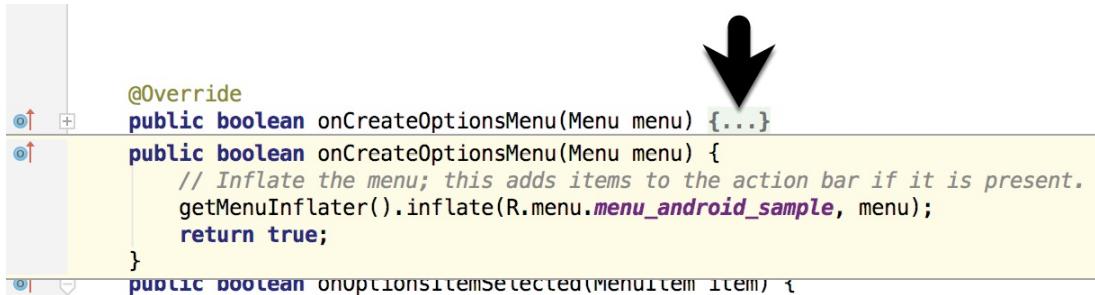


Figure 8-14

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 8-15](#)):

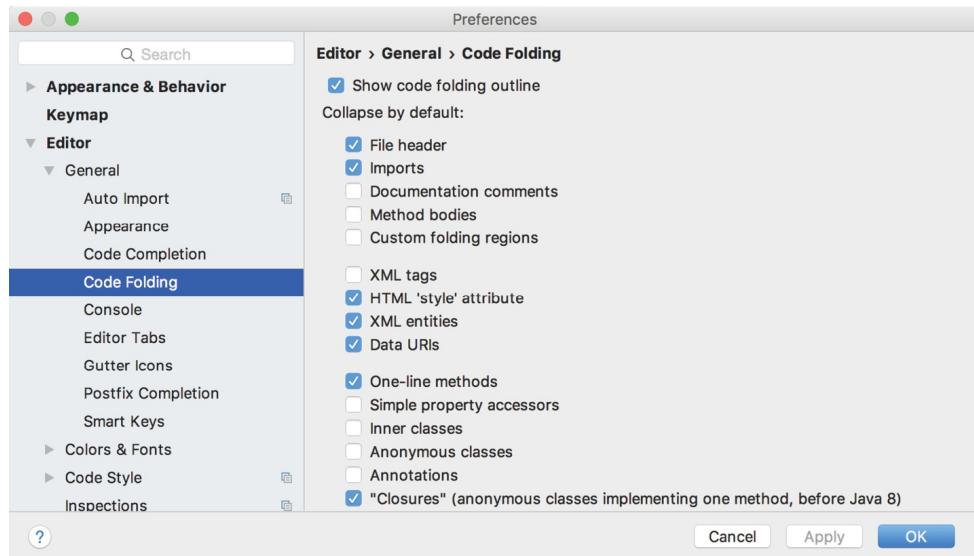


Figure 8-15

8.9 Quick Documentation Lookup

Context sensitive 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 8-16](#), for example, shows the documentation for the Android Snackbar class.



Figure 8-16

Once displayed, the documentation popup can be moved around the screen as needed. Clicking on the push pin icon located in the right-hand corner of the popup title bar will ensure that the popup remains visible once focus moves back to the editor, leaving the documentation visible as a reference while typing code.

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 web site), 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-Alt-L* on macOS) keyboard shortcut sequence. To display the *Reformat Code* dialog ([Figure 8-17](#)) use the *Ctrl-Alt-Shift-L* (*Cmd-Alt-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.

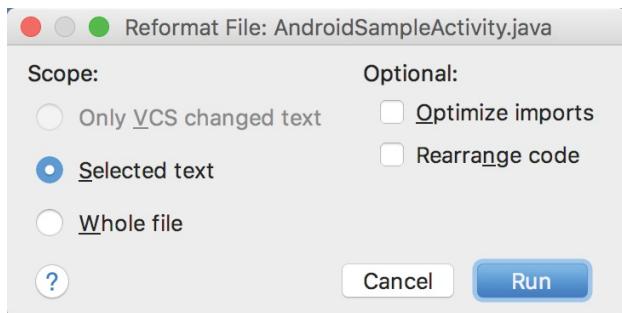


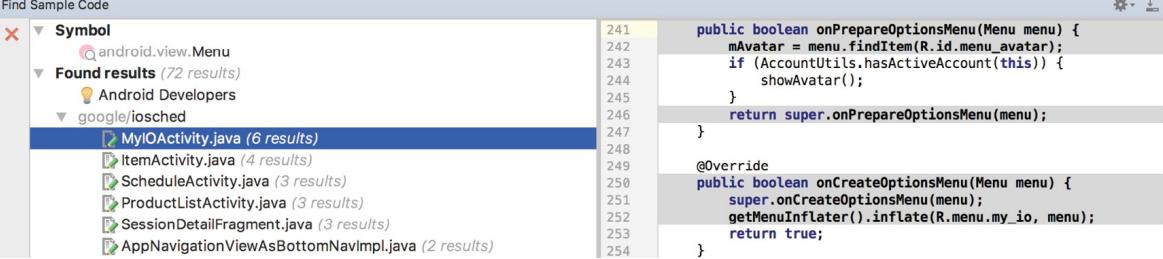
Figure 8-17

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 and, from the 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 8-18](#)) 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:



The screenshot shows the 'Find Sample Code' dialog in Android Studio. The left pane displays a tree view of search results under 'Symbol android.view.Menu'. The 'Found results' section shows 72 results, with 'MyIOActivity.java (6 results)' being the selected item. Other listed files include ItemActivity.java, ScheduleActivity.java, ProductListActivity.java, SessionDetailFragment.java, and AppNavigationViewAsBottomNavImpl.java. The right pane shows the source code for MyIOActivity.java, specifically the onPrepareOptionsMenu and onCreateOptionsMenu methods.

```
public boolean onPrepareOptionsMenu(Menu menu) {  
    mAvatar = menu.findItem(R.id.menu_avatar);  
    if (AccountUtils.hasActiveAccount(this)) {  
        showAvatar();  
    }  
    return super.onPrepareOptionsMenu(menu);  
}  
  
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
    super.onCreateOptionsMenu(menu);  
    getMenuInflater().inflate(R.menu.my_io, menu);  
    return true;  
}
```

Figure 8-18

8.12 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 and documentation lookup.

9. An Overview of the Android Architecture

So far in this book, steps have been taken to set up an environment suitable for the development of Android applications using Android Studio. An initial step has also been taken into the process of application development through the creation of a simple Android Studio application project.

Before delving further into the practical matters of Android application development, however, it is important to gain an understanding of some of the more abstract concepts of both the Android SDK and Android development in general. Gaining a clear understanding of these concepts now will provide a sound foundation on which to build further knowledge.

Starting with an overview of the Android architecture in this chapter, and continuing in the next few chapters of this book, the goal is to provide a detailed overview of the fundamentals of Android development.

9.1 The Android Software Stack

Android is structured in the form of a software stack comprising applications, an operating system, run-time environment, middleware, services and libraries. This architecture can, perhaps, best be represented visually as outlined in [Figure 9-1](#). Each layer of the stack, and the corresponding elements within each layer, are tightly integrated and carefully tuned to provide the optimal application development and execution environment for mobile devices. The remainder of this chapter will work through the different layers of the Android stack, starting at the bottom with the Linux Kernel.

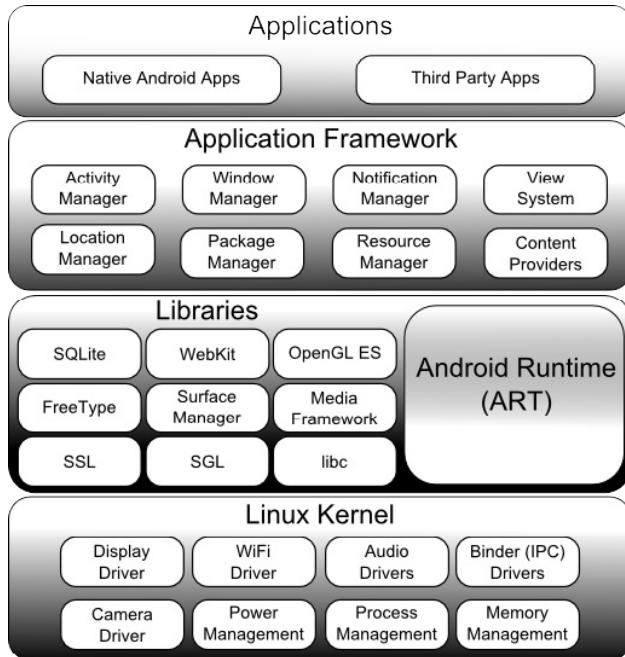


Figure 9-1

9.2 The Linux Kernel

Positioned at the bottom of the Android software stack, the Linux Kernel provides a level of abstraction between the device hardware and the upper layers of the Android software stack. Based on Linux version 2.6, the kernel provides preemptive multitasking, low-level core system services such as memory, process and power management in addition to providing a network stack and device drivers for hardware such as the device display, Wi-Fi and audio.

The original Linux kernel was developed in 1991 by Linus Torvalds and was combined with a set of tools, utilities and compilers developed by Richard Stallman at the Free Software Foundation to create a full operating system referred to as GNU/Linux. Various Linux distributions have been derived from these basic underpinnings such as Ubuntu and Red Hat Enterprise Linux.

It is important to note, however, that Android uses only the Linux kernel. That said, it is worth noting that the Linux kernel was originally developed for use in traditional computers in the form of desktops and servers. In fact, Linux is now most widely deployed in mission critical enterprise server environments. It is a testament to both the power of today's mobile devices and the efficiency and performance of the Linux kernel that we find this

software at the heart of the Android software stack.

9.3 Android Runtime – ART

When an Android app is built within Android Studio it is compiled into an intermediate bytecode format (referred to as DEX format). When the application is subsequently loaded onto the device, the Android Runtime (ART) uses a process referred to as Ahead-of-Time (AOT) compilation to translate the bytecode down to the native instructions required by the device processor. This format is known as Executable and Linkable Format (ELF).

Each time the application is subsequently launched, the ELF executable version is run, resulting in faster application performance and improved battery life.

This contrasts with the Just-in-Time (JIT) compilation approach used in older Android implementations whereby the bytecode was translated within a virtual machine (VM) each time the application was launched.

9.4 Android Libraries

In addition to a set of standard Java development libraries (providing support for such general purpose tasks as string handling, networking and file manipulation), the Android development environment also includes the Android Libraries. These are a set of Java-based libraries that are specific to Android development. Examples of libraries in this category include the application framework libraries in addition to those that facilitate user interface building, graphics drawing and database access.

A summary of some key core Android libraries available to the Android developer is as follows:

- **android.app** – Provides access to the application model and is the cornerstone of all Android applications.
- **android.content** – Facilitates content access, publishing and messaging between applications and application components.
- **android.database** – Used to access data published by content providers and includes SQLite database management classes.
- **android.graphics** – A low-level 2D graphics drawing API including colors, points, filters, rectangles and canvases.

- **android.hardware** – Presents an API providing access to hardware such as the accelerometer and light sensor.
- **android.opengl** – A Java interface to the OpenGL ES 3D graphics rendering API.
- **android.os** – Provides applications with access to standard operating system services including messages, system services and inter-process communication.
- **android.media** – Provides classes to enable playback of audio and video.
- **android.net** – A set of APIs providing access to the network stack. Includes *android.net.wifi*, which provides access to the device's wireless stack.
- **android.print** – Includes a set of classes that enable content to be sent to configured printers from within Android applications.
- **android.provider** – A set of convenience classes that provide access to standard Android content provider databases such as those maintained by the calendar and contact applications.
- **android.text** – Used to render and manipulate text on a device display.
- **android.util** – A set of utility classes for performing tasks such as string and number conversion, XML handling and date and time manipulation.
- **android.view** – The fundamental building blocks of application user interfaces.
- **android.widget** - A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
- **android.webkit** – A set of classes intended to allow web-browsing capabilities to be built into applications.

Having covered the Java-based libraries in the Android runtime, it is now time to turn our attention to the C/C++ based libraries contained in this layer of the Android software stack.

9.4.1 C/C++ Libraries

The Android runtime core libraries outlined in the preceding section are Java-based and provide the primary APIs for developers writing Android applications. It is important to note, however, that the core libraries do not perform much of the actual work and are, in fact, essentially Java “wrappers” around a set of C/C++ based libraries. When making calls, for example, to the *android.opengl* library to draw 3D graphics on the device display, the library actually ultimately makes calls to the *OpenGL ES* C++ library which, in turn, works with the underlying Linux kernel to perform the drawing tasks.

C/C++ libraries are included to fulfill a wide and diverse range of functions including 2D and 3D graphics drawing, Secure Sockets Layer (SSL) communication, SQLite database management, audio and video playback, bitmap and vector font rendering, display subsystem and graphic layer management and an implementation of the standard C system library (libc).

In practice, the typical Android application developer will access these libraries solely through the Java based Android core library APIs. In the event that direct access to these libraries is needed, this can be achieved using the Android Native Development Kit (NDK), the purpose of which is to call the native methods of non-Java or Kotlin programming languages (such as C and C++) from within Java code using the Java Native Interface (JNI).

9.5 Application Framework

The Application Framework is a set of services that collectively form the environment in which Android applications run and are managed. This framework implements the concept that Android applications are constructed from reusable, interchangeable and replaceable components. This concept is taken a step further in that an application is also able to *publish* its capabilities along with any corresponding data so that they can be found and reused by other applications.

The Android framework includes the following key services:

- **Activity Manager** – Controls all aspects of the application lifecycle and activity stack.
- **Content Providers** – Allows applications to publish and share data with other applications.
- **Resource Manager** – Provides access to non-code embedded resources

such as strings, color settings and user interface layouts.

- **Notifications Manager** – Allows applications to display alerts and notifications to the user.
- **View System** – An extensible set of views used to create application user interfaces.
- **Package Manager** – The system by which applications are able to find out information about other applications currently installed on the device.
- **Telephony Manager** – Provides information to the application about the telephony services available on the device such as status and subscriber information.
- **Location Manager** – Provides access to the location services allowing an application to receive updates about location changes.

9.6 Applications

Located at the top of the Android software stack are the applications. These comprise both the native applications provided with the particular Android implementation (for example web browser and email applications) and the third party applications installed by the user after purchasing the device.

9.7 Summary

A good Android development knowledge foundation requires an understanding of the overall architecture of Android. Android is implemented in the form of a software stack architecture consisting of a Linux kernel, a runtime environment and corresponding libraries, an application framework and a set of applications. Applications are predominantly written in Java or Kotlin and compiled down to bytecode format within the Android Studio build environment. When the application is subsequently installed on a device, this bytecode is compiled down by the Android Runtime (ART) to the native format used by the CPU. The key goals of the Android architecture are performance and efficiency, both in application execution and in the implementation of reuse in application design.

10. The Anatomy of an Android Application

Regardless of your prior programming experiences, be it Windows, macOS, Linux or even iOS based, the chances are good that Android development is quite unlike anything you have encountered before.

The objective of this chapter, therefore, is to provide an understanding of the high-level concepts behind the architecture of Android applications. In doing so, we will explore in detail both the various components that can be used to construct an application and the mechanisms that allow these to work together to create a cohesive application.

10.1 Android Activities

Those familiar with object-oriented programming languages such as Java, Kotlin, C++ or C# will be familiar with the concept of encapsulating elements of application functionality into classes that are then instantiated as objects and manipulated to create an application. Since Android applications are written in Java and Kotlin, this is still very much the case. Android, however, also takes the concept of re-usable components to a higher level.

Android applications are created by bringing together one or more components known as *Activities*. An activity is a single, standalone module of application functionality that usually correlates directly to a single user interface screen and its corresponding functionality. An appointments application might, for example, have an activity screen that displays appointments set up for the current day. The application might also utilize a second activity consisting of a screen where new appointments may be entered by the user.

Activities are intended as fully reusable and interchangeable building blocks that can be shared amongst different applications. An existing email application, for example, might contain an activity specifically for composing and sending an email message. A developer might be writing an application that also has a requirement to send an email message. Rather than develop an email composition activity specifically for the new application, the developer can simply use the activity from the existing email application.

Activities are created as subclasses of the Android *Activity* class and must be implemented so as to be entirely independent of other activities in the application. In other words, a shared activity cannot rely on being called at a known point in a program flow (since other applications may make use of the activity in unanticipated ways) and one activity cannot directly call methods or access instance data of another activity. This, instead, is achieved using *Intents* and *Content Providers*.

By default, an activity cannot return results to the activity from which it was invoked. If this functionality is required, the activity must be specifically started as a *sub-activity* of the originating activity.

10.2 Android Intents

Intents are the mechanism by which one activity is able to launch another and implement the flow through the activities that make up an application. Intents consist of a description of the operation to be performed and, optionally, the data on which it is to be performed.

Intents can be *explicit*, in that they request the launch of a specific activity by referencing the activity by class name, or *implicit* by stating either the type of action to be performed or providing data of a specific type on which the action is to be performed. In the case of implicit intents, the Android runtime will select the activity to launch that most closely matches the criteria specified by the Intent using a process referred to as *Intent Resolution*.

10.3 Broadcast Intents

Another type of Intent, the *Broadcast Intent*, is a system wide intent that is sent out to all applications that have registered an “interested” *Broadcast Receiver*. The Android system, for example, will typically send out Broadcast Intents to indicate changes in device status such as the completion of system start up, connection of an external power source to the device or the screen being turned on or off.

A Broadcast Intent can be *normal* (asynchronous) in that it is sent to all interested Broadcast Receivers at more or less the same time, or *ordered* in that it is sent to one receiver at a time where it can be processed and then either aborted or allowed to be passed to the next Broadcast Receiver.

10.4 Broadcast Receivers

Broadcast Receivers are the mechanism by which applications are able to respond to Broadcast Intents. A Broadcast Receiver must be registered by an application and configured with an *Intent Filter* to indicate the types of broadcast in which it is interested. When a matching intent is broadcast, the receiver will be invoked by the Android runtime regardless of whether the application that registered the receiver is currently running. The receiver then has 5 seconds in which to complete any tasks required of it (such as launching a Service, making data updates or issuing a notification to the user) before returning. Broadcast Receivers operate in the background and do not have a user interface.

10.5 Android Services

Android Services are processes that run in the background and do not have a user interface. They can be started and subsequently managed from activities, Broadcast Receivers or other Services. Android Services are ideal for situations where an application needs to continue performing tasks but does not necessarily need a user interface to be visible to the user. Although Services lack a user interface, they can still notify the user of events using notifications and *toasts* (small notification messages that appear on the screen without interrupting the currently visible activity) and are also able to issue Intents.

Services are given a higher priority by the Android runtime than many other processes and will only be terminated as a last resort by the system in order to free up resources. In the event that the runtime does need to kill a Service, however, it will be automatically restarted as soon as adequate resources once again become available. A Service can reduce the risk of termination by declaring itself as needing to run in the *foreground*. This is achieved by making a call to `startForeground()`. This is only recommended for situations where termination would be detrimental to the user experience (for example, if the user is listening to audio being streamed by the Service).

Example situations where a Service might be a practical solution include, as previously mentioned, the streaming of audio that should continue when the application is no longer active, or a stock market tracking application that needs to notify the user when a share hits a specified price.

10.6 Content Providers

Content Providers implement a mechanism for the sharing of data between applications. Any application can provide other applications with access to its underlying data through the implementation of a Content Provider including the ability to add, remove and query the data (subject to permissions). Access to the data is provided via a Universal Resource Identifier (URI) defined by the Content Provider. Data can be shared in the form of a file or an entire SQLite database.

The native Android applications include a number of standard Content Providers allowing applications to access data such as contacts and media files.

The Content Providers currently available on an Android system may be located using a *Content Resolver*.

10.7 The Application Manifest

The glue that pulls together the various elements that comprise an application is the Application Manifest file. It is within this XML based file that the application outlines the activities, services, broadcast receivers, data providers and permissions that make up the complete application.

10.8 Application Resources

In addition to the manifest file and the Dex files that contain the byte code, an Android application package will also typically contain a collection of *resource files*. These files contain resources such as the strings, images, fonts and colors that appear in the user interface together with the XML representation of the user interface layouts. By default, these files are stored in the /res sub-directory of the application project's hierarchy.

10.9 Application Context

When an application is compiled, a class named *R* is created that contains references to the application resources. The application manifest file and these resources combine to create what is known as the *Application Context*. This context, represented by the Android *Context* class, may be used in the application code to gain access to the application resources at runtime. In addition, a wide range of methods may be called on an application's context to gather information and make changes to the application's environment at runtime.

10.1 Summary

A number of different elements can be brought together in order to create an Android application. In this chapter, we have provided a high-level overview of Activities, Services, Intents and Broadcast Receivers together with an overview of the manifest file and application resources.

Maximum reuse and interoperability are promoted through the creation of individual, standalone modules of functionality in the form of activities and intents, while data sharing between applications is achieved by the implementation of content providers.

While activities are focused on areas where the user interacts with the application (an activity essentially equating to a single user interface screen), background processing is typically handled by Services and Broadcast Receivers.

The components that make up the application are outlined for the Android runtime system in a manifest file which, combined with the application's resources, represents the application's context.

Much has been covered in this chapter that is most likely new to the average developer. Rest assured, however, that extensive exploration and practical use of these concepts will be made in subsequent chapters to ensure a solid knowledge foundation on which to build your own applications.

11. Understanding Android Application and Activity Lifecycles

In the preceding few 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. In order 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.

11.1 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 in order 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.

11.2 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 11-1](#), a process can be in one of the following five states at any given time:

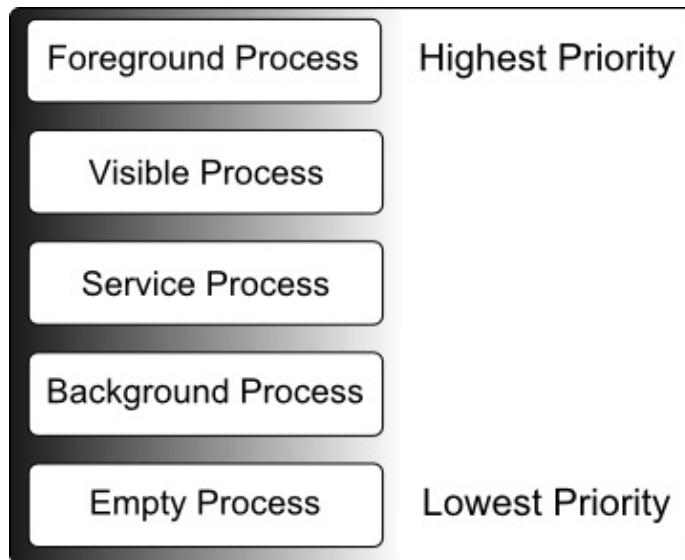


Figure 11-1

11.2.1 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.

11.2.2 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.

11.2.3 Service Process

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

11.2.4 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 in the event that 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.

11.2.5 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.

11.3 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.

11.4 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*.

11.5 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 11-2](#).

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. In the event that 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.

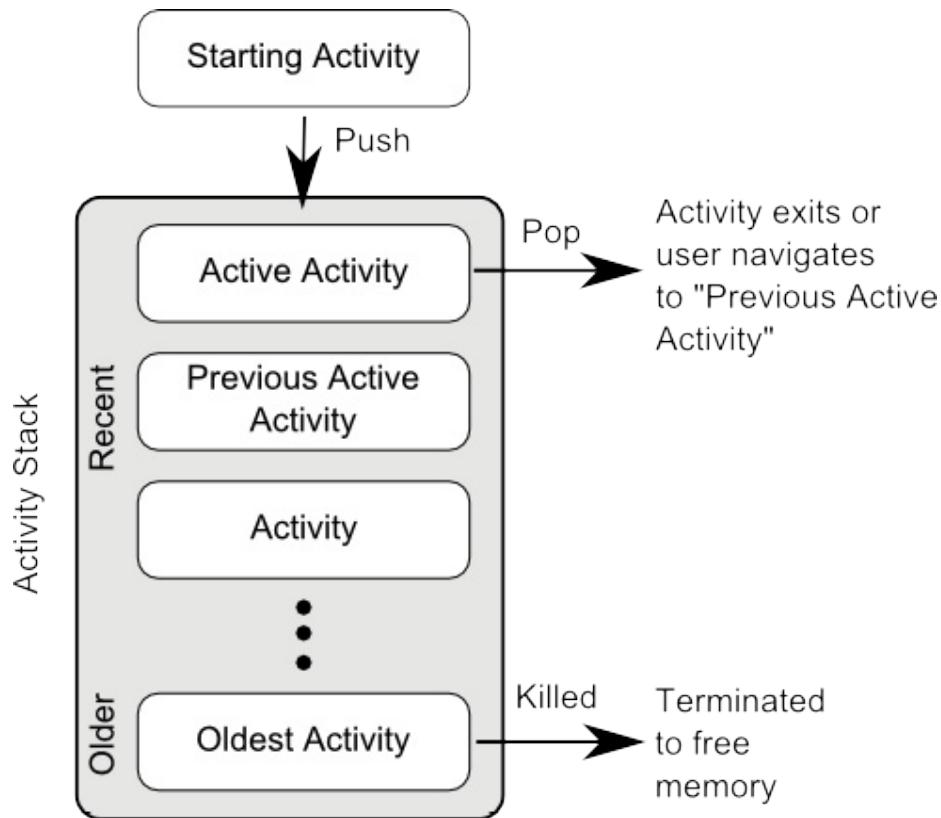


Figure 11-2

11.6 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.

11.7 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 in order 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.

11.8 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. The steps involved in gracefully handling state changes will be covered in detail in the next chapter entitled [“Handling Android Activity State Changes”](#).

11.9 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 in order 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, a topic that is covered in the next chapter.

12. Handling Android Activity State Changes

Based on the information outlined in the chapter entitled “[Understanding Android Application and Activity Lifecycles](#)” it is now evident that the activities that make up an application pass through a variety of different states during the course of the application’s lifespan. The change from one state to the other is imposed by the Android runtime system and is, therefore, largely beyond the control of the activity itself. That said, in most instances the runtime will provide the activity in question with a notification of the impending state change, thereby giving it time to react accordingly. In most cases, this will involve saving or restoring data relating to the state of the activity and its user interface.

The primary objective of this chapter is to provide a high-level overview of the ways in which an activity may be notified of a state change and to outline the areas where it is advisable to save or restore state information. Having covered this information, the chapter will then touch briefly on the subject of *activity lifetimes*.

12.1 The Activity Class

With few exceptions, activities in an application are created as subclasses of either the Android *Activity* class, or another class that is, itself, subclassed from the *Activity* class (for example the *AppCompatActivity* or *FragmentActivity* classes).

Consider, for example, the simple *AndroidSample* project created in “[Creating an Example Android App in Android Studio](#)”. Load this project into the Android Studio environment and locate the *AndroidSampleActivity.java* file (located in *app -> java -> com.<your domain>.androidsample*). Having located the file, double-click on it to load it into the editor where it should read as follows:

```
package com.ebookfrenzy.androidsample;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
```

```
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;

public class AndroidSampleActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_android_sample);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab =
            (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Snackbar.make(view,
                    "Replace with your own action", Snackbar.LENGTH_LONG)
                    .setAction("Action", null).show();
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it
        // is present.
        getMenuInflater().inflate(R.menu.menu_android_sample, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
    }
}
```

```

        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}

```

When the project was created, we instructed Android Studio also to create an initial activity named *AndroidSampleActivity*. As is evident from the above code, the *AndroidSampleActivity* class is a subclass of the *AppCompatActivity* class.

A review of the reference documentation for the *AppCompatActivity* class would reveal that it is itself a subclass of the *Activity* class. This can be verified within the Android Studio editor using the *Hierarchy* tool window. With the *AndroidSampleActivity.java* file loaded into the editor, click on *AppCompatActivity* in the *class* declaration line and press the *Ctrl-H* keyboard shortcut. The hierarchy tool window will subsequently appear displaying the class hierarchy for the selected class. As illustrated in [Figure 12-1](#), *AppCompatActivity* is clearly subclassed from the *FragmentActivity* class which is itself ultimately a subclass of the *Activity* class:

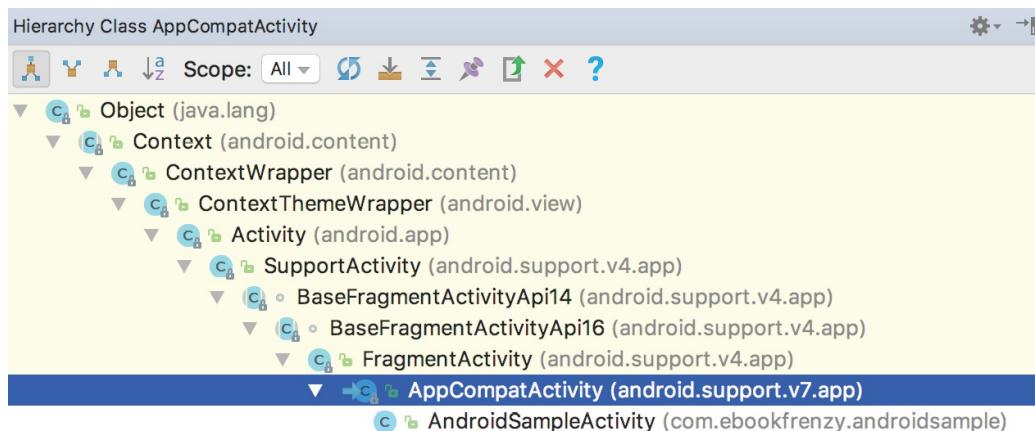


Figure 12-1

The *Activity* class and its subclasses contain a range of methods that are intended to be called by the Android runtime to notify an activity that its state is changing. For the purposes of this chapter, we will refer to these as the *activity lifecycle methods*. An activity class simply needs to *override* these methods and implement the necessary functionality within them in order to react accordingly to state changes.

One such method is named *onCreate()* and, turning once again to the above code fragment, we can see that this method has already been overridden and implemented for us in the *AndroidSampleActivity* class. In a later section we will explore in detail both *onCreate()* and the other relevant lifecycle methods of the Activity class.

12.2 Dynamic State vs. Persistent State

A key objective of Activity lifecycle management is ensuring that the state of the activity is saved and restored at appropriate times. When talking about *state* in this context we mean the data that is currently being held within the activity and the appearance of the user interface. The activity might, for example, maintain a data model in memory that needs to be saved to a database, content provider or file. Such state information, because it persists from one invocation of the application to another, is referred to as the *persistent state*.

The appearance of the user interface (such as text entered into a text field but not yet committed to the application's internal data model) is referred to as the *dynamic state*, since it is typically only retained during a single invocation of the application (and also referred to as *user interface state* or *instance state*).

Understanding the differences between these two states is important because both the ways they are saved, and the reasons for doing so, differ.

The purpose of saving the persistent state is to avoid the loss of data that may result from an activity being killed by the runtime system while in the background. The dynamic state, on the other hand, is saved and restored for reasons that are slightly more complex.

Consider, for example, that an application contains an activity (which we will refer to as *Activity A*) containing a text field and some radio buttons. During the course of using the application, the user enters some text into the text field and makes a selection from the radio buttons. Before performing an action to save these changes, however, the user then switches to another activity causing *Activity A* to be pushed down the Activity Stack and placed into the background. After some time, the runtime system ascertains that memory is low and consequently kills *Activity A* to free up resources. As far as the user is concerned, however, *Activity A* was simply placed into the

background and is ready to be moved to the foreground at any time. On returning *Activity A* to the foreground the user would, quite reasonably, expect the entered text and radio button selections to have been retained. In this scenario, however, a new instance of *Activity A* will have been created and, if the dynamic state was not saved and restored, the previous user input lost.

The main purpose of saving dynamic state, therefore, is to give the perception of seamless switching between foreground and background activities, regardless of the fact that activities may actually have been killed and restarted without the user's knowledge.

The mechanisms for saving persistent and dynamic state will become clearer in the following sections of this chapter.

12.3 The Android Activity Lifecycle Methods

As previously explained, the `Activity` class contains a number of lifecycle methods which act as event handlers when the state of an `Activity` changes. The primary methods supported by the `Android Activity` class are as follows:

- **`onCreate(Bundle savedInstanceState)`** – The method that is called when the activity is first created and the ideal location for most initialization tasks to be performed. The method is passed an argument in the form of a `Bundle` object that may contain dynamic state information (typically relating to the state of the user interface) from a prior invocation of the activity.
- **`onRestart()`** – Called when the activity is about to restart after having previously been stopped by the runtime system.
- **`onStart()`** – Always called immediately after the call to the `onCreate()` or `onRestart()` methods, this method indicates to the activity that it is about to become visible to the user. This call will be followed by a call to `onResume()` if the activity moves to the top of the activity stack, or `onStop()` in the event that it is pushed down the stack by another activity.
- **`onResume()`** – Indicates that the activity is now at the top of the activity stack and is the activity with which the user is currently interacting.
- **`onPause()`** – Indicates that a previous activity is about to become the

foreground activity. This call will be followed by a call to either the `onResume()` or `onStop()` method depending on whether the activity moves back to the foreground or becomes invisible to the user. Steps may be taken within this method to store *persistent state* information not yet saved by the app. To avoid delays in switching between activities, time consuming operations such as storing data to a database or performing network operations should be avoided within this method. This method should also ensure that any CPU intensive tasks such as animation are stopped.

- **onStop()** – The activity is now no longer visible to the user. The two possible scenarios that may follow this call are a call to `onRestart()` in the event that the activity moves to the foreground again, or `onDestroy()` if the activity is being terminated.
- **onDestroy()** – The activity is about to be destroyed, either voluntarily because the activity has completed its tasks and has called the `finish()` method or because the runtime is terminating it either to release memory or due to a configuration change (such as the orientation of the device changing). It is important to note that a call will not always be made to `onDestroy()` when an activity is terminated.
- **onConfigurationChanged()** – Called when a configuration change occurs for which the activity has indicated it is not to be restarted. The method is passed a Configuration object outlining the new device configuration and it is then the responsibility of the activity to react to the change.

In addition to the lifecycle methods outlined above, there are two methods intended specifically for saving and restoring the *dynamic state* of an activity:

- **onRestoreInstanceState(Bundle savedInstanceState)** – This method is called immediately after a call to the `onStart()` method in the event that the activity is restarting from a previous invocation in which state was saved. As with `onCreate()`, this method is passed a Bundle object containing the previous state data. This method is typically used in situations where it makes more sense to restore a previous state after the initialization of the activity has been performed in `onCreate()` and

onStart().

- **onSaveInstanceState(Bundle outState)** – Called before an activity is destroyed so that the current *dynamic state* (usually relating to the user interface) can be saved. The method is passed the Bundle object into which the state should be saved and which is subsequently passed through to the *onCreate()* and *onRestoreInstanceState()* methods when the activity is restarted. Note that this method is only called in situations where the runtime ascertains that dynamic state needs to be saved.

When overriding the above methods in an activity, it is important to remember that, with the exception of *onRestoreInstanceState()* and *onSaveInstanceState()*, the method implementation must include a call to the corresponding method in the *Activity* super class. For example, the following method overrides the *onRestart()* method but also includes a call to the super class instance of the method:

```
protected void onRestart() {  
    super.onRestart();  
    Log.i(TAG, "onRestart");  
}
```

Failure to make this super class call in method overrides will result in the runtime throwing an exception during execution of the activity. While calls to the super class in the *onRestoreInstanceState()* and *onSaveInstanceState()* methods are optional (they can, for example, be omitted when implementing custom save and restoration behavior) there are considerable benefits to using them, a subject that will be covered in the chapter entitled ["Saving and Restoring the State of an Android Activity"](#).

12.4 Activity Lifetimes

The final topic to be covered involves an outline of the *entire*, *visible* and *foreground* lifetimes through which an activity will transition during execution:

- **Entire Lifetime** –The term “entire lifetime” is used to describe everything that takes place within an activity between the initial call to the *onCreate()* method and the call to *onDestroy()* prior to the activity

terminating.

- **Visible Lifetime** – Covers the periods of execution of an activity between the call to `onStart()` and `onStop()`. During this period the activity is visible to the user though may not be the activity with which the user is currently interacting.
- **Foreground Lifetime** – Refers to the periods of execution between calls to the `onResume()` and `onPause()` methods.

It is important to note that an activity may pass through the *foreground* and *visible* lifetimes multiple times during the course of the *entire* lifetime.

The concepts of lifetimes and lifecycle methods are illustrated in [Figure 12-2](#):

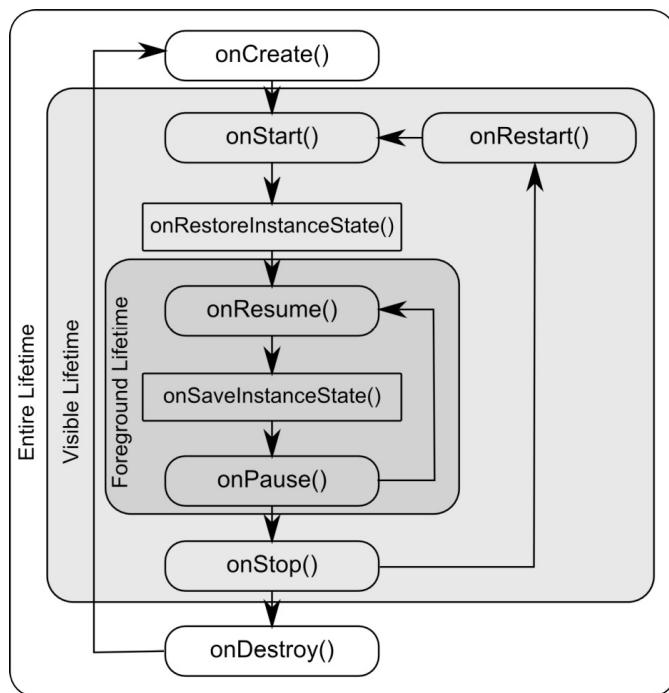


Figure 12-2

12.5 Disabling Configuration Change Restarts

As previously outlined, an activity may indicate that it is not to be restarted in the event of certain configuration changes. This is achieved by adding an `android:configChanges` directive to the activity element within the project manifest file. The following manifest file excerpt, for example, indicates that the activity should not be restarted in the event of configuration changes relating to orientation or device-wide font size:

```
<activity android:name=".DemoActivity"
```

```
    android:configChanges="orientation|fontScale"
    android:label="@string/app_name">
```

12.6 Summary

All activities are derived from the Android *Activity* class which, in turn, contains a number of event methods that are designed to be called by the runtime system when the state of an activity changes. By overriding these methods, an activity can respond to state changes and, where necessary, take steps to save and restore the current state of both the activity and the application. Activity state can be thought of as taking two forms. The persistent state refers to data that needs to be stored between application invocations (for example to a file or database). Dynamic state, on the other hand, relates instead to the current appearance of the user interface.

In this chapter, we have highlighted the lifecycle methods available to activities and covered the concept of activity lifetimes. In the next chapter, entitled ["Android Activity State Changes by Example"](#), we will implement an example application that puts much of this theory into practice.

13. Android Activity State Changes by Example

The previous chapters have discussed in some detail the different states and lifecycles of the activities that comprise an Android application. In this chapter, we will put the theory of handling activity state changes into practice through the creation of an example application. The purpose of this example application is to provide a real world demonstration of an activity as it passes through a variety of different states within the Android runtime.

In the next chapter, entitled "[Saving and Restoring the State of an Android Activity](#)", the example project constructed in this chapter will be extended to demonstrate the saving and restoration of dynamic activity state.

13.1 Creating the State Change Example Project

The first step in this exercise is to create the new project. Begin by launching Android Studio and, if necessary, closing any currently open projects using the *File -> Close Project* menu option so that the Welcome screen appears.

Select the *Start a new Android Studio project* quick start option from the welcome screen and, within the resulting new project dialog, enter *StateChange* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of a Basic Activity named *StateChangeActivity* with a corresponding layout named *activity_state_change*.

Upon completion of the project creation process, the *StateChange* project should be listed in the Project tool window located along the left-hand edge of the Android Studio main window with the *content_state_change.xml* layout file pre-loaded into the Layout Editor as illustrated in [Figure 13-1](#).

The next action to take involves the design of the content area of the user interface for the activity. This is stored in a file named *content_state_change.xml* which should already be loaded into the Layout Editor tool. If it is not, navigate to it in the project tool window where it can

be found in the *app -> res -> layout* folder. Once located, double-clicking on the file will load it into the Android Studio Layout Editor tool.

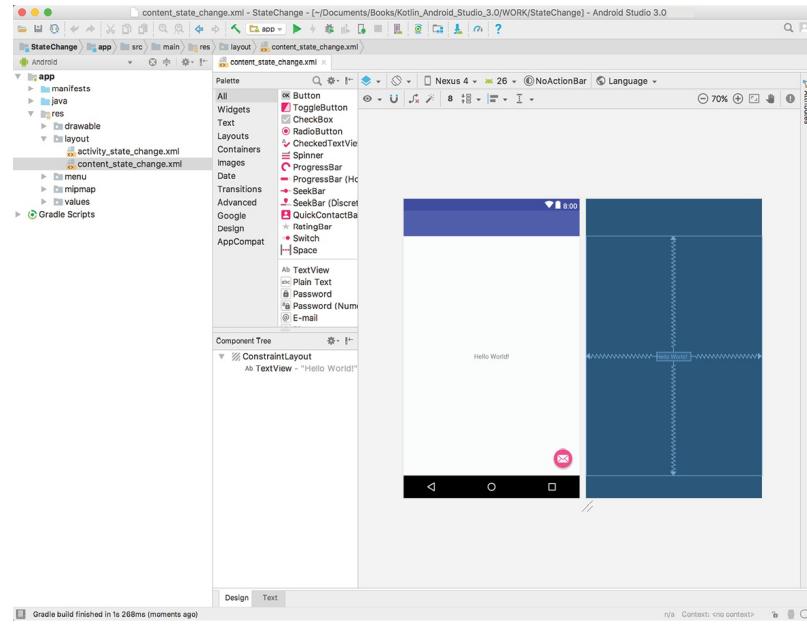


Figure 13-1

13.2 Designing the User Interface

With the user interface layout loaded into the Layout Editor tool, it is now time to design the user interface for the example application. Instead of the “Hello world!” TextView currently present in the user interface design, the activity actually requires an EditText view. Select the TextView object in the Layout Editor canvas and press the Delete key on the keyboard to remove it from the design.

From the Palette located on the left side of the Layout Editor, select the *Text* category and, from the list of text components, click and drag a *Plain Text* component over to the visual representation of the device screen. Move the component to the center of the display so that the center guidelines appear and drop it into place so that the layout resembles that of [Figure 13-2](#).

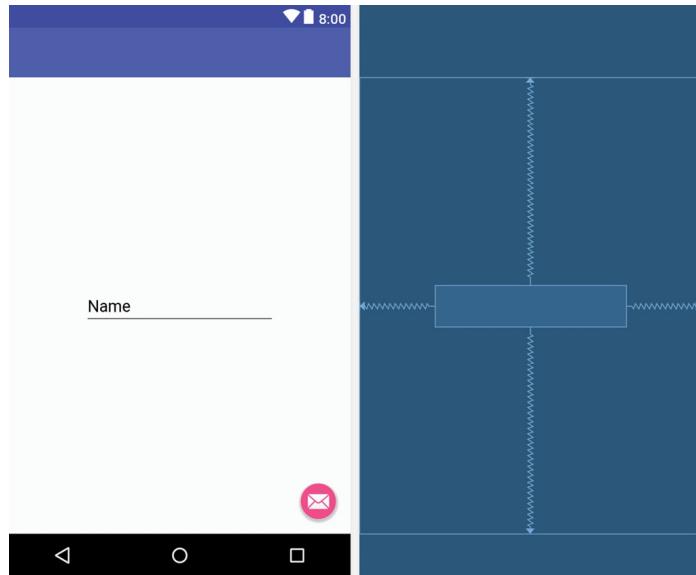


Figure 13-2

When using the `EditText` widget it is necessary to specify an *input type* for the view. This simply defines the type of text or data that will be entered by the user. For example, if the input type is set to *Phone*, the user will be restricted to entering numerical digits into the view. Alternatively, if the input type is set to *TextCapCharacters*, the input will default to upper case characters. Input type settings may also be combined.

For the purposes of this example, we will set the input type to support general text input. To do so, select the `EditText` widget in the layout and locate the *inputType* entry within the Attributes tool window. Click on the current setting to open the list of options and, within the list, switch off *textPersonName* and enable *text* before clicking on the OK button.

By default the `EditText` is displaying text which reads “Name”. Remaining within the Attributes panel, delete this from the *text* property field so that the view is blank within the layout.

13.3 Overriding the Activity Lifecycle Methods

At this point, the project contains a single activity named *StateChangeActivity*, which is derived from the Android *AppCompatActivity* class. The source code for this activity is contained within the *StateChangeActivity.java* file which should already be open in an editor session and represented by a tab in the editor tab bar. In the event that the file is no longer open, navigate to it in the Project tool window panel (*app -> java*

-> *com.ebookfrenzy.statechange* -> *StateChangeActivity*) and double-click on it to load the file into the editor. Once loaded the code should read as follows:

```
package com.ebookfrenzy.statechange;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;

public class StateChangeActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_state_change);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab =
                (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Snackbar.make(view, "Replace with your own action",
                        Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it
        getMenuInflater().inflate(R.menu.menu_state_change, menu);
        return true;
    }
}
```

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}
}

```

So far the only lifecycle method overridden by the activity is the *onCreate()* method which has been implemented to call the super class instance of the method before setting up the user interface for the activity. We will now modify this method so that it outputs a diagnostic message in the Android Studio Logcat panel each time it executes. For this, we will use the *Log* class, which requires that we import *android.util.Log* and declare a tag that will enable us to filter these messages in the log output:

```

package com.ebookfrenzy.statechange;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.util.Log;

public class StateChangeActivity extends AppCompatActivity {

    private static final String TAG = "StateChange";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

```

```

        setContentView(R.layout.activity_state_change);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab = (FloatingActionButton)
        findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Snackbar.make(view, "Replace with your own action",
                        Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
            }
        });

        Log.i(TAG, "onCreate");
    }

    .
    .
}

```

The next task is to override some more methods, with each one containing a corresponding log call. These override methods may be added manually or generated using the *Alt-Insert* keyboard shortcut as outlined in the chapter entitled [*"The Basics of the Android Studio Code Editor"*](#). Note that the Log calls will still need to be added manually if the methods are being auto-generated:

```

@Override
protected void onStart() {
    super.onStart();
    Log.i(TAG, "onStart");
}

@Override
protected void onResume() {
    super.onResume();
    Log.i(TAG, "onResume");
}

@Override
protected void onPause() {
    super.onPause();
}

```

```

        Log.i(TAG, "onPause");
    }

@Override
protected void onStop() {
    super.onStop();
    Log.i(TAG, "onStop");
}

@Override
protected void onRestart() {
    super.onRestart();
    Log.i(TAG, "onRestart");
}

@Override
protected void onDestroy() {
    super.onDestroy();
    Log.i(TAG, "onDestroy");
}

@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    Log.i(TAG, "onSaveInstanceState");
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    Log.i(TAG, "onRestoreInstanceState");
}

```

13.4 Filtering the Logcat Panel

The purpose of the code added to the overridden methods in *StateChangeActivity.java* is to output logging information to the *Logcat* tool window. This output can be configured to display all events relating to the device or emulator session, or restricted to those events that relate to the currently selected app. The output can also be further restricted to only those log events that match a specified filter.

Display the Logcat tool window and click on the filter menu (marked as B in

[Figure 13-3](#)) to review the available options. When this menu is set to *Show only selected application*, only those messages relating to the app selected in the menu marked as A will be displayed in the Logcat panel. Choosing *No Filter*, on the other hand, will display all the messages generated by the device or emulator.



Figure 13-3

Before running the application, it is worth demonstrating the creation of a filter which, when selected, will further restrict the log output to ensure that only those log messages containing the tag declared in our activity are displayed.

From the filter menu (B), select the *Edit Filter Configuration* menu option. In the *Create New Logcat Filter* dialog ([Figure 13-4](#)), name the filter *Lifecycle* and, in the *Log Tag* field, enter the *Tag* value declared in *StateChangeActivity.java* (in the above code example this was *StateChange*).

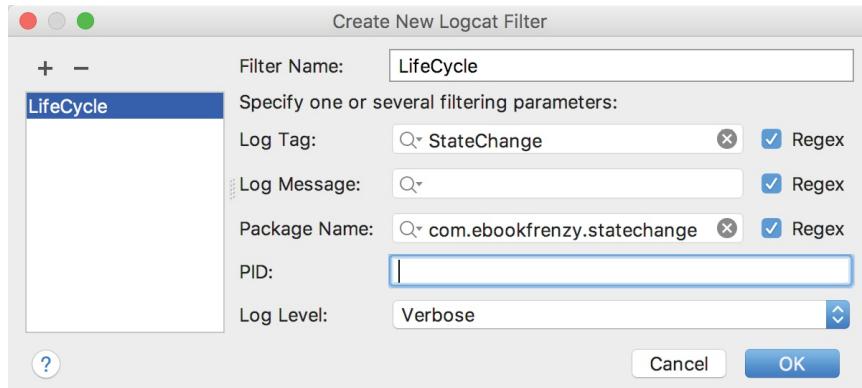


Figure 13-4

Enter the package identifier in the *Package Name* field (clicking on the search icon in the text field will drop down a menu from which the package name may be selected) and, when the changes are complete, click on the *OK* button to create the filter and dismiss the dialog. Instead of listing *No Filters*, the newly created filter should now be selected in the Logcat tool window.

13.5 Running the Application

For optimal results, the application should be run on a physical Android device, details of which can be found in the chapter entitled "["Testing Android Studio Apps on a Physical Android Device"](#)". With the device configured and connected to the development computer, click on the run button represented by a green triangle located in the Android Studio toolbar as shown in [Figure 13-5](#) below, select the *Run -> Run...* menu option or use the Shift+F10 keyboard shortcut:

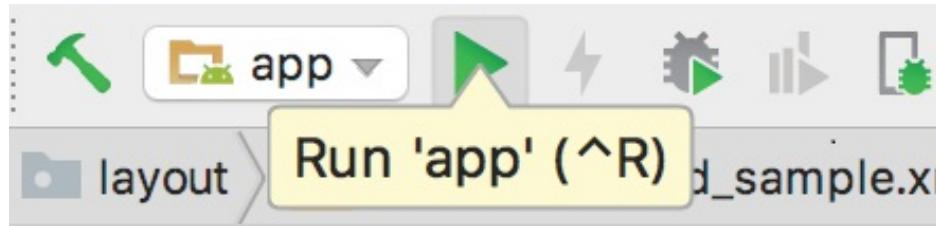


Figure 13-5

Select the physical Android device from the *Choose Device* dialog if it appears (assuming that you have not already configured it to be the default target). After Android Studio has built the application and installed it on the device it should start up and be running in the foreground.

A review of the Logcat panel should indicate which methods have so far been triggered (taking care to ensure that the *Lifecycle* filter created in the preceding section is selected to filter out log events that are not currently of interest to us):



Figure 13-6

13.6 Experimenting with the Activity

With the diagnostics working, it is now time to exercise the application with a view to gaining an understanding of the activity lifecycle state changes. To begin with, consider the initial sequence of log events in the Logcat panel:

```
onCreate  
onStart  
onResume
```

Clearly, the initial state changes are exactly as outlined in ["Understanding](#)

[Android Application and Activity Lifecycles](#). Note, however, that a call was not made to `onRestoreInstanceState()` since the Android runtime detected that there was no state to restore in this situation.

Tap on the Home icon in the bottom status bar on the device display and note the sequence of method calls reported in the log as follows:

```
onPause  
onSaveInstanceState  
onStop
```

In this case, the runtime has noticed that the activity is no longer in the foreground, is not visible to the user and has stopped the activity, but not without providing an opportunity for the activity to save the dynamic state. Depending on whether the runtime ultimately destroyed the activity or simply restarted it, the activity will either be notified it has been restarted via a call to `onRestart()` or will go through the creation sequence again when the user returns to the activity.

As outlined in [“Understanding Android Application and Activity Lifecycles”](#), the destruction and recreation of an activity can be triggered by making a configuration change to the device, such as rotating from portrait to landscape. To see this in action, simply rotate the device while the `StateChange` application is in the foreground. When using the emulator, device rotation may be simulated using the rotation button located in the emulator toolbar. The resulting sequence of method calls in the log should read as follows:

```
onPause  
onSaveInstanceState  
onStop  
onDestroy  
onCreate  
onStart  
onRestoreInstanceState  
onResume
```

Clearly, the runtime system has given the activity an opportunity to save state before being destroyed and restarted.

13.7 Summary

The old adage that a picture is worth a thousand words holds just as true for examples when learning a new programming paradigm. In this chapter, we

have created an example Android application for the purpose of demonstrating the different lifecycle states through which an activity is likely to pass. In the course of developing the project in this chapter, we also looked at a mechanism for generating diagnostic logging information from within an activity.

In the next chapter, we will extend the *StateChange* example project to demonstrate how to save and restore an activity's dynamic state.

14. Saving and Restoring the State of an Android Activity

If the previous few chapters have achieved their objective, it should now be a little clearer as to the importance of saving and restoring the state of a user interface at particular points in the lifetime of an activity.

In this chapter, we will extend the example application created in [“Android Activity State Changes by Example”](#) to demonstrate the steps involved in saving and restoring state when an activity is destroyed and recreated by the runtime system.

A key component of saving and restoring dynamic state involves the use of the Android SDK *Bundle* class, a topic that will also be covered in this chapter.

14.1 Saving Dynamic State

An activity, as we have already learned, is given the opportunity to save dynamic state information via a call from the runtime system to the activity’s implementation of the *onSaveInstanceState()* method. Passed through as an argument to the method is a reference to a *Bundle* object into which the method will need to store any dynamic data that needs to be saved. The *Bundle* object is then stored by the runtime system on behalf of the activity and subsequently passed through as an argument to the activity’s *onCreate()* and *onRestoreInstanceState()* methods if and when they are called. The data can then be retrieved from the *Bundle* object within these methods and used to restore the state of the activity.

14.2 Default Saving of User Interface State

In the previous chapter, the diagnostic output from the *StateChange* example application showed that an activity goes through a number of state changes when the device on which it is running is rotated sufficiently to trigger an orientation change.

Launch the *StateChange* application once again, this time entering some text into the *EditText* field prior to performing the device rotation. Having rotated the device, the following state change sequence should appear in the Logcat

window:

```
onPause  
onSaveInstanceState  
onStop  
onDestroy  
onCreate  
onStart  
onRestoreInstanceState  
onResume
```

Clearly this has resulted in the activity being destroyed and re-created. A review of the user interface of the running application, however, should show that the text entered into the EditText field has been preserved. Given that the activity was destroyed and recreated, and that we did not add any specific code to make sure the text was saved and restored, this behavior requires some explanation.

In actual fact most of the view widgets included with the Android SDK already implement the behavior necessary to automatically save and restore state when an activity is restarted. The only requirement to enable this behavior is for the `onSaveInstanceState()` and `onRestoreInstanceState()` override methods in the activity to include calls to the equivalent methods of the super class:

```
@Override  
protected void onSaveInstanceState(Bundle outState) {  
    super.onSaveInstanceState(outState);  
}  
  
@Override  
protected void onRestoreInstanceState(Bundle savedInstanceState) {  
    super.onRestoreInstanceState(savedInstanceState);  
}
```

The automatic saving of state for a user interface view can be disabled in the XML layout file by setting the `android:saveEnabled` property to `false`. For the purposes of an example, we will disable the automatic state saving mechanism for the EditText view in the user interface layout and then add code to the application to manually save and restore the state of the view.

To configure the EditText view such that state will not be saved and restored in the event that the activity is restarted, edit the `content_state_change.xml` file so that the entry for the view reads as follows (note that the XML can be

edited directly by clicking on the *Text* tab on the bottom edge of the Layout Editor panel):

```
<EditText  
    android:id="@+id/editText"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:ems="10"  
    android:inputType="text"  
    android:saveEnabled="false"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintLeft_toLeftOf="parent"  
    app:layout_constraintRight_toRightOf="parent"  
    app:layout_constraintTop_toTopOf="parent" />
```

After making the change, run the application, enter text and rotate the device to verify that the text is no longer saved and restored before proceeding.

14.3 The Bundle Class

For situations where state needs to be saved beyond the default functionality provided by the user interface view components, the Bundle class provides a container for storing data using a *key-value pair* mechanism. The *keys* take the form of string values, while the *values* associated with those *keys* can be in the form of a primitive value or any object that implements the Android *Parcelable* interface. A wide range of classes already implements the *Parcelable* interface. Custom classes may be made “parcelable” by implementing the set of methods defined in the *Parcelable* interface, details of which can be found in the Android documentation at:

<http://developer.android.com/reference/android/os/Parcelable.html>

The Bundle class also contains a set of methods that can be used to get and set key-value pairs for a variety of data types including both primitive types (including Boolean, char, double and float values) and objects (such as Strings and CharSequences).

For the purposes of this example, and having disabled the automatic saving of text for the *EditText* view, we need to make sure that the text entered into the *EditText* field by the user is saved into the *Bundle* object and subsequently restored. This will serve as a demonstration of how to manually save and restore state within an Android application and will be achieved using the *putCharSequence()* and *getCharSequence()* methods of the *Bundle* class

respectively.

14.4 Saving the State

The first step in extending the *StateChange* application is to make sure that the text entered by the user is extracted from the *EditText* component within the *onSaveInstanceState()* method of the *StateChangeActivity* activity, and then saved as a key-value pair into the *Bundle* object.

In order to extract the text from the *EditText* object we first need to identify that object in the user interface. Clearly, this involves bridging the gap between the Java code for the activity (contained in the *StateChangeActivity.java* source code file) and the XML representation of the user interface (contained within the *content_state_change.xml* resource file). In order to extract the text entered into the *EditText* component we need to gain access to that user interface object.

Each component within a user interface has associated with it a unique identifier. By default, the Layout Editor tool constructs the ID for a newly added component from the object type. If more than one view of the same type is contained in the layout the type name is followed by a sequential number (though this can, and should, be changed to something more meaningful by the developer). As can be seen by checking the *Component Tree* panel within the Android Studio main window when the *content_state_change.xml* file is selected and the Layout Editor tool displayed, the *EditText* component has been assigned the ID *editText*:

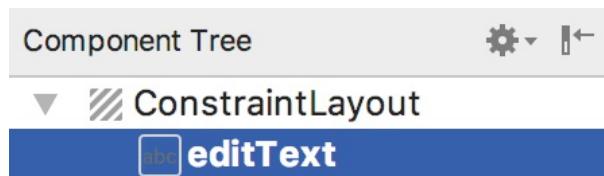


Figure 14-1

As outlined in the chapter entitled "[The Anatomy of an Android Application](#)", all of the resources that make up an application are compiled into a class named *R*. Amongst those resources are those that define layouts, including the layout for our current activity. Within the *R* class is a subclass named *layout*, which contains the layout resources, and within that subclass is our *content_state_change* layout. With this knowledge, we can make a call to the *findViewById()* method of our activity object to get a reference to the *EditText*

object as follows:

```
final EditText editText = (EditText) findViewById(R.id.editText);
```

Having either obtained a reference to the `EditText` object and assigned it to a variable, we can now obtain the text that it contains via the object's `getText()` method, which, in turn, returns the current text:

```
CharSequence userText = editText.getText();
```

Finally, we can save the text using the `Bundle` object's `putCharSequence()` method, passing through the key (this can be any string value but in this instance, we will declare it as "savedText") and the `userText` object as arguments:

```
outState.putCharSequence("savedText", userText);
```

Bringing this all together gives us a modified `onSaveInstanceState()` method in the `StateChangeActivity.java` file that reads as follows (noting also the additional import directive for `android.widget.EditText`):

```
package com.ebookfrenzy.statechange;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.util.Log;
import android.widget.EditText;

public class StateChangeActivity extends AppCompatActivity {
    .
    .
    .
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        Log.i(TAG, "onSaveInstanceState");

        final EditText editText =
            (EditText) findViewById(R.id.editText);
        CharSequence userText = editText.getText();
        outState.putCharSequence("savedText", userText);
    }
}
```

.
. .

Now that steps have been taken to save the state, the next phase is to ensure that it is restored when needed.

14.5 Restoring the State

The saved dynamic state can be restored in those lifecycle methods that are passed the Bundle object as an argument. This leaves the developer with the choice of using either `onCreate()` or `onRestoreInstanceState()`. The method to use will depend on the nature of the activity. In instances where state is best restored after the activity's initialization tasks have been performed, the `onRestoreInstanceState()` method is generally more suitable. For the purposes of this example we will add code to the `onRestoreInstanceState()` method to extract the saved state from the Bundle using the "savedText" key. We can then display the text on the editText component using the object's `setText()` method:

```
@Override  
protected void onRestoreInstanceState(Bundle savedInstanceState) {  
    super.onRestoreInstanceState(savedInstanceState);  
    Log.i(TAG, "onRestoreInstanceState");  
  
    final EditText editText =  
        (EditText) findViewById(R.id.editText);  
  
    CharSequence userText =  
        savedInstanceState.getCharSequence("savedText");  
  
    editText.setText(userText);  
}
```

14.6 Testing the Application

All that remains is once again to build and run the *StateChange* application. Once running and in the foreground, touch the EditText component and enter some text before rotating the device to another orientation. Whereas the text changes were previously lost, the new text is retained within the editText component thanks to the code we have added to the activity in this chapter.

Having verified that the code performs as expected, comment out the

super.onSaveInstanceState() and *super.onRestoreInstanceState()* calls from the two methods, re-launch the app and note that the text is still preserved after a device rotation. The default save and restoration system has essentially been replaced by a custom implementation, thereby providing a way to dynamically and selectively save and restore state within an activity.

14.7 Summary

The saving and restoration of dynamic state in an Android application is simply a matter of implementing the appropriate code in the appropriate lifecycle methods. For most user interface views, this is handled automatically by the Activity super class. In other instances, this typically consists of extracting values and settings within the *onSaveInstanceState()* method and saving the data as key-value pairs within the Bundle object passed through to the activity by the runtime system.

State can be restored in either the *onCreate()* or the *onRestoreInstanceState()* methods of the activity by extracting values from the Bundle object and updating the activity based on the stored values.

In this chapter, we have used these techniques to update the *StateChange* project so that the Activity retains changes through the destruction and subsequent recreation of an activity.

15. Understanding Android Views, View Groups and Layouts

With the possible exception of listening to streaming audio, a user's interaction with an Android device is primarily visual and tactile in nature. All of this interaction takes place through the user interfaces of the applications installed on the device, including both the built-in applications and any third party applications installed by the user. It should come as no surprise, therefore, that a key element of developing Android applications involves the design and creation of user interfaces.

Within this chapter, the topic of Android user interface structure will be covered, together with an overview of the different elements that can be brought together to make up a user interface; namely Views, View Groups and Layouts.

15.1 Designing for Different Android Devices

The term "Android device" covers a vast array of tablet and smartphone products with different screen sizes and resolutions. As a result, application user interfaces must now be carefully designed to ensure correct presentation on as wide a range of display sizes as possible. A key part of this is ensuring that the user interface layouts resize correctly when run on different devices. This can largely be achieved through careful planning and the use of the layout managers outlined in this chapter.

It is also important to keep in mind that the majority of Android based smartphones and tablets can be held by the user in both portrait and landscape orientations. A well-designed user interface should be able to adapt to such changes and make sensible layout adjustments to utilize the available screen space in each orientation.

15.2 Views and View Groups

Every item in a user interface is a subclass of the Android *View* class (to be precise *android.view.View*). The Android SDK provides a set of pre-built views that can be used to construct a user interface. Typical examples include standard items such as the Button, CheckBox, ProgressBar and TextView classes. Such views are also referred to as *widgerts* or *components*. For

requirements that are not met by the widgets supplied with the SDK, new views may be created either by subclassing and extending an existing class, or creating an entirely new component by building directly on top of the `View` class.

A view can also be comprised of multiple other views (otherwise known as a *composite view*). Such views are subclassed from the Android `ViewGroup` class (`android.view.ViewGroup`) which is itself a subclass of `View`. An example of such a view is the `RadioGroup`, which is intended to contain multiple `RadioButton` objects such that only one can be in the “on” position at any one time. In terms of structure, composite views consist of a single parent view (derived from the `ViewGroup` class and otherwise known as a *container view* or *root element*) that is capable of containing other views (known as *child views*).

Another category of `ViewGroup` based container view is that of the layout manager.

15.3 Android Layout Managers

In addition to the widget style views discussed in the previous section, the SDK also includes a set of views referred to as *layouts*. Layouts are container views (and, therefore, subclassed from `ViewGroup`) designed for the sole purpose of controlling how child views are positioned on the screen.

The Android SDK includes the following layout views that may be used within an Android user interface design:

- **ConstraintLayout** – Introduced in Android 7, use of this layout manager is recommended for most layout requirements. `ConstraintLayout` allows the positioning and behavior of the views in a layout to be defined by simple constraint settings assigned to each child view. The flexibility of this layout allows complex layouts to be quickly and easily created without the necessity to nest other layout types inside each other, resulting in improved layout performance. `ConstraintLayout` is also tightly integrated into the Android Studio Layout Editor tool. Unless otherwise stated, this is the layout of choice for the majority of examples in this book.
- **LinearLayout** – Positions child views in a single row or column

depending on the orientation selected. A *weight* value can be set on each child to specify how much of the layout space that child should occupy relative to other children.

- **TableLayout** – Arranges child views into a grid format of rows and columns. Each row within a table is represented by a *TableRow* object child, which, in turn, contains a view object for each cell.
- **FrameLayout** – The purpose of the FrameLayout is to allocate an area of screen, typically for the purposes of displaying a single view. If multiple child views are added they will, by default, appear on top of each other positioned in the top left-hand corner of the layout area. Alternate positioning of individual child views can be achieved by setting gravity values on each child. For example, setting a *center_vertical* gravity on a child will cause it to be positioned in the vertical center of the containing FrameLayout view.
- **RelativeLayout** – The RelativeLayout allows child views to be positioned relative both to each other and the containing layout view through the specification of alignments and margins on child views. For example, child *View A* may be configured to be positioned in the vertical and horizontal center of the containing RelativeLayout view. *View B*, on the other hand, might also be configured to be centered horizontally within the layout view, but positioned 30 pixels above the top edge of *View A*, thereby making the vertical position *relative* to that of *View A*. The RelativeLayout manager can be of particular use when designing a user interface that must work on a variety of screen sizes and orientations.
- **AbsoluteLayout** – Allows child views to be positioned at specific X and Y coordinates within the containing layout view. Use of this layout is discouraged since it lacks the flexibility to respond to changes in screen size and orientation.
- **GridLayout** – A GridLayout instance is divided by invisible lines that form a grid containing rows and columns of cells. Child views are then placed in cells and may be configured to cover multiple cells both horizontally and vertically allowing a wide range of layout options to be quickly and easily implemented. Gaps between components in a GridLayout may be implemented by placing a special type of view

called a *Space* view into adjacent cells, or by setting margin parameters.

- **CoordinatorLayout** – Introduced as part of the Android Design Support Library with Android 5.0, the CoordinatorLayout is designed specifically for coordinating the appearance and behavior of the app bar across the top of an application screen with other view elements. When creating a new activity using the Basic Activity template, the parent view in the main layout will be implemented using a CoordinatorLayout instance. This layout manager will be covered in greater detail starting with the chapter entitled [“Working with the Floating Action Button and Snackbar”](#).

When considering the use of layouts in the user interface for an Android application it is worth keeping in mind that, as will be outlined in the next section, these can be nested within each other to create a user interface design of just about any necessary level of complexity.

15.4 The View Hierarchy

Each view in a user interface represents a rectangular area of the display. A view is responsible for what is drawn in that rectangle and for responding to events that occur within that part of the screen (such as a touch event).

A user interface screen is comprised of a view hierarchy with a *root view* positioned at the top of the tree and child views positioned on branches below. The child of a container view appears on top of its parent view and is constrained to appear within the bounds of the parent view’s display area. Consider, for example, the user interface illustrated in [Figure 15-1](#):

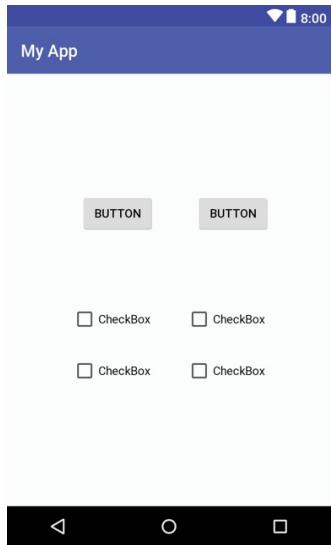


Figure 15-1

In addition to the visible button and checkbox views, the user interface actually includes a number of layout views that control how the visible views are positioned. [Figure 15-2](#) shows an alternative view of the user interface, this time highlighting the presence of the layout views in relation to the child views:

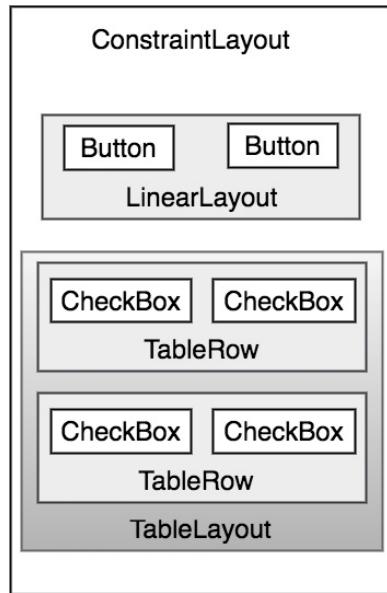


Figure 15-2

As was previously discussed, user interfaces are constructed in the form of a view hierarchy with a root view at the top. This being the case, we can also visualize the above user interface example in the form of the view tree illustrated in [Figure 15-3](#):

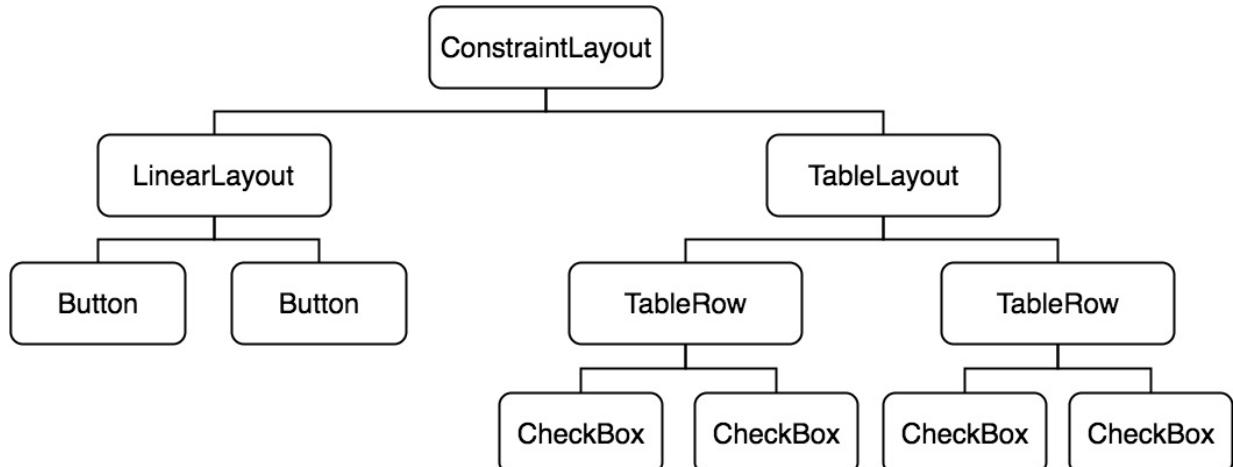


Figure 15-3

The view hierarchy diagram gives probably the clearest overview of the relationship between the various views that make up the user interface shown in [Figure 15-1](#). When a user interface is displayed to the user, the Android runtime walks the view hierarchy, starting at the root view and working down the tree as it renders each view.

15.5 Creating User Interfaces

With a clearer understanding of the concepts of views, layouts and the view hierarchy, the following few chapters will focus on the steps involved in creating user interfaces for Android activities. In fact, there are three different approaches to user interface design: using the Android Studio Layout Editor tool, handwriting XML layout resource files or writing Java code, each of which will be covered.

15.6 Summary

Each element within a user interface screen of an Android application is a view that is ultimately subclassed from the `android.view.View` class. Each view represents a rectangular area of the device display and is responsible both for what appears in that rectangle and for handling events that take place within the view's bounds. Multiple views may be combined to create a single *composite view*. The views within a composite view are children of a *container view* which is generally a subclass of `android.view.ViewGroup` (which is itself a subclass of `android.view.View`). A user interface is comprised of views constructed in the form of a view hierarchy.

The Android SDK includes a range of pre-built views that can be used to

create a user interface. These include basic components such as text fields and buttons, in addition to a range of layout managers that can be used to control the positioning of child views. In the event that the supplied views do not meet a specific requirement, custom views may be created, either by extending or combining existing views, or by subclassing *android.view.View* and creating an entirely new class of view.

User interfaces may be created using the Android Studio Layout Editor tool, handwriting XML layout resource files or by writing Java code. Each of these approaches will be covered in the chapters that follow.

16. A Guide to the Android Studio Layout Editor Tool

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

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

16.1 Basic vs. Empty Activity Templates

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

When creating a new Android Studio project, a number of different templates are available to be used as the starting point for the user interface of the main activity. The most basic of these templates are the Basic Activity and Empty Activity templates. Although these seem similar at first glance, there are actually considerable differences between the two options.

The Empty Activity template creates a single layout file consisting of a ConstraintLayout manager instance containing a TextView object as shown in [Figure 16-1](#):

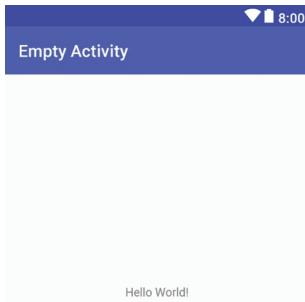


Figure 16-1

The Basic Activity, on the other hand, consists of two layout files. The top level layout file has a CoordinatorLayout as the root view, a configurable app bar, a menu preconfigured with a single menu item (A in [Figure 16-2](#)), a floating action button (B) and a reference to the second layout file in which the layout for the content area of the activity user interface is declared:



Figure 16-2

Clearly the Empty Activity template is useful if you need neither a floating action button nor a menu in your activity and do not need the special app bar behavior provided by the CoordinatorLayout such as options to make the app bar and toolbar collapse from view during certain scrolling operations (a

topic covered in the chapter entitled ["Working with the AppBar and Collapsing Toolbar Layouts"](#)). The Basic Activity is useful, however, in that it provides these elements by default. In fact, it is often quicker to create a new activity using the Basic Activity template and delete the elements you do not require than to use the Empty Activity template and manually implement behavior such as collapsing toolbars, a menu or floating action button.

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

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

1. Double-click on the main *activity* layout file located in the Project tool window under *app -> res -> layout* to load it into the Layout Editor. This will be the layout file prefixed with *activity_* and not the content file prefixed with *content_*.
2. With the layout loaded into the Layout Editor tool, select the floating action button and tap the keyboard *Delete* key to remove the object from the layout.
3. Locate and edit the Java code for the activity (located under *app -> java -> <package name> -> <activity class name>*) and remove the floating action button code from the *onCreate* method as follows:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);  
    setSupportActionBar(toolbar);  
  
    FloatingActionButton fab =  
    (FloatingActionButton) findViewById(R.id.fab);  
    fab.setOnClickListener(new View.OnClickListener() {  
            @Override  
            public void onClick(View view) {  
                Snackbar.make(view, "Replace with your own action",  
                    Snackbar.LENGTH_LONG)  
                    .setAction("Action", null).show();
```

```
    }  
});  
}  
}
```

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

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

16.2 The Android Studio Layout Editor

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

Under the surface, the Layout Editor tool actually constructs an XML resource file containing the definition of the user interface that is being designed. As such, the Layout Editor tool operates in two distinct modes referred to as *Design mode* and *Text mode*.

16.3 Design Mode

In design mode, the user interface can be visually manipulated by directly working with the view palette and the graphical representation of the layout. [Figure 16-3](#) highlights the key areas of the Android Studio Layout Editor tool in design mode:

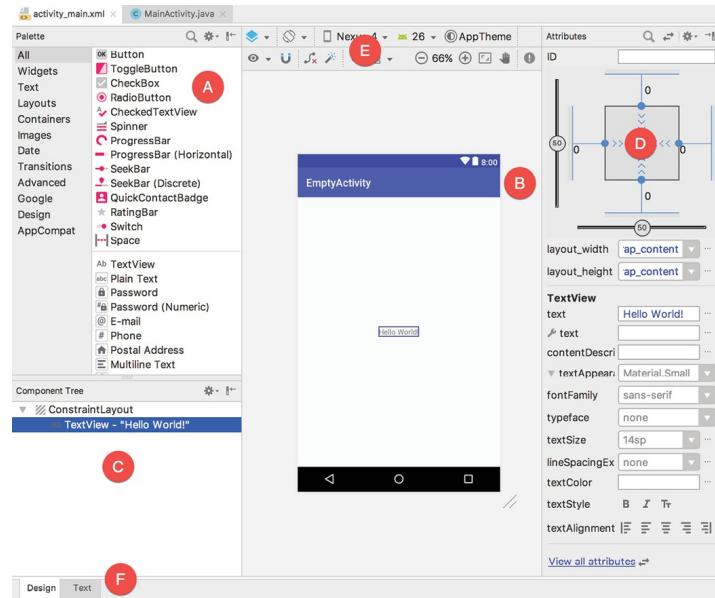


Figure 16-3

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

B – Device Screen – The device screen provides a visual “what you see is what you get” representation of the user interface layout as it is being designed. This layout allows for direct manipulation of the design in terms of allowing views to be selected, deleted, moved and resized. The device model represented by the layout can be changed at any time using a menu located in the toolbar.

C – Component Tree – As outlined in the previous chapter ([“Understanding Android Views, View Groups and Layouts”](#)) user interfaces are constructed using a hierarchical structure. The component tree provides a visual overview of the hierarchy of the user interface design. Selecting an element from the component tree will cause the corresponding view in the layout to be selected. Similarly, selecting a view from the device screen layout will select that view in the component tree hierarchy.

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

changes to be made.

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

F – Mode Switching Tabs – The tabs located along the lower edge of the Layout Editor provide a way to switch back and forth between the Layout Editor tool's text and design modes.

16.4 The Palette

The Layout Editor palette is organized into two panels designed to make it easy to locate and preview view components for addition to a layout design. The category panel (marked A in [Figure 16-4](#)) lists the different categories of view components supported by the Android SDK. When a category is selected from the list, the second panel (B) updates to display a list of the components that fall into that category:

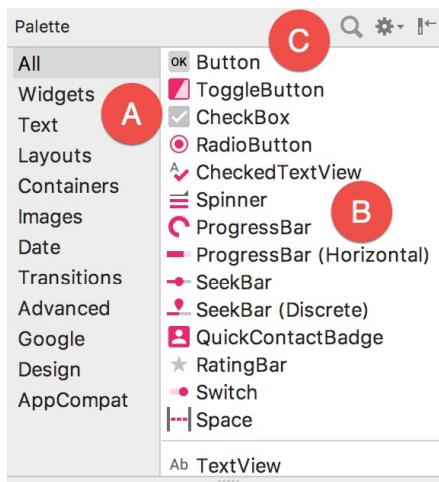


Figure 16-4

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

A search for a specific component within the currently selected category may be initiated by clicking on the search button (marked C in [Figure 16-4](#) above)

in the palette toolbar and typing in the component name. As characters are typed, matching results will appear in real-time within the component list panel. If you are unsure of the category in which the component resides, simply select the All category either before or during the search operation.

16.5 Pan and Zoom

When first opened, the Layout Editor will size the layout canvas so that it fits within the available space. Zooming in and out of the layout can be achieved using the plus and minus buttons located in the editor toolbar. When the view is zoomed in, it can be useful to pan around the layout to locate a particular area of the design. Although this can be achieved using the scrollbars, another option is to use the pan menu bar button highlighted in [Figure 16-5](#). Once selected, a pan and zoom panel will appear containing a lens which can be moved to alter the currently visible area of the layout.

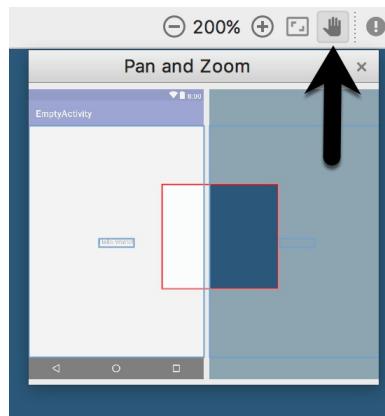


Figure 16-5

16.6 Design and Layout Views

When the Layout Editor tool is in Design mode, the layout can be viewed in two different ways. The view shown in [Figure 16-3](#) above is the Design view and shows the layout and widgets as they will appear in the running app. A second mode, referred to as Layout or Blueprint view can be shown either instead of, or concurrently with the Design view. The toolbar menu shown in [Figure 16-6](#) provides options to display the Design, Blueprint, or both views. A fourth option, *Force Refresh Layout*, causes the layout to rebuild and redraw. This can be useful when the layout enters an unexpected state or is not accurately reflecting the current design settings:

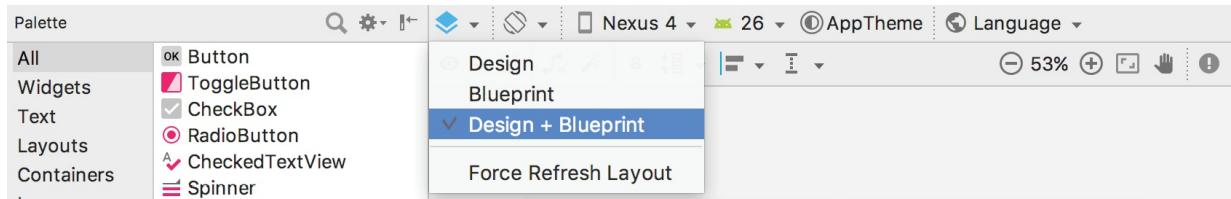


Figure 16-6

Whether to display the layout view, design view or both is a matter of personal preference. A good approach is to begin with both displayed as shown in [Figure 16-7](#):

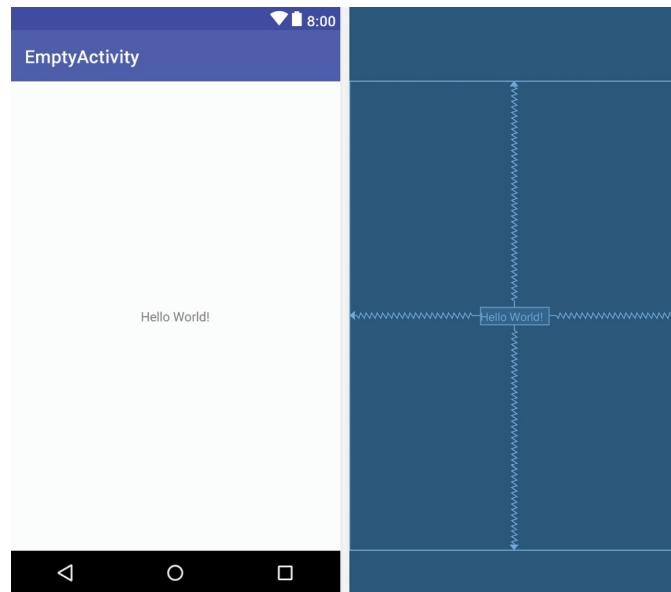


Figure 16-7

16.7 Text Mode

It is important to keep in mind when using the Android Studio Layout Editor tool that all it is really doing is providing a user friendly approach to creating XML layout resource files. At any time during the design process, the underlying XML can be viewed and directly edited simply by clicking on the *Text* tab located at the bottom of the Layout Editor tool panel. To return to design mode, simply click on the *Design* tab.

[Figure 16-8](#) highlights the key areas of the Android Studio Layout Editor tool in text mode:

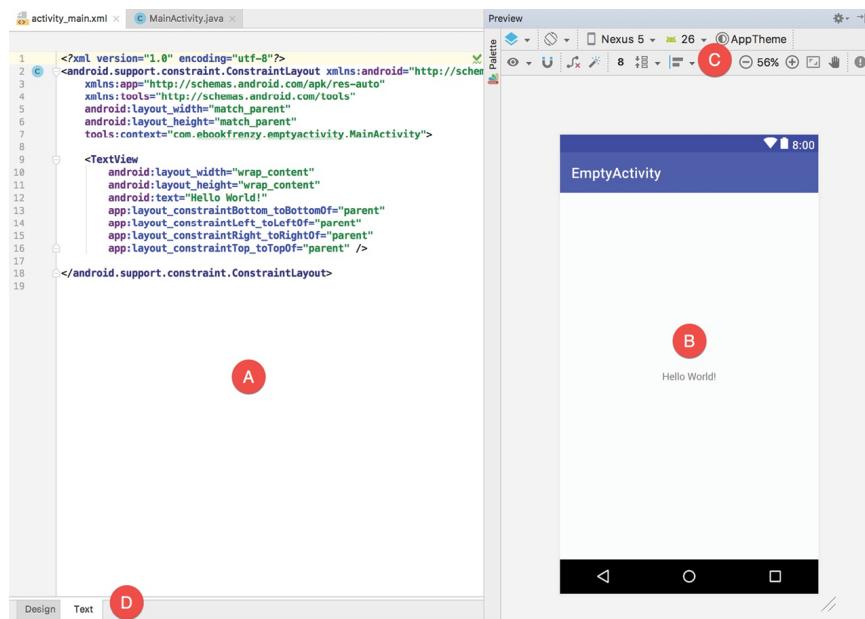


Figure 16-8

A – Editor – The editor panel displays the XML that makes up the current user interface layout design. This is the full Android Studio editor environment containing all of the features previously outlined in the [The Basics of the Android Studio Code Editor](#) chapter of this book.

B – Preview – As changes are made to the XML in the editor, these changes are visually reflected in the preview window. This provides instant visual feedback on the XML changes as they are made in the editor, thereby avoiding the need to switch back and forth between text and design mode to see changes. The preview also allows direct manipulation and design of the layout just as if the layout were in Design mode, with visual changes being reflected in the editor panel in real-time. As with Design mode, both the Design and Layout views may be displayed using the toolbar buttons highlighted in [Figure 16-6](#) above.

C – Toolbar – The toolbar in text mode provides access to the same functions available in design mode.

D - Mode Switching Tabs – The tabs located along the lower edge of the Layout Editor provide a way to switch back and forth between the Layout Editor tool's Text and Design modes.

16.8 Setting Attributes

The Attributes panel provides access to all of the available settings for the

currently selected component. By default, the attributes panel shows the most commonly changed attributes for the currently selected component in the layout. [Figure 16-9](#), for example, shows this subset of attributes for the TextView widget:

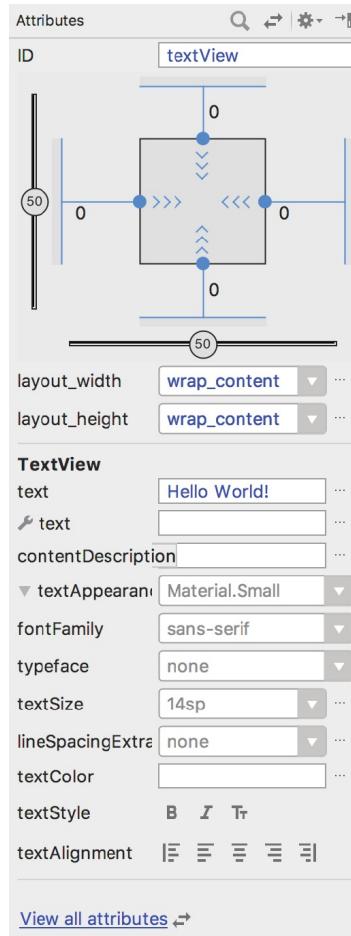


Figure 16-9

To access all of the attributes for the currently selected widget, click on the button highlighted in [Figure 16-10](#), or use the *View all attributes* link at the bottom of the attributes panel:

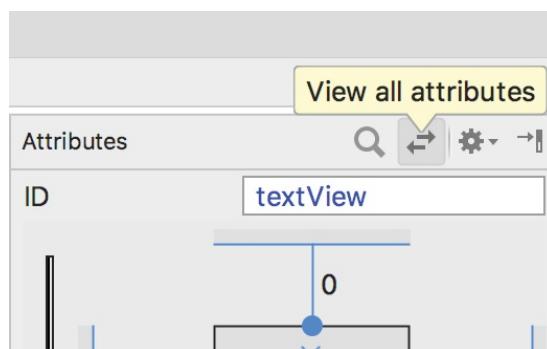


Figure 16-
10

A search for a specific attribute may also be performed by selecting the search button in the toolbar of the attributes tool window and typing in the attribute name. Select the *View all attributes* button or link either before or during a search to ensure that all of the attributes for the currently selected component are included in the results.

Some attributes contain a button displaying three dots. This indicates that a settings dialog is available to assist in selecting a suitable property value. To display the dialog, simply click on the button. Attributes for which a finite number of valid options are available will present a drop down menu ([Figure 16-11](#)) from which a selection may be made.

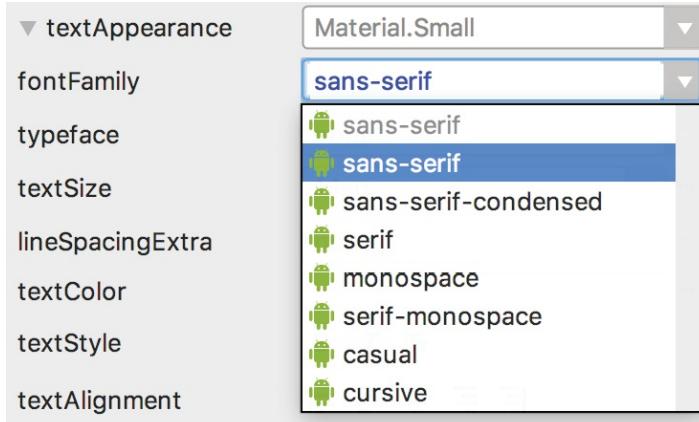


Figure 16-
11

16.9 Configuring Favorite Attributes

The attributes included on the initial subset attribute list may be extended by configuring *favorite attributes*. To add an attribute to the favorites list, display all the attributes for the currently selected component and hover the mouse pointer so that it is positioned to the far left of the attribute entry within the attributes tool window. A star icon will appear to the left of the attribute name which, when clicked, will add the property to the favorites list. [Figure 16-12](#), for example, shows the autoText, background and backgroundTint attributes for a TextView widget configured as favorite attributes:

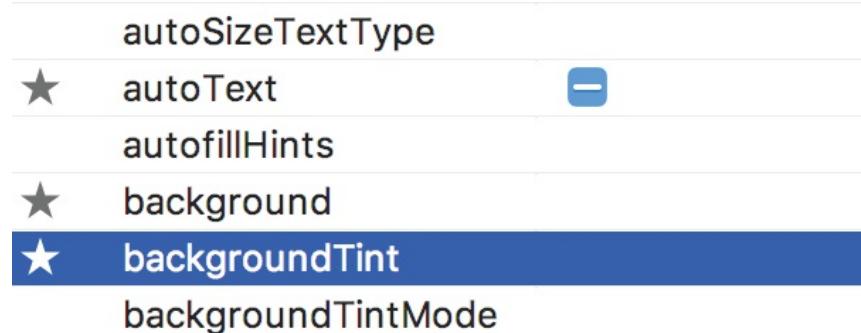


Figure 16-
12

Once added as favorites, the attributes will be listed beneath the *Favorite Attributes* section in the subject attributes list:

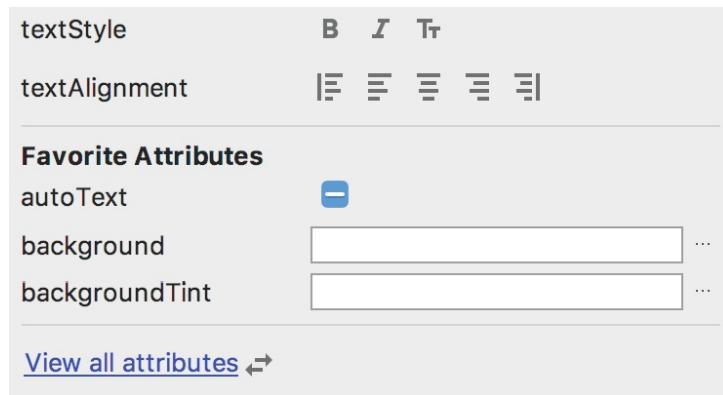


Figure 16-
13

16.10 Creating a Custom Device Definition

The device menu in the Layout Editor toolbar ([Figure 16-14](#)) provides a list of preconfigured device types which, when selected, will appear as the device screen canvas. In addition to the pre-configured device types, any AVD instances that have previously been configured within the Android Studio environment will also be listed within the menu. To add additional device configurations, display the device menu, select the *Add Device Definition...* option and follow the steps outlined in the chapter entitled [*"Creating an Android Virtual Device \(AVD\) in Android Studio"*](#).

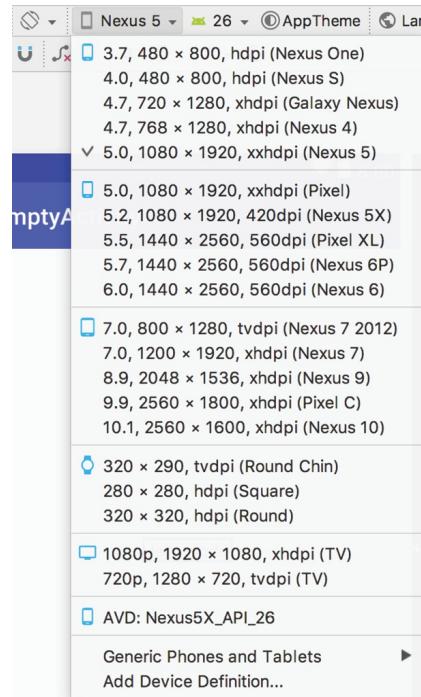


Figure 16-
14

16.11 Changing the Current Device

As an alternative to the device selection menu, the current device format may be changed by clicking on the resize handle located next to the bottom right-hand corner of the device screen ([Figure 16-15](#)) and dragging to select an alternate device display format. As the screen resizes, markers will appear indicating the various size options and orientations available for selection:

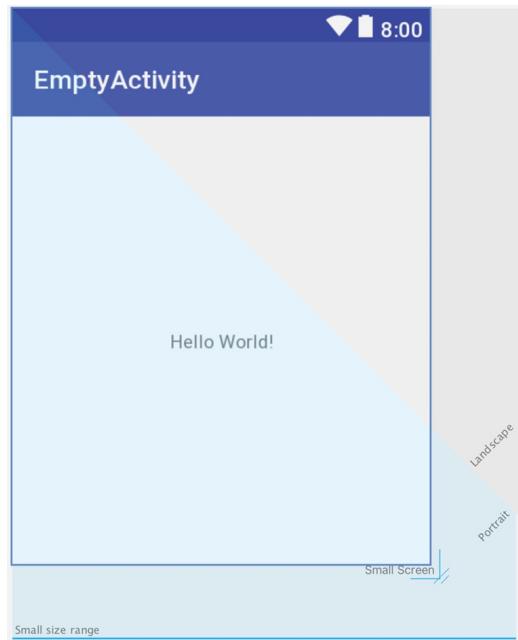


Figure 16-
15

16.1 Summary

A key part of developing Android applications involves the creation of the user interface. Within the Android Studio environment, this is performed using the Layout Editor tool which operates in two modes. In design mode, view components are selected from a palette and positioned on a layout representing an Android device screen and configured using a list of attributes. In text mode, the underlying XML that represents the user interface layout can be directly edited, with changes reflected in a preview screen. These modes combine to provide an extensive and intuitive user interface design environment.

17. A Guide to the Android ConstraintLayout

As discussed in the chapter entitled "[A Guide to the Android ConstraintLayout](#)", Android provides a number of layout managers for the purpose of designing user interfaces. With Android 7, Google has introduced a new layout that is intended to address many of the shortcomings of the older layout managers. This new layout, called ConstraintLayout, combines a simple, expressive and flexible layout system with powerful features built into the Android Studio Layout Editor tool to ease the creation of responsive user interface layouts that adapt automatically to different screen sizes and changes in device orientation.

This chapter will outline the basic concepts of ConstraintLayout while the next chapter will provide a detailed overview of how constraint-based layouts can be created using ConstraintLayout within the Android Studio Layout Editor tool.

17.1 How ConstraintLayout Works

In common with all other layouts, ConstraintLayout is responsible for managing the positioning and sizing behavior of the visual components (also referred to as widgets) it contains. It does this based on the constraint connections that are set on each child widget.

In order to fully understand and use ConstraintLayout, it is important to gain an appreciation of the following key concepts:

- Constraints
- Margins
- Opposing Constraints
- Constraint Bias
- Chains
- Chain Styles
- Barriers

17.1.1 Constraints

Constraints are essentially sets of rules that dictate the way in which a widget is aligned and distanced in relation to other widgets, the sides of the containing ConstraintLayout and special elements called *guidelines*. Constraints also dictate how the user interface layout of an activity will respond to changes in device orientation, or when displayed on devices of differing screen sizes. In order to be adequately configured, a widget must have sufficient constraint connections such that its position can be resolved by the ConstraintLayout layout engine in both the horizontal and vertical planes.

17.1.2 Margins

A margin is a form of constraint that specifies a fixed distance. Consider a Button object that needs to be positioned near the top right-hand corner of the device screen. This might be achieved by implementing margin constraints from the top and right-hand edges of the Button connected to the corresponding sides of the parent ConstraintLayout as illustrated in [Figure 17-1](#):

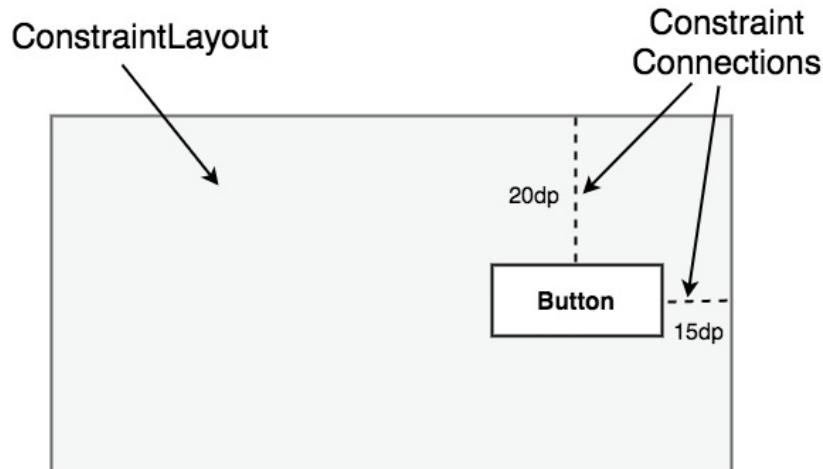


Figure 17-1

As indicated in the above diagram, each of these constraint connections has associated with it a margin value dictating the fixed distances of the widget from two sides of the parent layout. Under this configuration, regardless of screen size or the device orientation, the Button object will always be positioned 20 and 15 device-independent pixels (dp) from the top and right-hand edges of the parent ConstraintLayout respectively as specified by the

two constraint connections.

While the above configuration will be acceptable for some situations, it does not provide any flexibility in terms of allowing the ConstraintLayout layout engine to adapt the position of the widget in order to respond to device rotation and to support screens of different sizes. To add this responsiveness to the layout it is necessary to implement opposing constraints.

17.1.3 Opposing Constraints

Two constraints operating along the same axis on a single widget are referred to as *opposing constraints*. In other words, a widget with constraints on both its left and right-hand sides is considered to have horizontally opposing constraints. [Figure 17-2](#), for example, illustrates the addition of both horizontally and vertically opposing constraints to the previous layout:

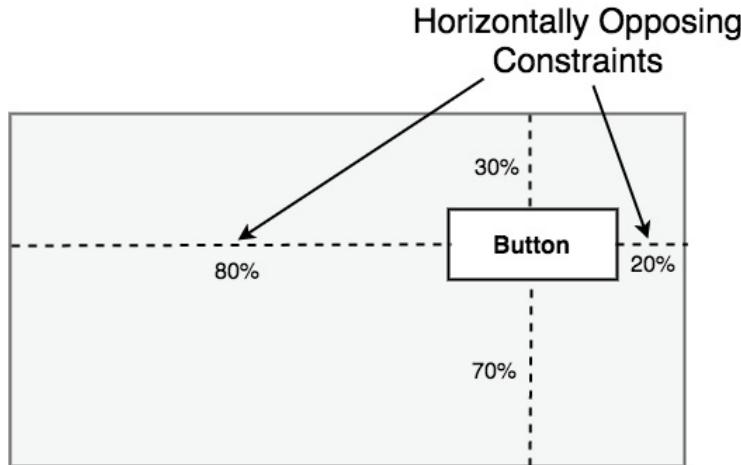


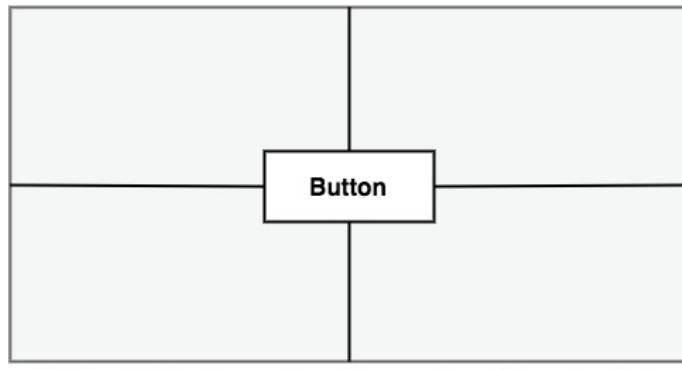
Figure 17-2

The key point to understand here is that once opposing constraints are implemented on a particular axis, the positioning of the widget becomes percentage rather than coordinate based. Instead of being fixed at 20dp from the top of the layout, for example, the widget is now positioned at a point 30% from the top of the layout. In different orientations and when running on larger or smaller screens, the Button will always be in the same location relative to the dimensions of the parent layout.

It is now important to understand that the layout outlined in [Figure 17-2](#) has been implemented using not only opposing constraints, but also by applying *constraint bias*.

17.1.4 Constraint Bias

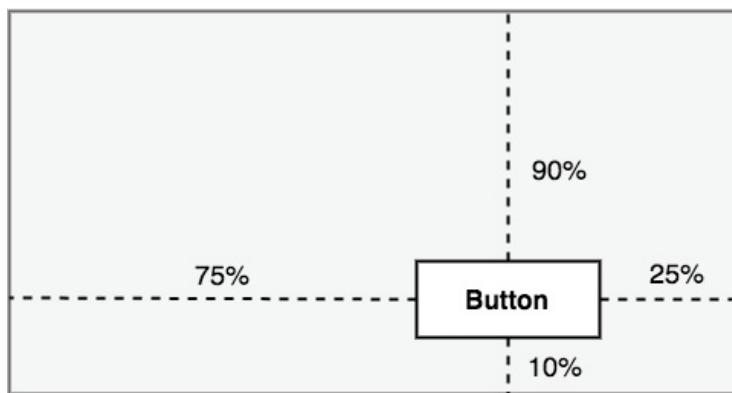
It has now been established that a widget in a ConstraintLayout can potentially be subject to opposing constraint connections. By default, opposing constraints are equal, resulting in the corresponding widget being centered along the axis of opposition. [Figure 17-3](#), for example, shows a widget centered within the containing ConstraintLayout using opposing horizontal and vertical constraints:



Widget Centered by Opposing Constraints

Figure 17-3

To allow for the adjustment of widget position in the case of opposing constraints, the ConstraintLayout implements a feature known as *constraint bias*. Constraint bias allows the positioning of a widget along the axis of opposition to be biased by a specified percentage in favor of one constraint. [Figure 17-4](#), for example, shows the previous constraint layout with a 75% horizontal bias and 10% vertical bias:



Widget Offset using Constraint Bias

Figure 17-4

The next chapter, entitled "[“A Guide to using ConstraintLayout in Android Studio”](#)", will cover these concepts in greater detail and explain how these

features have been integrated into the Android Studio Layout Editor tool. In the meantime, however, a few more areas of the ConstraintLayout class need to be covered.

17.1.5 Chains

ConstraintLayout chains provide a way for the layout behavior of two or more widgets to be defined as a group. Chains can be declared in either the vertical or horizontal axis and configured to define how the widgets in the chain are spaced and sized.

Widgets are chained when connected together by bi-directional constraints.

[Figure 17-5](#), for example, illustrates three widgets chained in this way:

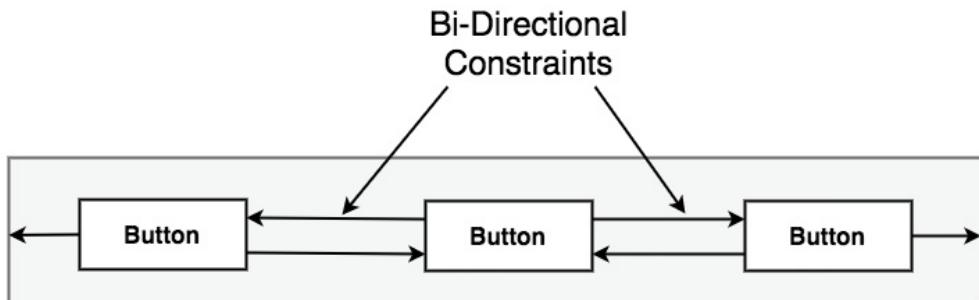


Figure 17-5

The first element in the chain is the *chain head* which translates to the top widget in a vertical chain or, in the case of a horizontal chain, the left-most widget. The layout behavior of the entire chain is primarily configured by setting attributes on the chain head widget.

17.1.6 Chain Styles

The layout behavior of a ConstraintLayout chain is dictated by the *chain style* setting applied to the chain head widget. The ConstraintLayout class currently supports the following chain layout styles:

- **Spread Chain** – The widgets contained within the chain are distributed evenly across the available space. This is the default behavior for chains.



Figure 17-6

- **Spread Inside Chain** – The widgets contained within the chain are spread evenly between the chain head and the last widget in the chain. The head and last widgets are not included in the distribution of spacing.



Figure 17-7

- **Weighted Chain** – Allows the space taken up by each widget in the chain to be defined via weighting properties.

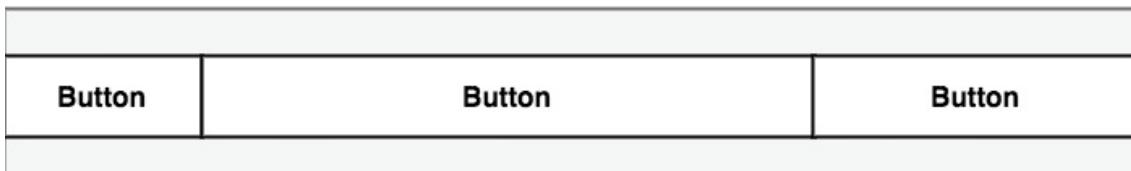


Figure 17-8

- **Packed Chain** – The widgets that make up the chain are packed together without any spacing. A bias may be applied to control the horizontal or vertical positioning of the chain in relation to the parent container.

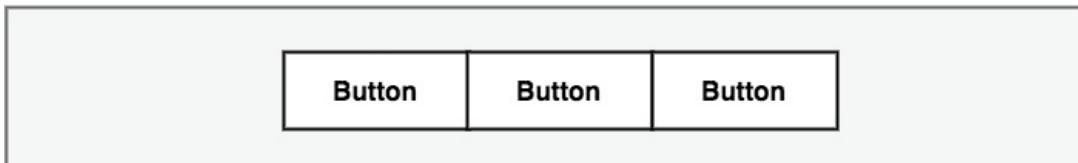


Figure 17-9

17.2 Baseline Alignment

So far, this chapter has only referred to constraints that dictate alignment relative to the sides of a widget (typically referred to as side constraints). A common requirement, however, is for a widget to be aligned relative to the content that it displays rather than the boundaries of the widget itself. To

address this need, ConstraintLayout provides *baseline alignment* support.

As an example, assume that the previous theoretical layout from [Figure 17-1](#) requires a TextView widget to be positioned 40dp to the left of the Button. In this case, the TextView needs to be *baseline aligned* with the Button view. This means that the text within the Button needs to be vertically aligned with the text within the TextView. The additional constraints for this layout would need to be connected as illustrated in [Figure 17-10](#):

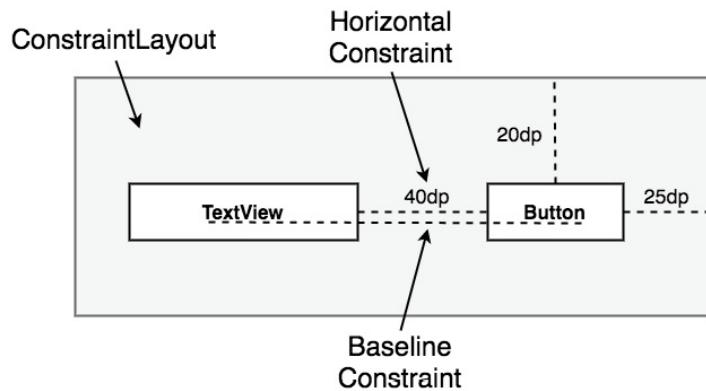


Figure 17-
10

The TextView is now aligned vertically along the baseline of the Button and positioned 40dp horizontally from the Button object's left-hand edge.

17.3 Working with Guidelines

Guidelines are special elements available within the ConstraintLayout that provide an additional target to which constraints may be connected. Multiple guidelines may be added to a ConstraintLayout instance which may, in turn, be configured in horizontal or vertical orientations. Once added, constraint connections may be established from widgets in the layout to the guidelines. This is particularly useful when multiple widgets need to be aligned along an axis. In [Figure 17-11](#), for example, three Button objects contained within a ConstraintLayout are constrained along a vertical guideline:

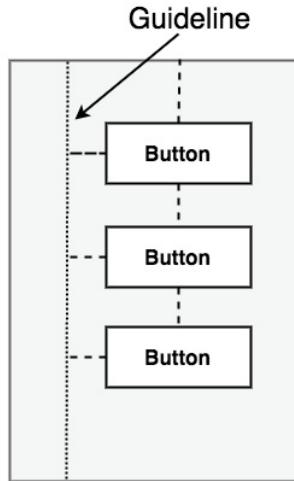


Figure 17-
11

17.4 Configuring Widget Dimensions

Controlling the dimensions of a widget is a key element of the user interface design process. The `ConstraintLayout` provides three options which can be set on individual widgets to manage sizing behavior. These settings are configured individually for height and width dimensions:

- **Fixed** – The widget is fixed to specified dimensions.
- **Match Constraint** – Allows the widget to be resized by the layout engine to satisfy the prevailing constraints. Also referred to as the `AnySize` or `MATCH_CONSTRAINT` option.
- **Wrap Content** – The size of the widget is dictated by the content it contains (i.e. text or graphics).

17.5 Working with Barriers

Rather like guidelines, barriers are virtual views that can be used to constrain views within a layout. As with guidelines, a barrier can be vertical or horizontal and one or more views may be constrained to it (to avoid confusion, these will be referred to as *constrained views*). Unlike guidelines where the guideline remains at a fixed position within the layout, however, the position of a barrier is defined by a set of so called *reference views*. Barriers were introduced to address an issue that occurs with some frequency involving overlapping views. Consider, for example, the layout illustrated in

[Figure 17-12](#) below:

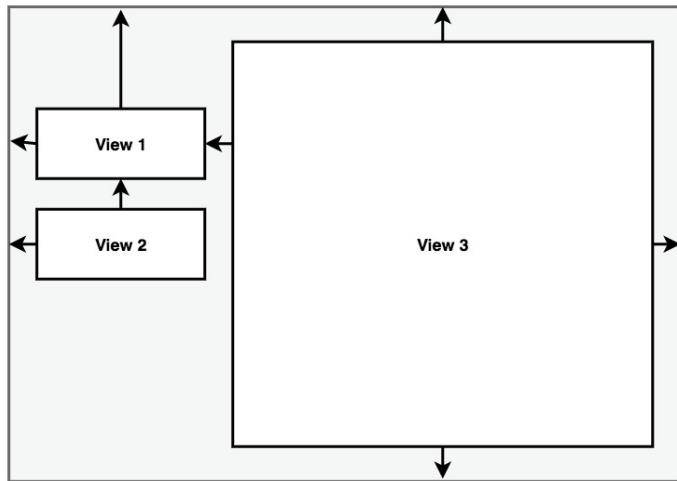


Figure 17-
12

The key points to note about the above layout is that the width of View 3 is set to match constraint mode, and the left-hand edge of the view is connected to the right hand edge of View 1. As currently implemented, an increase in width of View 1 will have the desired effect of reducing the width of View 3:

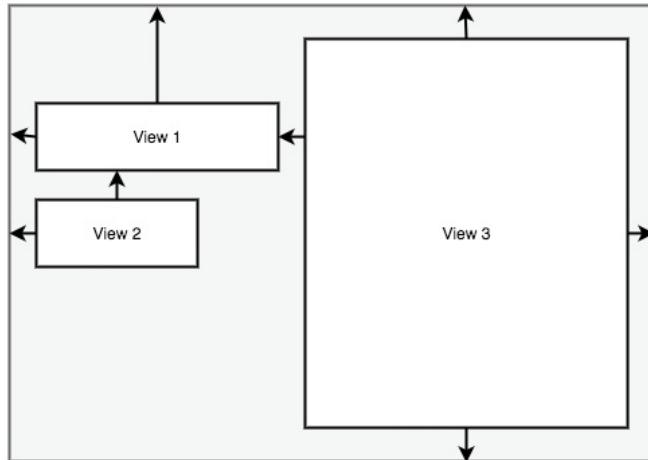


Figure 17-
13

A problem arises, however, if View 2 increases in width instead of View 1:

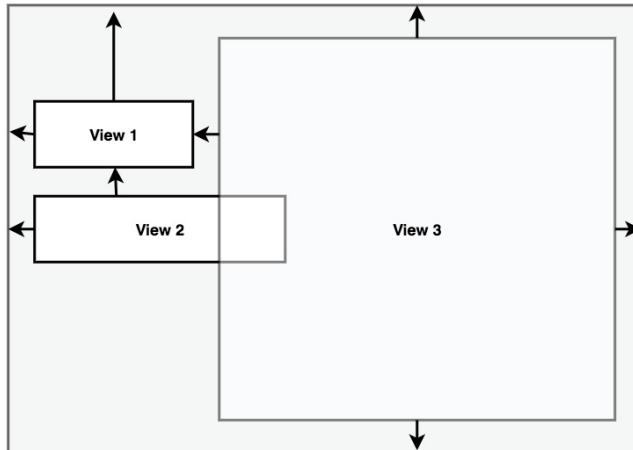


Figure 17-
14

Clearly because View 3 is only constrained by View 1, it does not resize to accommodate the increase in width of View 2 causing the views to overlap.

A solution to this problem is to add a vertical barrier and assign Views 1 and 2 as the barrier's *reference views* so that they control the barrier position. The left-hand edge of View 3 will then be constrained in relation to the barrier, making it a *constrained view*.

Now when either View 1 or View 2 increase in width, the barrier will move to accommodate the widest of the two views, causing the width of View 3 change in relation to the new barrier position:

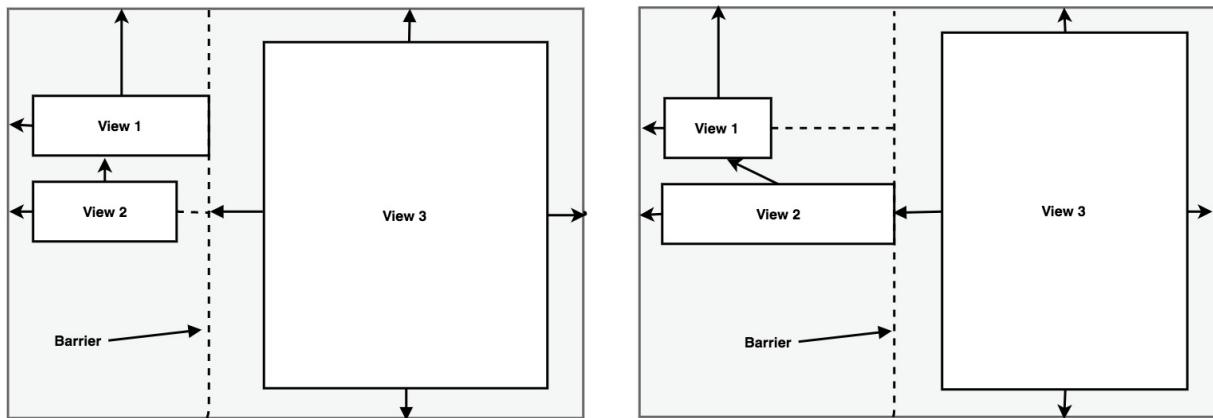


Figure 17-
15

When working with barriers there is no limit to the number of reference views and constrained views that can be associated with a single barrier.

17.6 Ratios

The dimensions of a widget may be defined using ratio settings. A widget could, for example, be constrained using a ratio setting such that, regardless of any resizing behavior, the width is always twice the height dimension.

17.7 ConstraintLayout Advantages

ConstraintLayout provides a level of flexibility that allows many of the features of older layouts to be achieved with a single layout instance where it would previously have been necessary to nest multiple layouts. This has the benefit of avoiding the problems inherent in layout nesting by allowing so called “flat” or “shallow” layout hierarchies to be designed leading both to less complex layouts and improved user interface rendering performance at runtime.

ConstraintLayout was also implemented with a view to addressing the wide range of Android device screen sizes available on the market today. The flexibility of ConstraintLayout makes it easier for user interfaces to be designed that respond and adapt to the device on which the app is running.

Finally, as will be demonstrated in the chapter entitled [“A Guide to using ConstraintLayout in Android Studio”](#), the Android Studio Layout Editor tool has been enhanced specifically for ConstraintLayout-based user interface design.

17.8 ConstraintLayout Availability

Although introduced with Android 7, ConstraintLayout is provided as a separate support library from the main Android SDK and is compatible with older Android versions as far back as API Level 9 (Gingerbread). This allows apps that make use of this new layout to run on devices running much older versions of Android.

17.9 Summary

ConstraintLayout is a layout manager introduced with Android 7. It is designed to ease the creation of flexible layouts that adapt to the size and orientation of the many Android devices now on the market. ConstraintLayout uses constraints to control the alignment and positioning of widgets in relation to the parent ConstraintLayout instance, guidelines, barriers and the other widgets in the layout. ConstraintLayout is the default

layout for newly created Android Studio projects and is the recommended choice when designing user interface layouts. With this simple yet flexible approach to layout management, complex and responsive user interfaces can be implemented with surprising ease.

18. 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.

18.1 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 18-1](#), side by side:

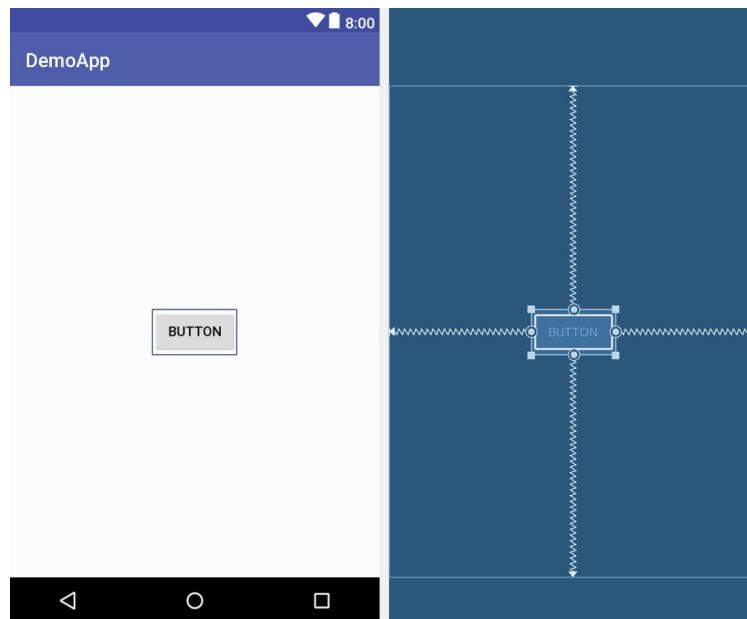


Figure 18-1

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 18-1](#) 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 18-2](#):

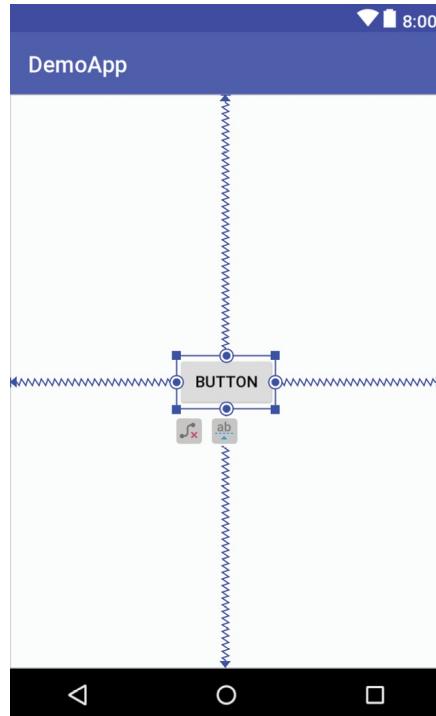


Figure 18-2

The appearance of constraint connections in both views can be change using the toolbar menu shown in [Figure 18-3](#):

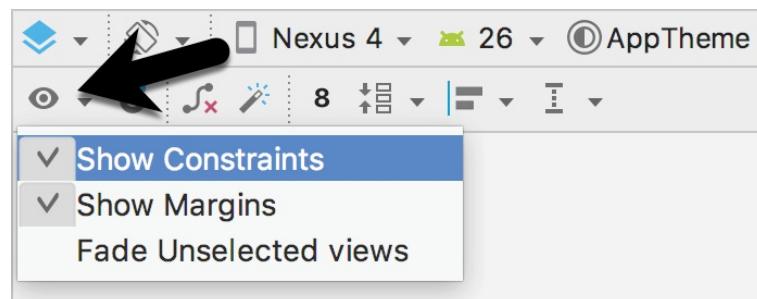


Figure 18-3

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.

18.2 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 18-4](#):

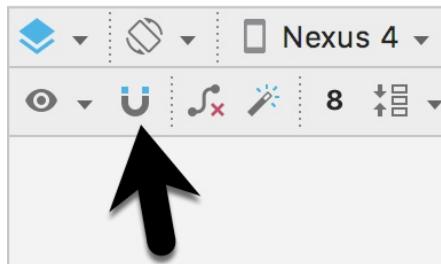


Figure 18-4

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. In the event that any of the automatic constraint connections fail to provide the desired behavior, these may be changed manually as outlined later in this chapter.

18.3 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 18-5](#)) is clicked:

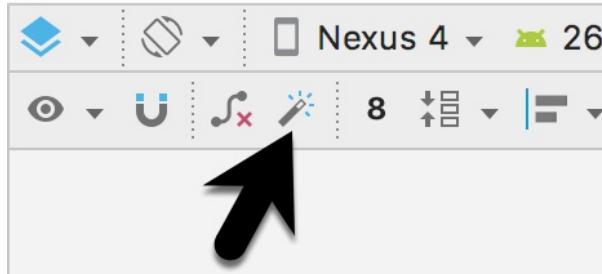


Figure 18-5

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.

18.4 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 18-6](#):

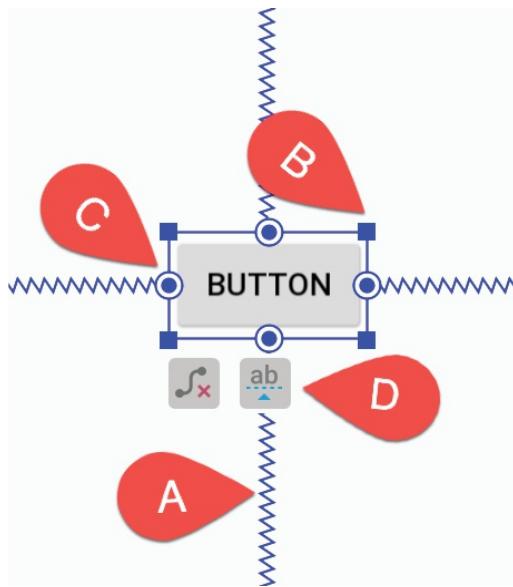


Figure 18-6

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 18-7](#). When connecting to the side of another widget, simply drag the line to the side constraint handle of that widget and, when it turns green, release the line.

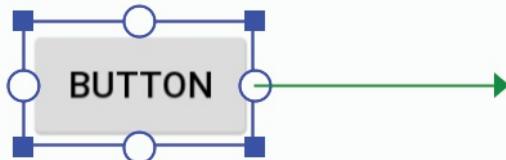


Figure 18-7

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 click on the button displaying the letters 'ab' (referenced by marker D in [Figure 18-6](#)). To establish a constraint connection from a baseline constraint handle, simply hover the mouse pointer over the handle until it begins to flash before clicking and dragging to the target (such as the baseline anchor of another widget as shown in [Figure 18-8](#)). When the destination anchor begins to flash green, release the mouse button to make the constraint connection:



Figure 18-8

To hide the baseline anchors, simply click on the baseline button a second time.

18.5 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 18-9](#). 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:

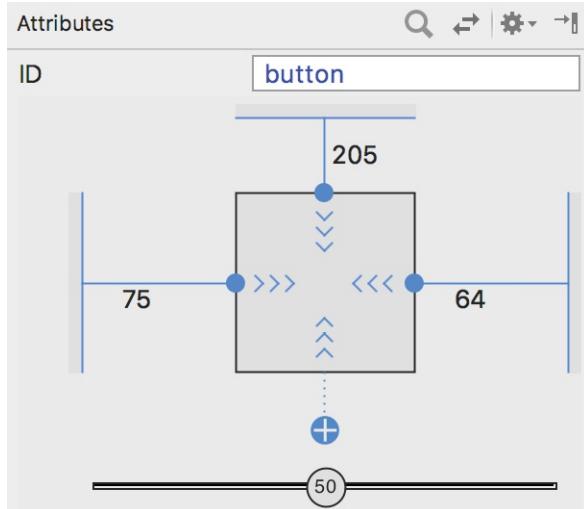


Figure 18-9

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.

18.6 Deleting Constraints

To delete an individual constraint, simply click within the anchor to which it is connected. The constraint will then be deleted from the layout (when hovering over the anchor it will glow red to indicate that clicking will perform a deletion):

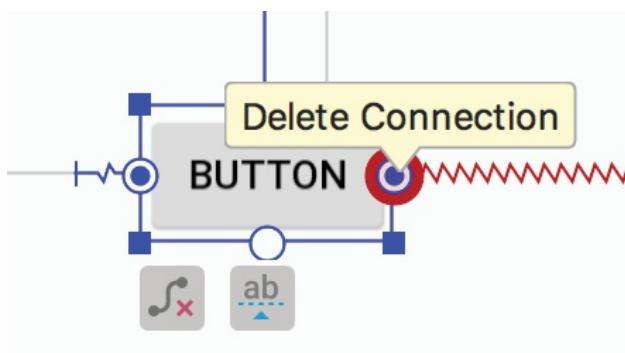


Figure 18-

Alternatively, remove all of the constraints on a widget by selecting it and clicking on the *Delete All Constraints* button which appears next to the widget when it is selected in the layout as highlighted in [Figure 18-11](#):

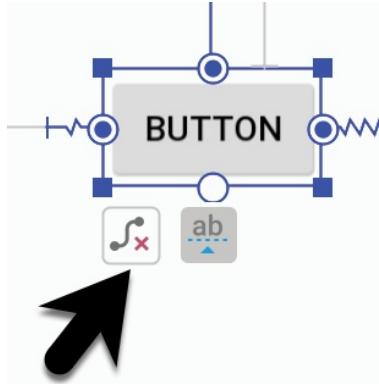


Figure 18-
11

To remove all of the constraints from every widget in a layout, use the toolbar button highlighted in [Figure 18-12](#):

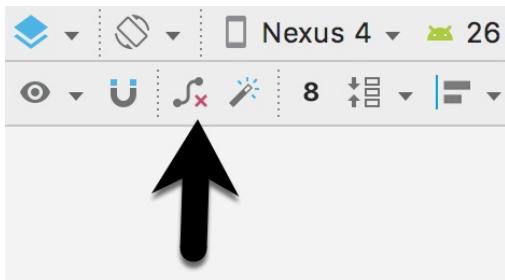


Figure 18-
12

18.7 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 18-13](#). 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.

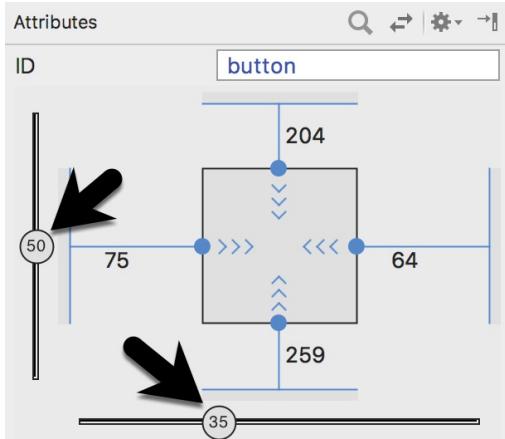


Figure 18-
13

18.8 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 18-14](#):

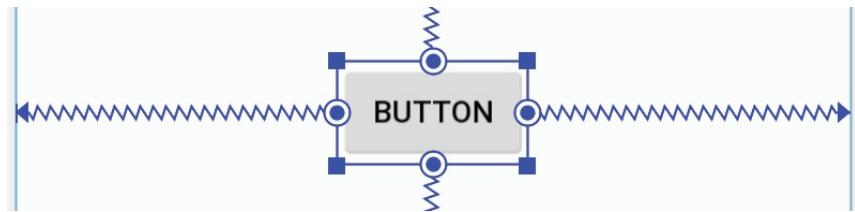


Figure 18-
14

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 18-15](#), for example, shows the same button with 100% horizontal bias applied:

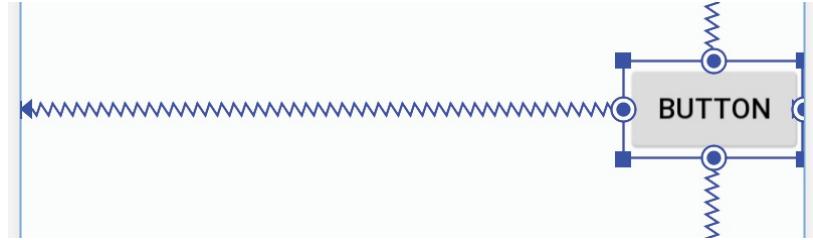


Figure 18-
15

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 18-16](#), 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%.

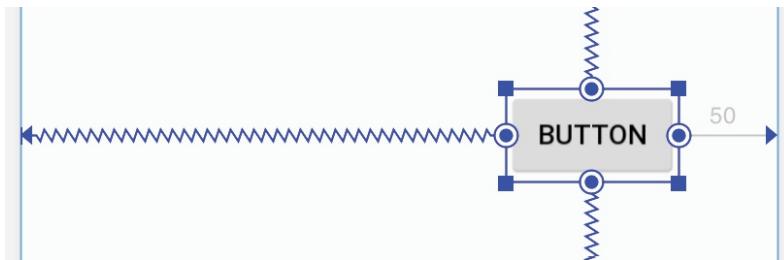


Figure 18-
16

Existing margin values on a widget can be modified from within the Inspector. As can be seen in [Figure 18-17](#), a dropdown 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.

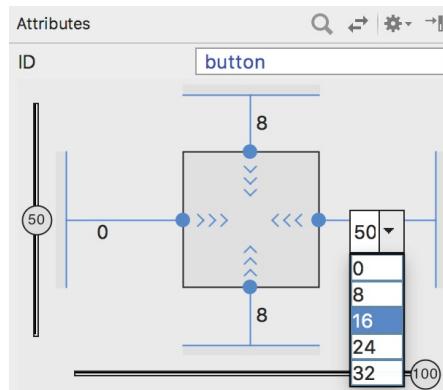


Figure 18-
17

The default margin for new constraints can be changed at any time using the option in the toolbar highlighted in [Figure 18-18](#):

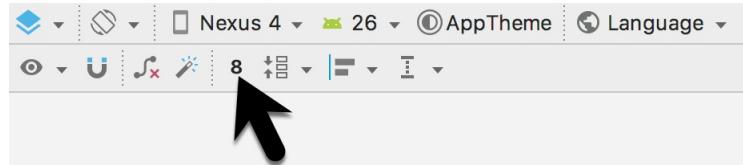


Figure 18-
18

18.9 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 18-19](#).

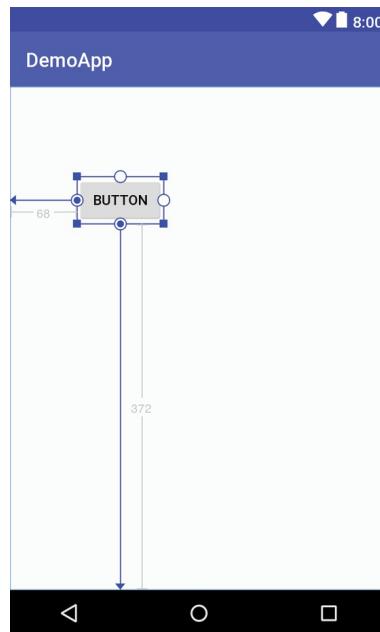


Figure 18-
19

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 18-20](#)). 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 18-
20

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):

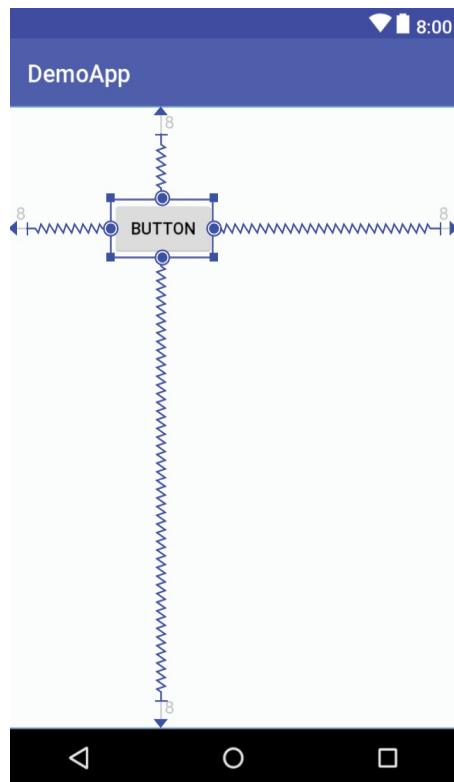


Figure 18-
21

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:

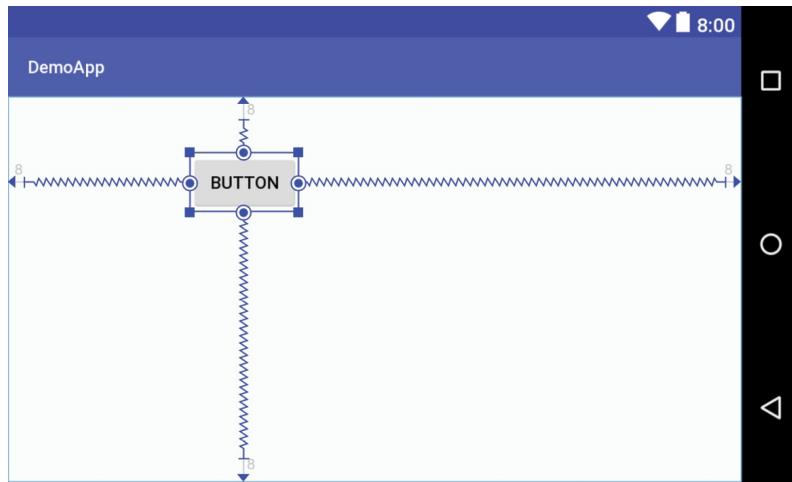


Figure 18-
22

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.

18.10 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 18-23](#):

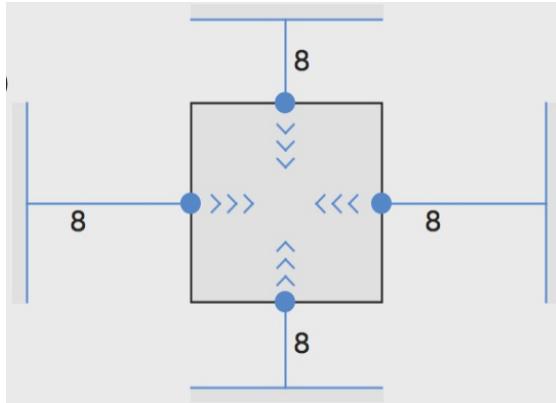


Figure 18-
23

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 18-1

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 18-24](#), for example, only the width of the view has been set to match constraint:



Figure 18-
24

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 18-25](#)). When used, the currently selected widget will increase in size horizontally or vertically to fill the available space around it.

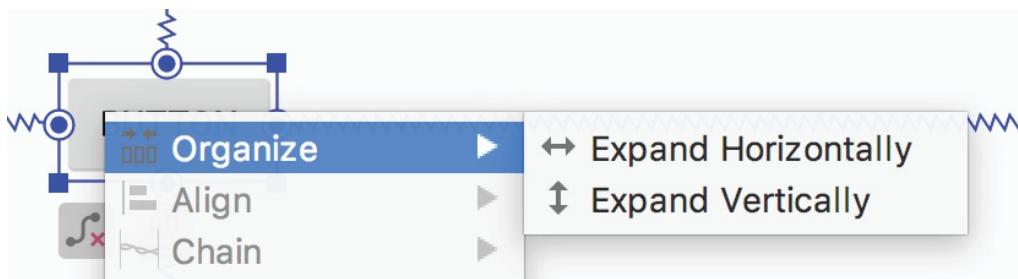


Figure 18-
25

18.11 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 *Add Vertical Guideline* or *Add Horizontal Guideline* menu option or using the toolbar menu options as shown in [Figure 18-26](#):

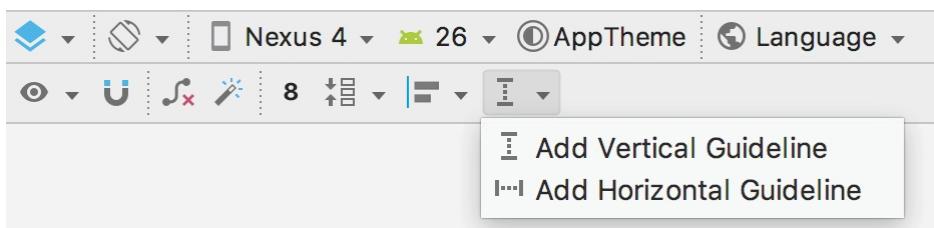


Figure 18-
26

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 18-27](#), 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.

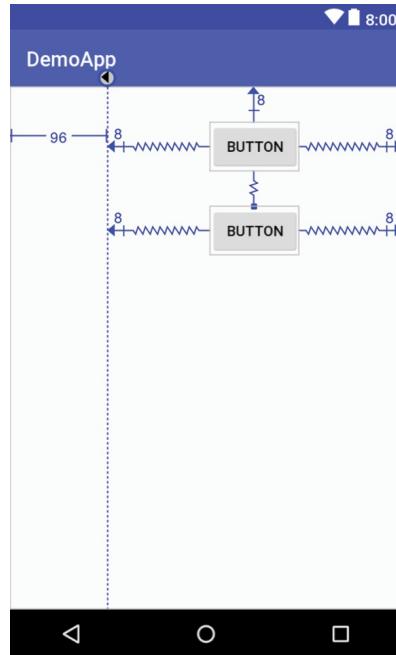


Figure 18-
27

Alternatively, the guideline may be positioned as a percentage of 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 18-28](#), for example, shows a guideline positioned based on percentage:

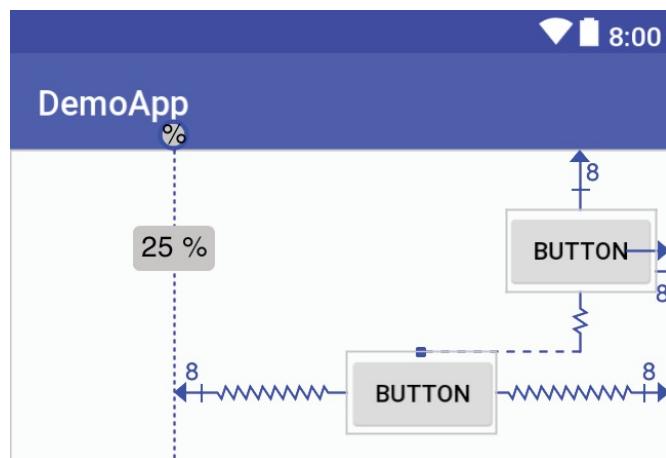


Figure 18-
28

18.12 Adding Barriers

Barriers are added by right-clicking on the layout and selecting either the *Add Vertical Barrier* or *Add Horizontal Barrier* menu option, or using the toolbar menu options as shown in [Figure 18-29](#):

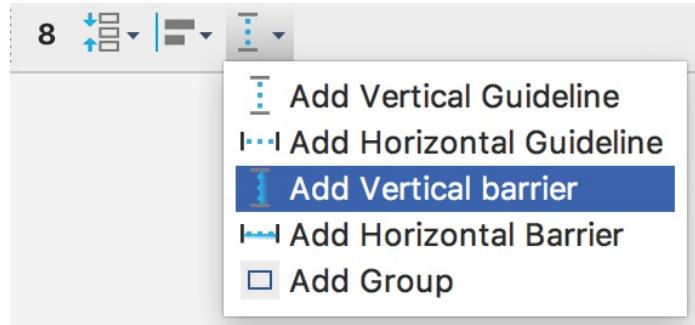


Figure 18-
29

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

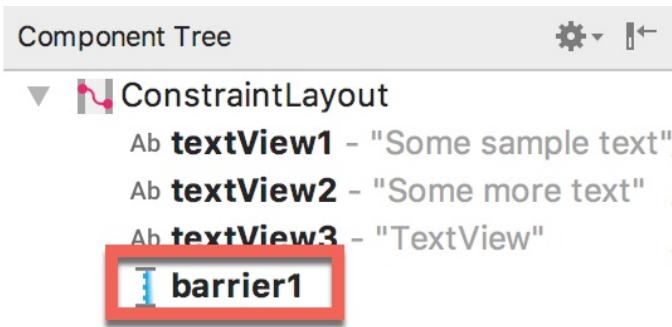


Figure 18-
30

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 18-31](#), for example, widgets named textView1 and textView2 have been assigned as the reference widgets for barrier1:

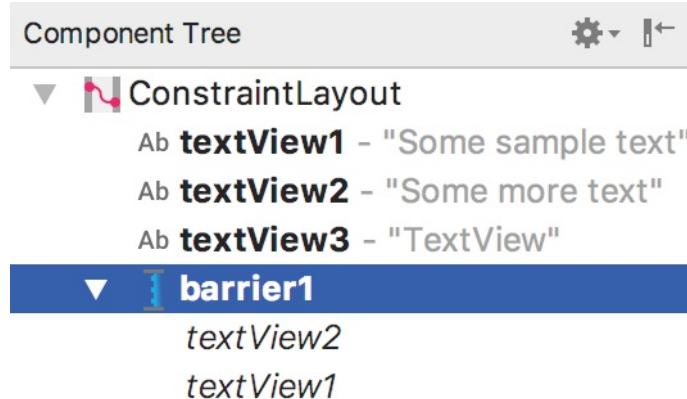


Figure 18-

31

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:

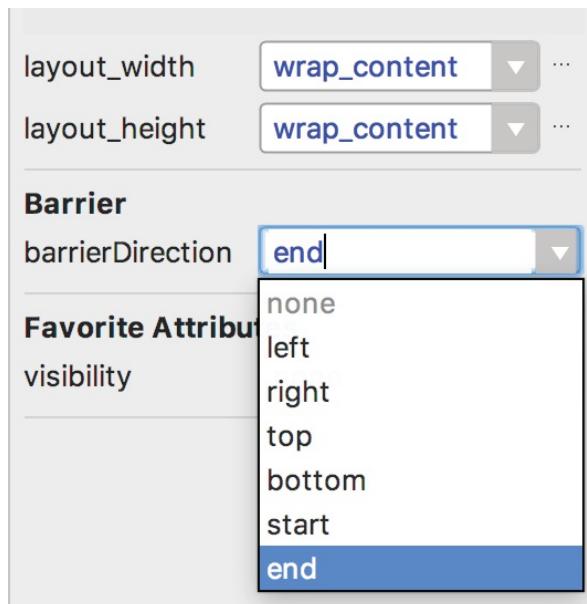


Figure 18-

32

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*:

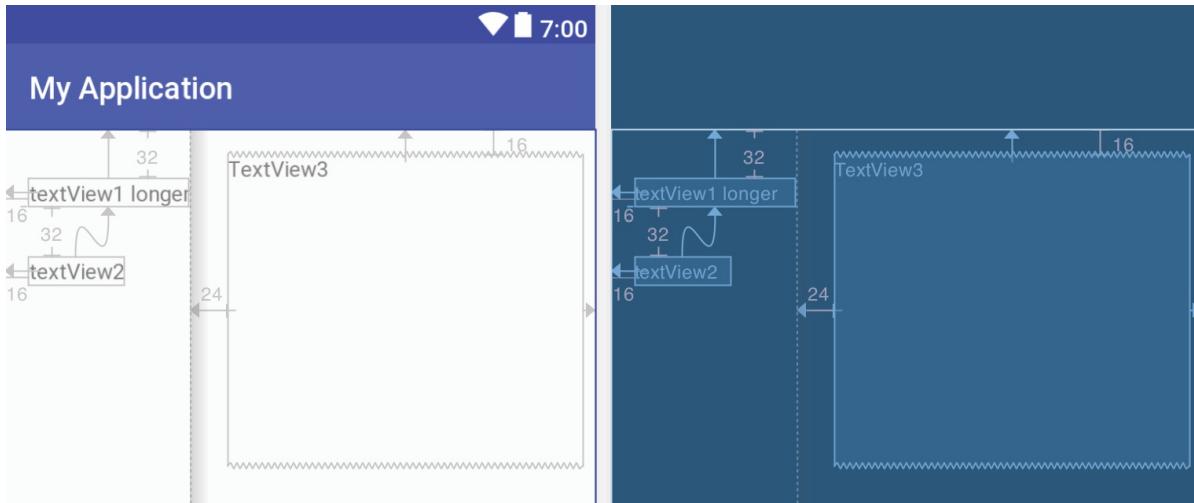


Figure 18-
33

18.13 Widget Group Alignment

The Android Studio Layout Editor tool provides a range of alignment 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:

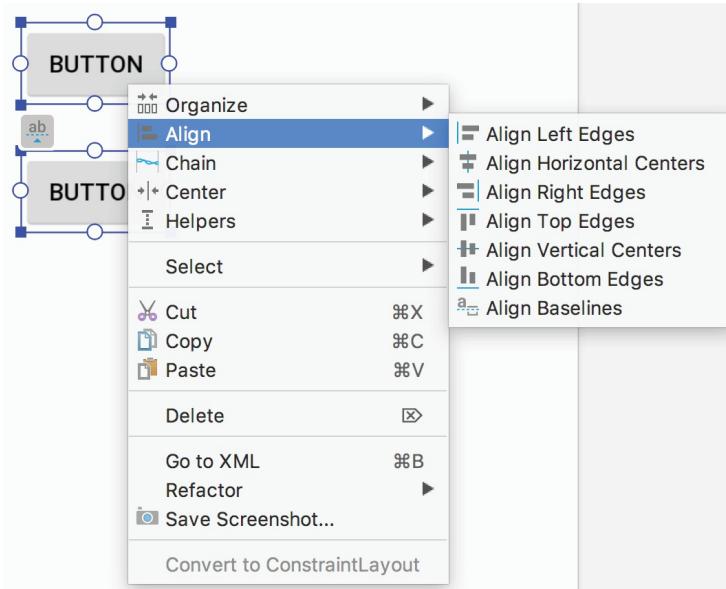


Figure 18-
34

As shown in [Figure 18-35](#) below, these options are also available as buttons in

the Layout Editor toolbar:

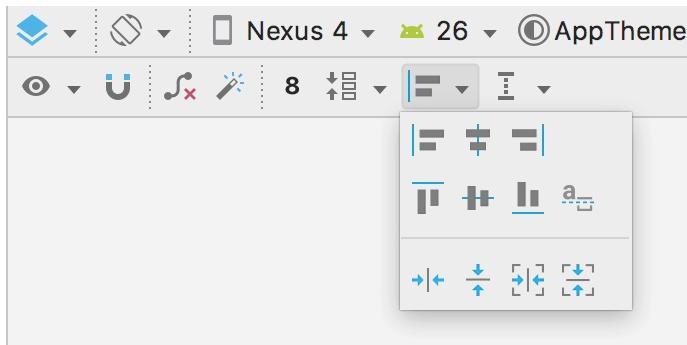


Figure 18-
35

18.14 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:

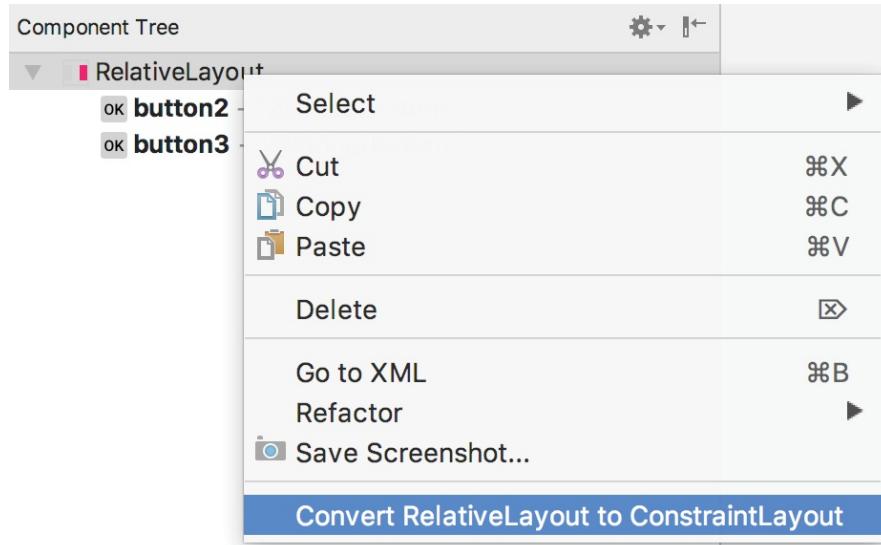


Figure 18-
36

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.

18.15 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.

19. Working with ConstraintLayout Chains and Ratios in Android Studio

The previous chapters have introduced the key features of the `ConstraintLayout` class and outlined the best practices for `ConstraintLayout`-based user interface design within the Android Studio Layout Editor. Although the concepts of `ConstraintLayout` chains and ratios were outlined in the chapter entitled "[A Guide to the Android ConstraintLayout](#)", we have not yet addressed how to make use of these features within the Layout Editor. The focus of this chapter, therefore, is to provide practical steps on how to create and manage chains and ratios when using the `ConstraintLayout` class.

19.1 Creating a Chain

Chains may be implemented either by adding a few lines to the XML layout resource file of an activity or by using some chain specific features of the Layout Editor.

Consider a layout consisting of three `Button` widgets constrained so as to be positioned in the top-left, top-center and top-right of the `ConstraintLayout` parent as illustrated in [Figure 19-1](#):

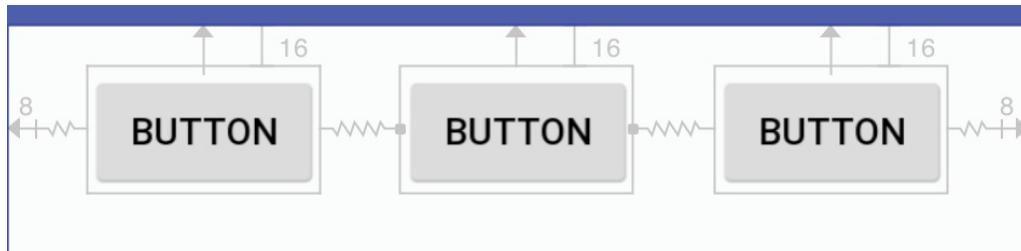


Figure 19-1

To represent such a layout, the XML resource layout file might contain the following entries for the button widgets:

```
<Button  
    android:id="@+id/button1"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginStart="8dp"  
    android:layout_marginTop="16dp"  
    android:text="Button"  
    app:layout_constraintStart_toStartOf="parent"
```

```

    app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintEnd_toStartOf="@+id/button3"
    app:layout_constraintStart_toEndOf="@+id/button1"
    app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

```

As currently configured, there are no bi-directional constraints to group these widgets into a chain. To address this, additional constraints need to be added from the right-hand side of button1 to the left side of button2, and from the left side of button3 to the right side of button2 as follows:

```

<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintRight_toLeftOf="@+id/button2" />

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"

```

```

        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="16dp"
        android:text="Button"
        app:layout_constraintEnd_toStartOf="@+id/button3"
        app:layout_constraintStart_toEndOf="@+id/button1"
        app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintLeft_toRightOf="@+id/button2" />
```

With these changes, the widgets now have bi-directional horizontal constraints configured. This essentially constitutes a ConstraintLayout chain which is represented visually within the Layout Editor by chain connections as shown in [Figure 19-2](#) below. Note that in this configuration the chain has defaulted to the spread chain style.



Figure 19-2

A chain may also be created by right-clicking on one of the views and selecting the *Chain -> Create Horizontal Chain* or *Chain -> Create Vertical Chain* menu options.

19.2 Changing the Chain Style

If no chain style is configured, the ConstraintLayout will default to the spread chain style. The chain style can be altered by selecting any of the widgets in the chain and clicking on the chain button as highlighted in [Figure 19-3](#):

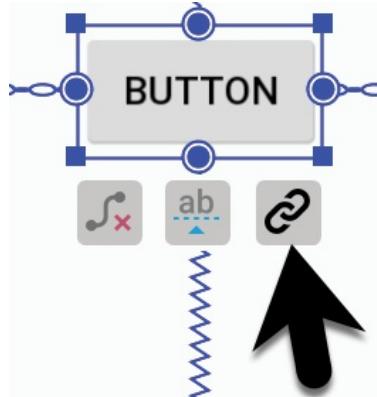


Figure 19-3

Each time the chain button is clicked the style will switch to another setting in the order of spread, spread inside and packed.

Alternatively, the style may be specified in the Attributes tool window by clicking on the *View all attributes* link, unfolding the *Constraints* section and changing either the *horizontal_chainStyle* or *vertical_chainStyle* property depending on the orientation of the chain:

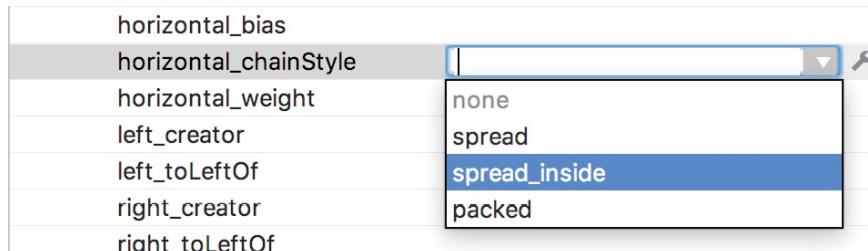


Figure 19-4

19.3 Spread Inside Chain Style

[Figure 19-5](#) illustrates the effect of changing the chain style to spread inside chain style using the above techniques:

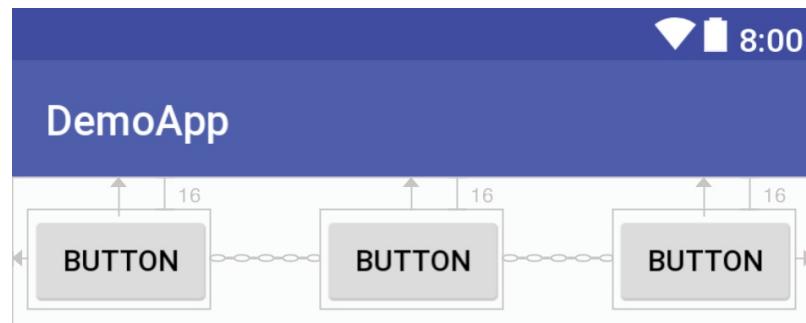


Figure 19-5

19.4 Packed Chain Style

Using the same technique, changing the chain style property to *packed* causes the layout to change as shown in [Figure 19-6](#):

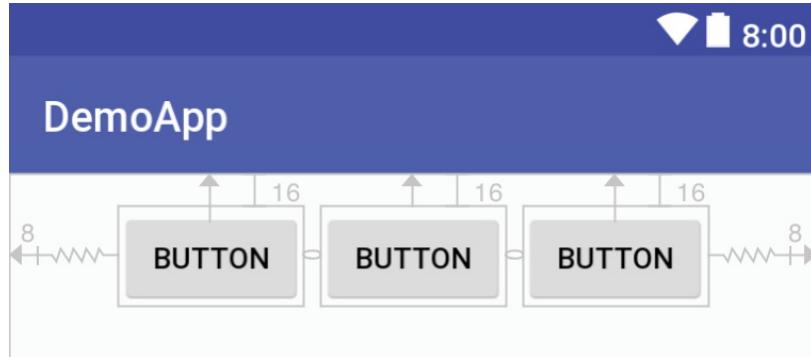


Figure 19-6

19.5 Packed Chain Style with Bias

The positioning of the packed chain may be influenced by applying a bias value. The bias can be any value between 0.0 and 1.0, with 0.5 representing the center of the parent. Bias is controlled by selecting the chain head widget and assigning a value to the *horizontal_bias* or *vertical_bias* attribute in the Attributes panel. [Figure 19-7](#) shows a packed chain with a horizontal bias setting of 0.2:

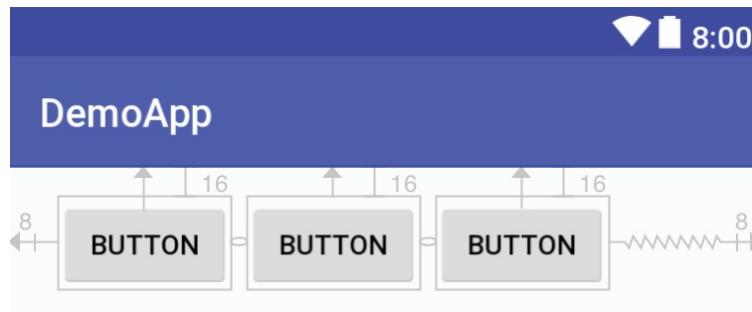


Figure 19-7

19.6 Weighted Chain

The final area of chains to explore involves weighting of the individual widgets to control how much space each widget in the chain occupies within the available space. A weighted chain may only be implemented using the *spread* chain style and any widget within the chain that is to respond to the *weight* property must have the corresponding dimension property (height for a vertical chain and width for a horizontal chain) configured for *match constraint* mode. Match constraint mode for a widget dimension may be

configured by selecting the widget, displaying the Attributes panel and changing the dimension to *match_constraint*. In [Figure 19-8](#), for example, the *layout_width* constraint for button1 has been set to *match_constraint* to indicate that the width of the widget is to be determined based on the prevailing constraint settings:

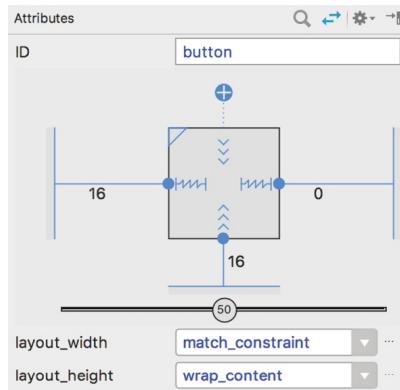


Figure 19-8

Assuming that the spread chain style has been selected, and all three buttons have been configured such that the width dimension is set to match the constraints, the widgets in the chain will expand equally to fill the available space:

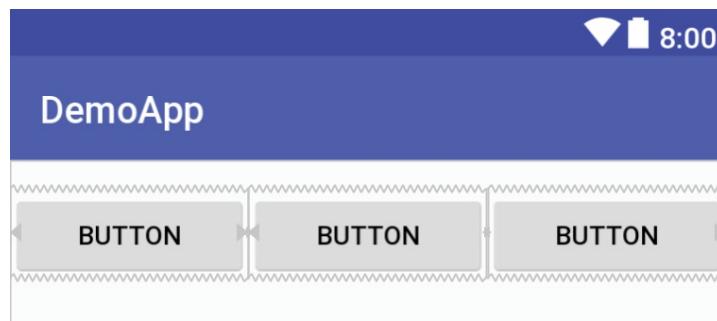


Figure 19-9

The amount of space occupied by each widget relative to the other widgets in the chain can be controlled by adding weight properties to the widgets. [Figure 19-10](#) shows the effect of setting the *horizontal_weight* property to 4 on button1, and to 2 on both button2 and button3:

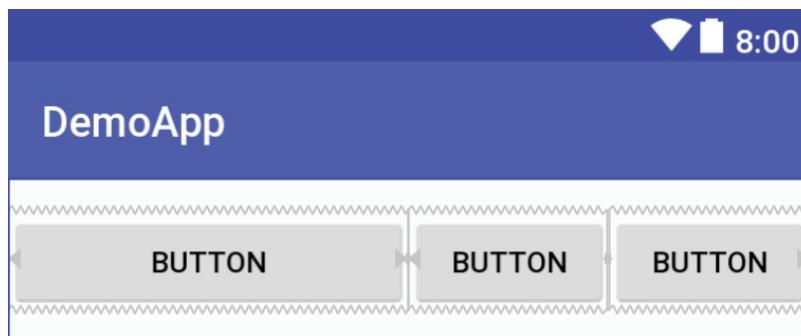


Figure 19-
10

As a result of these weighting values, button1 occupies half of the space (4/8), while button2 and button3 each occupy one quarter (2/8) of the space.

19.7 Working with Ratios

ConstraintLayout ratios allow one dimension of a widget to be sized relative to the widget's other dimension (otherwise known as aspect ratio). An aspect ratio setting could, for example, be applied to an ImageView to ensure that its width is always twice its height.

A dimension ratio constraint is configured by setting the constrained dimension to match constraint mode and configuring the *layout_constraintDimensionRatio* attribute on that widget to the required ratio. This ratio value may be specified either as a float value or a *width:height* ratio setting. The following XML excerpt, for example, configures a ratio of 2:1 on an ImageView widget:

```
<ImageView  
    android:layout_width="0dp"  
    android:layout_height="100dp"  
    android:id="@+id/imageView"  
    app:layout_constraintDimensionRatio="2:1" />
```

The above example demonstrates how to configure a ratio when only one dimension is set to match constraint. A ratio may also be applied when both dimensions are set to match constraint mode. This involves specifying the ratio preceded with either an H or a W to indicate which of the dimensions is constrained relative to the other.

Consider, for example, the following XML excerpt for an ImageView object:

```
<ImageView  
    android:layout_width="0dp"
```

```

    android:layout_height="0dp"
    android:id="@+id/imageView"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintDimensionRatio="w,1:3" />

```

In the above example the height will be defined subject to the constraints applied to it. In this case constraints have been configured such that it is attached to the top and bottom of the parent view, essentially stretching the widget to fill the entire height of the parent. The width dimension, on the other hand, has been constrained to be one third of the ImageView's height dimension. Consequently, whatever size screen or orientation the layout appears on, the ImageView will always be the same height as the parent and the width one third of that height.

The same results may also be achieved without the need to manually edit the XML resource file. Whenever a widget dimension is set to match constraint mode, a ratio control toggle appears in the Inspector area of the property panel. [Figure 19-11](#), for example, shows the layout width and height attributes of a button widget set to match constraint mode and 100dp respectively, and highlights the ratio control toggle in the widget sizing preview:

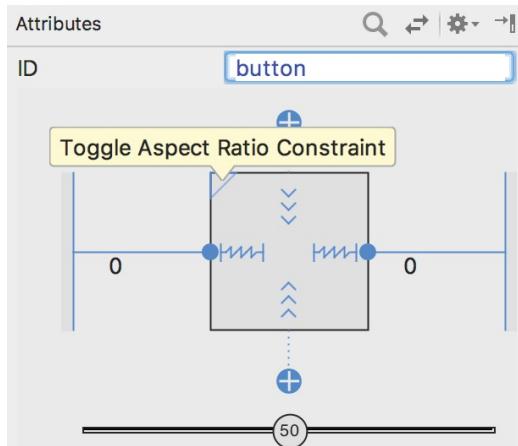


Figure 19-
11

By default the ratio sizing control is toggled off. Clicking on the control enables the ratio constraint and displays an additional field where the ratio may be changed:

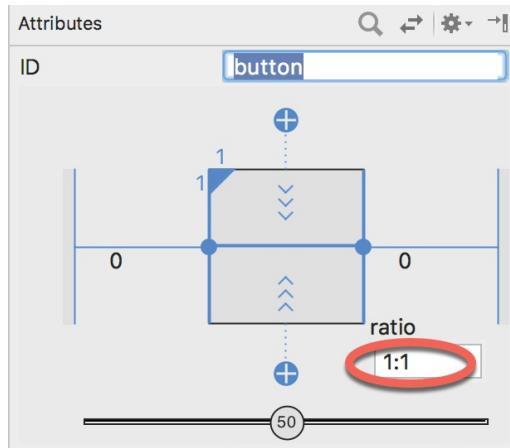


Figure 19-
12

19.8 Summary

Both chains and ratios are powerful features of the ConstraintLayout class intended to provide additional options for designing flexible and responsive user interface layouts within Android applications. As outlined in this chapter, the Android Studio Layout Editor has been enhanced to make it easier to use these features during the user interface design process.

20. An Android Studio Layout Editor ConstraintLayout Tutorial

By far the easiest and most productive way to design a user interface for an Android application is to make use of the Android Studio Layout Editor tool. The goal of this chapter is to provide an overview of how to create a ConstraintLayout-based user interface using this approach. The exercise included in this chapter will also be used as an opportunity to outline the creation of an activity starting with a “bare-bones” Android Studio project.

Having covered the use of the Android Studio Layout Editor, the chapter will also introduce the Layout Inspector tool.

20.1 An Android Studio Layout Editor Tool Example

The first step in this phase of the example is to create a new Android Studio project. Begin, therefore, by launching Android Studio and closing any previously opened projects by selecting the *File -> Close Project* menu option. Within the Android Studio welcome screen click on the *Start a new Android Studio project* quick start option to display the first screen of the new project dialog.

Enter *LayoutSample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button and setting the minimum SDK to API 14: Android 4.0 (IceCreamSandwich).

In previous examples, we have requested that Android Studio create a template activity for the project. We will, however, be using this tutorial to learn how to create an entirely new activity and corresponding layout resource file manually, so click *Next* once again and make sure that the *Add No Activity* option is selected before clicking on *Finish* to create the new project.

20.2 Creating a New Activity

Once the project creation process is complete, the Android Studio main window should appear with no tool windows open.

The next step in the project is to create a new activity. This will be a valuable learning exercise since there are many instances in the course of developing

Android applications where new activities need to be created from the ground up.

Begin by displaying the Project tool window using the Alt-1/Cmd-1 keyboard shortcut. Once displayed, unfold the hierarchy by clicking on the right facing arrows next to the entries in the Project window. The objective here is to gain access to the `app -> java -> com.ebookfrenzy.layoutsample` folder in the project hierarchy. Once the package name is visible, right-click on it and select the `New -> Activity -> Empty Activity` menu option as illustrated in [Figure 20-1](#):

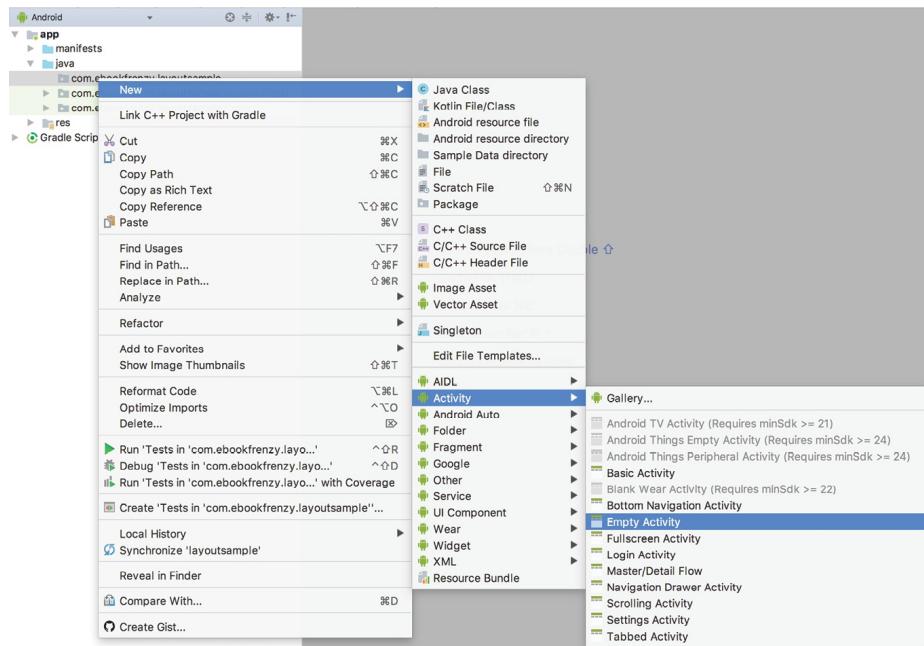


Figure 20-1

In the resulting *New Activity* dialog, name the new activity *LayoutSampleActivity* and the layout *activity_layout_sample*. The activity will, of course, need a layout resource file so make sure that the *Generate Layout File* option is enabled.

In order for an application to be able to run on a device it needs to have an activity designated as the *launcher activity*. Without a launcher activity, the operating system will not know which activity to start up when the application first launches and the application will fail to start. Since this example only has one activity, it needs to be designated as the launcher activity for the application so make sure that the *Launcher Activity* option is enabled before clicking on the *Finish* button.

At this point Android Studio should have added two files to the project. The Java source code file for the activity should be located in the *app -> java -> com.ebookfrenzy.layoutsample* folder.

In addition, the XML layout file for the user interface should have been created in the *app -> res -> layout* folder. Note that the Empty Activity template was chosen for this activity so the layout is contained entirely within the *activity_layout_sample.xml* file and there is no separate content layout file.

Finally, the new activity should have been added to the *AndroidManifest.xml* file and designated as the launcher activity. The manifest file can be found in the project window under the *app -> manifests* folder and should contain the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ebookfrenzy.layoutsample">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".LayoutSampleActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

20.3 Preparing the Layout Editor Environment

Locate and double-click on the *activity_layout_sample.xml* layout file located in the *app -> res -> layout* folder to load it into the Layout Editor tool. Since the purpose of this tutorial is to gain experience with the use of constraints,

turn off the Autoconnect feature using the button located in the Layout Editor toolbar. Once disabled, the button will appear with a line through it as is the case in [Figure 20-2](#):

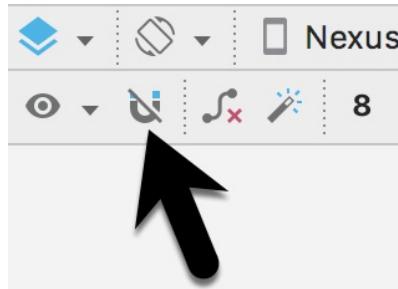


Figure 20-2

The user interface design will also make use of the ImageView object to display an image. Before proceeding, this image should be added to the project ready for use later in the chapter. This file is named *galaxys6.png* and can be found in the *project_icons* folder of the sample code download available from the following URL:

<http://www.ebookfrenzy.com/retail/androidstudio30/index.php>

Locate this image in the file system navigator for your operating system and copy the image file. Right-click on the *app -> res -> drawable* entry in the Project tool window and select Paste from the menu to add the file to the folder. When the copy dialog appears, click on OK to accept the default settings.

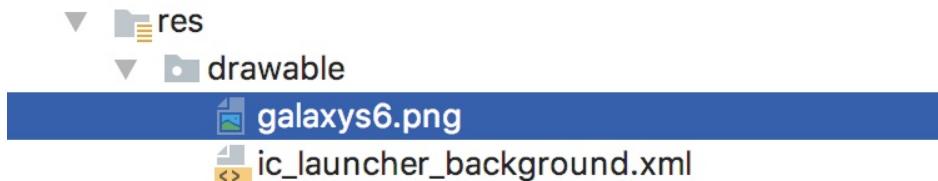


Figure 20-3

20.4 Adding the Widgets to the User Interface

From within the *Images* palette category, drag an ImageView object into the center of the display view. Note that horizontal and vertical dashed lines appear indicating the center axes of the display. When centered, release the mouse button to drop the view into position. Once placed within the layout, the Resources dialog will appear seeking the image to be displayed within the view. In the search bar located at the top of the dialog, enter “galaxy” to locate the *galaxys6.png* resource as illustrated in [Figure 20-4](#).

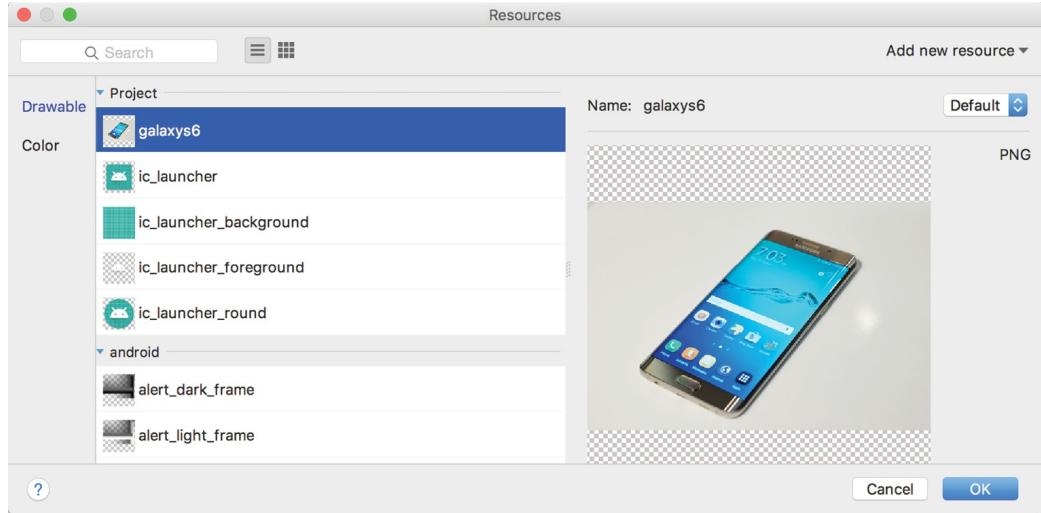


Figure 20-4

Select the image and click on OK to assign it to the ImageView object. If necessary, adjust the size of the ImageView using the resize handles and reposition it in the center of the layout. At this point the layout should match [Figure 20-5](#):

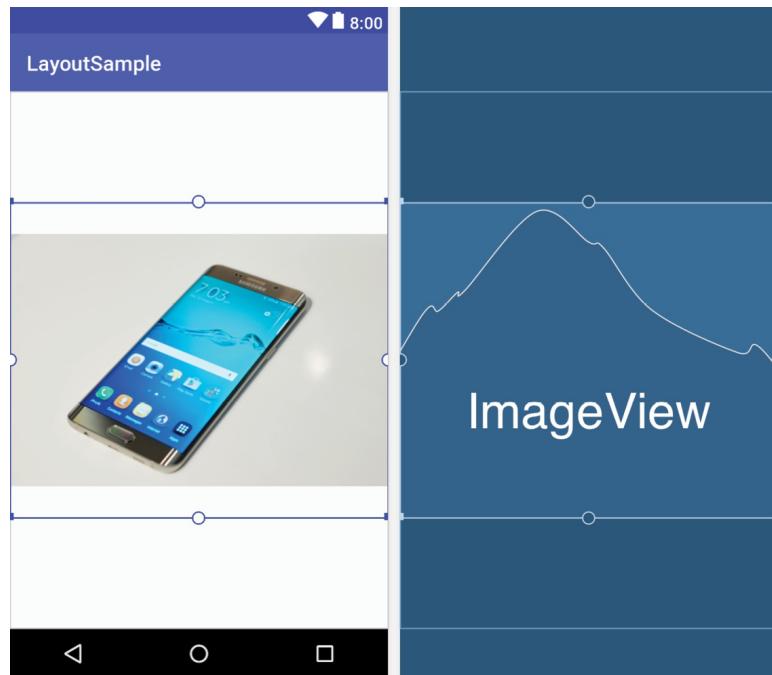


Figure 20-5

Click and drag a TextView object from the Text section of the palette and position it so that it appears above the ImageView as illustrated in [Figure 20-6](#).

Using the Attributes panel, change the *textSize* property to 24sp, the

textAlignment setting to center and the text to “Samsung Galaxy S6”.

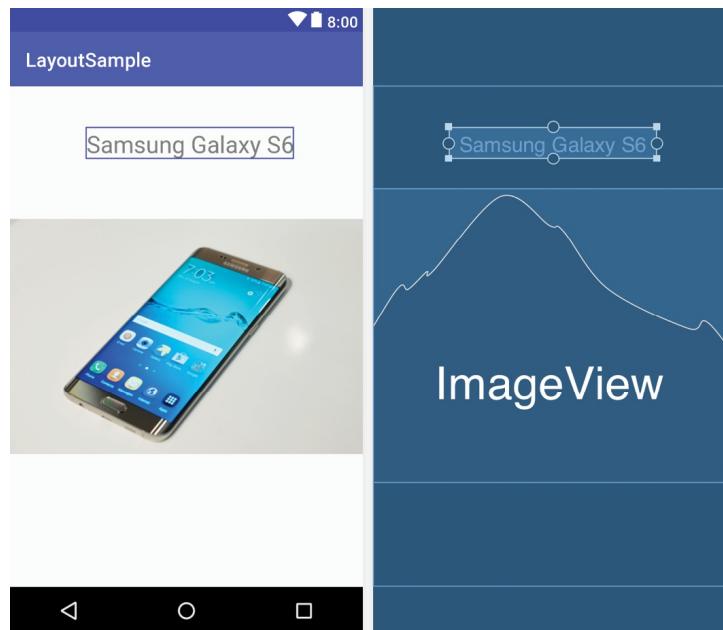


Figure 20-6

Next, add three Button widgets along the bottom of the layout and set the text attributes of these views to “Buy Now”, “Pricing” and “Details”. The completed layout should now match [Figure 20-7](#):



Figure 20-7

At this point, the widgets are not sufficiently constrained for the layout engine to be able to position and size the widgets at runtime. Were the app to

run now, all of the widgets would be positioned in the top left-hand corner of the display.

With the widgets added to the layout, use the device rotation button located in the Layout Editor toolbar (indicated by the arrow in [Figure 20-8](#)) to view the user interface in landscape orientation:

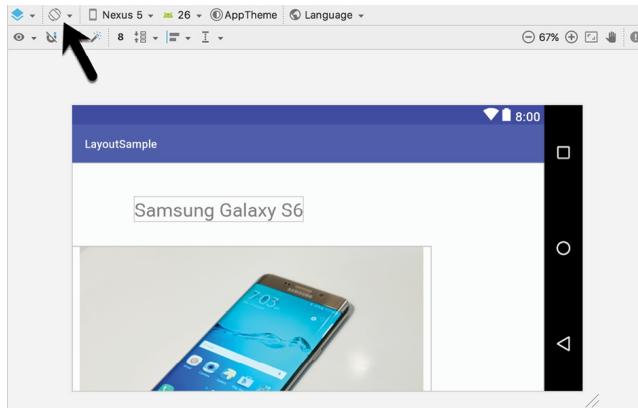


Figure 20-8

The absence of constraints results in a layout that fails to adapt to the change in device orientation, leaving the content off center and with part of the image and all three buttons positioned beyond the viewable area of the screen. Clearly some work still needs to be done to make this into a responsive user interface.

20.5 Adding the Constraints

Constraints are the key to creating layouts that can adapt to device orientation changes and different screen sizes. Begin by rotating the layout back to portrait orientation and selecting the TextView widget located above the ImageView. With the widget selected, establish constraints from the left, right and top sides of the TextView to the corresponding sides of the parent ConstraintLayout as shown in [Figure 20-9](#):

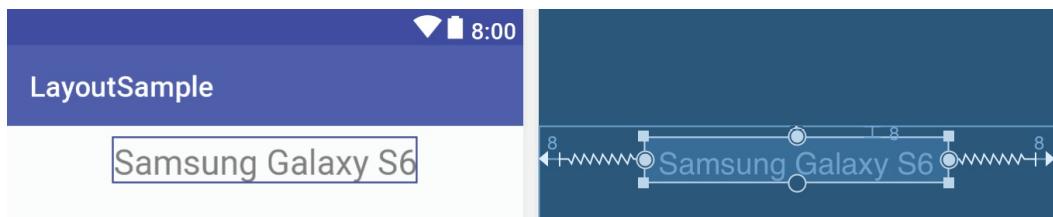


Figure 20-9

With the TextView widget constrained, select the ImageView instance and establish opposing constraints on the left and right-hand sides with each

connected to the corresponding sides of the parent layout. Next, establish a constraint connection from the top of the ImageView to the bottom of the TextView and from the bottom of the ImageView to the top of the center Button widget. If necessary, click and drag the ImageView so that it is still positioned in the vertical center of the layout.

With the ImageView still selected, use the Inspector in the attributes panel to change the top and bottom margins on the ImageView to 24 and 8 respectively and to change both the widget height and width dimension properties to *match_constraint* so that the widget will resize to match the constraints. These settings will allow the layout engine to enlarge and reduce the size of the ImageView when necessary to accommodate layout changes:

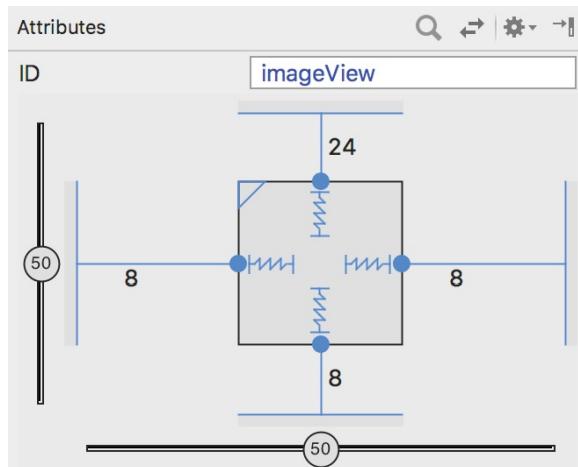


Figure 20-
10

[Figure 20-11](#), shows the currently implemented constraints for the ImageView in relation to the other elements in the layout:

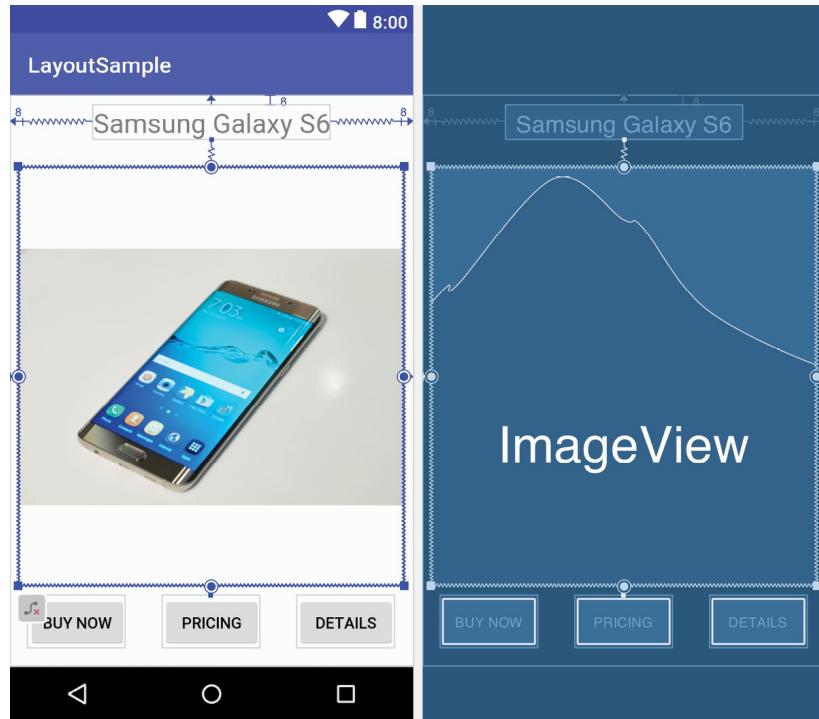


Figure 20-
11

The final task is to add constraints to the three Button widgets. For this example, the buttons will be placed in a chain. Begin by turning on Autoconnect within the Layout Editor by clicking the toolbar button highlighted in [Figure 20-2](#).

Next, click on the Buy Now button and then shift-click on the other two buttons so that all three are selected. Right-click on the Buy Now button and select the *Chain -> Create Horizontal Chain* menu option from the resulting menu. By default, the chain will be displayed using the spread style which is the correct behavior for this example.

Finally, establish a constraint between the bottom of the Buy Now button and the bottom of the layout. Repeat this step for the remaining buttons.

On completion of these steps the buttons should be constrained as outlined in [Figure 20-12](#):

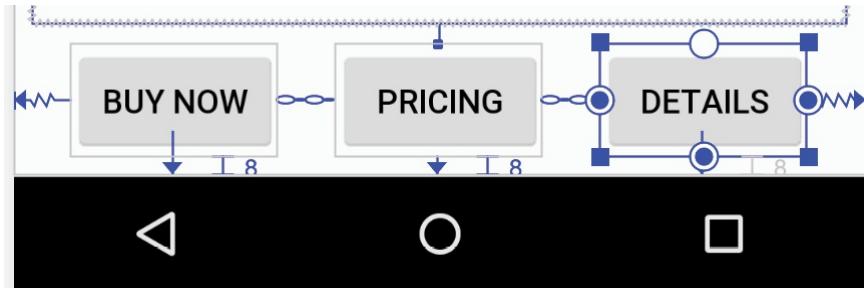


Figure 20-
12

20.6 Testing the Layout

With the constraints added to the layout, rotate the screen into landscape orientation and verify that the layout adapts to accommodate the new screen dimensions.

While the Layout Editor tool provides a useful visual environment in which to design user interface layouts, when it comes to testing there is no substitute for testing the running app. Launch the app on a physical Android device or emulator session and verify that the user interface reflects the layout created in the Layout Editor. [Figure 20-13](#), for example, shows the running app in landscape orientation:

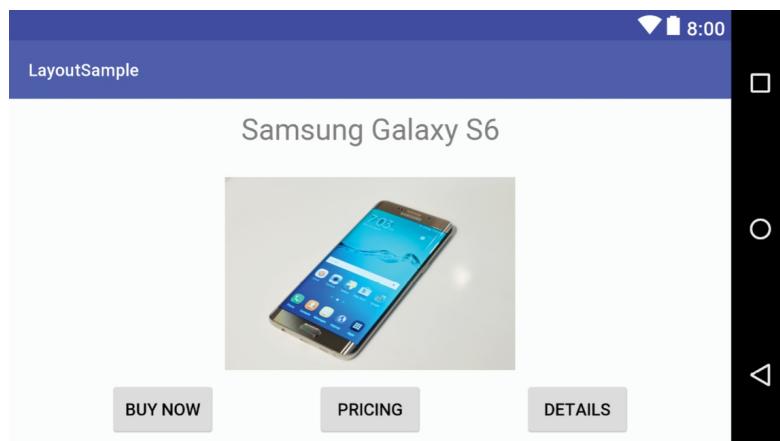


Figure 20-
13

The very simple user interface design is now complete. Designing a more complex user interface layout is a continuation of the steps outlined above. Simply drag and drop views onto the display, position, constrain and set properties as needed.

20.7 Using the Layout Inspector

The hierarchy of components that make up a user interface layout may be viewed at any time using the Layout Inspector tool. In order to access this information the app must be running on a device or emulator. Once the app is running, select the *Tools -> Android -> Layout Inspector* menu option followed by the process to be inspected.

Once the inspector loads, the left most panel (A) shows the hierarchy of components that make up the user interface layout. The center panel (B) shows a visual representation of the layout design. Clicking on a widget in the visual layout will cause that item to highlight in the hierarchy list making it easy to find where a visual component is situated relative to the overall layout hierarchy.

Finally, the rightmost panel (marked C in [Figure 20-14](#)) contains all of the property settings for the currently selected component, allowing for in-depth analysis of the component's internal configuration.

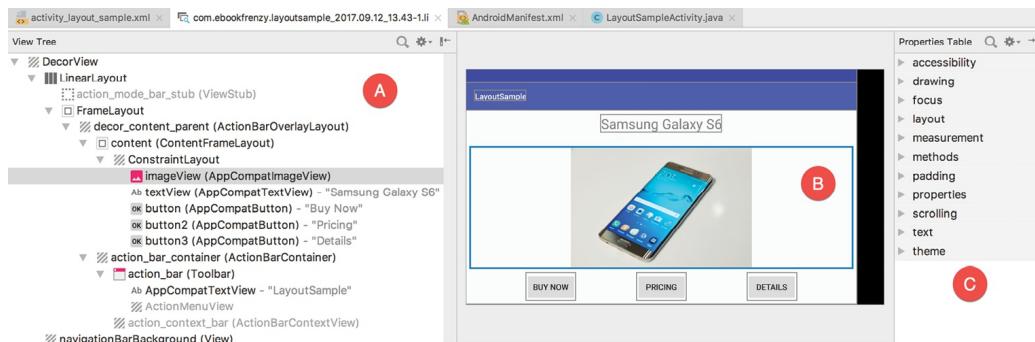


Figure 20-
14

20.8 Summary

The Layout Editor tool in Android Studio has been tightly integrated with the `ConstraintLayout` class. This chapter has worked through the creation of an example user interface intended to outline the ways in which a `ConstraintLayout`-based user interface can be implemented using the Layout Editor tool in terms of adding widgets and setting constraints. This chapter also introduced the Layout Inspector tool which is useful for analyzing the structural composition of a user interface layout.

21. Manual XML Layout Design in Android Studio

While the design of layouts using the Android Studio Layout Editor tool greatly improves productivity, it is still possible to create XML layouts by manually editing the underlying XML. This chapter will introduce the basics of the Android XML layout file format.

21.1 Manually Creating an XML Layout

The structure of an XML layout file is actually quite straightforward and follows the hierarchical approach of the view tree. The first line of an XML resource file should ideally include the following standard declaration:

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

This declaration should be followed by the root element of the layout, typically a container view such as a layout manager. This is represented by both opening and closing tags and any properties that need to be set on the view. The following XML, for example, declares a ConstraintLayout view as the root element, assigns the ID *activity_main* and sets *match_parent* attributes such that it fills all the available space of the device display:

```
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res-auto"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main_activity"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    tools:context="com.ebookfrenzy.demoapp.MainActivity">

</android.support.constraint.ConstraintLayout>
```

Note that in the above example the layout element is also configured with padding on each side of 16dp (density independent pixels). Any specification of spacing in an Android layout must be specified using one of the following units of measurement:

- **in** – Inches.
- **mm** – Millimeters.
- **pt** – Points (1/72 of an inch).
- **dp** – Density-independent pixels. An abstract unit of measurement based on the physical density of the device display relative to a 160dpi display baseline.
- **sp** – Scale-independent pixels. Similar to dp but scaled based on the user's font preference.
- **px** – Actual screen pixels. Use is not recommended since different displays will have different pixels per inch. Use *dp* in preference to this unit.

Any children that need to be added to the ConstraintLayout parent must be *nested* within the opening and closing tags. In the following example a Button widget has been added as a child of the ConstraintLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res-auto"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    tools:context="com.ebookfrenzy.demoapp.MainActivity">

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

</android.support.constraint.ConstraintLayout>
```

As currently implemented, the button has no constraint connections. At runtime, therefore, the button will appear in the top left-hand corner of the screen (though indented 16dp by the padding assigned to the parent layout).

If opposing constraints are added to the sides of the button, however, it will appear centered within the layout:

```
<Button  
    android:text="Button"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/button"  
    app:layout_constraintLeft_toLeftOf="@+id/activity_main"  
    app:layout_constraintTop_toTopOf="@+id/activity_main"  
    app:layout_constraintRight_toRightOf="@+id/activity_main"  
    app:layout_constraintBottom_toBottomOf="@+id/activity_main" />
```

Note that each of the constraints is attached to the element named *activity_main* which is, in this case, the parent ConstraintLayout instance.

To add a second widget to the layout, simply embed it within the body of ConstraintLayout element. The following modification, for example, adds a TextView widget to the layout:

```
<?xml version="1.0" encoding="utf-8"?>  
<android.support.constraint.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/activity_main"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:paddingLeft="16dp"  
    android:paddingRight="16dp"  
    android:paddingTop="16dp"  
    android:paddingBottom="16dp"  
    tools:context="com.ebookfrenzy.demoapp.MainActivity">  
  
<Button  
    android:text="Button"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/button"  
    app:layout_constraintLeft_toLeftOf="@+id/activity_main"  
    app:layout_constraintTop_toTopOf="@+id/activity_main"  
    app:layout_constraintRight_toRightOf="@+id/activity_main"  
    app:layout_constraintBottom_toBottomOf="@+id/activity_main" />  
  
<TextView
```

```
    android:text="TextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/textView" />

</android.support.constraint.ConstraintLayout>
```

Once again, the absence of constraints on the newly added TextView will cause it to appear in the top left-hand corner of the layout at runtime. The following modifications add opposing constraints connected to the parent layout to center the widget horizontally, together with a constraint connecting the bottom of the TextView to the top of the button with a margin of 72dp:

```
<TextView
    android:text="TextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/textView"
    app:layout_constraintLeft_toLeftOf="@+id/activity_main"
    app:layout_constraintRight_toRightOf="@+id/activity_main"
    app:layout_constraintBottom_toTopOf="@+id/button"
    android:layout_marginBottom="72dp" />
```

Also, note that the Button and TextView views have a number of attributes declared. Both views have been assigned IDs and configured to display text strings represented by string resources named *button_string* and *text_string* respectively. Additionally, the *wrap_content* height and width properties have been declared on both objects so that they are sized to accommodate the content (in this case the text referenced by the string resource value).

Viewed from within the Preview panel of the Layout Editor in Text mode, the above layout will be rendered as shown in [Figure 21-1](#):

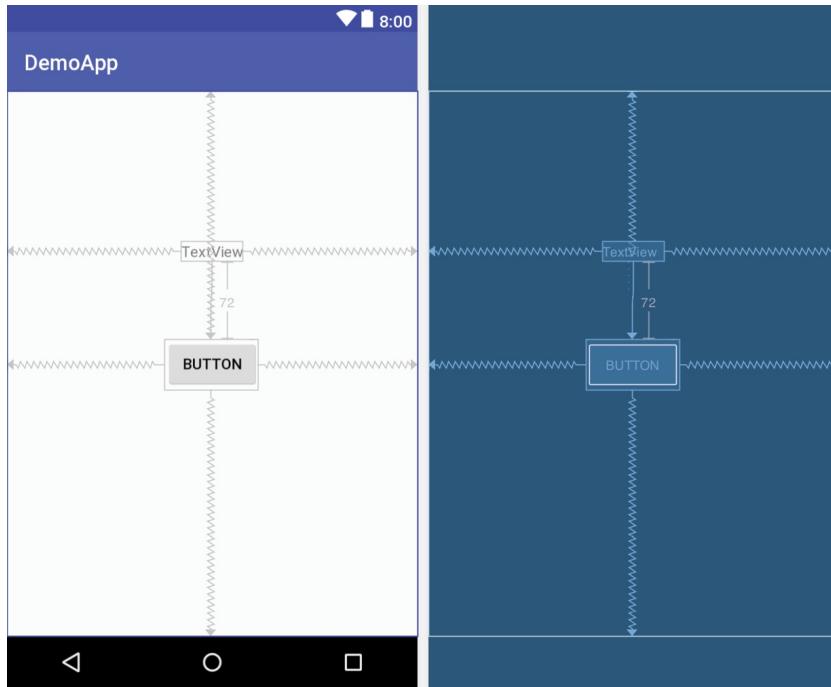


Figure 21-1

21.2 Manual XML vs. Visual Layout Design

When to write XML manually as opposed to using the Layout Editor tool in design mode is a matter of personal preference. There are, however, advantages to using design mode.

First, design mode will generally be quicker given that it avoids the necessity to type lines of XML. Additionally, design mode avoids the need to learn the intricacies of the various property values of the Android SDK view classes. Rather than continually refer to the Android documentation to find the correct keywords and values, most properties can be located by referring to the Attributes panel.

All the advantages of design mode aside, it is important to keep in mind that the two approaches to user interface design are in no way mutually exclusive. As an application developer, it is quite likely that you will end up creating user interfaces within design mode while performing fine-tuning and layout tweaks of the design by directly editing the generated XML resources. Both views of the interface design are, after all, displayed side by side within the Android Studio environment making it easy to work seamlessly on both the XML and the visual layout.

21.3 Summary

The Android Studio Layout Editor tool provides a visually intuitive method for designing user interfaces. Using a drag and drop paradigm combined with a set of property editors, the tool provides considerable productivity benefits to the application developer.

User interface designs may also be implemented by manually writing the XML layout resource files, the format of which is well structured and easily understood.

The fact that the Layout Editor tool generates XML resource files means that these two approaches to interface design can be combined to provide a “best of both worlds” approach to user interface development.

22. Managing Constraints using Constraint Sets

Up until this point in the book, all user interface design tasks have been performed using the Android Studio Layout Editor tool, either in text or design mode. An alternative to writing XML resource files or using the Android Studio Layout Editor is to write Java code to directly create, configure and manipulate the view objects that comprise the user interface of an Android activity. Within the context of this chapter, we will explore some of the advantages and disadvantages of writing Java code to create a user interface before describing some of the key concepts such as view properties and the creation and management of layout constraints.

In the next chapter, an example project will be created and used to demonstrate some of the typical steps involved in this approach to Android user interface creation.

22.1 Java Code vs. XML Layout Files

There are a number of key advantages to using XML resource files to design a user interface as opposed to writing Java code. In fact, Google goes to considerable lengths in the Android documentation to extol the virtues of XML resources over Java code. As discussed in the previous chapter, one key advantage to the XML approach includes the ability to use the Android Studio Layout Editor tool, which, itself, generates XML resources. A second advantage is that once an application has been created, changes to user interface screens can be made by simply modifying the XML file, thereby avoiding the necessity to recompile the application. Also, even when hand writing XML layouts, it is possible to get instant feedback on the appearance of the user interface using the preview feature of the Android Studio Layout Editor tool. In order to test the appearance of a Java created user interface the developer will, inevitably, repeatedly cycle through a loop of writing code, compiling and testing in order to complete the design work.

In terms of the strengths of the Java coding approach to layout creation, perhaps the most significant advantage that Java has over XML resource files comes into play when dealing with dynamic user interfaces. XML resource

files are inherently most useful when defining static layouts, in other words layouts that are unlikely to change significantly from one invocation of an activity to the next. Java code, on the other hand, is ideal for creating user interfaces dynamically at run-time. This is particularly useful in situations where the user interface may appear differently each time the activity executes subject to external factors.

A knowledge of working with user interface components in Java code can also be useful when dynamic changes to a static XML resource based layout need to be performed in real-time as the activity is running.

Finally, some developers simply prefer to write Java code than to use layout tools and XML, regardless of the advantages offered by the latter approaches.

22.2 Creating Views

As previously established, the Android SDK includes a toolbox of view classes designed to meet most of the basic user interface design needs. The creation of a view in Java is simply a matter of creating instances of these classes, passing through as an argument a reference to the activity with which that view is to be associated.

The first view (typically a container view to which additional child views can be added) is displayed to the user via a call to the *setContentView()* method of the activity. Additional views may be added to the root view via calls to the object's *addView()* method.

When working with Java code to manipulate views contained in XML layout resource files, it is necessary to obtain the ID of the view. The same rule holds true for views created in Java. As such, it is necessary to assign an ID to any view for which certain types of access will be required in subsequent Java code. This is achieved via a call to the *setId()* method of the view object in question. In later code, the ID for a view may be obtained via the object's *getId()* method.

22.3 View Attributes

Each view class has associated with it a range of *attributes*. These property settings are set directly on the view instances and generally define how the view object will appear or behave. Examples of attributes are the text that appears on a Button object, or the background color of a ConstraintLayout

view. Each view class within the Android SDK has a pre-defined set of methods that allow the user to *set* and *get* these property values. The Button class, for example, has a *setText()* method which can be called from within Java code to set the text displayed on the button to a specific string value. The background color of a *ConstraintLayout* object, on the other hand, can be set with a call to the object's *setBackgroundColor()* method.

22.4 Constraint Sets

While property settings are internal to view objects and dictate how a view appears and behaves, *constraint sets* are used to control how a view appears relative to its parent view and other sibling views. Every *ConstraintLayout* instance has associated with it a set of constraints that define how its child views are positioned and constrained.

The key to working with constraint sets in Java code is the *ConstraintSet* class. This class contains a range of methods that allow tasks such as creating, configuring and applying constraints to a *ConstraintLayout* instance. In addition, the current constraints for a *ConstraintLayout* instance may be copied into a *ConstraintSet* object and used to apply the same constraints to other layouts (with or without modifications).

A *ConstraintSet* instance is created just like any other Java object:

```
ConstraintSet set = new ConstraintSet();
```

Once a constraint set has been created, methods can be called on the instance to perform a wide range of tasks.

22.4.1 Establishing Connections

The *connect()* method of the *ConstraintSet* class is used to establish constraint connections between views. The following code configures a constraint set in which the left-hand side of a Button view is connected to the right-hand side of an EditText view with a margin of 70dp:

```
set.connect(button1.getId(), ConstraintSet.LEFT,
           editText1.getId(), ConstraintSet.RIGHT, 70);
```

22.4.2 Applying Constraints to a Layout

Once the constraint set is configured, it must be applied to a *ConstraintLayout* instance before it will take effect. A constraint set is applied via a call to the *applyTo()* method, passing through a reference to the layout object to which the settings are to be applied:

```
set.applyTo(myLayout);
```

22.4.3 Parent Constraint Connections

Connections may also be established between a child view and its parent ConstraintLayout by referencing the ConstraintSet.PARENT_ID constant. In the following example, the constraint set is configured to connect the top edge of a Button view to the top of the parent layout with a margin of 100dp:

```
set.connect(button1.getId(), ConstraintSet.TOP,  
          ConstraintSet.PARENT_ID, ConstraintSet.TOP, 100);
```

22.4.4 Sizing Constraints

A number of methods are available for controlling the sizing behavior of views. The following code, for example, sets the horizontal size of a Button view to *wrap_content* and the vertical size of an ImageView instance to a maximum of 250dp:

```
set.constrainWidth(button1.getId(), ConstraintSet.WRAP_CONTENT);  
set.constrainMaxHeight(imageView1.getId(), 250);
```

22.4.5 Constraint Bias

As outlined in the chapter entitled "["A Guide to using ConstraintLayout in Android Studio"](#)", when a view has opposing constraints it is centered along the axis of the constraints (i.e. horizontally or vertically). This centering can be adjusted by applying a bias along the particular axis of constraint. When using the Android Studio Layout Editor, this is achieved using the controls in the Attributes tool window. When working with a constraint set, however, bias can be added using the *setHorizontalBias()* and *setVerticalBias()* methods, referencing the view ID and the bias as a floating point value between 0 and 1.

The following code, for example, constrains the left and right-hand sides of a Button to the corresponding sides of the parent layout before applying a 25% horizontal bias:

```
set.connect(button1.getId(), ConstraintSet.LEFT,  
          ConstraintSet.PARENT_ID, ConstraintSet.LEFT, 0);  
set.connect(button1.getId(), ConstraintSet.RIGHT,  
          ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);  
set.setHorizontalBias(button1.getId(), 0.25f);
```

22.4.6 Alignment Constraints

Alignments may also be applied using a constraint set. The full set of

alignment options available with the Android Studio Layout Editor may also be configured using a constraint set via the `centerVertically()` and `centerHorizontally()` methods, both of which take a variety of arguments depending on the alignment being configured. In addition, the `center()` method may be used to center a view between two other views.

In the code below, `button2` is positioned so that it is aligned horizontally with `button1`:

```
set.centerHorizontally(button2.getId(), button1.getId());
```

22.4.7 Copying and Applying Constraint Sets

The current constraint set for a `ConstraintLayout` instance may be copied into a constraint set object using the `clone()` method. The following line of code, for example, copies the constraint settings from a `ConstraintLayout` instance named `myLayout` into a constraint set object:

```
set.clone(myLayout);
```

Once copied, the constraint set may be applied directly to another layout or, as in the following example, modified before being applied to the second layout:

```
ConstraintSet set = new ConstraintSet();
set.clone(myLayout);
set.constrainWidth(button1.getId(), ConstraintSet.WRAP_CONTENT);
set.applyTo(mySecondLayout);
```

22.4.8 ConstraintLayout Chains

Vertical and horizontal chains may also be created within a constraint set using the `createHorizontalChain()` and `createVerticalChain()` methods. The syntax for using these methods is as follows:

```
createHorizontalChain(int leftId, int leftSide, int rightId,
                     int rightSide, int[] chainIds, float[] weights, int style);
```

Based on the above syntax, the following example creates a horizontal spread chain that starts with `button1` and ends with `button4`. In between these views are `button2` and `button3` with weighting set to zero for both:

```
int[] chainViews = {button2.getId(), button3.getId()};
float[] chainWeights = {0, 0};

set.createHorizontalChain(button1.getId(), ConstraintSet.LEFT,
                        button4.getId(), ConstraintSet.RIGHT,
                        chainViews, chainWeights,
```

```
        ConstraintSet.CHAIN_SPREAD);
```

A view can be removed from a chain by passing the ID of the view to be removed through to either the *removeFromHorizontalChain()* or *removeFromVerticalChain()* methods. A view may be added to an existing chain using either the *addToHorizontalChain()* or *addToVerticalChain()* methods. In both cases the methods take as arguments the IDs of the views between which the new view is to be inserted as follows:

```
set.addToHorizontalChain(newViewId, leftViewId, rightViewId);
```

22.4.9 Guidelines

Guidelines are added to a constraint set using the *create()* method and then positioned using the *setGuidelineBegin()*, *setGuidelineEnd()* or *setGuidelinePercent()* methods. In the following code, a vertical guideline is created and positioned 50% across the width of the parent layout. The left side of a button view is then connected to the guideline with no margin:

```
set.create(R.id.myGuideline, ConstraintSet.VERTICAL_GUIDELINE);
set.setGuidelinePercent(R.id.myGuideline, 0.5f);

set.connect(button.getId(), ConstraintSet.LEFT,
           R.id.myGuideline, ConstraintSet.RIGHT, 0);

set.applyTo(layout);
```

22.4.10 Removing Constraints

A constraint may be removed from a view in a constraint set using the *clear()* method, passing through as arguments the view ID and the anchor point for which the constraint is to be removed:

```
set.clear(button.getId(), ConstraintSet.LEFT);
```

Similarly, all of the constraints on a view may be removed in a single step by referencing only the view in the *clear()* method call:

```
set.clear(button.getId());
```

22.4.11 Scaling

The scale of a view within a layout may be adjusted using the *ConstraintSet* *setScaleX()* and *setScaleY()* methods which take as arguments the view on which the operation is to be performed together with a float value indicating the scale. In the following code, a button object is scaled to twice its original width and half the height:

```
set.setScaleX(myButton.getId(), 2f);
```

```
set.setScaleY(myButton.getId(), 0.5f);
```

22.4.1 Rotation

A view may be rotated on either the X or Y axis using the *setRotationX()* and *setRotationY()* methods respectively both of which must be passed the ID of the view to be rotated and a float value representing the degree of rotation to be performed. The pivot point on which the rotation is to take place may be defined via a call to the *setTransformPivot()*, *setTransformPivotX()* and *setTransformPivotY()* methods. The following code rotates a button view 30 degrees on the Y axis using a pivot point located at point 500, 500:

```
set.setTransformPivot(button.getId(), 500, 500);
set.setRotationY(button.getId(), 30);
set.applyTo(layout);
```

Having covered the theory of constraint sets and user interface creation from within Java code, the next chapter will work through the creation of an example application with the objective of putting this theory into practice. For more details on the ConstraintSet class, refer to the reference guide at the following URL:

<https://developer.android.com/reference/android/support/constraint/ConstraintSet>

22.5 Summary

As an alternative to writing XML layout resource files or using the Android Studio Layout Editor tool, Android user interfaces may also be dynamically created in Java code.

Creating layouts in Java code consists of creating instances of view classes and setting attributes on those objects to define required appearance and behavior.

How a view is positioned and sized relative to its ConstraintLayout parent view and any sibling views is defined through the use of constraint sets. A constraint set is represented by an instance of the ConstraintSet class which, once created, can be configured using a wide range of method calls to perform tasks such as establishing constraint connections, controlling view sizing behavior and creating chains.

With the basics of the ConstraintSet class covered in this chapter, the next chapter will work through a tutorial that puts these features to practical use.

23. An Android ConstraintSet Tutorial

The previous chapter introduced the basic concepts of creating and modifying user interface layouts in Java code using the `ConstraintLayout` and `ConstraintSet` classes. This chapter will take these concepts and put them into practice through the creation of an example layout created entirely in Java code and without using the Android Studio Layout Editor tool.

23.1 Creating the Example Project in Android Studio

Launch Android Studio and select the *Start a new Android Studio project* option from the quick start menu in the welcome screen.

In the new project configuration dialog, enter *JavaLayout* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *JavaLayoutActivity* with a corresponding layout named *activity_java_layout*.

Once the project has been created, the *JavaLayoutActivity.java* file should automatically load into the editing panel. As we have come to expect, Android Studio has created a template activity and overridden the *onCreate()* method, providing an ideal location for Java code to be added to create a user interface.

23.2 Adding Views to an Activity

The *onCreate()* method is currently designed to use a resource layout file for the user interface. Begin, therefore, by deleting this line from the method:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_java_layout);  
}
```

The next modification is to add a `ConstraintLayout` object with a single `Button` view child to the activity. This involves the creation of new instances of the `ConstraintLayout` and `Button` classes. The `Button` view then needs to be

added as a child to the ConstraintLayout view which, in turn, is displayed via a call to the `setContentView()` method of the activity instance:

```
package com.ebookfrenzy.javalayout;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.constraint.ConstraintSet;
import android.support.constraint.ConstraintLayout;
import android.widget.Button;
import android.widget.EditText;

public class JavaLayoutActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        configureLayout();
    }

    private void configureLayout() {
        Button myButton = new Button(this);
        ConstraintLayout myLayout = new ConstraintLayout(this);
        myLayout.addView(myButton);
        setContentView(myLayout);
    }
}
```

When new instances of user interface objects are created in this way, the constructor methods must be passed the context within which the object is being created which, in this case, is the current activity. Since the above code resides within the activity class, the context is simply referenced by the standard `this` keyword:

```
Button myButton = new Button(this);
```

Once the above additions have been made, compile and run the application (either on a physical device or an emulator). Once launched, the visible result will be a button containing no text appearing in the top left-hand corner of the ConstraintLayout view as shown in [Figure 23-1](#):



Figure 23-1

23.3 Setting View Attributes

For the purposes of this exercise, we need the background of the ConstraintLayout view to be blue and the Button view to display text that reads “Press Me” on a yellow background. Both of these tasks can be achieved by setting attributes on the views in the Java code as outlined in the following code fragment. In order to allow the text on the button to be easily translated to other languages it will be added as a String resource. Within the Project tool window, locate the *app -> res -> values -> strings.xml* file and modify it to add a resource value for the “Press Me” string:

```
<resources>
    <string name="app_name">JavaLayout</string>
    <string name="press_me">Press Me</string>
</resources>
```

Although this is the recommended way to handle strings that are directly referenced in code, to avoid repetition of this step throughout the remainder of the book, many subsequent code samples will directly enter strings into the code.

Once the string is stored as a resource it can be accessed from within code as follows:

```
getString(R.string.press_me)
```

With the string resource created, add code to the *configureLayout()* method to set the button text and color attributes:

```
.
.
import android.graphics.Color;

public class JavaLayoutActivity extends AppCompatActivity {
```

```

private void configureLayout() {
    Button myButton = new Button(this);
    myButton.setText(getString(R.string.press_me));
    myButton.setBackgroundColor(Color.YELLOW);

    ConstraintLayout myLayout = new ConstraintLayout(this);
    myLayout.setBackgroundColor(Color.BLUE);

    myLayout.addView(myButton);
    setContentView(myLayout);
}

```

When the application is now compiled and run, the layout will reflect the property settings such that the layout will appear with a blue background and the button will display the assigned text on a yellow background.

23.4 Creating View IDs

When the layout is complete it will consist of a Button and an EditText view. Before these views can be referenced within the methods of the ConstraintSet class, they must be assigned unique view IDs. The first step in this process is to create a new resource file containing these ID values.

Right click on the *app -> res -> values* folder, select the *New -> Values resource file* menu option and name the new resource file *id.xml*. With the resource file created, edit it so that it reads as follows:

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <item name="myButton" type="id" />
    <item name="myEditText" type="id" />
</resources>

```

At this point in the tutorial, only the Button has been created, so edit the *createLayout()* method to assign the corresponding ID to the object:

```

private void configureLayout() {
    Button myButton = new Button(this);
    myButton.setText(getString(R.string.press_me));
    myButton.setBackgroundColor(Color.YELLOW);
    myButton.setId(R.id.myButton);

    .
    .
}

```

23.5 Configuring the Constraint Set

In the absence of any constraints, the ConstraintLayout view has placed the Button view in the top left corner of the display. In order to instruct the layout view to place the button in a different location, in this case centered both horizontally and vertically, it will be necessary to create a ConstraintSet instance, initialize it with the appropriate settings and apply it to the parent layout.

For this example, the button needs to be configured so that the width and height are constrained to the size of the text it is displaying and the view centered within the parent layout. Edit the *onCreate()* method once more to make these changes:

```
private void configureLayout() {  
    Button myButton = new Button(this);  
    myButton.setText(getString(R.string.press_me));  
    myButton.setBackgroundColor(Color.YELLOW);  
    myButton.setId(R.id.myButton);  
  
    ConstraintLayout myLayout = new ConstraintLayout(this);  
    myLayout.setBackgroundColor(Color.BLUE);  
  
    myLayout.addView(myButton);  
    setContentView(myLayout);  
  
    ConstraintSet set = new ConstraintSet();  
  
    set.constrainHeight(myButton.getId(),  
        ConstraintSet.WRAP_CONTENT);  
    set.constrainWidth(myButton.getId(),  
        ConstraintSet.WRAP_CONTENT);  
  
    set.connect(myButton.getId(), ConstraintSet.LEFT,  
        ConstraintSet.PARENT_ID, ConstraintSet.LEFT, 0);  
    set.connect(myButton.getId(), ConstraintSet.RIGHT,  
        ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);  
    set.connect(myButton.getId(), ConstraintSet.TOP,  
        ConstraintSet.PARENT_ID, ConstraintSet.TOP, 0);  
    set.connect(myButton.getId(), ConstraintSet.BOTTOM,  
        ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0);  
  
    set.applyTo(myLayout);  
}
```

With the initial constraints configured, compile and run the application and verify that the Button view now appears in the center of the layout:

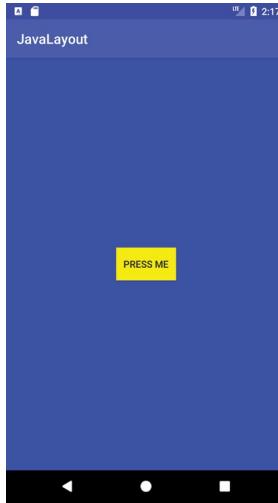


Figure 23-2

23.6 Adding the EditText View

The next item to be added to the layout is the EditText view. The first step is to create the EditText object, assign it the ID as declared in the *id.xml* resource file and add it to the layout. The code changes to achieve these steps now need to be made to the *onCreate()* method as follows:

```
private void configureLayout() {  
    Button myButton = new Button(this);  
    myButton.setText(getString(R.string.press_me));  
    myButton.setBackgroundColor(Color.YELLOW);  
    myButton.setId(R.id.myButton);  
  
    EditTextView myEditText = new EditTextView(this);  
    myEditText.setId(R.id.myEditText);  
  
    ConstraintLayout myLayout = new ConstraintLayout(this);  
    myLayout.setBackgroundColor(Color.BLUE);  
  
    myLayout.addView(myButton);  
    myLayout.addView(myEditText);  
  
    setContentView(myLayout);  
    .  
    .  
}
```

The EditText widget is intended to be sized subject to the content it is displaying, centered horizontally within the layout and positioned 70dp above the existing Button view. Add code to the *onCreate()* method so that it reads as follows:

```
.  
. .  
  
set.connect(myButton.getId(), ConstraintSet.LEFT,  
           ConstraintSet.PARENT_ID, ConstraintSet.LEFT, 0);  
set.connect(myButton.getId(), ConstraintSet.RIGHT,  
           ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);  
set.connect(myButton.getId(), ConstraintSet.TOP,  
           ConstraintSet.PARENT_ID, ConstraintSet.TOP, 0);  
set.connect(myButton.getId(), ConstraintSet.BOTTOM,  
           ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0);  
  
set.constrainHeight(myEditText.getId(),  
                    ConstraintSet.WRAP_CONTENT);  
set.constrainWidth(myEditText.getId(),  
                   ConstraintSet.WRAP_CONTENT);  
  
set.connect(myEditText.getId(), ConstraintSet.LEFT,  
           ConstraintSet.PARENT_ID, ConstraintSet.LEFT, 0);  
set.connect(myEditText.getId(), ConstraintSet.RIGHT,  
           ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);  
set.connect(myEditText.getId(), ConstraintSet.BOTTOM,  
           myButton.getId(), ConstraintSet.TOP, 70);  
  
set.applyTo(myLayout);
```

A test run of the application should show the EditText field centered above the button with a margin of 70dp.

23.7 Converting Density Independent Pixels (dp) to Pixels (px)

The next task in this exercise is to set the width of the EditText view to 200dp. As outlined in the chapter entitled "["An Android Studio Layout Editor ConstraintLayout Tutorial"](#)" when setting sizes and positions in user interface layouts it is better to use density independent pixels (dp) rather than pixels (px). In order to set a position using dp it is necessary to convert a dp value to

a px value at runtime, taking into consideration the density of the device display. In order, therefore, to set the width of the EditText view to 200dp, the following code needs to be added to the class:

```
package com.ebookfrenzy.javayout;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.constraint.ConstraintLayout;
import android.support.constraint.ConstraintSet;
import android.widget.Button;
import android.widget.EditText;
import android.graphics.Color;
import android.content.res.Resources;
import android.util.TypedValue;

public class JavaLayoutActivity extends AppCompatActivity {

    private int convertToPx(int value) {
        Resources r = getResources();
        int px = (int) TypedValue.applyDimension(
            TypedValue.COMPLEX_UNIT_DIP, value,
            r.getDisplayMetrics());
        return px;
    }

    private void configureLayout() {
        Button myButton = new Button(this);
        myButton.setText(getString(R.string.press_me));
        myButton.setBackgroundColor(Color.YELLOW);
        myButton.setId(R.id.myButton);

        EditText myEditText = new EditText(this);
        myEditText.setId(R.id.myEditText);

        int px = convertToPx(200);
        myEditText.setWidth(px);

    }
}
```

Compile and run the application one more time and note that the width of the EditText view has changed as illustrated in [Figure 23-3](#):

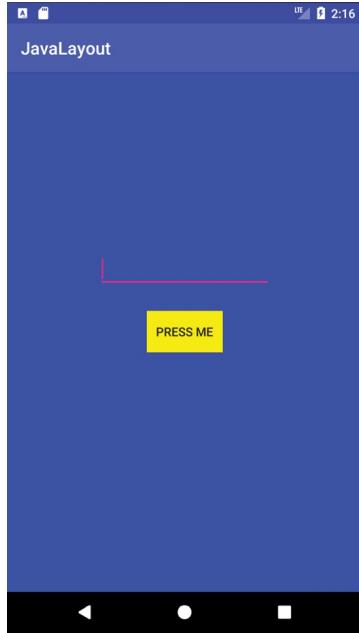


Figure 23-3

23.8 Summary

The example activity created in this chapter has, of course, created a similar user interface (the change in background color and view type notwithstanding) as that created in the earlier "["Manual XML Layout Design in Android Studio"](#)" chapter. If nothing else, this chapter should have provided an appreciation of the level to which the Android Studio Layout Editor tool and XML resources shield the developer from many of the complexities of creating Android user interface layouts.

There are, however, instances where it makes sense to create a user interface in Java. This approach is most useful, for example, when creating dynamic user interface layouts.

24. A Guide to using Instant Run in Android Studio

Now that some of the basic concepts of Android development using Android Studio have been covered, now is a good time to introduce the Android Studio Instant Run feature. As all experienced developers know, every second spent waiting for an app to compile and run is time better spent writing and refining code.

24.1 Introducing Instant Run

Prior to the introduction of Instant Run, each time a change to a project needed to be tested Android Studio would recompile the code, convert it to Dex format, generate the APK package file and install it on the device or emulator. Having performed these steps the app would finally be launched ready for testing. Even on a fast development system this is a process that takes a considerable amount of time to complete. It is not uncommon for it to take a minute or more for this process to complete for a large application.

Instant Run, in contrast, allows many code and resource changes within a project to be reflected nearly instantaneously within the app while it is already running on a device or emulator session.

Consider, for the purposes of an example, an app being developed in Android Studio which has already been launched on a device or emulator. If changes are made to resource settings or the code within a method, Instant Run will push the updated code and resources to the running app and dynamically “swap” the changes. The changes are then reflected in the running app without the need to build, deploy and relaunch the entire app. In many cases, this allows changes to be tested in a fraction of the time it would take without Instant Run.

24.2 Understanding Instant Run Swapping Levels

Not all project changes are fully supported by Instant Run and different changes result in a different level of “swap” being performed. There are three levels of Instant Run support, referred to as hot, warm and cold swapping:

- **Hot Swapping** – Hot swapping occurs when the code within an

existing method implementation is changed. The new method implementation is used next time it is called by the app. A hot swap occurs instantaneously and, if configured, is accompanied by a toast message on the device screen that reads “Applied code changes without activity restart”.

- **Warm Swapping** – When a change is made to a resource file of the project (for example a layout change or the modification of a string or color resource setting) an Instant Run warm swap is performed. A warm swap involves the restarting of the currently running activity. Typically the screen will flicker as the activity restarts. A warm swap is reported on the device screen by a toast message that reads “Applied changes, restarted activity”.
- **Cold Swapping** – Structural code changes such as the addition of a new method, a change to the signature of an existing method or a change to the class hierarchy of the project triggers a cold swap in which the entire app is restarted. In some conditions, such as the addition of new image resources to the project, the application package file (APK) will also be reinstalled during the swap.

24.3 Enabling and Disabling Instant Run

Instant Run is enabled and disabled via the Android Studio Settings screen. To view the current settings begin by selecting the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS). Within the Settings dialog select the *Build, Execution, Deployment* entry in the left-hand panel followed by *Instant Run* as shown in [Figure 24-1](#):

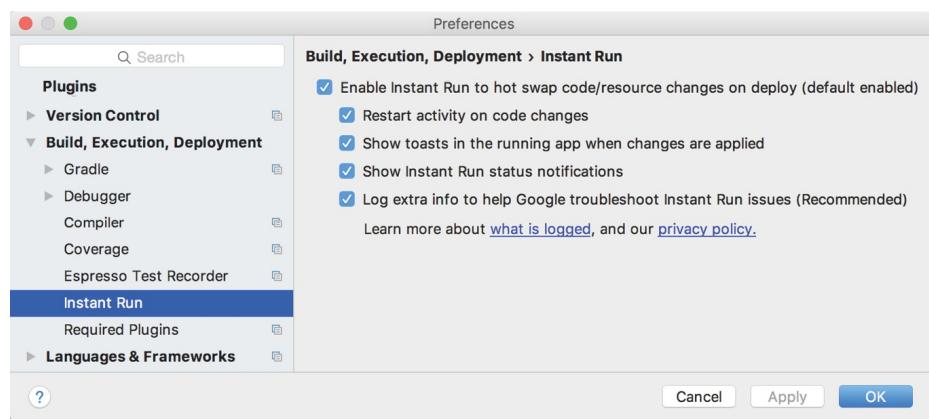


Figure 24-1

The options provided in the panel apply only to the current project. Each new project will start with the default settings. The first option controls whether or not Instant Run is enabled by default each time the project is opened in Android Studio. The *Restart activity on code changes* option forces Instant Run to restart the current activity every time a change is made, regardless of whether a hot swap could have been performed. The next option controls whether or not messages are displayed within Android Studio and the app indicating the type of Instant Run level performed. Finally, an option is provided to allow additional log information to be provided to Google to help in improving the reliability of the Instant Run feature.

24.4 Using Instant Run

When a project has been loaded into Android Studio, but is not yet running on a device or emulator, it can be launched as usual using either the run (marked A in [Figure 24-2](#)) or debug (B) button located in the toolbar:

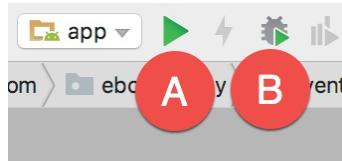


Figure 24-2

After the app has launched and is running, Android Studio will indicate the availability of Instant Run by enabling the *Apply Changes* button located immediately to the right of the run button as highlighted in [Figure 24-3](#):

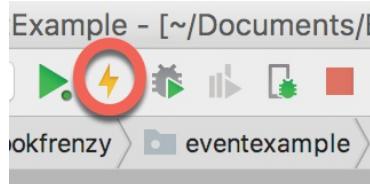


Figure 24-3

When it is enabled, clicking on the Apply Changes button will use Instant Run to update the running app.

24.5 An Instant Run Tutorial

Begin by launching Android Studio and creating a new project. Within the *New Project* dialog, enter *InstantRunDemo* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 23: Android 6.0 (Marshmallow). Continue to proceed through the screens, requesting the creation of a Basic Activity named *InstantRunDemoActivity* with a corresponding layout named *activity_instant_run_demo*.

Click on the *Finish* button to initiate the project creation process.

24.6 Triggering an Instant Run Hot Swap

Begin by clicking on the run button and selecting a suitable emulator or physical device as the run target. After clicking the run button, track the amount of time before the example app appears on the device or emulator.

Once running, click on the action button (the button displaying an envelope icon located in the lower right-hand corner of the screen). Note that a Snackbar instance appears displaying text which reads “Replace with your own action” as shown in [Figure 24-4](#):

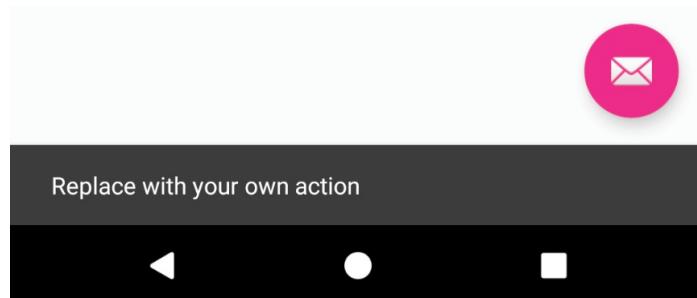


Figure 24-4

Once the app is running, the Apply Changes button should have been enabled indicating the availability of Instant Run. To see this in action, edit the *InstantRunDemoActivity.java* file, locate the *onCreate* method and modify the action code so that a different message is displayed when the action button is selected:

```
FloatingActionButton fab = (FloatingActionButton)
findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Snackbar.make(view, "Instant Run is Amazing!",
                     Snackbar.LENGTH_LONG)
            .setAction("Action", null).show();
    }
})
```

```
} );
```

With the code change implemented, click on the Apply Changes button and note that the toast message appears within a few seconds indicating the app has been updated. Tap the action button and note that the new message is now displayed in the Snackbar. Instant Run has successfully performed a hot swap.

24.7 Triggering an Instant Run Warm Swap

Any resource change should result in Instant Run performing a warm swap. Within Android Studio select the *app -> res -> layout -> content_instant_run_demo.xml* layout file. With the Layout Editor tool in Design mode, select the ConstraintLayout view within the Component Tree panel, switch the Attributes tool window to expert mode and locate the *background* property. Click on the button displaying three dots next to the background property text field, select a color from the Resources dialog and click on OK. With the background color of the activity content modified, click on the Apply Changes button once again. This time a warm swap will be performed and the currently running activity should quickly restart to adopt the new background color setting.

24.8 Triggering an Instant Run Cold Swap

As previously described, a cold swap triggers a complete restart of the running app. To experience an Instant Run cold swap, edit the *InstantRunDemoActivity.java* file and add a new method after the *onCreate* method as follows:

```
public void demoMethod() {  
}
```

Click on the Apply Changes button and note that the app now has to terminate and restart to accommodate the addition of the new method. Within Android Studio a message will appear indicating that the app was restarted due to a method being added:



Figure 24-5

24.9 The Run Button

When no apps are running, the run button appears as shown in [Figure 24-2](#). When an app is running, however, an additional green dot appears in the bottom right-hand corner of the button as shown in [Figure 24-6](#) below:



Figure 24-6

Although the Instant Run feature has improved significantly since being introduced it can still occasionally produce unexpected results when performing hot or warm swaps. It is worth being aware, therefore, that clicking the run button when an app is currently running will force a cold swap to be performed regardless of the changes made to the project.

24.10 Summary

Instant Run is a feature of Android Studio designed to significantly accelerate the code, build and run cycle. Using a swapping mechanism, Instant Run is able to push updates to the running application, in many cases without the need to re-install or even restart the app. Instant Run provides a number of different levels of support depending on the nature of the modification being applied to the project. These levels are referred to as hot, warm and cold swapping. This chapter has introduced the concepts of Instant Run and worked through some demonstrations of the different levels of swapping.

25. 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.

25.1 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.

In order 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. In order 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. In the event that 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.

25.2 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  
    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. 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.

25.3 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.

25.4 An Event Handling Example

In the remainder of this chapter, we will work through the creation of a simple 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.

Create a new project in Android Studio, entering *EventExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *EventExampleActivity* with a corresponding layout file named *activity_event_example*.

25.5 Designing the User Interface

The user interface layout for the *EventExampleActivity* class in this example is to consist of a ConstraintLayout, a Button and a TextView as illustrated in [Figure 25-1](#).



Figure 25-1

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

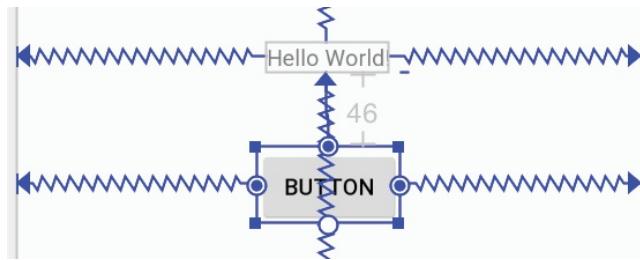


Figure 25-2

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 25-3](#)), display the warnings list and click on the Fix button to extract the text string on the button to a resource named *press_me*:

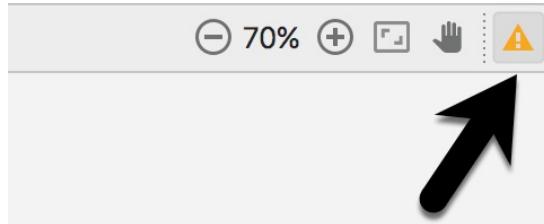


Figure 25-3

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*.

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

25.6 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 EventExampleActivity class.

If the *EventExampleActivity.java* 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 -> EventExampleActivity*) 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 android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class EventExample extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_event_example);
        Button button = (Button) findViewById(R.id.myButton);

        button.setOnClickListener(
            new Button.OnClickListener() {
                public void onClick(View v) {

                }
            }
        );
    }
}
```

The above code has now registered the event listener on the button and implemented the *onClick()* method. 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 *onClick()* callback method. 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
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_event_example);

    Button button = (Button) findViewById(R.id.myButton);

    button.setOnClickListener(
        new Button.OnClickListener() {
            public void onClick(View v) {
                TextView statusText =
                    (TextView) findViewById(R.id.statusText);
                statusText.setText("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.

25.7 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()* method in the above section of this chapter. The callback is declared as *void* and, as such, does not return a value to the Android framework after it has finished executing.

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 and callback method for long clicks to the button view in the example activity:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_event_example);

    Button button = (Button) findViewById(R.id.myButton);

    button.setOnClickListener(
        new Button.OnClickListener() {
            public void onClick(View v) {
                TextView statusText =
                    (TextView) findViewById(R.id.statusText);
                statusText.setText("Button clicked");
            }
        }
    );

    button.setOnLongClickListener(
        new Button.OnLongClickListener() {
            public boolean onLongClick(View v) {
                TextView statusText =
                    (TextView) findViewById(R.id.statusText);
                statusText.setText("Long button click");
                return true;
            }
        }
    );
}
```

Clearly, when a long click is detected, the *onLongClick()* callback method 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 such that the `onLongClick` listener now returns a *false* value:

```
button.setOnLongClickListener(
    new Button.OnLongClickListener() {
        public boolean onLongClick(View v) {
            TextView myTextView =
(TextView)findViewById(R.id.myTextView);
            myTextView.setText("Long button
click");
            return 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 click”. 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 `onClickListener` on the button was also interested in events of this type and subsequently called the `onClick` listener code.

25.8 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 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.

26. Android Touch and Multi-touch Event Handling

Most Android based devices use a touch screen as the primary interface between user and device. The previous chapter introduced the mechanism by which a touch on the screen translates into an action within a running Android application. There is, however, much more to touch event handling than responding to a single finger tap on a view object. Most Android devices can, for example, detect more than one touch at a time. Nor are touches limited to a single point on the device display. Touches can, of course, be dynamic as the user slides one or more points of contact across the surface of the screen.

Touches can also be interpreted by an application as a *gesture*. Consider, for example, that a horizontal swipe is typically used to turn the page of an eBook, or how a pinching motion can be used to zoom in and out of an image displayed on the screen.

The objective of this chapter is to highlight the handling of touches that involve motion and to explore the concept of intercepting multiple concurrent touches. The topic of identifying distinct gestures will be covered in the next chapter.

26.1 Intercepting Touch Events

Touch events can be intercepted by a view object through the registration of an *onTouchListener* event listener and the implementation of the corresponding *onTouch()* callback method. The following code, for example, ensures that any touches on a ConstraintLayout view instance named *myLayout* result in a call to the *onTouch()* method:

```
myLayout.setOnTouchListener(  
    new ConstraintLayout.OnTouchListener() {  
        public boolean onTouch(View v, MotionEvent m) {  
            // Perform tasks here  
  
            return true;  
        }  
    }  
);
```

As indicated in the code example, the *onTouch()* callback is required to return a Boolean value indicating to the Android runtime system whether or not the event should be passed on to other event listeners registered on the same view or discarded. The method is passed both a reference to the view on which the event was triggered and an object of type *MotionEvent*.

26.2 The MotionEvent Object

The *MotionEvent* object passed through to the *onTouch()* callback method is the key to obtaining information about the event. Information contained within the object includes the location of the touch within the view and the type of action performed. The *MotionEvent* object is also the key to handling multiple touches.

26.3 Understanding Touch Actions

An important aspect of touch event handling involves being able to identify the type of action performed by the user. The type of action associated with an event can be obtained by making a call to the *getActionMasked()* method of the *MotionEvent* object which was passed through to the *onTouch()* callback method. When the first touch on a view occurs, the *MotionEvent* object will contain an action type of *ACTION_DOWN* together with the coordinates of the touch. When that touch is lifted from the screen, an *ACTION_UP* event is generated. Any motion of the touch between the *ACTION_DOWN* and *ACTION_UP* events will be represented by *ACTION_MOVE* events.

When more than one touch is performed simultaneously on a view, the touches are referred to as *pointers*. In a multi-touch scenario, pointers begin and end with event actions of type *ACTION_POINTER_DOWN* and *ACTION_POINTER_UP* respectively. In order to identify the index of the pointer that triggered the event, the *getActionIndex()* callback method of the *MotionEvent* object must be called.

26.4 Handling Multiple Touches

The chapter entitled [*“An Overview and Example of Android Event Handling”*](#) began exploring event handling within the narrow context of a single touch event. In practice, most Android devices possess the ability to respond to multiple consecutive touches (though it is important to note that the number

of simultaneous touches that can be detected varies depending on the device). As previously discussed, each touch in a multi-touch situation is considered by the Android framework to be a *pointer*. Each pointer, in turn, is referenced by an *index* value and assigned an *ID*. The current number of pointers can be obtained via a call to the *getPointerCount()* method of the current *MotionEvent* object. The ID for a pointer at a particular index in the list of current pointers may be obtained via a call to the *MotionEvent getPointerId()* method. For example, the following code excerpt obtains a count of pointers and the ID of the pointer at index 0:

```
public boolean onTouch(View v, MotionEvent m) {  
    int pointerCount = m.getPointerCount();  
    int pointerId = m.getPointerId(0);  
    return true;  
}
```

Note that the pointer count will always be greater than or equal to 1 when the *onTouch* listener is triggered (since at least one touch must have occurred for the callback to be triggered).

A touch on a view, particularly one involving motion across the screen, will generate a stream of events before the point of contact with the screen is lifted. As such, it is likely that an application will need to track individual touches over multiple touch events. While the ID of a specific touch gesture will not change from one event to the next, it is important to keep in mind that the index value will change as other touch events come and go. When working with a touch gesture over multiple events, therefore, it is essential that the ID value be used as the touch reference in order to make sure the same touch is being tracked. When calling methods that require an index value, this should be obtained by converting the ID for a touch to the corresponding index value via a call to the *findPointerIndex()* method of the *MotionEvent* object.

26.5 An Example Multi-Touch Application

The example application created in the remainder of this chapter will track up to two touch gestures as they move across a layout view. As the events for each touch are triggered, the coordinates, index and ID for each touch will be displayed on the screen.

Create a new project in Android Studio, entering *MotionEvent* into the

Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *MotionEventActivity* with a corresponding layout file named *activity_motion_event*.

Click on the *Finish* button to initiate the project creation process.

26.6 Designing the Activity User Interface

The user interface for the application's sole activity is to consist of a ConstraintLayout view containing two TextView objects. Within the Project tool window, navigate to *app -> res -> layout* and double-click on the *activity_motion_event.xml* layout resource file to load it into the Android Studio Layout Editor tool.

Select and delete the default "Hello World!" TextView widget and then, with autoconnect enabled, drag and drop a new TextView widget so that it is centered horizontally and positioned at the 16dp margin line on the top edge of the layout:



Figure 26-1

Drag a second TextView widget and position and constrain it so that it is distanced by a 32dp margin from the bottom of the first widget:

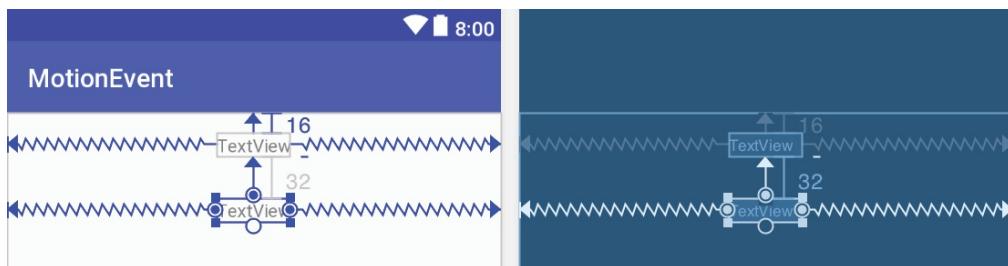


Figure 26-2

Using the Attributes tool window, change the IDs for the TextView widgets to *textView1* and *textView2* respectively. Change the text displayed on the

widgets to read “Touch One Status” and “Touch Two Status” and extract the strings to resources using the warning button in the top right-hand corner of the Layout Editor.

Select the ConstraintLayout entry in the Component Tree and use the Attributes panel to change the ID to *activity_motion_event*.

26.7 Implementing the Touch Event Listener

In order to receive touch event notifications it will be necessary to register a touch listener on the layout view within the *onCreate()* method of the *MotionEventActivity* activity class. Select the *MotionEventActivity.java* tab from the Android Studio editor panel to display the source code. Within the *onCreate()* method, add code to identify the ConstraintLayout view object, register the touch listener and implement code which, in this case, is going to call a second method named *handleTouch()* to which is passed the MotionEvent object:

```
package com.ebookfrenzy.motionevent;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
import android.support.constraint.ConstraintLayout;
import android.widget.TextView;

public class MotionEventActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_motion_event);

        ConstraintLayout myLayout =
            (ConstraintLayout) findViewById(R.id.activity_motion_event);

        myLayout.setOnTouchListener(
            new ConstraintLayout.OnTouchListener() {
                public boolean onTouch(View v,
                                      MotionEvent m) {
                    handleTouch(m);
                    return true;
                }
            }
        );
    }

    private void handleTouch(MotionEvent m) {
        if (m.getAction() == MotionEvent.ACTION_UP) {
            String str = "One";
        } else if (m.getAction() == MotionEvent.ACTION_DOWN) {
            String str = "Two";
        }
    }
}
```

```

        }
    }
);
}
.
.
.
}
```

The final task before testing the application is to implement the *handleTouch()* method called by the listener. The code for this method reads as follows:

```

void handleTouch(MotionEvent m)
{
    TextView textView1 =
(TextView) findViewById(R.id.textView1);
    TextView textView2 =
(TextView) findViewById(R.id.textView2);

    int pointerCount = m.getPointerCount();

    for (int i = 0; i < pointerCount; i++)
    {
        int x = (int) m.getX(i);
        int y = (int) m.getY(i);
        int id = m.getPointerId(i);
        int action = m.getActionMasked();
        int actionIndex = m.getActionIndex();
        String actionString;

        switch (action)
        {
            case MotionEvent.ACTION_DOWN:
                actionString = "DOWN";
                break;
            case MotionEvent.ACTION_UP:
                actionString = "UP";
                break;
            case MotionEvent.ACTION_POINTER_DOWN:
                actionString = "PNTR DOWN";
                break;
            case MotionEvent.ACTION_POINTER_UP:
                actionString = "PNTR UP";
                break;
        }
    }
}
```

```

        actionString = "PNTR UP";
        break;
    case MotionEvent.ACTION_MOVE:
        actionString = "MOVE";
        break;
    default:
        actionString = "";
    }

    String touchStatus = "Action: " + actionString + " "
Index: " + actionIndex + " ID: " + id + " X: " + x + " Y: " + y;

    if (id == 0)
        textView1.setText(touchStatus);
    else
        textView2.setText(touchStatus);
}
}

```

Before compiling and running the application, it is worth taking the time to walk through this code systematically to highlight the tasks that are being performed.

The code begins by obtaining references to the two TextView objects in the user interface and identifying how many pointers are currently active on the view:

```

TextView textView1 = (TextView) findViewById(R.id.textView1);
TextView textView2 = (TextView) findViewById(R.id.textView2);

int pointerCount = m.getPointerCount();

```

Next, the *pointerCount* variable is used to initiate a *for* loop which performs a set of tasks for each active pointer. The first few lines of the loop obtain the X and Y coordinates of the touch together with the corresponding event ID, action type and action index. Lastly, a string variable is declared:

```

for (int i = 0; i < pointerCount; i++)
{
    int x = (int) m.getX(i);
    int y = (int) m.getY(i);
    int id = m.getPointerId(i);
    int action = m.getActionMasked();
    int actionIndex = m.getActionIndex();
    String actionString;

```

Since action types equate to integer values, a *switch* statement is used to convert the action type to a more meaningful string value, which is stored in the previously declared *actionString* variable:

```
switch (action)
{
    case MotionEvent.ACTION_DOWN:
        actionString = "DOWN";
        break;
    case MotionEvent.ACTION_UP:
        actionString = "UP";
        break;
    case MotionEvent.ACTION_POINTER_DOWN:
        actionString = "PNTR DOWN";
        break;
    case MotionEvent.ACTION_POINTER_UP:
        actionString = "PNTR UP";
        break;
    case MotionEvent.ACTION_MOVE:
        actionString = "MOVE";
        break;
    default:
        actionString = "";
}
```

Finally, the string message is constructed using the *actionString* value, the action index, touch ID and X and Y coordinates. The ID value is then used to decide whether the string should be displayed on the first or second *TextView* object:

```
String touchStatus = "Action: " + actionString + " Index: "
+ actionIndex + " ID: " + id + " X: " + x + " Y: " + y;

if (id == 0)
    textView1.setText(touchStatus);
else
    textView2.setText(touchStatus);
```

26.8 Running the Example Application

Compile and run the application and, once launched, experiment with single and multiple touches on the screen and note that the text views update to reflect the events as illustrated in [Figure 26-3](#). When running on an emulator, multiple touches may be simulated by holding down the Ctrl (Cmd on

macOS) key while clicking the mouse button:

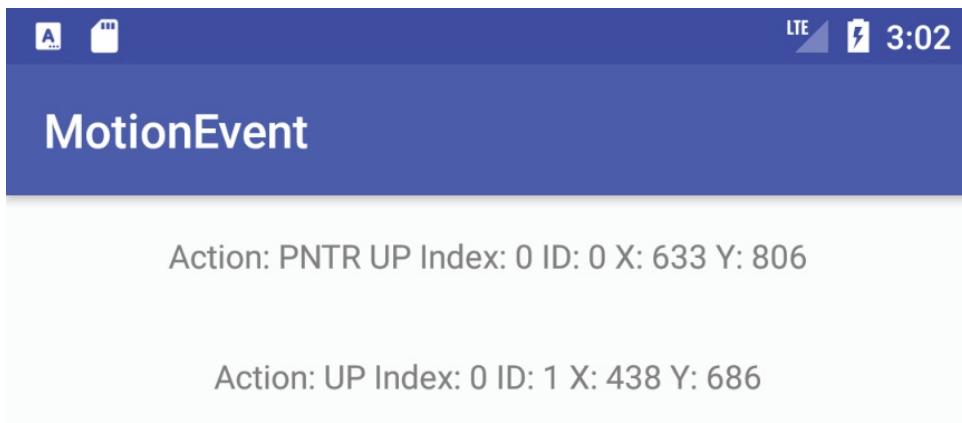


Figure 26-3

26.9 Summary

Activities receive notifications of touch events by registering an `onTouchListener` event listener and implementing the `onTouch()` callback method which, in turn, is passed a `MotionEvent` object when called by the Android runtime. This object contains information about the touch such as the type of touch event, the coordinates of the touch and a count of the number of touches currently in contact with the view.

When multiple touches are involved, each point of contact is referred to as a pointer with each assigned an index and an ID. While the index of a touch can change from one event to another, the ID will remain unchanged until the touch ends.

This chapter has worked through the creation of an example Android application designed to display the coordinates and action type of up to two simultaneous touches on a device display.

Having covered touches in general, the next chapter (entitled "["Detecting Common Gestures using the Android Gesture Detector Class"](#)") will look further at touch screen event handling through the implementation of gesture recognition.

27. Detecting Common Gestures using the Android Gesture Detector Class

The term “gesture” is used to define a contiguous sequence of interactions between the touch screen and the user. A typical gesture begins at the point that the screen is first touched and ends when the last finger or pointing device leaves the display surface. When correctly harnessed, gestures can be implemented as a form of communication between user and application. Swiping motions to turn the pages of an eBook, or a pinching movement involving two touches to zoom in or out of an image are prime examples of the ways in which gestures can be used to interact with an application.

The Android SDK provides mechanisms for the detection of both common and custom gestures within an application. Common gestures involve interactions such as a tap, double tap, long press or a swiping motion in either a horizontal or a vertical direction (referred to in Android nomenclature as a *fling*).

The goal of this chapter is to explore the use of the Android GestureDetector class to detect common gestures performed on the display of an Android device. The next chapter, entitled [“Implementing Custom Gesture and Pinch Recognition on Android”](#), will cover the detection of more complex, custom gestures such as circular motions and pinches.

27.1 Implementing Common Gesture Detection

When a user interacts with the display of an Android device, the *onTouchEvent()* method of the currently active application is called by the system and passed MotionEvent objects containing data about the user’s contact with the screen. This data can be interpreted to identify if the motion on the screen matches a common gesture such as a tap or a swipe. This can be achieved with very little programming effort by making use of the Android GestureDetectorCompat class. This class is designed specifically to receive motion event information from the application and to trigger method calls based on the type of common gesture, if any, detected.

The basic steps in detecting common gestures are as follows:

1. Declaration of a class which implements the

`GestureDetector.OnGestureListener` interface including the required `onFling()`, `onDown()`, `onScroll()`, `onShowPress()`, `onSingleTapUp()` and `onLongPress()` callback methods. Note that this can be either an entirely new class, or the enclosing activity class. In the event that double tap gesture detection is required, the class must also implement the `GestureDetector.OnDoubleTapListener` interface and include the corresponding `onDoubleTap()` method.

2. Creation of an instance of the Android `GestureDetectorCompat` class, passing through an instance of the class created in step 1 as an argument.
3. An optional call to the `setOnDoubleTapListener()` method of the `GestureDetectorCompat` instance to enable double tap detection if required.
4. Implementation of the `onTouchEvent()` callback method on the enclosing activity which, in turn, must call the `onTouchEvent()` method of the `GestureDetectorCompat` instance, passing through the current motion event object as an argument to the method.

Once implemented, the result is a set of methods within the application code that will be called when a gesture of a particular type is detected. The code within these methods can then be implemented to perform any tasks that need to be performed in response to the corresponding gesture.

In the remainder of this chapter, we will work through the creation of an example project intended to put the above steps into practice.

27.2 Creating an Example Gesture Detection Project

The goal of this project is to detect the full range of common gestures currently supported by the `GestureDetectorCompat` class and to display status information to the user indicating the type of gesture that has been detected.

Create a new project in Android Studio, entering *CommonGestures* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *CommonGesturesActivity* with a corresponding layout

resource file named *activity_common_gestures*.

Click on the *Finish* button to initiate the project creation process.

Once the new project has been created, navigate to the *app -> res -> layout -> activity_common_gestures.xml* file in the Project tool window and double-click on it to load it into the Layout Editor tool.

Within the Layout Editor tool, select the “Hello, World!” TextView component and, in the Attributes tool window, enter *gestureStatusText* as the ID.

27.3 Implementing the Listener Class

As previously outlined, it is necessary to create a class that implements the GestureDetector.OnGestureListener interface and, if double tap detection is required, the GestureDetector.OnDoubleTapListener interface. While this can be an entirely new class, it is also perfectly valid to implement this within the current activity class. For the purposes of this example, therefore, we will modify the CommonGesturesActivity class to implement these listener interfaces. Edit the *CommonGesturesActivity.java* file so that it reads as follows to declare the interfaces and to extract and store a reference to the TextView component in the user interface:

```
package com.ebookfrenzy.commongestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.GestureDetector;
import android.widget.TextView;

public class CommonGesturesActivity extends AppCompatActivity
    implements GestureDetector.OnGestureListener,
    GestureDetector.OnDoubleTapListener
{
    private TextView gestureText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_common_gestures);

        gestureText =
            (TextView) findViewById(R.id.gestureStatusText);
```

```
}

.

.

}
```

Declaring that the class implements the listener interfaces mandates that the corresponding methods also be implemented in the class:

```
package com.ebookfrenzy.commongestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.GestureDetector;
import android.widget.TextView;
import android.view.MotionEvent;

public class CommonGesturesActivity extends AppCompatActivity
    implements GestureDetector.OnGestureListener,
    GestureDetector.OnDoubleTapListener {

    private TextView gestureText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_common_gestures);
        gestureText =
            (TextView) findViewById(R.id.gestureStatusText);
    }

    @Override
    public boolean onDown(MotionEvent event) {
        gestureText.setText ("onDown");
        return true;
    }

    @Override
    public boolean onFling(MotionEvent event1, MotionEvent event2,
                          float velocityX, float velocityY) {
        gestureText.setText("onFling");
        return true;
    }
}
```

```
@Override
public void onLongPress(MotionEvent event) {
    gestureText.setText("onLongPress");
}

@Override
public boolean onScroll(MotionEvent e1, MotionEvent e2,
                       float distanceX, float distanceY) {
    gestureText.setText("onScroll");
    return true;
}

@Override
public void onShowPress(MotionEvent event) {
    gestureText.setText("onShowPress");
}

@Override
public boolean onSingleTapUp(MotionEvent event) {
    gestureText.setText("onSingleTapUp");
    return true;
}

@Override
public boolean onDoubleTap(MotionEvent event) {
    gestureText.setText("onDoubleTap");
    return true;
}

@Override
public boolean onDoubleTapEvent(MotionEvent event) {
    gestureText.setText("onDoubleTapEvent");
    return true;
}

@Override
public boolean onSingleTapConfirmed(MotionEvent event) {
    gestureText.setText("onSingleTapConfirmed");
    return true;
}

.
```

```
}
```

Note that many of these methods return *true*. This indicates to the Android Framework that the event has been consumed by the method and does not need to be passed to the next event handler in the stack.

27.4 Creating the GestureDetectorCompat Instance

With the activity class now updated to implement the listener interfaces, the next step is to create an instance of the `GestureDetectorCompat` class. Since this only needs to be performed once at the point that the activity is created, the best place for this code is in the `onCreate()` method. Since we also want to detect double taps, the code also needs to call the `setOnDoubleTapListener()` method of the `GestureDetectorCompat` instance:

```
package com.ebookfrenzy.commongestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.GestureDetector;
import android.widget.TextView;
import android.view.MotionEvent;
import android.support.v4.view.GestureDetectorCompat;

public class CommonGesturesActivity extends AppCompatActivity
    implements GestureDetector.OnGestureListener,
    GestureDetector.OnDoubleTapListener {

    private TextView gestureText;
    private GestureDetectorCompat gDetector;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_common_gestures);
        gestureText =
            (TextView) findViewById(R.id.gestureStatusText);

        this.gDetector = new GestureDetectorCompat(this, this);
        gDetector.setOnDoubleTapListener(this);
    }
}
```

```
}
```

27.5 Implementing the onTouchEvent() Method

If the application were to be compiled and run at this point, nothing would happen if gestures were performed on the device display. This is because no code has been added to intercept touch events and to pass them through to the GestureDetectorCompat instance. In order to achieve this, it is necessary to override the *onTouchEvent()* method within the activity class and implement it such that it calls the *onTouchEvent()* method of the GestureDetectorCompat instance. Remaining in the *CommonGesturesActivity.java* file, therefore, implement this method so that it reads as follows:

```
@Override  
public boolean onTouchEvent(MotionEvent event) {  
    this.gDetector.onTouchEvent(event);  
    // Be sure to call the superclass implementation  
    return super.onTouchEvent(event);  
}
```

27.6 Testing the Application

Compile and run the application on either a physical Android device or an AVD emulator. Once launched, experiment with swipes, presses, scrolling motions and double and single taps. Note that the text view updates to reflect the events as illustrated in [Figure 27-1](#):



Figure 27-1

27.7 Summary

Any physical contact between the user and the touch screen display of a device can be considered a “gesture”. Lacking the physical keyboard and mouse pointer of a traditional computer system, gestures are widely used as a method of interaction between user and application. While a gesture can be comprised of just about any sequence of motions, there is a widely used set of gestures with which users of touch screen devices have become familiar. A number of these so-called “common gestures” can be easily detected within an application by making use of the Android Gesture Detector classes. In this chapter, the use of this technique has been outlined both in theory and through the implementation of an example project.

Having covered common gestures in this chapter, the next chapter will look at detecting a wider range of gesture types including the ability to both design and detect your own gestures.

28. Implementing Custom Gesture and Pinch Recognition on Android

The previous chapter looked at the steps involved in detecting what are referred to as “common gestures” from within an Android application. In practice, however, a gesture can conceivably involve just about any sequence of touch motions on the display of an Android device. In recognition of this fact, the Android SDK allows custom gestures of just about any nature to be defined by the application developer and used to trigger events when performed by the user. This is a multistage process, the details of which are the topic of this chapter.

28.1 The Android Gesture Builder Application

The Android SDK allows developers to design custom gestures which are then stored in a gesture file bundled with an Android application package. These custom gesture files are most easily created using the *Gesture Builder* application which is bundled with the samples package supplied as part of the Android SDK. The creation of a gestures file involves launching the Gesture Builder application, either on a physical device or emulator, and “drawing” the gestures that will need to be detected by the application. Once the gestures have been designed, the file containing the gesture data can be pulled off the SD card of the device or emulator and added to the application project. Within the application code, the file is then loaded into an instance of the *GestureLibrary* class where it can be used to search for matches to any gestures performed by the user on the device display.

28.2 The GestureOverlayView Class

In order to facilitate the detection of gestures within an application, the Android SDK provides the *GestureOverlayView* class. This is a transparent view that can be placed over other views in the user interface for the sole purpose of detecting gestures.

28.3 Detecting Gestures

Gestures are detected by loading the gestures file created using the *Gesture Builder* app and then registering a *GesturePerformedListener* event listener on

an instance of the GestureOverlayView class. The enclosing class is then declared to implement both the *OnGesturePerformedListener* interface and the corresponding *onGesturePerformed* callback method required by that interface. In the event that a gesture is detected by the listener, a call to the *onGesturePerformed* callback method is triggered by the Android runtime system.

28.4 Identifying Specific Gestures

When a gesture is detected, the *onGesturePerformed* callback method is called and passed as arguments a reference to the GestureOverlayView object on which the gesture was detected, together with a Gesture object containing information about the gesture.

With access to the Gesture object, the GestureLibrary can then be used to compare the detected gesture to those contained in the gestures file previously loaded into the application. The GestureLibrary reports the probability that the gesture performed by the user matches an entry in the gestures file by calculating a *prediction score* for each gesture. A prediction score of 1.0 or greater is generally accepted to be a good match between a gesture stored in the file and that performed by the user on the device display.

28.5 Building and Running the Gesture Builder Application

The Gesture Builder application is bundled by default with the AVD emulator profile for most versions of the SDK. It is not, however, pre-installed on most physical Android devices. If the utility is pre-installed, it will be listed along with the other apps installed in the device or AVD instance. In the event that it is not installed, the source code for the utility is included with the sample code provided with this book. If you have not already done so, download this now using the following link:

<http://www.ebookfrenzy.com/retail/androidstudio30/index.php>

The source code for the Gesture Builder application is located within this archive in a folder named *GestureBuilder*.

The GestureBuilder project is based on Android 5.0.1 (API 21) so use the SDK Manager tool once again to ensure that this version of the Android SDK is installed before proceeding.

From the Android Studio welcome screen select the *Import project* option. Alternatively, from the Android Studio main window for an existing project, select the *File -> New -> Import Project...* menu option and, within the resulting dialog, navigate to and select the GestureBuilder folder within the samples directory and click on *OK*. At this point, Android Studio will import the project into the designated folder and convert it to match the Android Studio project file and build structure.

Once imported, install and run the GestureBuilder utility on an Android device attached to the development system.

28.6 Creating a Gestures File

Once the Gesture Builder application has loaded, it should indicate that no gestures have yet been created. To create a new gesture, click on the *Add gesture* button located at the bottom of the device screen, enter the name *Circle Gesture* into the *Name* text box and then “draw” a gesture using a circular motion on the screen as illustrated in [Figure 28-1](#). Assuming that the gesture appears as required (represented by the yellow line on the device screen), click on the *Done* button to add the gesture to the gestures file:

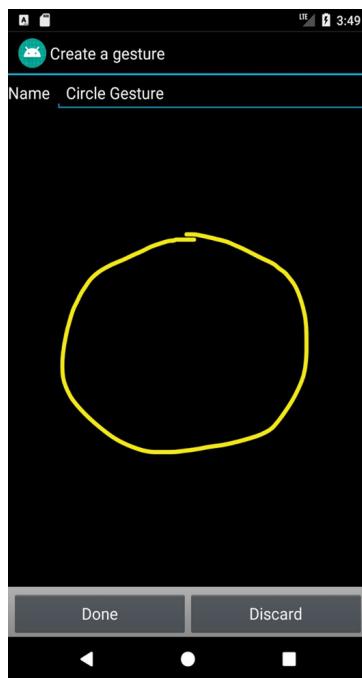


Figure 28-1

After the gesture has been saved, the Gesture Builder app will display a list of currently defined gestures, which, at this point, will consist solely of the new

Circle Gesture.

Repeat the gesture creation process to add a further gesture to the file. This should involve a two-stroke gesture creating an X on the screen named *X Gesture*. When creating gestures involving multiple strokes, be sure to allow as little time as possible between each stroke so that the builder knows that the strokes are part of the same gesture. Once this gesture has been added, the list within the Gesture Builder application should resemble that outlined in [Figure 28-2](#):

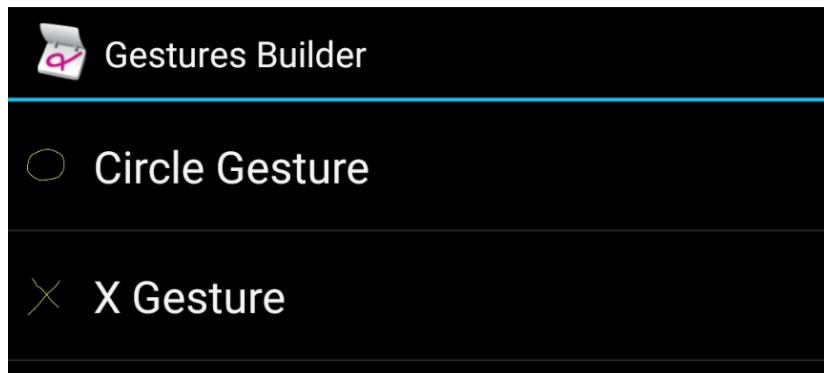


Figure 28-2

28.7 Creating the Example Project

Create a new project in Android Studio, entering *CustomGestures* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *CustomGesturesActivity* with a corresponding layout file named *activity_custom_gestures*.

Click on the *Finish* button to initiate the project creation process.

28.8 Extracting the Gestures File from the SD Card

As each gesture was created within the Gesture Builder application, it was added to a file named *gestures* located on the SD Card of the emulator or device on which the app was running. Before this file can be added to an Android Studio project, however, it must first be pulled off the SD Card and saved to the local file system. This is most easily achieved by using the

Android Studio Device File Explorer tool window. Display this tool using the *View -> Tool Windows -> Device File Explorer* menu option. Once displayed, select the device on which the gesture file was created from the dropdown menu, then navigate through the filesystem to the */sdcard* folder:

The screenshot shows the Device File Explorer interface with the title bar "Emulator Nexus_5X_API_26_2 Android 8.0.0, API 26". The tree view shows the following directory structure under "/sdcard":

Name	Permissions	Date	Size
etc	rw-rw----	2017-07-24 13:16	11 B
mnt	drwxr-xr-x	2017-07-24 13:16	220 B
oem	drwxr-xr-x	1969-12-31 19:00	40 B
proc	dr-xr-xr-x	2017-07-24 13:16	0 B
root	drwx-----	2017-04-18 20:57	40 B
sbin	drwxr-x---	1969-12-31 19:00	120 B
sdcard	lrwxrwxrwx	1969-12-31 19:00	21 B
Alarms	drwxrwx---	2017-07-13 10:57	4 KB
Android	drwxrwx---	2017-07-13 10:58	4 KB
DCIM	drwxrwx---	2017-07-13 10:57	4 KB
Download	drwxrwx---	2017-07-13 10:57	4 KB
Movies	drwxrwx---	2017-07-13 10:57	4 KB
Music	drwxrwx---	2017-07-13 10:57	4 KB
Notifications	drwxrwx---	2017-07-13 10:57	4 KB
Pictures	drwxrwx---	2017-07-13 10:57	4 KB
Podcasts	drwxrwx---	2017-07-13 10:57	4 KB
Ringtones	drwxrwx---	2017-07-13 10:57	4 KB
gestures	-rw-rw----	2017-07-24 13:21	1.1 KB
storage	drwxr-xr-x	2017-07-24 13:17	100 B
sys	dr-xr-xr-x	2017-07-24 13:16	0 B

Figure 28-3

Locate the *gestures* file in this folder, right click on it and select the *Save as...* menu and save the file to a temporary location.

Once the gestures file has been created and pulled off the SD Card, it is ready to be added to an Android Studio project as a resource file.

28.9 Adding the Gestures File to the Project

Within the Android Studio Project tool window, locate and right-click on the *res* folder (located under *app*) and select *New -> Directory* from the resulting menu. In the New Directory dialog, enter *raw* as the folder name and click on the *OK* button. Using the appropriate file explorer utility for your operating system type, locate the *gestures* file previously pulled from the SD Card and copy and paste it into the new *raw* folder in the Project tool window.

28.10 Designing the User Interface

This example application calls for a very simple user interface consisting of a

LinearLayout view with a GestureOverlayView layered on top of it to intercept any gestures performed by the user. Locate the *app -> res -> layout -> activity_custom_gestures.xml* file and double-click on it to load it into the Layout Editor tool.

Once loaded, switch to Text mode and modify the XML so that it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <android.gesture.GestureOverlayView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/gOverlay"
        android:layout_gravity="center_horizontal">

    </android.gesture.GestureOverlayView>
</LinearLayout>
```

28.1 Loading the Gestures File

Now that the gestures file has been added to the project, the next step is to write some code so that the file is loaded when the activity starts up. For the purposes of this project, the code to achieve this will be added to the *CustomGesturesActivity* class located in the *CustomGesturesActivity.java* source file as follows:

```
package com.ebookfrenzy.customgestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.gesture.GestureLibraries;
import android.gesture.GestureLibrary;
import android.gesture.GestureOverlayView;
import android.gesture.GestureOverlayView.OnGesturePerformedListener;

public class CustomGesturesActivity extends AppCompatActivity
    implements OnGesturePerformedListener {
```

```

private GestureLibrary gLibrary;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_custom_gestures);

    gestureSetup();
}

private void gestureSetup() {
    gLibrary =
        GestureLibraries.fromRawResource(this,
            R.raw.gestures);
    if (!gLibrary.load()) {
        finish();
    }
}

.
.
.
}

```

In addition to some necessary import directives, the above code also creates a *GestureLibrary* instance named *gLibrary* and then loads into it the contents of the gestures file located in the *raw* resources folder. The activity class has also been modified to implement the *OnGesturePerformedListener* interface, which requires the implementation of the *onGesturePerformed* callback method (which will be created in a later section of this chapter).

28.12 Registering the Event Listener

In order for the activity to receive notification that the user has performed a gesture on the screen, it is necessary to register the *OnGesturePerformedListener* event listener on the *gLayout* view, a reference to which can be obtained using the *findViewById* method as outlined in the following code fragment:

```

private void gestureSetup() {
    gLibrary =
        GestureLibraries.fromRawResource(this,
            R.raw.gestures);
    if (!gLibrary.load()) {

```

```

        finish();
    }

    GestureOverlayView gOverlay = findViewById(R.id.gOverlay);
    gOverlay.addOnGesturePerformedListener(this);
}

```

28.13 Implementing the onGesturePerformed Method

All that remains before an initial test run of the application can be performed is to implement the *OnGesturePerformed* callback method. This is the method which will be called when a gesture is performed on the GestureOverlayView instance:

```

package com.ebookfrenzy.customgestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.gesture.GestureLibraries;
import android.gesture.GestureLibrary;
import android.gesture.GestureOverlayView;
import android.gesture.GestureOverlayView.OnGesturePerformedListener;
import android.gesture.Prediction;
import android.widget.Toast;
import android.gesture.Gesture;
import java.util.ArrayList;

public class CustomGesturesActivity extends AppCompatActivity
implements OnGesturePerformedListener {

    private GestureLibrary gLibrary;
    .

    .

    public void onGesturePerformed(GestureOverlayView overlay,
Gesture
        gesture) {
        ArrayList<Prediction> predictions =
            gLibrary.recognize(gesture);

        if (predictions.size() > 0 && predictions.get(0).score > 1.0)
        {

            String action = predictions.get(0).name;

```

```
        Toast.makeText(this, action, Toast.LENGTH_SHORT).show();
    }
}

.
.
.
```

When a gesture on the gesture overlay view object is detected by the Android runtime, the *onGesturePerformed* method is called. Passed through as arguments are a reference to the GestureOverlayView object on which the gesture was detected together with an object of type *Gesture*. The *Gesture* class is designed to hold the information that defines a specific gesture (essentially a sequence of timed points on the screen depicting the path of the strokes that comprise a gesture).

The *Gesture* object is passed through to the *recognize()* method of our *gLibrary* instance, the purpose of which is to compare the current gesture with each gesture loaded from the gestures file. Once this task is complete, the *recognize()* method returns an *ArrayList* object containing a *Prediction* object for each comparison performed. The list is ranked in order from the best match (at position 0 in the array) to the worst. Contained within each prediction object is the name of the corresponding gesture from the gestures file and a prediction score indicating how closely it matches the current gesture.

The code in the above method, therefore, takes the prediction at position 0 (the closest match) makes sure it has a score of greater than 1.0 and then displays a *Toast* message (an Android class designed to display notification pop ups to the user) displaying the name of the matching gesture.

28.14 Testing the Application

Build and run the application on either an emulator or a physical Android device and perform the circle and swipe gestures on the display. When performed, the toast notification should appear containing the name of the gesture that was performed. Note that when a gesture is recognized, it is outlined on the display with a bright yellow line while gestures about which the overlay is uncertain appear as a faded yellow line. While useful during development, this is probably not ideal for a real world application. Clearly, therefore, there is still some more configuration work to do.

28.15 Configuring the GestureOverlayView

By default, the GestureOverlayView is configured to display yellow lines during gestures. The color used to draw recognized and unrecognized gestures can be defined via the `android:gestureColor` and `android:uncertainGestureColor` attributes. For example, to hide the gesture lines, modify the `activity_custom_gestures.xml` file in the example project as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <android.gesture.GestureOverlayView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/gOverlay"
        android:layout_gravity="center_horizontal"
        android:gestureColor="#00000000"
        android:uncertainGestureColor="#00000000" >
    </android.gesture.GestureOverlayView>
</LinearLayout>
```

On re-running the application, gestures should now be invisible (since they are drawn in white on the white background of the LinearLayout view).

28.16 Intercepting Gestures

The GestureOverlayView is, as previously described, a transparent overlay that may be positioned over the top of other views. This leads to the question as to whether events intercepted by the gesture overlay should then be passed on to the underlying views when a gesture has been recognized. This is controlled via the `android:eventsInterceptionEnabled` property of the GestureOverlayView instance. When set to true, the gesture events are not passed to the underlying views when a gesture is recognized. This can be a particularly useful setting when gestures are being performed over a view that might be configured to scroll in response to certain gestures. Setting this property to `true` will avoid gestures also being interpreted as instructions to the underlying view to scroll in a particular direction.

28.1 Detecting Pinch Gestures

Before moving on from touch handling in general and gesture recognition in particular, the last topic of this chapter is that of handling pinch gestures. While it is possible to create and detect a wide range of gestures using the steps outlined in the previous sections of this chapter it is, in fact, not possible to detect a pinching gesture (where two fingers are used in a stretching and pinching motion, typically to zoom in and out of a view or image) using the techniques discussed so far.

The simplest method for detecting pinch gestures is to use the Android *ScaleGestureDetector* class. In general terms, detecting pinch gestures involves the following three steps:

1. Declaration of a new class which implements the *SimpleOnScaleGestureListener* interface including the required *onScale()*, *onScaleBegin()* and *onScaleEnd()* callback methods.
2. Creation of an instance of the *ScaleGestureDetector* class, passing through an instance of the class created in step 1 as an argument.
3. Implementing the *onTouchEvent()* callback method on the enclosing activity which, in turn, calls the *onTouchEvent()* method of the *ScaleGestureDetector* class.

In the remainder of this chapter, we will create a very simple example designed to demonstrate the implementation of pinch gesture recognition.

28.1A Pinch Gesture Example Project

Create a new project in Android Studio, entering *PinchExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *PinchExampleActivity* with a layout resource file named *activity_pinch_example*.

Within the *activity_pinch_example.xml* file, select the default *TextView* object and use the Attributes tool window to set the ID to *myTextView*.

Locate and load the *PinchExampleActivity.java* file into the Android Studio

editor and modify the file as follows:

```
package com.ebookfrenzy.pinchexample;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.ScaleGestureDetector;
import android.view.ScaleGestureDetector.SimpleOnScaleGestureListener;
import android.widget.TextView;

public class PinchExampleActivity extends AppCompatActivity {

    TextView scaleText;
    ScaleGestureDetector scaleGestureDetector;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_pinch_example);

        scaleText = (TextView) findViewById(R.id.myTextView);

        scaleGestureDetector =
            new ScaleGestureDetector(this,
                new MyOnScaleGestureListener());
    }

    @Override
    public boolean onTouchEvent(MotionEvent event) {
        scaleGestureDetector.onTouchEvent(event);
        return true;
    }

    public class MyOnScaleGestureListener extends
        SimpleOnScaleGestureListener {

        @Override
        public boolean onScale(ScaleGestureDetector detector) {

            float scaleFactor = detector.getScaleFactor();

            if (scaleFactor > 1) {
```

```

        scaleText.setText("Zooming Out");
    } else {
        scaleText.setText("Zooming In");
    }
    return true;
}

@Override
public boolean onScaleBegin(ScaleGestureDetector detector) {
    return true;
}

@Override
public void onScaleEnd(ScaleGestureDetector detector) {

}
}

.
.
.
}

```

The code declares a new class named *MyOnScaleGestureListener* which extends the Android *SimpleOnScaleGestureListener* class. This interface requires that three methods (*onScale()*, *onScaleBegin()* and *onScaleEnd()*) be implemented. In this instance the *onScale()* method identifies the scale factor and displays a message on the text view indicating the type of pinch gesture detected.

Within the *onCreate()* method, a reference to the text view object is obtained and assigned to the *scaleText* variable. Next, a new *ScaleGestureDetector* instance is created, passing through a reference to the enclosing activity and an instance of our new *MyOnScaleGestureListener* class as arguments. Finally, an *onTouchEvent()* callback method is implemented for the activity, which simply calls the corresponding *onTouchEvent()* method of the *ScaleGestureDetector* object, passing through the *MotionEvent* object as an argument.

Compile and run the application on an emulator or physical Android device and perform pinching gestures on the screen, noting that the text view displays either the zoom in or zoom out message depending on the pinching motion. Pinching gestures may be simulated within the emulator by holding

down the Ctrl (or Cmd) key and clicking and dragging the mouse pointer as shown in [Figure 28-4](#):

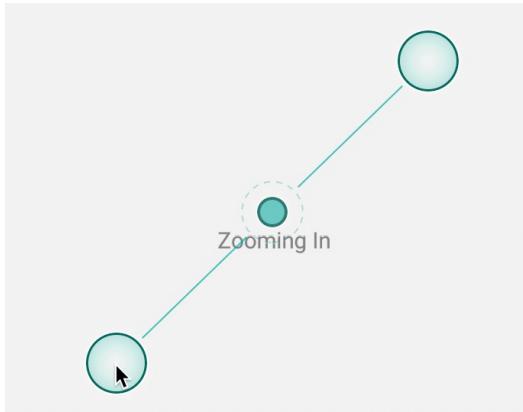


Figure 28-4

28.1 Summary

A gesture is essentially the motion of points of contact on a touch screen involving one or more strokes and can be used as a method of communication between user and application. Android allows gestures to be designed using the Gesture Builder application. Once created, gestures can be saved to a gestures file and loaded into an activity at application runtime using the GestureLibrary.

Gestures can be detected on areas of the display by overlaying existing views with instances of the transparent *GestureOverlayView* class and implementing an *OnGesturePerformedListener* event listener. Using the GestureLibrary, a ranked list of matches between a gesture performed by the user and the gestures stored in a gestures file may be generated, using a prediction score to decide whether a gesture is a close enough match.

Pinch gestures may be detected through the implementation of the *ScaleGestureDetector* class, an example of which was also provided in this chapter.

29. An Introduction to Android Fragments

As you progress through the chapters of this book it will become increasingly evident that many of the design concepts behind the Android system were conceived with the goal of promoting reuse of, and interaction between, the different elements that make up an application. One such area that will be explored in this chapter involves the use of Fragments.

This chapter will provide an overview of the basics of fragments in terms of what they are and how they can be created and used within applications. The next chapter will work through a tutorial designed to show fragments in action when developing applications in Android Studio, including the implementation of communication between fragments.

29.1 What is a Fragment?

A fragment is a self-contained, modular section of an application's user interface and corresponding behavior that can be embedded within an activity. Fragments can be assembled to create an activity during the application design phase, and added to or removed from an activity during application runtime to create a dynamically changing user interface.

Fragments may only be used as part of an activity and cannot be instantiated as standalone application elements. That being said, however, a fragment can be thought of as a functional “sub-activity” with its own lifecycle similar to that of a full activity.

Fragments are stored in the form of XML layout files and may be added to an activity either by placing appropriate `<fragment>` elements in the activity's layout file, or directly through code within the activity's class implementation.

Before starting to use Fragments in an Android application, it is important to be aware that Fragments were not introduced to Android until version 3.0 of the Android SDK. An application that uses Fragments must, therefore, make use of the `android-support-v4` Android Support Library in order to be compatible with older Android versions. The steps to achieve this will be covered in the next chapter, entitled [“Using Fragments in Android Studio - An Example”](#).

29.2 Creating a Fragment

The two components that make up a fragment are an XML layout file and a corresponding Java class. The XML layout file for a fragment takes the same format as a layout for any other activity layout and can contain any combination and complexity of layout managers and views. The following XML layout, for example, is for a fragment consisting simply of a `RelativeLayout` with a red background containing a single `TextView`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/red" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/fragone_label_text"
        android:textAppearance="?android:attr/textAppearanceLarge" />
</RelativeLayout>
```

The corresponding class to go with the layout must be a subclass of the Android *Fragment* class. If the application is to be compatible with devices running versions of Android predating version 3.0 then the class file must import `android.support.v4.app.Fragment`. The class should, at a minimum, override the `onCreateView()` method which is responsible for loading the fragment layout. For example:

```
package com.example.myfragmentdemo;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class FragmentOne extends Fragment {

    @Override
```

```

        public View onCreateView(LayoutInflater inflater,
                               ViewGroup container,
                               Bundle savedInstanceState) {
            // Inflate the layout for this fragment
            return inflater.inflate(R.layout.fragment_one_layout,
                                   container, false);
    }
}

```

In addition to the `onCreateView()` method, the class may also override the standard `onActivityCreated()` method.

Note that in order to make the above fragment compatible with Android versions prior to version 3.0, the `Fragment` class from the v4 support library has been imported.

Once the fragment layout and class have been created, the fragment is ready to be used within application activities.

29.3 Adding a Fragment to an Activity using the Layout XML File

Fragments may be incorporated into an activity either by writing Java code or by embedding the fragment into the activity's XML layout file. Regardless of the approach used, a key point to be aware of is that when the support library is being used for compatibility with older Android releases, any activities using fragments must be implemented as a subclass of `FragmentActivity` instead of the `AppCompatActivity` class:

```

package com.example.myfragmentdemo;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.view.Menu;

public class FragmentDemoActivity extends FragmentActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fragment_demo);
    }
}

```

Fragments are embedded into activity layout files using the `<fragment>`

element. The following example layout embeds the fragment created in the previous section of this chapter into an activity layout:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".FragmentDemoActivity" >

    <fragment
        android:id="@+id/fragment_one"
        android:name="com.example.myfragmentdemo.myfragmentdemo.FragmentOne"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
        tools:layout="@layout/fragment_one_layout" />

</RelativeLayout>
```

The key properties within the `<fragment>` element are `android:name`, which must reference the class associated with the fragment, and `tools:layout`, which must reference the XML resource file containing the layout of the fragment.

Once added to the layout of an activity, fragments may be viewed and manipulated within the Android Studio Layout Editor tool. [Figure 29-1](#), for example, shows the above layout with the embedded fragment within the Android Studio Layout Editor:



Figure 29-1

29.4 Adding and Managing Fragments in Code

The ease of adding a fragment to an activity via the activity's XML layout file comes at the cost of the activity not being able to remove the fragment at runtime. In order to achieve full dynamic control of fragments during runtime, those activities must be added via code. This has the advantage that the fragments can be added, removed and even made to replace one another dynamically while the application is running.

When using code to manage fragments, the fragment itself will still consist of an XML layout file and a corresponding class. The difference comes when working with the fragment within the hosting activity. There is a standard sequence of steps when adding a fragment to an activity using code:

1. Create an instance of the fragment's class.
2. Pass any additional intent arguments through to the class.
3. Obtain a reference to the fragment manager instance.
4. Call the `beginTransaction()` method on the fragment manager instance.
This returns a fragment transaction instance.
5. Call the `add()` method of the fragment transaction instance, passing through as arguments the resource ID of the view that is to contain the fragment and the fragment class instance.
6. Call the `commit()` method of the fragment transaction.

The following code, for example, adds a fragment defined by the `FragmentOne` class so that it appears in the container view with an ID of `LinearLayout1`:

```
FragmentOne firstFragment = new FragmentOne();
firstFragment.setArguments(getIntent().getExtras());

FragmentManager fragManager =
getSupportFragmentManager();
FragmentTransaction transaction =
fragManager.beginTransaction();

transaction.add(R.id.LinearLayout1, firstFragment);
transaction.commit();
```

The above code breaks down each step into a separate statement for the

purposes of clarity. The last four lines can, however, be abbreviated into a single line of code as follows:

```
getSupportFragmentManager().beginTransaction()  
    .add(R.id.LinearLayout1, firstFragment).commit();
```

Once added to a container, a fragment may subsequently be removed via a call to the *remove()* method of the fragment transaction instance, passing through a reference to the fragment instance that is to be removed:

```
transaction.remove(firstFragment);
```

Similarly, one fragment may be replaced with another by a call to the *replace()* method of the fragment transaction instance. This takes as arguments the ID of the view containing the fragment and an instance of the new fragment. The replaced fragment may also be placed on what is referred to as the *back* stack so that it can be quickly restored in the event that the user navigates back to it. This is achieved by making a call to the *addToBackStack()* method of the fragment transaction object before making the *commit()* method call:

```
FragmentTwo secondFragment = new FragmentTwo();  
transaction.replace(R.id.LinearLayout1, secondFragment);  
transaction.addToBackStack(null);  
transaction.commit();
```

29.5 Handling Fragment Events

As previously discussed, a fragment is very much like a sub-activity with its own layout, class and lifecycle. The view components (such as buttons and text views) within a fragment are able to generate events just like those in a regular activity. This raises the question as to which class receives an event from a view in a fragment; the fragment itself, or the activity in which the fragment is embedded. The answer to this question depends on how the event handler is declared.

In the chapter entitled "[An Overview and Example of Android Event Handling](#)", two approaches to event handling were discussed. The first method involved configuring an event listener and callback method within the code of the activity. For example:

```
button.setOnClickListener(  
    new Button.OnClickListener() {  
        public void onClick(View v) {  
            // Code to be performed when  
            // the button is clicked
```

```
        }  
    }  
);
```

In the case of intercepting click events, the second approach involved setting the *android:onClick* property within the XML layout file:

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

The general rule for events generated by a view in a fragment is that if the event listener was declared in the fragment class using the event listener and callback method approach, then the event will be handled first by the fragment. If the *android:onClick* resource is used, however, the event will be passed directly to the activity containing the fragment.

29.6 Implementing Fragment Communication

Once one or more fragments are embedded within an activity, the chances are good that some form of communication will need to take place both between the fragments and the activity, and between one fragment and another. In fact, good practice dictates that fragments do not communicate directly with one another. All communication should take place via the encapsulating activity.

In order for an activity to communicate with a fragment, the activity must identify the fragment object via the ID assigned to it. Once this reference has been obtained, the activity can simply call the public methods of the fragment object.

Communicating in the other direction (from fragment to activity) is a little more complicated. In the first instance, the fragment must define a listener interface, which is then implemented within the activity class. For example, the following code declares an interface named ToolbarListener on a fragment class named ToolbarFragment. The code also declares a variable in which a reference to the activity will later be stored:

```
public class ToolbarFragment extends Fragment {  
  
    ToolbarListener activityCallback;
```

```
public interface ToolbarListener {  
    public void onButtonClick(int position, String text);  
}  
. . .  
}
```

The above code dictates that any class that implements the ToolbarListener interface must also implement a callback method named *onButtonClick* which, in turn, accepts an integer and a String as arguments.

Next, the *onAttach()* method of the fragment class needs to be overridden and implemented. This method is called automatically by the Android system when the fragment has been initialized and associated with an activity. The method is passed a reference to the activity in which the fragment is contained. The method must store a local reference to this activity and verify that it implements the ToolbarListener interface:

```
@Override  
public void onAttach(Context context) {  
    super.onAttach(context);  
  
    try {  
        activityCallback = (ToolbarListener) activity;  
    } catch (ClassCastException e) {  
        throw new ClassCastException(activity.toString()  
            + " must implement ToolbarListener");  
    }  
}
```

Upon execution of this example, a reference to the activity will be stored in the local *activityCallback* variable, and an exception will be thrown if that activity does not implement the ToolbarListener interface.

The next step is to call the callback method of the activity from within the fragment. When and how this happens is entirely dependent on the circumstances under which the activity needs to be contacted by the fragment. The following code, for example, calls the callback method on the activity when a button is clicked:

```
public void buttonClicked (View view) {  
    activityCallback.onButtonClick(arg1, arg2);  
}
```

All that remains is to modify the activity class so that it implements the ToolbarListener interface. For example:

```
public class FragmentExampleActivity extends FragmentActivity
    implements ToolbarFragment.ToolbarListener {

    public void onButtonClick(String arg1, int arg2) {
        // Implement code for callback method
    }

    .
    .
}
```

As we can see from the above code, the activity declares that it implements the ToolbarListener interface of the ToolbarFragment class and then proceeds to implement the *onButtonClick()* method as required by the interface.

29.7 Summary

Fragments provide a powerful mechanism for creating re-usable modules of user interface layout and application behavior, which, once created, can be embedded in activities. A fragment consists of a user interface layout file and a class. Fragments may be utilized in an activity either by adding the fragment to the activity's layout file, or by writing code to manage the fragments at runtime. Fragments added to an activity in code can be removed and replaced dynamically at runtime. All communication between fragments should be performed via the activity within which the fragments are embedded.

Having covered the basics of fragments in this chapter, the next chapter will work through a tutorial designed to reinforce the techniques outlined in this chapter.

30. Using Fragments in Android Studio - An Example

As outlined in the previous chapter, fragments provide a convenient mechanism for creating reusable modules of application functionality consisting of both sections of a user interface and the corresponding behavior. Once created, fragments can be embedded within activities.

Having explored the overall theory of fragments in the previous chapter, the objective of this chapter is to create an example Android application using Android Studio designed to demonstrate the actual steps involved in both creating and using fragments, and also implementing communication between one fragment and another within an activity.

30.1 About the Example Fragment Application

The application created in this chapter will consist of a single activity and two fragments. The user interface for the first fragment will contain a toolbar of sorts consisting of an EditText view, a SeekBar and a Button, all contained within a RelativeLayout view. The second fragment will consist solely of a TextView object, also contained within a RelativeLayout view.

The two fragments will be embedded within the main activity of the application and communication implemented such that when the button in the first fragment is pressed, the text entered into the EditText view will appear on the TextView of the second fragment using a font size dictated by the position of the SeekBar in the first fragment.

Since this application is intended to work on earlier versions of Android, it will also be necessary to make use of the appropriate Android support library.

30.2 Creating the Example Project

Create a new project in Android Studio with , entering *FragmentExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty

Activity named *FragmentExampleActivity* with a corresponding layout resource file named *activity_fragment_example*.

Click the *Finish* button to begin the project creation process.

30.3 Creating the First Fragment Layout

The next step is to create the user interface for the first fragment that will be used within our activity.

This user interface will, of course, reside in an XML layout file so begin by navigating to the *layout* folder located under *app -> res* in the Project tool window. Once located, right-click on the *layout* entry and select the *New -> Layout resource file* menu option as illustrated in [Figure 30-1](#):

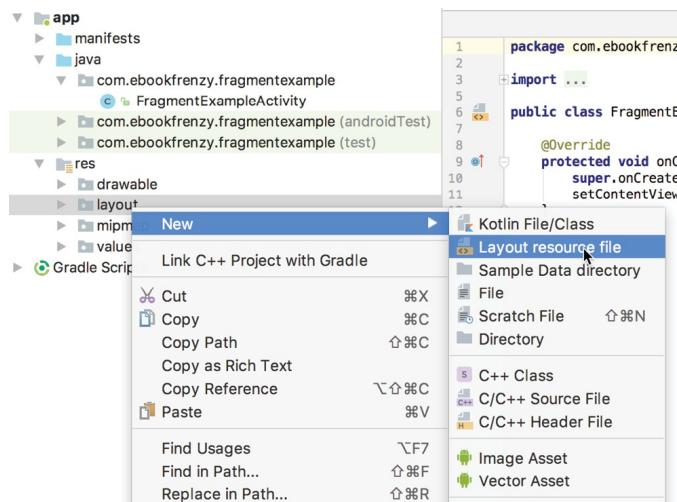


Figure 30-1

In the resulting dialog, name the layout *toolbar_fragment* and change the root element to *RelativeLayout* before clicking on OK to create the new resource file.

The new resource file will appear within the Layout Editor tool ready to be designed. Switch the Layout Editor to Text mode and modify the XML so that it reads as outlined in the following listing to add three new view elements to the layout:

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

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```

<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/seekBar1"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="17dp"
    android:text="Change Text" />

<EditText
    android:id="@+id/editText1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="16dp"
    android:ems="10"
    android:inputType="text" >
    <requestFocus />
</EditText>

<SeekBar
    android:id="@+id/seekBar1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentStart="true"
    android:layout_below="@+id/editText1"
    android:layout_marginTop="14dp"
    android:layout_alignParentLeft="true" />

</RelativeLayout>

```

Once the changes have been made, switch the Layout Editor tool back to Design mode and click on the warning button in the top right-hand corner of the design area. Select the hardcoded text warning, click the *Fix* button and assign the string to a resource named *change_text*.

Upon completion of these steps, the user interface layout should resemble that of [Figure 30-2](#):

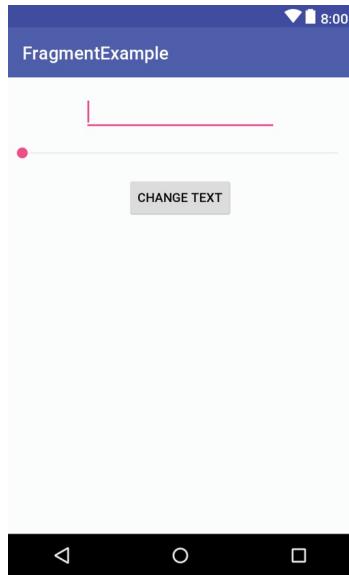


Figure 30-2

With the layout for the first fragment implemented, the next step is to create a class to go with it.

30.4 Creating the First Fragment Class

In addition to a user interface layout, a fragment also needs to have a class associated with it to do the actual work behind the scenes. Add a class for this purpose to the project by unfolding the *app -> java* folder in the Project tool window and right-clicking on the package name given to the project when it was created (in this instance *com.ebookfrenzy.fragmentexample*). From the resulting menu, select the *New -> Java Class* option. In the resulting *Create New Class* dialog, name the class *ToolbarFragment* and click on *OK* to create the new class.

Once the class has been created it should, by default, appear in the editing panel where it will read as follows:

```
package com.ebookfrenzy.fragmentexample;

/**
 * Created by <name> on <date>.
 */
public class ToolbarFragment {
```

For the time being, the only changes to this class are the addition of some import directives and the overriding of the *onCreateView()* method to make

sure the layout file is inflated and displayed when the fragment is used within an activity. The class declaration also needs to indicate that the class extends the Android Fragment class:

```
package com.ebookfrenzy.fragmentexample;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class ToolbarFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
                            ViewGroup container, Bundle
                            savedInstanceState) {

        // Inflate the layout for this fragment
        View view = inflater.inflate(R.layout.toolbar_fragment,
                                    container, false);
        return view;
    }
}
```

Later in this chapter, more functionality will be added to this class. Before that, however, we need to create the second fragment.

30.5 Creating the Second Fragment Layout

Add a second new Android XML layout resource file to the project, once again selecting a RelativeLayout as the root element. Name the layout *text_fragment* and click OK. When the layout loads into the Layout Editor tool, change to Text mode and modify the XML to add a TextView to the fragment layout as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
```

```

    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:text="Fragment Two"
    android:textAppearance="?android:attr/textAppearanceLarge" />
</RelativeLayout>

```

Once the XML changes have been made, switch back to Design mode and extract the string to a resource named *fragment_two*. Upon completion of these steps, the user interface layout for this second fragment should resemble that of [Figure 30-3](#).

As with the first fragment, this one will also need to have a class associated with it. Right-click on *app* -> *java* -> *com.ebookfrenzy.fragmentexample* in the Project tool window. From the resulting menu, select the *New* -> *Java Class* option. Name the fragment *TextFragment* and click *OK* to create the class.

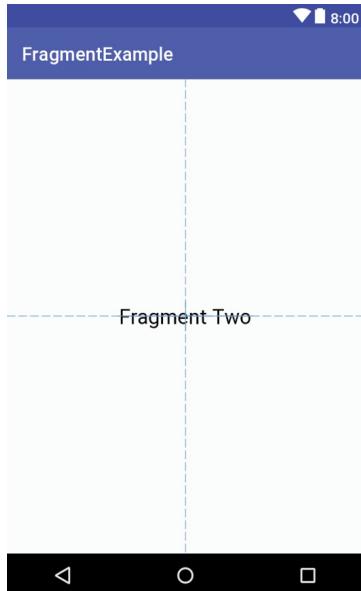


Figure 30-3

Edit the new *TextFragment.java* class file and modify it to implement the *onCreateView()* method and designate the class as extending the Android *Fragment* class:

```

package com.ebookfrenzy.fragmentexample;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;

```

```

import android.view.ViewGroup;

public class TextFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.text_fragment,
                           container, false);

        return view;
    }
}

```

Now that the basic structure of the two fragments has been implemented, they are ready to be embedded in the application's main activity.

30.6 Adding the Fragments to the Activity

The main activity for the application has associated with it an XML layout file named *activity_fragment_example.xml*. For the purposes of this example, the fragments will be added to the activity using the `<fragment>` element within this file. Using the Project tool window, navigate to the *app -> res -> layout* section of the *FragmentExample* project and double-click on the *activity_fragment_example.xml* file to load it into the Android Studio Layout Editor tool.

With the Layout Editor tool in Design mode, select and delete the default *TextView* object from the layout and select the *Layouts* category in the palette. Drag the `<fragment>` component from the list of layouts and drop it onto the layout so that it is centered horizontally and positioned such that the dashed line appears indicating the top layout margin:

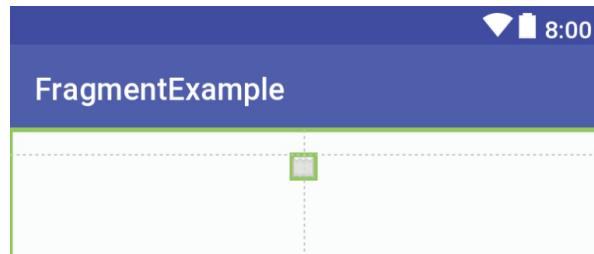


Figure 30-4

On dropping the fragment onto the layout, a dialog will appear displaying a

list of Fragments available within the current project as illustrated in [Figure 30-5](#):



Figure 30-5

Select the ToolbarFragment entry from the list and click on the OK button to dismiss the Fragments dialog. Once added, click on the red warning button in the top right-hand corner of the layout editor to display the warnings panel. An *unknown fragments* message ([Figure 30-6](#)) will be listed indicating that the Layout Editor tool needs to know which fragment to display during the preview session. Display the ToolbarFragment fragment by clicking on the *Use @layout/toolbar_fragment* link within the message:

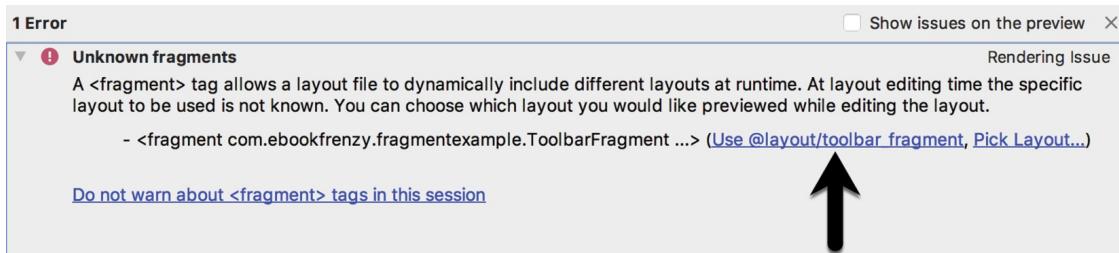


Figure 30-6

Click and drag another <fragment> entry from the panel and position it so that it is centered horizontally and positioned beneath the bottom edge of the first fragment. When prompted, select the *TextFragment* entry from the fragment dialog before clicking on the OK button. When the rendering message appears, click on the *Use @layout/text_fragment* option. Establish a constraint connection between the top edge of the TextFragment and the bottom edge of the ToolbarFragment.

Note that the fragments are now visible in the layout as demonstrated in [Figure 30-7](#):

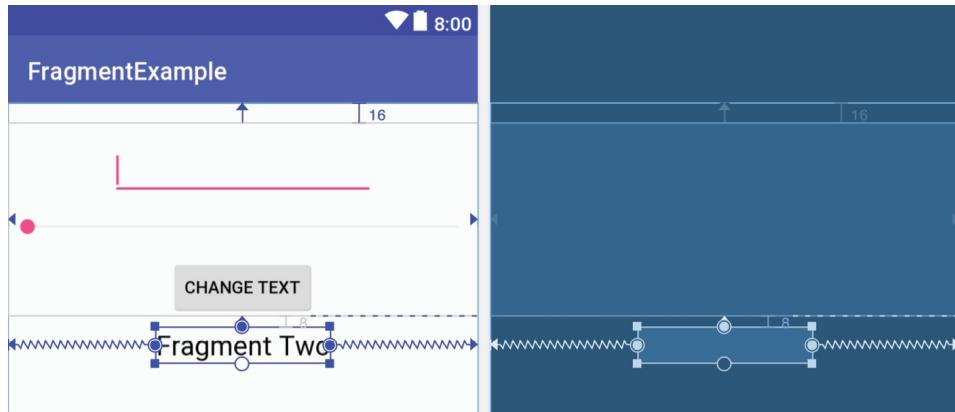


Figure 30-7

Before proceeding to the next step, select the `TextFragment` instance in the layout and, within the Attributes tool window, change the ID of the fragment to `text_fragment`.

30.7 Making the Toolbar Fragment Talk to the Activity

When the user touches the button in the toolbar fragment, the fragment class is going to need to get the text from the `EditTextView` and the current value of the `SeekBar` and send them to the text fragment. As outlined in ["An Introduction to Android Fragments"](#), fragments should not communicate with each other directly, instead using the activity in which they are embedded as an intermediary.

The first step in this process is to make sure that the toolbar fragment responds to the button being clicked. We also need to implement some code to keep track of the value of the `SeekBar` view. For the purposes of this example, we will implement these listeners within the `ToolbarFragment` class. Select the `ToolbarFragment.java` file and modify it so that it reads as shown in the following listing:

```
package com.ebookfrenzy.fragmentexample;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.content.Context;
import android.widget.Button;
import android.widget.EditText;
```

```
import android.widget.SeekBar;
import android.widget.SeekBar.OnSeekBarChangeListener;

public class ToolbarFragment extends Fragment implements OnSeekBarChangeListener {

    private static int seekvalue = 10;
    private static EditText edittext;

    @Override
    public View onCreateView(LayoutInflater inflater,
                            ViewGroup container, Bundle
                            savedInstanceState) {

        // Inflate the layout for this fragment
        View view = inflater.inflate(R.layout.toolbar_fragment,
                                     container, false);

        edittext = (EditText) view.findViewById(R.id.editText1);
        final SeekBar seekbar =
            (SeekBar) view.findViewById(R.id.seekBar1);

        seekbar.setOnSeekBarChangeListener(this);

        final Button button =
            (Button) view.findViewById(R.id.button1);

        button.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                buttonClicked(v);
            }
        });

        return view;
    }

    public void buttonClicked (View view) {

    }

    @Override
    public void onProgressChanged(SeekBar seekBar, int progress,
                                boolean fromUser) {
        seekvalue = progress;
    }
}
```

```

}

@Override
public void onStartTrackingTouch(SeekBar arg0) {

}

@Override
public void onStopTrackingTouch(SeekBar arg0) {

}
}
}

```

Before moving on, we need to take some time to explain the above code changes. First, the class is declared as implementing the OnSeekBarChangeListener interface. This is because the user interface contains a SeekBar instance and the fragment needs to receive notifications when the user slides the bar to change the font size. Implementation of the OnSeekBarChangeListener interface requires that the *onProgressChanged()*, *onStartTrackingTouch()* and *onStopTrackingTouch()* methods be implemented. These methods have been implemented but only the *onProgressChanged()* method is actually required to perform a task, in this case storing the new value in a variable named seekvalue which has been declared at the start of the class. Also declared is a variable in which to store a reference to the EditText object.

The *onCreateView()* method has been modified to obtain references to the EditText, SeekBar and Button views in the layout. Once a reference to the button has been obtained it is used to set up an onClickListener on the button which is configured to call a method named *buttonClicked()* when a click event is detected. This method is also then implemented, though at this point it does not do anything.

The next phase of this process is to set up the listener that will allow the fragment to call the activity when the button is clicked. This follows the mechanism outlined in the previous chapter:

```

public class ToolbarFragment extends Fragment
    implements OnSeekBarChangeListener {

    private static int seekvalue = 10;
    private static EditText edittext;

```

```
ToolbarListener activityCallback;

public interface ToolbarListener {
    public void onButtonClick(int position, String text);
}

@Override
public void onAttach(Context context) {
    super.onAttach(context);
    try {
        activityCallback = (ToolbarListener) context;
    } catch (ClassCastException e) {
        throw new ClassCastException(context.toString()
            + " must implement ToolbarListener");
    }
}

@Override
public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
    // Inflate the layout for this fragment

    View view =
        inflater.inflate(R.layout.toolbar_fragment,
            container, false);

    edittext = (EditText)
        view.findViewById(R.id.editText1);
    final SeekBar seekbar =
        (SeekBar) view.findViewById(R.id.seekBar1);

    seekbar.setOnSeekBarChangeListener(this);

    final Button button =
        (Button) view.findViewById(R.id.button1);
    button.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
            buttonClicked(v);
        }
    });
}

return view;
```

```

    }

    public void buttonClicked (View view) {
        activityCallback.onButtonClick(seekvalue,
            edittext.getText().toString());
    }

    .
    .
    .
}

```

The above implementation will result in a method named *onButtonClick()* belonging to the activity class being called when the button is clicked by the user. All that remains, therefore, is to declare that the activity class implements the newly created ToolbarListener interface and to implement the *onButtonClick()* method.

Since the Android Support Library is being used for fragment support in earlier Android versions, the activity also needs to be changed to subclass from *FragmentActivity* instead of *AppCompatActivity*. Bringing these requirements together results in the following modified *FragmentExampleActivity.java* file:

```

package com.ebookfrenzy.fragmentexample;

import android.support.v7.app.AppCompatActivity;
import android.support.v4.app.FragmentActivity;
import android.os.Bundle;

public class FragmentExampleActivity extends FragmentActivity implements
    ToolbarListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fragment_example);
    }

    public void onButtonClick(int fontsize, String text) {
    }
}

```

With the code changes as they currently stand, the toolbar fragment will detect when the button is clicked by the user and call a method on the activity

passing through the content of the EditText field and the current setting of the SeekBar view. It is now the job of the activity to communicate with the Text Fragment and to pass along these values so that the fragment can update the TextView object accordingly.

30.8 Making the Activity Talk to the Text Fragment

As outlined in [*"An Introduction to Android Fragments"*](#), an activity can communicate with a fragment by obtaining a reference to the fragment class instance and then calling public methods on the object. As such, within the TextFragment class we will now implement a public method named *changeTextProperties()* which takes as arguments an integer for the font size and a string for the new text to be displayed. The method will then use these values to modify the TextView object. Within the Android Studio editing panel, locate and modify the *TextFragment.java* file to add this new method and to add code to the *onCreateView()* method to obtain the ID of the TextView object:

```
package com.ebookfrenzy.fragmentexample;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class TextFragment extends Fragment {

    private static TextView textview;

    @Override
    public View onCreateView(LayoutInflater inflater,
                            ViewGroup container,
                            Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.text_fragment,
                                    container, false);

        textview = (TextView) view.findViewById(R.id.textView1);

        return view;
    }
}
```

```

public void changeTextProperties(int fontsize, String text)
{
    textview.setTextSize(fontsize);
    textview.setText(text);
}
}

```

When the TextFragment fragment was placed in the layout of the activity, it was given an ID of *text_fragment*. Using this ID, it is now possible for the activity to obtain a reference to the fragment instance and call the *changeTextProperties()* method on the object. Edit the *FragmentExampleActivity.java* file and modify the *onButtonClick()* method as follows:

```

public void onButtonClick(int fontsize, String text) {

    TextFragment textFragment =
        (TextFragment)
        getSupportFragmentManager().findFragmentById(R.id.text_fragment);

    textFragment.changeTextProperties(fontsize, text);
}

```

30.9 Testing the Application

With the coding for this project now complete, the last remaining task is to run the application. When the application is launched, the main activity will start and will, in turn, create and display the two fragments. When the user touches the button in the toolbar fragment, the *onButtonClick()* method of the activity will be called by the toolbar fragment and passed the text from the EditText view and the current value of the SeekBar. The activity will then call the *changeTextProperties()* method of the second fragment, which will modify the TextView to reflect the new text and font size:

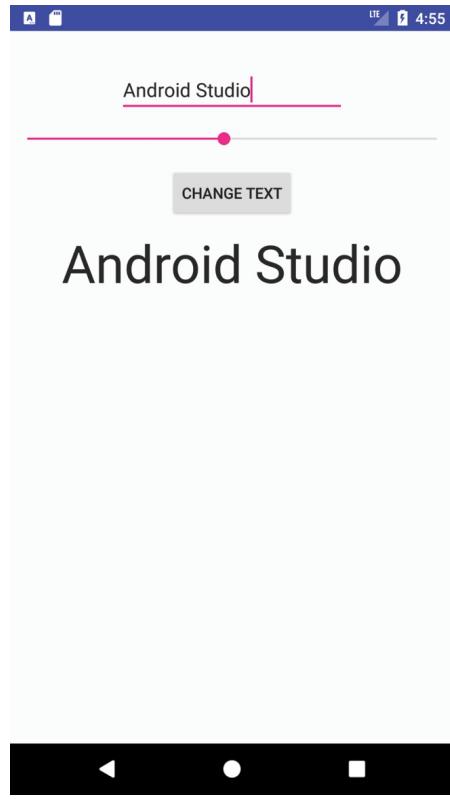


Figure 30-8

30.10 Summary

The goal of this chapter was to work through the creation of an example project intended specifically to demonstrate the steps involved in using fragments within an Android application. Topics covered included the use of the Android Support Library for compatibility with Android versions predating the introduction of fragments, the inclusion of fragments within an activity layout and the implementation of inter-fragment communication.

31. Creating and Managing Overflow Menus on Android

An area of user interface design that has not yet been covered in this book relates to the concept of menus within an Android application. Menus provide a mechanism for offering additional choices to the user beyond the view components that are present in the user interface layout. While there are a number of different menu systems available to the Android application developer, this chapter will focus on the more commonly used Overflow menu. The chapter will cover the creation of menus both manually via XML and visually using the Android Studio Layout Editor tool.

31.1 The Overflow Menu

The overflow menu (also referred to as the options menu) is a menu that is accessible to the user from the device display and allows the developer to include other application options beyond those included in the user interface of the application. The location of the overflow menu is dependent upon the version of Android that is running on the device. With the Android 4.0 release and later, the overflow menu button is located in the top right-hand corner ([Figure 31-1](#)) in the action toolbar represented by the stack of three squares:

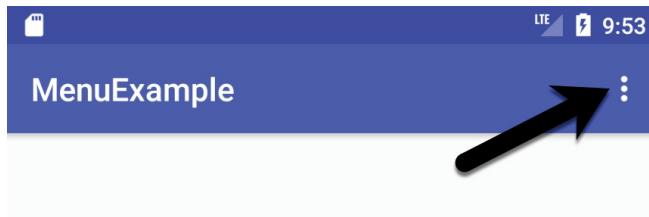


Figure 31-1

31.2 Creating an Overflow Menu

The items in a menu can be declared within an XML file, which is then inflated and displayed to the user on demand. This involves the use of the `<menu>` element, containing an `<item>` sub-element for each menu item. The following XML, for example, defines a menu consisting of two menu items relating to color choices:

```
<menu 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"
tools:context=".MenuExampleActivity" >
<item
    android:id="@+id/menu_red"
    android:orderInCategory="1"
    app:showAsAction="never"
    android:title="@string/red_string"/>
    <item
        android:id="@+id/menu_green"
        android:orderInCategory="2"
        app:showAsAction="never"
        android:title="@string/green_string"/>
</menu>

```

In the above XML, the *android:orderInCategory* property dictates the order in which the menu items will appear within the menu when it is displayed. The *app:showAsAction* property, on the other hand, controls the conditions under which the corresponding item appears as an item within the action bar itself. If set to *if Room*, for example, the item will appear in the action bar if there is enough room. [Figure 31-2](#) shows the effect of setting this property to *ifRoom* for both menu items:



Figure 31-2

This property should be used sparingly to avoid over cluttering the action bar. By default, a menu XML file is created by Android Studio when a new Android application project is created. This file is located in the *app -> res -> menu* project folder and contains a single menu item entitled “Settings”:

```

<menu 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"
      tools:context=".MainActivity">
    <item android:id="@+id/action_settings"
          android:title="@string/action_settings"
          android:orderInCategory="100"
          app:showAsAction="never" />

```

```
</menu>
```

This menu is already configured to be displayed when the user selects the overflow menu on the user interface when the app is running, so simply modify this one to meet your needs.

31.3 Displaying an Overflow Menu

An overflow menu is created by overriding the *onCreateOptionsMenu()* method of the corresponding activity and then inflating the menu's XML file. For example, the following code creates the menu contained within a menu XML file named *menu_menu_example*:

```
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
    getMenuInflater().inflate(R.menu.menu_menu_example, menu);  
    return true;  
}
```

As with the menu XML file, Android Studio will already have overridden this method in the main activity of a newly created Android application project. In the event that an overflow menu is not required in your activity, either remove or comment out this method.

31.4 Responding to Menu Item Selections

Once a menu has been implemented, the question arises as to how the application receives notification when the user makes menu item selections. All that an activity needs to do to receive menu selection notifications is to override the *onOptionsItemSelected()* method. Passed as an argument to this method is a reference to the selected menu item. The *getItemId()* method may then be called on the item to obtain the ID which may, in turn, be used to identify which item was selected. For example:

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
  
    switch (item.getItemId()) {  
        case R.id.menu_red:  
            // Red item was selected  
            return true;  
        case R.id.menu_green:  
            // Green item was selected  
            return true;  
    }  
}
```

```

        default:
            return super.onOptionsItemSelected(item);
    }
}

```

31.5 Creating Checkable Item Groups

In addition to configuring independent menu items, it is also possible to create groups of menu items. This is of particular use when creating checkable menu items whereby only one out of a number of choices can be selected at any one time. Menu items can be assigned to a group by wrapping them in the `<group>` tag. The group is declared as checkable using the `android:checkableBehavior` property, setting the value to either *single*, *all* or *none*. The following XML declares that two menu items make up a group wherein only one item may be selected at any given time:

```

<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/menu_red"
            android:title="@string/red_string"/>
        <item
            android:id="@+id/menu_green"
            android:title="@string/green_string"/>
    </group>
</menu>

```

When a menu group is configured to be checkable, a small circle appears next to the item in the menu as illustrated in [Figure 31-3](#). It is important to be aware that the setting and unsetting of this indicator does not take place automatically. It is, therefore, the responsibility of the application to check and uncheck the menu item.

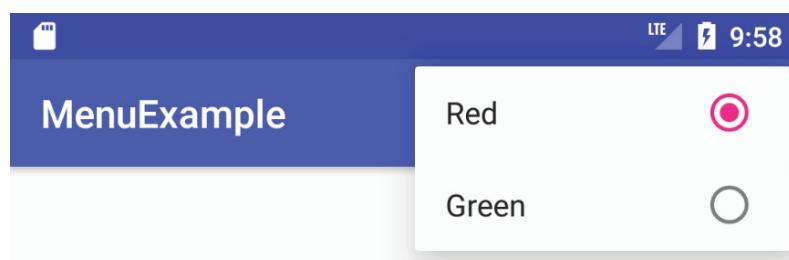


Figure 31-3

Continuing the color example used previously in this chapter, this would be

implemented as follows:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {

    switch (item.getItemId()) {
        case R.id.menu_red:
            if (item.isChecked()) item.setChecked(false);
            else item.setChecked(true);
            return true;
        case R.id.menu_green:
            if (item.isChecked()) item.setChecked(false);
            else item.setChecked(true);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

31.6 Menus and the Android Studio Menu Editor

Android Studio allows menus to be designed visually simply by loading the menu resource file into the Menu Editor tool, dragging and dropping menu elements from a palette and setting properties. This considerably eases the menu design process, though it is important to be aware that it is still necessary to write the code in the *onOptionsItemSelected()* method to implement the menu behavior.

To visually design a menu, locate the menu resource file and double-click on it to load it into the Menu Editor tool. [Figure 31-4](#), for example, shows the default menu resource file for a basic activity loaded into the Menu Editor:

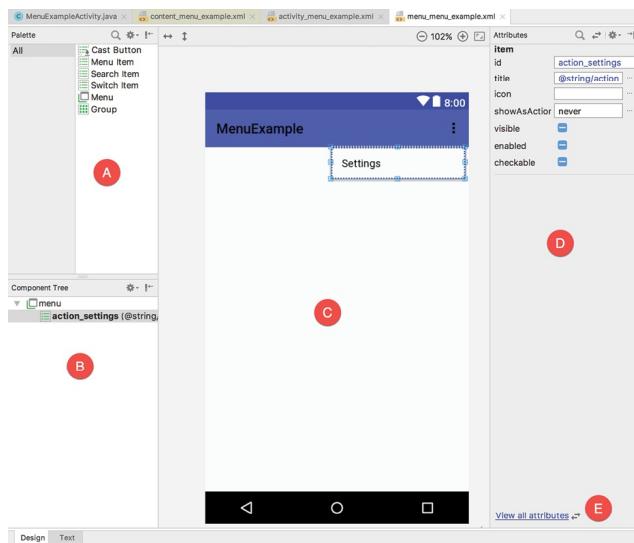


Figure 31-4

The palette (A) contains items that can be added to the menu contained in the design area (C). The Component Tree (B) is a useful tool for identifying the hierarchical structure of the menu. The Attributes panel (D) contains a subset of common attributes for the currently selected item. The view all attributes link (E) may be used to access the full list of attributes.

New elements may be added to the menu by dragging and dropping objects either onto the layout canvas or the Component Tree. When working with menus in the Layout Editor tool, it will sometimes be easier to drop the items onto the Component Tree since this provides greater control over where the item is placed within the tree. This is of particular use, for example, when adding items to a group.

Although the Menu Editor provides a visual approach to constructing menus, the underlying menu is still stored in XML format which may be viewed and edited manually by switching from Design to Text mode using the tab marked F in the above figure.

31.7 Creating the Example Project

To see the overflow menu in action, create a new project in Android Studio, entering *MenuExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of a basic

activity named *MenuExampleActivity* with a corresponding layout file named *activity_menu_example*.

When the project has been created, navigate to the *app -> res -> layout* folder in the Project tool window and double-click on the *content_menu_example.xml* file to load it into the Android Studio Menu Editor tool. Switch the tool to Design mode, select the ConstraintLayout from the Component Tree panel and enter *layoutView* into the ID field of the Attributes panel.

31.8 Designing the Menu

Within the Project tool window, locate the project's *app -> res -> menu -> menu_menu_example.xml* file and double-click on it to load it into the Layout Editor tool. Select and delete the default Settings menu item added by Android Studio so that the menu currently has no items.

From the palette, click and drag a menu *group* object onto the title bar of the layout canvas as highlighted in [Figure 31-5](#):



Figure 31-5

Although the group item has been added, it will be invisible within the layout. To verify the presence of the element, refer to the Component Tree panel where the group will be listed as a child of the menu:

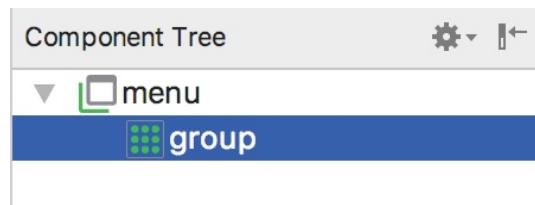


Figure 31-6

Select the *group* entry in the Component Tree and, referring to the Attributes panel, set the *checkableBehavior* property to *single* so that only one group menu item can be selected at any one time:

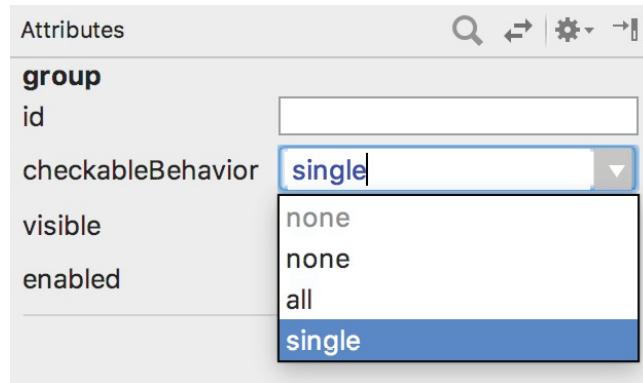


Figure 31-7

Next, drag four *Menu Item* elements from the palette and drop them onto the *group* element in the Component Tree. Select the first item and use the Attributes panel to change the title to “Red” and the ID to *menu_red*:

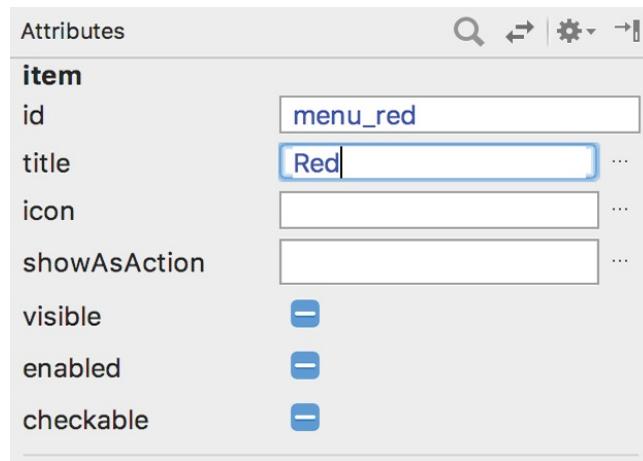


Figure 31-8

Repeat these steps for the remaining three menu items setting the titles to “Green”, “Yellow” and “Blue” with matching IDs of *menu_green*, *menu_yellow* and *menu_blue*. Use the warning buttons to the right of the menu items in the Component Tree panel to extract the strings to resources:



Figure 31-9

On completion of these steps, the menu layout should match that shown in [Figure 31-10](#) below:

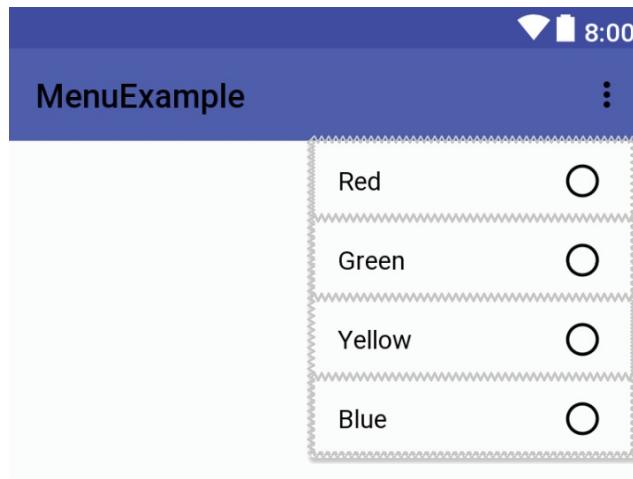


Figure 31-
10

Switch the Layout Editor tool to Text mode and review the XML representation of the menu which should match the following listing:

```
<menu 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"  
      tools:context="com.ebookfrenzy.menuexample.MenuExampleActivity">  
  
    <group android:checkableBehavior="single">  
        <item android:title="@string/red_string"  
              android:id="@+id/menu_red" />  
        <item android:title="@string/green_string"  
              android:id="@+id/menu_green" />  
        <item android:title="@string/yellow_string"  
              android:id="@+id/menu_yellow" />  
        <item android:title="@string/blue_string"  
              android:id="@+id/menu_blue" />  
    </group>  
</menu>
```

31.9 Modifying the `onOptionsItemSelected()` Method

When items are selected from the menu, the overridden `onOptionsItemSelected()` method of the application's activity will be called. The role of this method will be to identify which item was selected and change the background color of the layout view to the corresponding color.

Locate and double-click on the *app -> java -> com.ebookfrenzy.menuexample -> MenuExampleActivity* file and modify the method as follows:

```
package com.ebookfrenzy.menuexample;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.support.constraint.ConstraintLayout;

public class MenuExampleActivity extends AppCompatActivity {

    .
    .
    .

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {

        ConstraintLayout mainLayout =
                (ConstraintLayout) findViewById(R.id.layoutView);

        switch (item.getItemId()) {
            case R.id.menu_red:
                if (item.isChecked()) item.setChecked(false);
                else item.setChecked(true);
                mainLayout.setBackgroundColor(android.graphics.Color.RED);
                return true;
            case R.id.menu_green:
                if (item.isChecked()) item.setChecked(false);
                else item.setChecked(true);
                mainLayout.setBackgroundColor(android.graphics.Color.GREEN);
                return true;
            case R.id.menu_yellow:
                if (item.isChecked()) item.setChecked(false);
                else item.setChecked(true);
                mainLayout.setBackgroundColor(android.graphics.Color.YELLOW);
                return true;
            case R.id.menu_blue:
                if (item.isChecked()) item.setChecked(false);
                else item.setChecked(true);
                mainLayout.setBackgroundColor(android.graphics.Color.BLUE);
                return true;
        }
    }
}
```

```

        else item.setChecked(true);
        mainLayout.setBackgroundColor(android.graphics.Color.I
        return true;
    default:
        return super.onOptionsItemSelected(item);
    }
}
.
.
.
}

```

31.10 Testing the Application

Build and run the application on either an emulator or physical Android device. Using the overflow menu, select menu items and verify that the layout background color changes appropriately. Note that the currently selected color is displayed as the checked item in the menu.

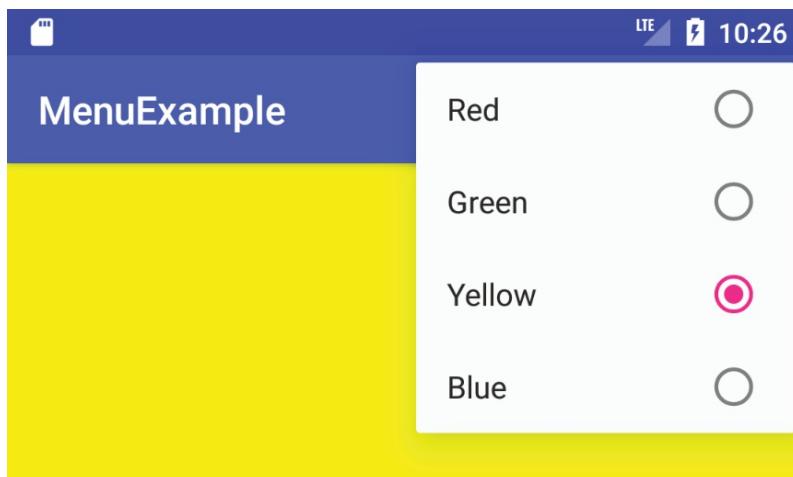


Figure 31-

11

31.11 Summary

The Android overflow menu is accessed from the far right of the actions toolbar at the top of the display of the running app. This menu provides a location for applications to provide additional options to the user.

The structure of the menu is most easily defined within an XML file and the application activity receives notifications of menu item selections by overriding and implementing the *onOptionsItemSelected()* method.

32. Animating User Interfaces with the Android Transitions Framework

The Android Transitions framework was introduced as part of the Android 4.4 KitKat release and is designed to make it easy for you, as an Android developer, to add animation effects to the views that make up the screens of your applications. As will be outlined in both this and subsequent chapters, animated effects such as making the views in a user interface gently fade in and out of sight and glide smoothly to new positions on the screen can be implemented with just a few simple lines of code when using the Transitions framework in Android Studio.

32.1 Introducing Android Transitions and Scenes

Transitions allow the changes made to the layout and appearance of the views in a user interface to be animated during application runtime. While there are a number of different ways to implement Transitions from within application code, perhaps the most powerful mechanism involves the use of *Scenes*. A scene represents either the entire layout of a user interface screen, or a subset of the layout (represented by a `ViewGroup`).

To implement transitions using this approach, scenes are defined that reflect the two different user interface states (these can be thought of as the “before” and “after” scenes). One scene, for example, might consist of a `EditText`, `Button` and `TextView` positioned near the top of the screen. The second scene might remove the `Button` view and move the remaining `EditText` and `TextView` objects to the bottom of the screen to make room for the introduction of a `MapView` instance. Using the transition framework, the changes between these two scenes can be animated so that the `Button` fades from view, the `EditText` and `TextView` slide to the new locations and the map gently fades into view.

Scenes can be created in code from `ViewGroups`, or implemented in layout resource files that are loaded into `Scene` instances at application runtime.

Transitions can also be implemented dynamically from within application code. Using this approach, scenes are created by referencing collections of user interface views in the form of `ViewGroups` with transitions then being

performed on those elements using the `TransitionManager` class, which provides a range of methods for triggering and managing the transitions between scenes.

Perhaps the simplest form of transition involves the use of the `beginDelayedTransition()` method of the `TransitionManager` class. When called and passed the `ViewGroup` representing a scene, any subsequent changes to any views within that scene (such as moving, resizing, adding or deleting views) will be animated by the Transition framework.

The actual animation is handled by the Transition framework via instances of the `Transition` class. Transition instances are responsible for detecting changes to the size, position and visibility of the views within a scene and animating those changes accordingly.

By default, transitions will be animated using a set of criteria defined by the `AutoTransition` class. Custom transitions can be created either via settings in XML transition files or directly within code. Multiple transitions can be combined together in a `TransitionSet` and configured to be performed either in parallel or sequentially.

32.2 Using Interpolators with Transitions

The Transitions framework makes extensive use of the Android Animation framework to implement animation effects. This fact is largely incidental when using transitions since most of this work happens behind the scenes, thereby shielding the developer from some of the complexities of the Animation framework. One area where some knowledge of the Animation framework is beneficial when using Transitions, however, involves the concept of interpolators.

Interpolators are a feature of the Android Animation framework that allow animations to be modified in a number of pre-defined ways. At present the Animation framework provides the following interpolators, all of which are available for use in customizing transitions:

- **AccelerateDecelerateInterpolator** – By default, animation is performed at a constant rate. The `AccelerateDecelerateInterpolator` can be used to cause the animation to begin slowly and then speed up in the middle before slowing down towards the end of the sequence.

- **AccelerateInterpolator** – As the name suggests, the AccelerateInterpolator begins the animation slowly and accelerates at a specified rate with no deceleration at the end.
- **AnticipateInterpolator** – The AnticipateInterpolator provides an effect similar to that of a sling shot. The animated view moves in the opposite direction to the configured animation for a short distance before being flung forward in the correct direction. The amount of backward force can be controlled through the specification of a tension value.
- **AnticipateOvershootInterpolator** – Combines the effect provided by the AnticipateInterpolator with the animated object overshooting and then returning to the destination position on the screen.
- **BounceInterpolator** – Causes the animated view to bounce on arrival at its destination position.
- **CycleInterpolator** – Configures the animation to be repeated a specified number of times.
- **DecelerateInterpolator** – The DecelerateInterpolator causes the animation to begin quickly and then decelerate by a specified factor as it nears the end.
- **LinearInterpolator** – Used to specify that the animation is to be performed at a constant rate.
- **OvershootInterpolator** – Causes the animated view to overshoot the specified destination position before returning. The overshoot can be configured by specifying a tension value.

As will be demonstrated in this and later chapters, interpolators can be specified both in code and XML files.

32.3 Working with Scene Transitions

Scenes can be represented by the content of an Android Studio XML layout file. The following XML, for example, could be used to represent a scene consisting of three button views within a RelativeLayout parent:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/RelativeLayout1"
```

```

    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:onClick="goToScene2"
        android:text="@string/one_string" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignParentTop="true"
        android:onClick="goToScene1"
        android:text="@string/two_string" />

    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/three_string" />

</RelativeLayout>

```

Assuming that the above layout resides in a file named *scene1_layout.xml* located in the *res/layout* folder of the project, the layout can be loaded into a scene using the *getSceneForLayout()* method of the Scene class. For example:

```
Scene scene1 = Scene.getSceneForLayout(rootContainer,
    R.layout.scene1_layout, this);
```

Note that the method call requires a reference to the root container. This is the view at the top of the view hierarchy in which the scene is to be displayed. To display a scene to the user without any transition animation, the *enter()* method is called on the scene instance:

```
scene1.enter();
```

Transitions between two scenes using the default AutoTransition class can be triggered using the go() method of the TransitionManager class:

```
TransitionManager.go(scene2);
```

Scene instances can be created easily in code by bundling the view elements into one or more ViewGroups and then creating a scene from those groups.

For example:

```
Scene scene1 = Scene(viewGroup1);
Scene scene2 = Scene(viewGroup2, viewGroup3);
```

32.4 Custom Transitions and TransitionSets in Code

The examples outlined so far in this chapter have used the default transition settings in which resizing, fading and motion are animated using pre-configured behavior. These can be modified by creating custom transitions which are then referenced during the transition process. Animations are categorized as either *change bounds* (relating to changes in the position and size of a view) and *fade* (relating to the visibility or otherwise of a view).

A single Transition can be created as follows:

```
Transition myChangeBounds = new ChangeBounds();
```

This new transition can then be used when performing a transition:

```
TransitionManager.go(scene2, myChangeBounds);
```

Multiple transitions may be bundled together into a TransitionSet instance. The following code, for example, creates a new TransitionSet object consisting of both change bounds and fade transition effects:

```
TransitionSet myTransition = new TransitionSet();
myTransition.addTransition(new ChangeBounds());
myTransition.addTransition(new Fade());
```

Transitions can be configured to target specific views (referenced by view ID). For example, the following code will configure the previous fade transition to target only the view with an ID that matches *myButton1*:

```
TransitionSet myTransition = new TransitionSet();
myTransition.addTransition(new ChangeBounds());
Transition fade = new Fade();
fade.addTarget(R.id.myButton1);
myTransition.addTransition(fade);
```

Additional aspects of the transition may also be customized, such as the duration of the animation. The following code specifies the duration over which the animation is to be performed:

```
Transition changeBounds = new ChangeBounds();  
changeBounds.setDuration(2000);
```

As with Transition instances, once a TransitionSet instance has been created, it can be used in a transition via the TransitionManager class. For example:

```
TransitionManager.go(scene1, myTransition);
```

32.5 Custom Transitions and TransitionSets in XML

While custom transitions can be implemented in code, it is often easier to do so via XML transition files using the <fade> and <changeBounds> tags together with some additional options. The following XML includes a single changeBounds transition:

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

As with the code based approach to working with transitions, each transition entry in a resource file may be customized. The XML below, for example, configures a duration for a change bounds transition:

```
<changeBounds android:duration="5000" >
```

Multiple transitions may be bundled together using the <transitionSet> element:

```
<?xml version="1.0" encoding="utf-8"?>  
<transitionSet  
    xmlns:android="http://schemas.android.com/apk/res/android" >  
  
<fade  
    android:duration="2000"  
    android:fadingMode="fade_out" />  
  
<changeBounds  
    android:duration="5000" >  
  
    <targets>  
        <target android:targetId="@+id/button2" />  
    </targets>  
  
</changeBounds>  
  
<fade  
    android:duration="2000"  
    android:fadingMode="fade_in" />  
</transitionSet>
```

Transitions contained within an XML resource file should be stored in the `res/transition` folder of the project in which they are being used and must be inflated before being referenced in the code of an application. The following code, for example, inflates the transition resources contained within a file named `transition.xml` and assigns the results to a reference named `myTransition`:

```
Transition myTransition = TransitionInflater.from(this)
    .inflateTransition(R.transition.transition);
```

Once inflated, the new transition can be referenced in the usual way:

```
TransitionManager.go(scene1, myTransition);
```

By default, transition effects within a `TransitionSet` are performed in parallel. To instruct the Transition framework to perform the animations sequentially, add the appropriate `android:transitionOrdering` property to the `transitionSet` element of the resource file:

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

<transitionSet
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:transitionOrdering="sequential">

    <fade
        android:duration="2000"
        android:fadingMode="fade_out" />

    <changeBounds
        android:duration="5000" >
    </changeBounds>
</transitionSet>
```

Change the value from “sequential” to “together” to indicate that the animation sequences are to be performed in parallel.

32.6 Working with Interpolators

As previously discussed, interpolators can be used to modify the behavior of a transition in a variety of ways and may be specified either in code or via the settings within a transition XML resource file.

When working in code, new interpolator instances can be created by calling the constructor method of the required interpolator class and, where appropriate, passing through values to further modify the interpolator

behavior:

- AccelerateDecelerateInterpolator()
- AccelerateInterpolator(float factor)
- AnticipateInterpolator(float tension)
- AnticipateOvershootInterpolator(float tension)
- BounceInterpolator()
- CycleInterpolator(float cycles)
- DecelerateInterpolator(float factor)
- LinearInterpolator()
- OvershootInterpolator(float tension)

Once created, an interpolator instance can be attached to a transition using the `setInterpolator()` method of the `Transition` class. The following code, for example, adds a bounce interpolator to a change bounds transition:

```
Transition changeBounds = new ChangeBounds();  
changeBounds.setInterpolator(new BounceInterpolator());
```

Similarly, the following code adds an accelerate interpolator to the same transition, specifying an acceleration factor of 1.2:

```
changeBounds.setInterpolator(new AccelerateInterpolator(1.2f));
```

In the case of XML based transition resources, a default interpolator is declared using the following syntax:

```
android:interpolator="@android:anim/<interpolator_element>"
```

In the above syntax, `<interpolator_element>` must be replaced by the resource ID of the corresponding interpolator selected from the following list:

- accelerate_decelerate_interpolator
- accelerate_interpolator
- anticipate_interpolator
- anticipate_overshoot_interpolator
- bounce_interpolator
- cycle_interpolator
- decelerate_interpolator

- linear_interpolator
- overshoot_interpolator

The following XML fragment, for example, adds a bounce interpolator to a change bounds transition contained within a transition set:

```
<?xml version="1.0" encoding="utf-8"?>
<transitionSet
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:transitionOrdering="sequential">

    <changeBounds
        android:interpolator="@android:anim/bounce_interpolator"
        android:duration="2000" />

    <fade
        android:duration="1000"
        android:fadingMode="fade_in" />
</transitionSet>
```

This approach to adding interpolators to transitions within XML resources works well when the default behavior of the interpolator is required. The task becomes a little more complex when the default behavior of an interpolator needs to be changed. Take, for example, the cycle interpolator. The purpose of this interpolator is to make an animation or transition repeat a specified number of times. In the absence of a *cycles* attribute setting, the cycle interpolator will perform only one cycle. Unfortunately, there is no way to directly specify the number of cycles (or any other interpolator attribute for that matter) when adding an interpolator using the above technique. Instead, a custom interpolator must be created and then referenced within the transition file.

32.7 Creating a Custom Interpolator

A custom interpolator must be declared in a separate XML file and stored within the *res/anim* folder of the project. The name of the XML file will be used by the Android system as the resource ID for the custom interpolator.

Within the custom interpolator XML resource file, the syntax should read as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<interpolatorElement xmlns:android="http://schemas.android.com/apk/re:
```

In the above syntax, *interpolatorElement* must be replaced with the element name of the required interpolator selected from the following list:

- accelerateDecelerateInterpolator
- accelerateInterpolator
- anticipateInterpolator
- anticipateOvershootInterpolator
- bounceInterpolator
- cycleInterpolator
- decelerateInterpolator
- linearInterpolator
- overshootInterpolator

The *attribute* keyword is replaced by the name attribute of the interpolator for which the value is to be changed (for example *tension* to change the tension attribute of an overshoot interpolator). Finally, *value* represents the value to be assigned to the specified attribute. The following XML, for example, contains a custom cycle interpolator configured to cycle 7 times:

```
<?xml version="1.0" encoding="utf-8"?>
<cycleInterpolator xmlns:android="http://schemas.android.com/apk/res/android"
    android:cycles="7" android:pivotX="50%" android:pivotY="50%">
```

Assuming that the above XML was stored in a resource file named *my_cycle.xml* stored in the *res/anim* project folder, the custom interpolator could be added to a transition resource file using the following XML syntax:

```
<changeBounds
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="5000"
    android:interpolator="@anim/my_cycle" >
```

32.8 Using the beginDelayedTransition Method

Perhaps the simplest form of Transition based user interface animation involves the use of the *beginDelayedTransition()* method of the *TransitionManager* class. This method is passed a reference to the root view of the viewgroup representing the scene for which animation is required. Subsequent changes to the views within that sub view will then be animated using the default transition settings:

```
myLayout = (ViewGroup) findViewById(R.id.myLayout);  
TransitionManager.beginDelayedTransition(myLayout);  
// Make changes to the scene here
```

If behavior other than the default animation behavior is required, simply pass a suitably configured Transition or TransitionSet instance through to the method call:

```
TransitionManager.beginDelayedTransition(myLayout, myTransition);
```

32.9 Summary

The Android 4.4 KitKat SDK release introduced the Transition Framework, the purpose of which is to simplify the task of adding animation to the views that make up the user interface of an Android application. With some simple configuration and a few lines of code, animation effects such as movement, visibility and resizing of views can be animated by making use of the Transition framework. A number of different approaches to implementing transitions are available involving a combination of Java code and XML resource files. The animation effects of transitions may also be enhanced through the use of a range of interpolators.

Having covered some of the theory of Transitions in Android, the next two chapters will put this theory into practice by working through some example Android Studio based transition implementations.

33. An Android Transition Tutorial using beginDelayedTransition

The previous chapter, entitled “[Animating User Interfaces with the Android Transitions Framework](#)”, provided an introduction to the animation of user interfaces using the Android Transitions framework. This chapter uses a tutorial based approach to demonstrate Android transitions in action using the *beginDelayedTransition()* method of the *TransitionManager* class.

The next chapter will create a more complex example that uses layout files and transition resource files to animate the transition from one scene to another within an application.

33.1 Creating the Android Studio TransitionDemo Project

Create a new project in Android Studio, entering *TransitionDemo* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 19: Android 4.4 (KitKat). Continue to proceed through the screens, requesting the creation of an Empty Activity named *TransitionDemoActivity* with a layout resource file named *activity_transition_demo*.

33.2 Preparing the Project Files

The first example transition animation will be implemented through the use of the *beginDelayedTransition()* method of the *TransitionManager* class. If Android Studio does not automatically load the file, locate and double-click on the *app -> res -> layout -> activity_transition_demo.xml* file in the Project tool window panel to load it into the Layout Editor tool.

Switch the Layout Editor to Design mode, delete the “Hello World!” *TextView*, drag a *Button* from the Widget section of the Layout Editor palette and position it in the top left-hand corner of the device screen layout. Once positioned, select the button and use the Attributes tool window to specify an ID value of *myButton*.

Select the ConstraintLayout entry in the Component Tree tool window and use the Attributes window to set the ID to *myLayout*.

33.3 Implementing beginDelayedTransition Animation

The objective for the initial phase of this tutorial is to implement a touch handler so that when the user taps on the layout view the button view moves to the lower right-hand corner of the screen.

Open the *TransitionDemoActivity.java* file (located in the Project tool window under *app -> java -> com.ebookfrenzy.transitiondemo*) and modify the *onCreate()* method to implement the *onTouch* handler:

```
package com.ebookfrenzy.transitiondemo;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.constraint.ConstraintLayout;
import android.support.constraint.ConstraintSet;
import android.view.MotionEvent;
import android.widget.Button;
import android.view.View;

public class TransitionDemoActivity extends AppCompatActivity {

    ConstraintLayout myLayout;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_transition_demo);

        myLayout = (ConstraintLayout) findViewById(R.id.myLayout);

        myLayout.setOnTouchListener(
            new ConstraintLayout.OnTouchListener() {
                public boolean onTouch(View v,
                                      MotionEvent m) {
                    handleTouch();
                    return true;
                }
            });
    }
}
```

```
}
```

The above code simply sets up a touch listener on the ConstraintLayout container and configures it to call a method named *handleTouch()* when a touch is detected. The next task, therefore, is to implement the *handleTouch()* method as follows:

```
public void handleTouch() {  
    Button button = (Button) findViewById(R.id.myButton);  
  
    button.setMinimumWidth(500);  
    button.setMinimumHeight(350);  
  
    ConstraintSet set = new ConstraintSet();  
  
    set.connect(R.id.myButton, ConstraintSet.BOTTOM,  
               ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0);  
  
    set.connect(R.id.myButton, ConstraintSet.RIGHT,  
               ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);  
  
    set.constrainWidth(R.id.myButton, ConstraintSet.WRAP_CONTENT);  
  
    set.applyTo(myLayout);  
}
```

This method obtains a reference to the button view in the user interface layout and sets new minimum height and width attributes so that the button increases in size.

A ConstraintSet object is then created and configured with constraints that will position the button in the lower right-hand corner of the parent layout. This constraint set is then applied to the layout.

Test the code so far by compiling and running the application. Once launched, touch the background (not the button) and note that the button moves and resizes as illustrated in [Figure 33-1](#):

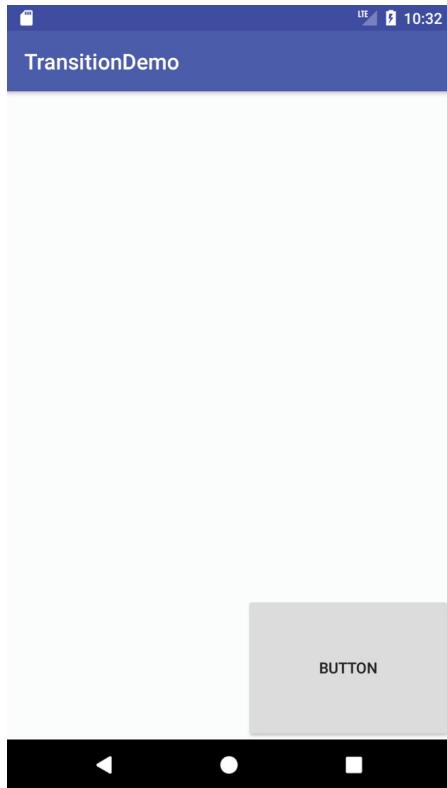


Figure 33-1

Although the layout changes took effect, they did so instantly and without any form of animation. This is where the call to the *beginDelayedTransition()* method of the *TransitionManager* class comes in. All that is needed to add animation to this layout change is the addition of a single line of code before the layout changes are implemented. Remaining within the *TransitionDemoActivity.java* file, modify the code as follows:

```
import android.transition.TransitionManager;  
  
public void handleTouch() {  
    Button button = (Button) findViewById(R.id.myButton);  
  
    TransitionManager.beginDelayedTransition(myLayout);  
  
    ConstraintSet set = new ConstraintSet();  
  
    button.setMinimumWidth(500);  
    button.setMinimumHeight(350);
```

```

        set.connect(R.id.myButton, ConstraintSet.BOTTOM,
                    ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0);

        set.connect(R.id.myButton, ConstraintSet.RIGHT,
                    ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);

        set.constrainWidth(R.id.myButton,
ConstraintSet.WRAP_CONTENT);

        set.applyTo(myLayout);
    }
}

.
.
.
}

```

Compile and run the application once again and note that the transition is now animated.

33.4 Customizing the Transition

The final task in this example is to modify the `changeBounds` transition so that it is performed over a longer duration and incorporates a bounce effect when the view reaches its new screen location. This involves the creation of a `Transition` instance with appropriate duration interpolator settings which is, in turn, passed through as an argument to the `beginDelayedTransition()` method:

```

.
.

import android.transition.ChangeBounds;
import android.transition.Transition;
import android.view.animation.BounceInterpolator;
.

.

public void handleTouch() {
    Button button = (Button) findViewById(R.id.myButton);

    Transition changeBounds = new ChangeBounds();
    changeBounds.setDuration(3000);
    changeBounds.setInterpolator(new BounceInterpolator());

```

```

TransitionManager.beginDelayedTransition(myLayout,
    changeBounds) ;

TransitionManager.beginDelayedTransition(myLayout) ;

ConstraintSet set = new ConstraintSet() ;

button.setMinimumWidth(500) ;
button.setMinimumHeight(350) ;

set.connect(R.id.myButton, ConstraintSet.BOTTOM,
    ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0) ;

set.connect(R.id.myButton, ConstraintSet.RIGHT,
    ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0) ;

set.constrainWidth(R.id.myButton, ConstraintSet.WRAP_CONTENT) ;

set.applyTo(myLayout) ;
}

```

When the application is now executed, the animation will slow to match the new duration setting and the button will bounce on arrival at the bottom right-hand corner of the display.

33.5 Summary

The most basic form of transition animation involves the use of the *beginDelayedTransition()* method of the TransitionManager class. Once called, any changes in size and position of the views in the next user interface rendering frame, and within a defined view group, will be animated using the specified transitions. This chapter has worked through a simple Android Studio example that demonstrates the use of this approach to implementing transitions.

34. Implementing Android Scene Transitions – A Tutorial

This chapter will build on the theory outlined in the chapter entitled [“Animating User Interfaces with the Android Transitions Framework”](#) by working through the creation of a project designed to demonstrate transitioning from one scene to another using the Android Transition framework.

34.1 An Overview of the Scene Transition Project

The application created in this chapter will consist of two scenes, each represented by an XML layout resource file. A transition will then be used to animate the changes from one scene to another. The first scene will consist of three button views. The second scene will contain two of the buttons from the first scene positioned at different locations on the screen. The third button will be absent from the second scene. Once the transition has been implemented, movement of the first two buttons will be animated with a bounce effect. The third button will gently fade into view as the application transitions back to the first scene from the second.

34.2 Creating the Android Studio SceneTransitions Project

Create a new project in Android Studio, entering *SceneTransitions* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 19: Android 4.4 (KitKat). Continue to proceed through the screens, requesting the creation of an Empty Activity named *SceneTransitionsActivity* with a corresponding layout file named *activity_scene_transitions*.

34.3 Identifying and Preparing the Root Container

When working with transitions it is important to identify the root container for the scenes. This is essentially the parent layout container into which the scenes are going to be displayed. When the project was created, Android

Studio created a layout resource file in the *app -> res -> layout* folder named *activity_scene_transitions.xml* and containing a single layout container and *TextView*. When the application is launched, this is the first layout that will be displayed to the user on the device screen and for the purposes of this example, a *RelativeLayout* manager within this layout will act as the root container for the two scenes.

Begin by locating the *activity_scene_transitions.xml* layout resource file and loading it into the Android Studio Layout Editor tool. Switch to Text mode and replace the existing XML with the following to implement the *RelativeLayout* with an ID of *rootContainer*:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/and:
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/rootContainer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.ebookfrenzy.scenetransitions.SceneTransitionsActiv:
</RelativeLayout>
```

34.4 Designing the First Scene

The first scene is going to consist of a layout containing three button views. Create this layout resource file by right-clicking on the *app -> res -> layout* entry in the Project tool window and selecting the *New -> Layout resource file...* menu option. In the resulting dialog, name the file *scene1_layout* and enter *android.support.constraint.ConstraintLayout* as the root element before clicking on *OK*.

When the newly created layout file has loaded into the Layout Editor tool, check that Autoconnect mode is enabled, drag a Button view from the Widgets section of the palette onto the layout canvas and position it in the top left-hand corner of the layout view so that the dashed margin guidelines appear as illustrated in [Figure 34-1](#). Drop the Button view at this position, select it and change the text value in the Attributes tool window to “One”.

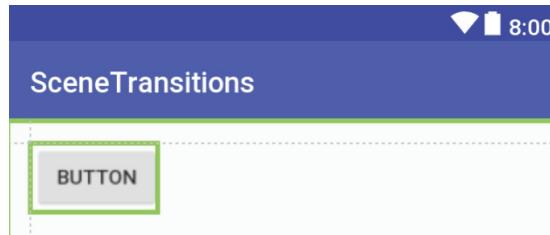


Figure 34-1

Drag a second Button view from the palette and position it in the top right-hand corner of the layout view so that the margin guidelines appear. Repeating the steps for the first button, assign text that reads “Two” to the button.

Drag a third Button view and position it so that it is centered both horizontally and vertically within the layout, this time configuring the button text to read “Three”.

Click on the warning button in the top right-hand corner of the Layout Editor and work through the list of I18N warnings, extracting the three button strings to resource values.

On completion of the above steps, the layout for the first scene should resemble that shown in [Figure 34-2](#):

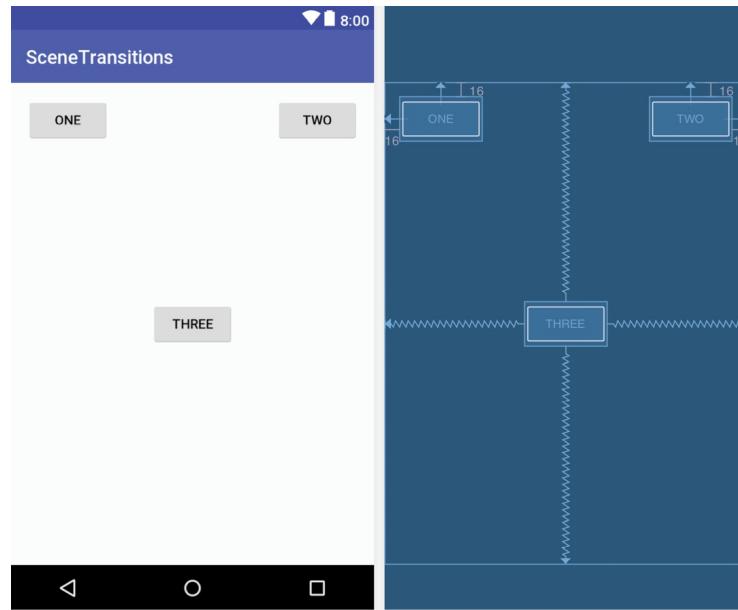


Figure 34-2

Select the “One” button and, using the Attributes tool window, configure the onClick attribute to call a method named `goToScene2`. Repeat this steps for the “Two” button, this time entering a method named `goToScene1` into the

onClick field.

34.5 Designing the Second Scene

The second scene is simply a modified version of the first scene. The first and second buttons will still be present but will be located in the bottom right and left-hand corners of the layout respectively. The third button, on the other hand, will no longer be present in the second scene.

For the purposes of avoiding duplicated effort, the layout file for the second scene will be created by copying and modifying the *scene1_layout.xml* file. Within the Project tool window, locate the *app -> res -> layout -> scene1_layout.xml* file, right-click on it and select the *Copy* menu option. Right-click on the *layout* folder, this time selecting the *Paste* menu option and change the name of the file to *scene2_layout.xml* when prompted to do so.

Double-click on the new *scene2_layout.xml* file to load it into the Layout Editor tool and switch to Design mode if necessary. Use the *Clear all Constraints* button located in the toolbar to remove the current constraints from the layout.

Select and delete the “Three” button and move the first and second buttons to the bottom right and bottom left locations as illustrated in [Figure 34-3](#):

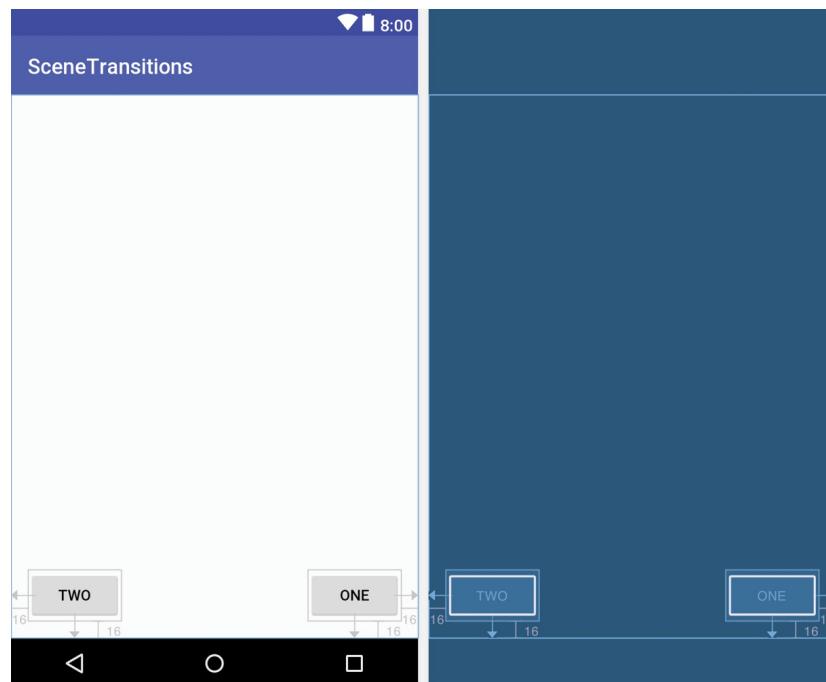


Figure 34-3

34.6 Entering the First Scene

If the application were to be run now, only the blank layout represented by the *activity_scene_transitions.xml* file would be displayed. Some code must, therefore, be added to the *onCreate()* method located in the *SceneTransitionsActivity.java* file so that the first scene is presented when the activity is created. This can be achieved as follows:

```
package com.ebookfrenzy.scenetransitions;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.transition.Scene;
import android.transition.Transition;
import android.transition.TransitionManager;
import android.view.ViewGroup;
import android.view.View;

public class SceneTransitionsActivity extends AppCompatActivity {

    ViewGroup rootContainer;
    Scene scene1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_scene_transitions);

        rootContainer =
            (ViewGroup) findViewById(R.id.rootContainer);

        scene1 = Scene.getSceneForLayout(rootContainer,
            R.layout.scene1_layout, this);

        scene1.enter();
    }
}
```

The code added to the activity class declares some variables in which to store references to the root container and first scene and obtains a reference to the root container view. The *getSceneForLayout()* method of the *Scene* class is then used to create a scene from the layout contained in the *scene1_layout.xml* file to convert that layout into a scene. The scene is then

entered via the `enter()` method call so that it is displayed to the user.

Compile and run the application at this point and verify that scene 1 is displayed after the application has launched.

34.7 Loading Scene 2

Before implementing the transition between the first and second scene it is first necessary to add some code to load the layout from the `scene2_layout.xml` file into a Scene instance. Remaining in the `SceneTransitionsActivity.java` file, therefore, add this code as follows:

```
public class SceneTransitionsActivity extends AppCompatActivity {

    ViewGroup rootContainer;
    Scene scene1;
    Scene scene2;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_scene_transitions);

        rootContainer =
            (ViewGroup) findViewById(R.id.rootContainer);

        scene1 = Scene.getSceneForLayout(rootContainer,
            R.layout.scene1_layout, this);

        scene2 = Scene.getSceneForLayout(rootContainer,
            R.layout.scene2_layout, this);

        scene1.enter();
    }
}
```

34.8 Implementing the Transitions

The first and second buttons have been configured to call methods named `goToScene2` and `goToScene1` respectively when selected. As the method names suggest, it is the responsibility of these methods to trigger the transitions between the two scenes. Add these two methods within the

SceneTransitionsActivity.java file so that they read as follows:

```
public void goToScene2 (View view)
{
    TransitionManager.go (scene2);
}

public void goToScene1 (View view)
{
    TransitionManager.go (scene1);
}
```

Run the application and note that selecting the first two buttons causes the layout to switch between the two scenes. Since we have yet to configure any transitions, these layout changes are not yet animated.

34.9 Adding the Transition File

All of the transition effects for this project will be implemented within a single transition XML resource file. As outlined in the chapter entitled [“Animating User Interfaces with the Android Transitions Framework”](#), transition resource files must be placed in the *app -> res -> transition* folder of the project. Begin, therefore, by right-clicking on the *res* folder in the Project tool window and selecting the *New -> Directory* menu option. In the resulting dialog, name the new folder *transition* and click on the *OK* button. Right-click on the new transition folder, this time selecting the *New -> File* option and name the new file *transition.xml*.

With the newly created *transition.xml* file selected and loaded into the editing panel, add the following XML content to add a transition set that enables the change bounds transition animation with a duration attribute setting:

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

<transitionSet
    xmlns:android="http://schemas.android.com/apk/res/android">

    <changeBounds
        android:duration="2000">
    </changeBounds>

</transitionSet>
```

34.10 Loading and Using the Transition Set

Although a transition resource file has been created and populated with a change bounds transition, this will have no effect until some code is added to load the transitions into a TransitionManager instance and reference it in the scene changes. The changes to achieve this are as follows:

```
package com.ebookfrenzy.scenetransitions;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.transition.Scene;
import android.transition.Transition;
import android.transition.TransitionInflater;
import android.transition.TransitionManager;
import android.view.ViewGroup;
import android.view.View;

public class SceneTransitionsActivity extends AppCompatActivity {

    ViewGroup rootContainer;
    Scene scene1;
    Scene scene2;
    Transition transitionMgr;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_scene_transitions);

        rootContainer =
            (ViewGroup) findViewById(R.id.rootContainer);

        transitionMgr = TransitionInflater.from(this)
            .inflateTransition(R.transition.transition);

        scene1 = Scene.getSceneForLayout(rootContainer,
            R.layout.scene1_layout, this);

        scene2 = Scene.getSceneForLayout(rootContainer,
            R.layout.scene2_layout, this);

        scene1.enter();
    }
}
```

```

public void goToScene2 (View view)
{
    TransitionManager.go(scene2, transitionMgr);
}

public void goToScene1 (View view)
{
    TransitionManager.go(scene1, transitionMgr);
}

.
.
}

```

When the application is now run the two buttons will gently glide to their new positions during the transition.

34.1 Configuring Additional Transitions

With the transition file integrated into the project, any number of additional transitions may be added to the file without the need to make any further changes to the Java source code of the activity. Take, for example, the following changes to the *transition.xml* file to add a bounce interpolator to the change bounds transition, introduce a fade-in transition targeted at the third button and to change the transitions such that they are performed sequentially:

```

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

<transitionSet
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:transitionOrdering="sequential" >

    <fade
        android:duration="2000"
        android:fadingMode="fade_in">

        <targets>
            <target android:targetId="@+id/button3" />
        </targets>
    </fade>

    <changeBounds
        android:duration="2000"
        android:interpolatorName="bounce">

```

```
    android:interpolator="@android:anim/bounce_interpolator">
  </changeBounds>
</transitionSet>
```

Buttons one and two will now bounce on arriving at the end destinations and button three will gently fade back into view when transitioning to scene 1 from scene 2.

Take some time to experiment with different transitions and interpolators by making changes to the *transition.xml* file and re-running the application.

34.1 Summary

Scene based transitions provide a flexible approach to animating user interface layout changes within an Android application. This chapter has demonstrated the steps involved in animating the transition between the scenes represented by two layout resource files. In addition, the example also used a transition XML resource file to configure the transition animation effects between the two scenes.

35. Working with the Floating Action Button and Snackbar

One of the objectives of this chapter is to provide an overview of the concepts of material design. Originally introduced as part of Android 5.0, material design is a set of design guidelines that dictate how the Android user interface, and that of the apps running on Android, appear and behave.

As part of the implementation of the material design concepts, Google also introduced the Android Design Support Library. This library contains a number of different components that allow many of the key features of material design to be built into Android applications. Two of these components, the floating action button and Snackbar, will also be covered in this chapter prior to introducing many of the other components in subsequent chapters.

35.1 The Material Design

The overall appearance of the Android environment is defined by the principles of material design. Material design was created by the Android team at Google and dictates that the elements that make up the user interface of Android and the apps that run on it appear and behave in a certain way in terms of behavior, shadowing, animation and style. One of the tenets of the material design is that the elements of a user interface appear to have physical depth and a sense that items are constructed in layers of physical material. A button, for example, appears to be raised above the surface of the layout in which it resides through the use of shadowing effects. Pressing the button causes the button to flex and lift as though made of a thin material that ripples when released.

Material design also dictates the layout and behavior of many standard user interface elements. A key example is the way in which the app bar located at the top of the screen should appear and the way in which it should behave in relation to scrolling activities taking place within the main content of the activity.

In fact, material design covers a wide range of areas from recommended color styles to the way in which objects are animated. A full description of the

material design concepts and guidelines can be found online at the following link and is recommended reading for all Android developers:

<https://www.google.com/design/spec/material-design/introduction.html>

35.2 The Design Library

Many of the building blocks needed to implement Android applications that adopt the principles of material design are contained within the Android Design Support Library. This library contains a collection of user interface components that can be included in Android applications to implement much of the look, feel and behavior of material design. Two of the components from this library, the floating action button and Snackbar, will be covered in this chapter, while others will be introduced in later chapters.

35.3 The Floating Action Button (FAB)

The floating action button is a button which appears to float above the surface of the user interface of an app and is generally used to promote the most common action within a user interface screen. A floating action button might, for example, be placed on a screen to allow the user to add an entry to a list of contacts or to send an email from within the app. [Figure 35-1](#), for example, highlights the floating action button that allows the user to add a new contact within the standard Android Contacts app:

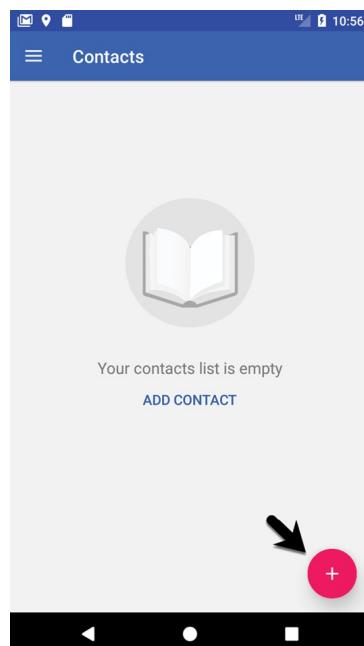


Figure 35-1

To conform with the material design guidelines, there are a number of rules that should be followed when using floating action buttons. Floating action buttons must be circular and can be either 56 x 56dp (Default) or 40 x 40dp (Mini) in size. The button should be positioned a minimum of 16dp from the edge of the screen on phones and 24dp on desktops and tablet devices. Regardless of the size, the button must contain an interior icon that is 24x24dp in size and it is recommended that each user interface screen have only one floating action button.

Floating action buttons can be animated or designed to morph into other items when touched. A floating action button could, for example, rotate when tapped or morph into another element such as a toolbar or panel listing related actions.

35.4 The Snackbar

The Snackbar component provides a way to present the user with information in the form of a panel that appears at the bottom of the screen as shown in [Figure 35-2](#). Snackbar instances contain a brief text message and an optional action button which will perform a task when tapped by the user. Once displayed, a Snackbar will either timeout automatically or can be removed manually by the user via a swiping action. During the appearance of the Snackbar the app will continue to function and respond to user interactions in the normal manner.

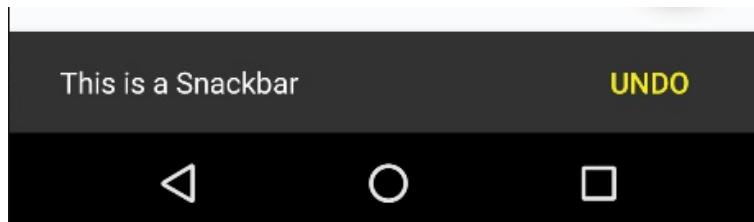


Figure 35-2

In the remainder of this chapter an example application will be created that makes use of the basic features of the floating action button and Snackbar to add entries to a list of items.

35.5 Creating the Example Project

Create a new project in Android Studio, entering *FabExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich).

Although it is possible to manually add a floating action button to an activity, it is much easier to use the Basic Activity template which includes a floating action button by default. Continue to proceed through the screens, therefore, requesting the creation of a Basic Activity named *FabExampleActivity* with corresponding layout and menu files named *activity_fab_example* and *menu_fab_example* respectively.

Click on the *Finish* button to initiate the project creation process.

35.6 Reviewing the Project

Since the Basic Activity template was selected, the activity contains two layout files. The *activity_fab_example.xml* file consists of a CoordinatorLayout manager containing entries for an app bar, a toolbar and a floating action button.

The *content_fab_example.xml* file represents the layout of the content area of the activity and contains a ConstraintLayout instance and a TextView. This file is embedded into the *activity_fab_example.xml* file via the following include directive:

```
<include layout="@layout/content_fab_example" />
```

The floating action button element within the *activity_fab_example.xml* file reads as follows:

```
<android.support.design.widget.FloatingActionButton  
    android:id="@+id/fab"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="bottom|end"  
    android:layout_margin="@dimen/fab_margin"  
    android:src="@android:drawable/ic_dialog_email" />
```

This declares that the button is to appear in the bottom right-hand corner of the screen with margins represented by the *fab_margin* identifier in the *values/dimens.xml* file (which in this case is set to 16dp). The XML further declares that the interior icon for the button is to take the form of the standard drawable built-in email icon.

The blank template has also configured the floating action button to display a Snackbar instance when tapped by the user. The code to implement this can

be found in the `onCreate()` method of the `FabExampleActivity.java` file and reads as follows:

```
FloatingActionButton fab =  
    (FloatingActionButton) findViewById(R.id.fab);  
  
fab.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View view) {  
        Snackbar.make(view, "Replace with your own action",  
            Snackbar.LENGTH_LONG)  
            .setAction("Action", null).show();  
    }  
});
```

The code obtains a reference to the floating action button via the button's ID and adds to it an `onClickListener` handler to be called when the button is tapped. This method simply displays a `Snackbar` instance configured with a message but no actions.

Finally, open the module level `build.gradle` file (*Gradle Scripts -> build.gradle (Module: App)*) and note that the Android design support library has been added as a dependency:

```
implementation 'com.android.support:design:26.1.0'
```

When the project is compiled and run the floating action button will appear at the bottom of the screen as shown in [Figure 35-3](#):

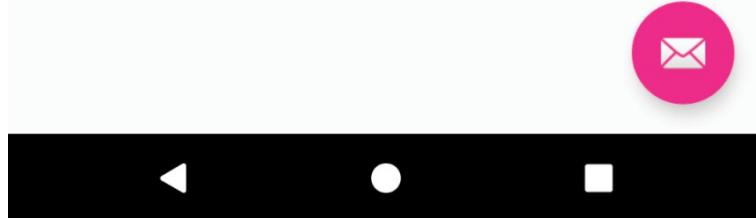


Figure 35-3

Tapping the floating action button will trigger the `onClickListener` handler method causing the `Snackbar` to appear at the bottom of the screen:

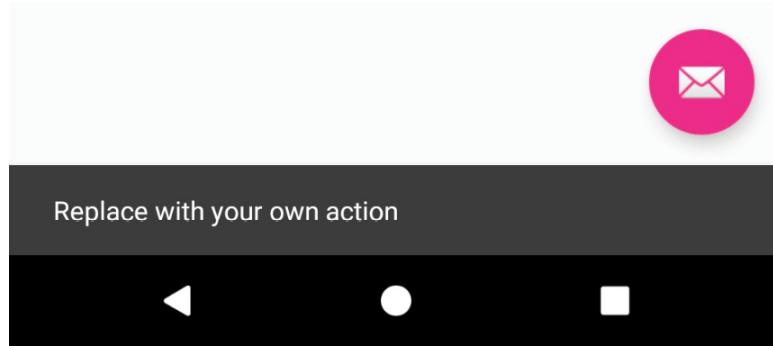


Figure 35-4

When the Snackbar appears on a narrower device (as is the case in [Figure 35-4](#) above) note that the floating action button is moved up to make room for the Snackbar to appear. This is handled for us automatically by the CoordinatorLayout container in the *activity_fab_example.xml* layout resource file.

35.7 Changing the Floating Action Button

Since the objective of this example is to configure the floating action button to add entries to a list, the email icon currently displayed on the button needs to be changed to something more indicative of the action being performed. The icon that will be used for the button is named *ic_add_entry.png* and can be found in the *project_icons* folder of the sample code download available from the following URL:

<http://www.ebookfrenzy.com/retail/androidstudio30/index.php>

Locate this image in the file system navigator for your operating system and copy the image file. Right-click on the *app -> res -> drawable* entry in the Project tool window and select Paste from the menu to add the file to the folder:

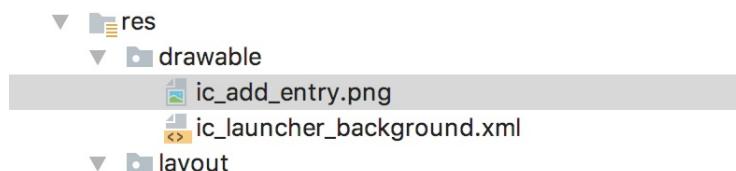


Figure 35-5

Next, edit the *activity_fab_example.xml* file and change the image source for the icon from `@android:drawable/ic_dialog_email` to `@drawable/ic_add_entry` as follows:

```
<android.support.design.widget.FloatingActionButton
```

```

    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/fab_margin"
    android:src="@drawable/ic_add_entry" />

```

Within the layout preview, the interior icon for the button will have changed to a plus sign.

The background color of the floating action button is defined by the *accentColor* property of the prevailing theme used by the application. The color assigned to this value is declared in the *colors.xml* file located under *app -> res -> values* in the Project tool window. Instead of editing this XML file directly a better approach is to use the Android Studio Theme Editor.

Select the *Tools -> Android -> Theme Editor* menu option to display the Theme Editor as illustrated in [Figure 35-6](#):

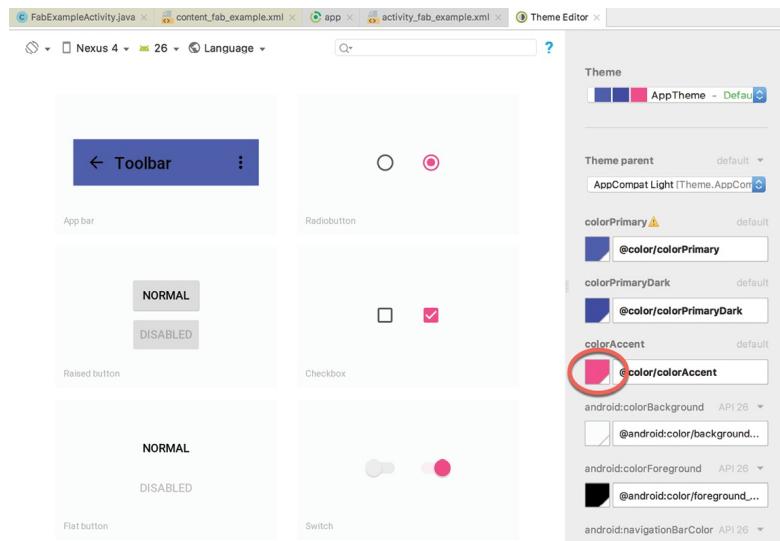


Figure 35-6

Click on the color swatch for the *colorAccent* setting (highlighted in the figure above) to display the color resource dialog. Within the color resource dialog, enter *holo_orange_light* into the search field and select the color from the list:

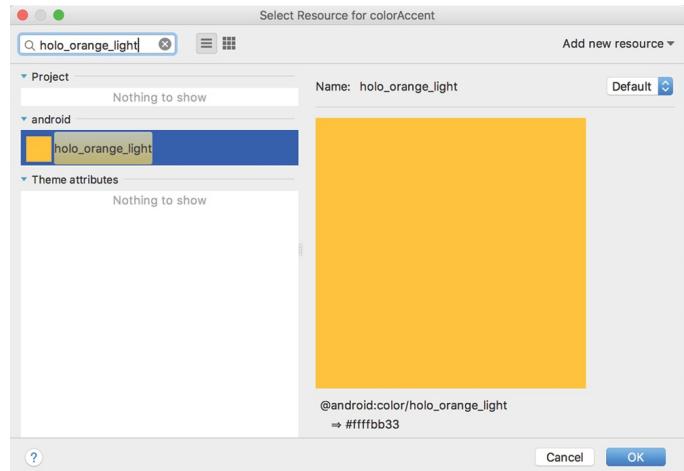


Figure 35-7

Click on the OK button to apply the new accentColor setting and return to the *activity_fab_example.xml* and verify that the floating action button now appears with an orange background.

35.8 Adding the ListView to the Content Layout

The next step in this tutorial is to add the ListView instance to the *content_fab_example.xml* file. The ListView class provides a way to display items in a list format and can be found in the *Containers* section of the Layout Editor tool palette.

Load the *content_fab_example.xml* file into the Layout Editor tool, select Design mode if necessary, and select and delete the default TextView object. Locate the ListView object in the Containers category of the palette and, with autoconnect mode enabled, drag and drop it onto the center of the layout canvas. Select the ListView object and change the ID to *listView* within the Attributes tool window. The Layout Editor should have sized the ListView to fill the entire container and established constraints on all four edges as illustrated in [Figure 35-8](#):

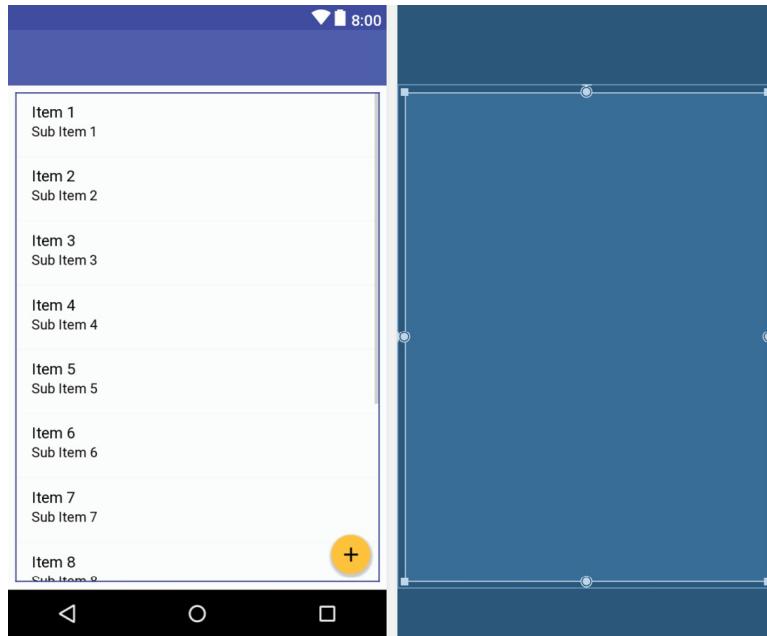


Figure 35-8

35.9 Adding Items to the ListView

Each time the floating action button is tapped by the user, a new item will be added to the ListView in the form of the prevailing time and date. To achieve this, some changes need to be made to the *FabExampleActivity.java* file.

Begin by modifying the *onCreate()* method to obtain a reference to the ListView instance and to initialize an adapter instance to allow us to add items to the list in the form of an array:

```
import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.ArrayAdapter;
import android.widget.ListView;

import java.util.ArrayList;

public class FabExampleActivity extends AppCompatActivity {

    ArrayList<String> listItems = new ArrayList<String>();
```

```

ArrayAdapter<String> adapter;
private ListView myListview;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_fab_example);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    myListview = (ListView) findViewById(R.id.listView);

    adapter = new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1,
        listItems);
    myListview.setAdapter(adapter);

    FloatingActionButton fab = (FloatingActionButton)
        findViewById(R.id.fab);
    fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action",
                Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
        }
    });
}

.
.
.
}

```

The ListView needs an array of items to display, an adapter to manage the items in that array and a layout definition to dictate how items are to be presented to the user.

In the above code changes, the items are stored in an ArrayList instance assigned to an adapter that takes the form of an ArrayAdapter. The items added to the list will be displayed in the ListView using the *simple_list_item_1* layout, a built-in layout that is provided with Android to display simple string based items in a ListView instance.

Next, edit the onClickListener code for the floating action button to display a different message in the Snackbar and to call a method to add an item to the list:

```
FloatingActionButton fab = (FloatingActionButton)
                        findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        addListItem();
        Snackbar.make(view, "Item added to list",
                Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
    }
});
```

Remaining within the *FabExampleActivity.java* file, add the *addListItem()* method as follows:

```
package com.ebookfrenzy.fabexample;
.

.

.

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

public class FabExampleActivity extends AppCompatActivity {

    .

    .

    private void addListItem() {

        SimpleDateFormat dateformat =
            new SimpleDateFormat("HH:mm:ss MM/dd/yyyy",
                Locale.US);
        listItems.add(dateformat.format(new Date()));
        adapter.notifyDataSetChanged();
    }
    .
    .
}
```

The code in the *addListItem()* method identifies and formats the current date and time and adds it to the list items array. The array adapter assigned to the

`ListView` is then notified that the list data has changed, causing the `ListView` to update to display the latest list items.

Compile and run the app and test that tapping the floating action button adds new time and date entries to the `ListView`, displaying the `Snackbar` each time as shown in [Figure 35-9](#):



Figure 35-9

35.10 Adding an Action to the Snackbar

The final task in this project is to add an action to the `Snackbar` that allows the user to undo the most recent addition to the list. Edit the `FabExampleActivity.java` file and modify the `Snackbar` creation code to add an action titled “Undo” configured with an `onClickListener` named `undoOnClickListener`:

```
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        addListItem();
```

```

        Snackbar.make(view, "Item added to list",
                Snackbar.LENGTH_LONG)
                .setAction("Undo", undoOnClickListener).show();

    }
});

}

```

Within the *FabExampleActivity.java* file add the listener handler:

```

View.OnClickListener undoOnClickListener = new View.OnClickListener()
{
    @Override
    public void onClick(View view) {
        listItems.remove(listItems.size() -1);
        adapter.notifyDataSetChanged();
        Snackbar.make(view, "Item removed", Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
    }
};

```

The code in the `onClick` method identifies the location of the last item in the list array and removes it from the list before triggering the list view to perform an update. A new Snackbar is then displayed indicating that the last item has been removed from the list.

Run the app once again and add some items to the list. On the final addition, tap the Undo button in the Snackbar ([Figure 35-10](#)) to remove the last item from the list:

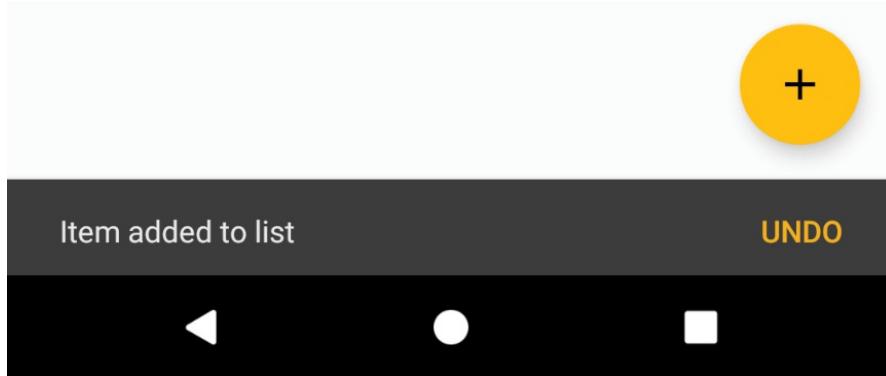


Figure 35-
10

It is also worth noting that the Undo button appears using the same color assigned to the `accentColor` property via the Theme Editor earlier in the chapter.

35.1 Summary

This chapter has provided a general overview of material design, the floating action button and Snackbar before working through an example project that makes use of these features

Both the floating action button and the Snackbar are part of the material design approach to user interface implementation in Android. The floating action button provides a way to promote the most common action within a particular screen of an Android application. The Snackbar provides a way for an application to both present information to the user and also allow the user to take action upon it.

36. Creating a Tabbed Interface using the TabLayout Component

The previous chapter outlined the concept of material design in Android and introduced two of the components provided by the design support library in the form of the floating action button and the Snackbar. This chapter will demonstrate how to use another of the design library components, the TabLayout, which can be combined with the ViewPager class to create a tab based interface within an Android activity.

36.1 An Introduction to the ViewPager

Although not part of the design support library, the ViewPager is a useful companion class when used in conjunction with the TabLayout component to implement a tabbed user interface. The primary role of the ViewPager is to allow the user to flip through different pages of information where each page is most typically represented by a layout fragment. The fragments that are associated with the ViewPager are managed by an instance of the FragmentPagerAdapter class.

At a minimum the pager adapter assigned to a ViewPager must implement two methods. The first, named *getCount()*, must return the total number of page fragments available to be displayed to the user. The second method, *getItem()*, is passed a page number and must return the corresponding fragment object ready to be presented to the user.

36.2 An Overview of the TabLayout Component

As previously discussed, TabLayout is one of the components introduced as part of material design and is included in the design support library. The purpose of the TabLayout is to present the user with a row of tabs which can be selected to display different pages to the user. The tabs can be fixed or scrollable, whereby the user can swipe left or right to view more tabs than will currently fit on the display. The information displayed on a tab can be text-based, an image or a combination of text and images. [Figure 36-1](#), for example, shows the tab bar for the Android phone app consisting of three tabs displaying images:



Figure 36-1

[Figure 36-2](#), on the other hand, shows a TabLayout configuration consisting of four tabs displaying text in a scrollable configuration:



Figure 36-2

The remainder of this chapter will work through the creation of an example project that demonstrates the use of the TabLayout component together with a ViewPager and four fragments.

36.3 Creating the TabLayoutDemo Project

Create a new project in Android Studio, entering *TabLayoutDemo* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich).

Continue through the configuration screens requesting the creation of a Basic Activity named *TabLayoutDemoActivity* with a corresponding layout file named *activity_tab_layout_demo*. Click on the *Finish* button to initiate the project creation process.

Once the project has been created, load the *content_tab_layout_demo.xml* file into the Layout Editor tool, select “Hello World” TextView object, and then delete it.

36.4 Creating the First Fragment

Each of the tabs on the TabLayout will display a different fragment when selected. Create the first of these fragments by right-clicking on the *app -> java -> com.ebookfrenzy.tablayoutdemo* entry in the Project tool window and selecting the *New -> Fragment -> Fragment (Blank)* option. In the resulting dialog, enter *Tab1Fragment* into the *Fragment Name:* field and *fragment_tab1* into the *Fragment Layout Name:* field. Enable the *Create layout XML?* option and disable both the *Include fragment factory methods?* and *Include interface*

callbacks? options before clicking on the *Finish* button to create the new fragment:

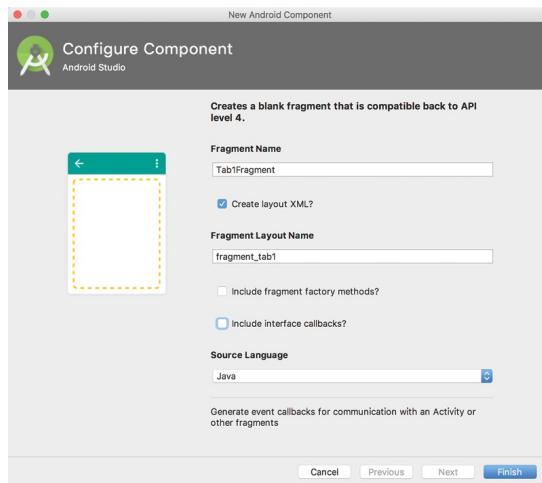


Figure 36-3

Load the newly created *fragment_tab1.xml* file (located under *app -> res -> layout*) into the Layout Editor tool, right-click on the *FrameLayout* entry in the Component Tree panel and select the *Convert FrameLayout to ConstraintLayout* menu option. In the resulting dialog, verify that all conversion options are selected before clicking on OK.

Once the layout has been converted to a *ConstraintLayout*, delete the *TextView* from the layout. From the Palette, locate the *TextView* widget and drag and drop it so that it is positioned in the center of the layout. Edit the text property on the object so that it reads “Tab 1 Fragment” and extract the string to a resource named *tab_1_fragment*, at which point the layout should match that of [Figure 36-4](#):

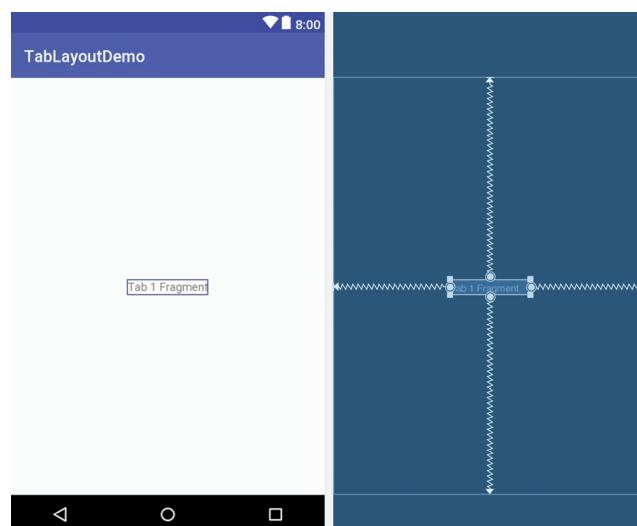


Figure 36-4

36.5 Duplicating the Fragments

So far, the project contains one of the four required fragments. Instead of creating the remaining three fragments using the previous steps it would be quicker to duplicate the first fragment. Each fragment consists of a layout XML file and a Java class file, each of which needs to be duplicated.

Right-click on the *fragment_tab1.xml* file in the Project tool window and select the Copy option from the resulting menu. Right-click on the *layout* entry, this time selecting the Paste option. In the resulting dialog, name the new layout file *fragment_tab2.xml* before clicking the OK button. Edit the new *fragment_tab2.xml* file and change the text on the Text View to “Tab 2 Fragment”, following the usual steps to extract the string to a resource named *tab_2_fragment*.

To duplicate the Tab1Fragment class file, right-click on the class listed under *app -> java -> com.ebookfrenzy.tablayoutdemo* and select Copy. Right-click on the *com.ebookfrenzy.tablayoutdemo* entry and select Paste. In the Copy Class dialog, enter Tab2Fragment into the *New name:* field and click on OK. Edit the new *Tab2Fragment.java* file and change the *onCreateView()* method to inflate the *fragment_tab2* layout file:

```
@Override  
public View onCreateView(LayoutInflater inflater, ViewGroup  
container,  
                           Bundle savedInstanceState) {  
    // Inflate the layout for this fragment  
    return inflater.inflate(R.layout.fragment_tab2, container,  
false);  
}
```

Perform the above duplication steps twice more to create the fragment layout and class files for the remaining two fragments. On completion of these steps the project structure should match that of [Figure 36-5](#):

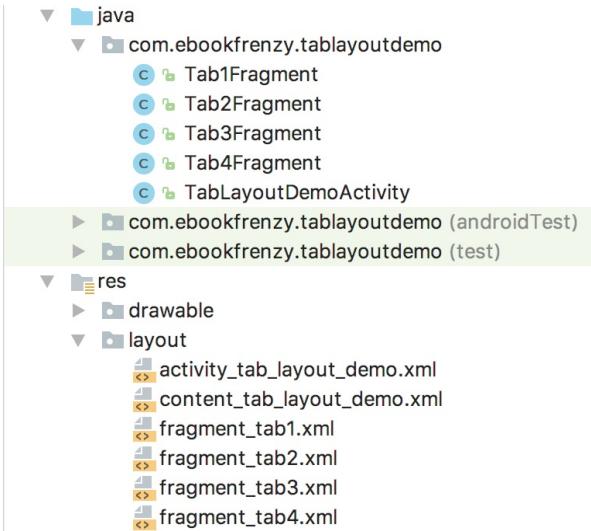


Figure 36-5

36.6 Adding the TabLayout and ViewPager

With the fragment creation process now complete, the next step is to add the TabLayout and ViewPager to the main activity layout file. Edit the *activity_tab_layout_demo.xml* file and add these elements as outlined in the following XML listing. Note that the TabLayout component is embedded into the AppBarLayout element while the ViewPager is placed after the AppBarLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    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"
    android:fitsSystemWindows="true"
    tools:context=".TabLayoutDemoActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr actionBarSize">
```

```

        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay" />

<android.support.design.widget.TabLayout
    android:id="@+id/tab_layout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:tabMode="fixed"
    app:tabGravity="fill"/>

</android.support.design.widget.AppBarLayout>

<android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
/>

<include layout="@layout/content_tab_layout_demo" />

<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/fab_margin"
    android:src="@android:drawable/ic_dialog_email" />

</android.support.design.widget.CoordinatorLayout>

```

36.7 Creating the Pager Adapter

This example will use the ViewPager approach to handling the fragments assigned to the TabLayout tabs. With the ViewPager added to the layout resource file, a new class which subclasses FragmentPagerAdapter needs to be added to the project to manage the fragments that will be displayed when the tab items are selected by the user.

Add a new class to the project by right-clicking on the *com.ebookfrenzy.tablayoutdemo* entry in the Project tool window and selecting the *New -> Java Class* menu option. In the new class dialog, enter *TabPagerAdapter* into the *Name:* field and click *OK*.

Edit the *TabPagerAdapter.java* file so that it reads as follows:

```
package com.ebookfrenzy.tablayoutdemo;

import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;

public class TabPagerAdapter extends FragmentPagerAdapter {

    int tabCount;

    public TabPagerAdapter(FragmentManager fm, int numberOfTabs) {
        super(fm);
        this.tabCount = numberOfTabs;
    }

    @Override
    public Fragment getItem(int position) {

        switch (position) {
            case 0:
                return new Tab1Fragment();
            case 1:
                return new Tab2Fragment();
            case 2:
                return new Tab3Fragment();
            case 3:
                return new Tab4Fragment();
            default:
                return null;
        }
    }

    @Override
    public int getCount() {
        return tabCount;
    }
}
```

The class is declared as extending the FragmentPagerAdapter class and a constructor is implemented allowing the number of pages required to be passed to the class when an instance is created. The *getItem()* method will be

called when a specific page is required. A switch statement is used to identify the page number being requested and to return a corresponding fragment instance. Finally, the `getCount()` method simply returns the count value passed through when the object instance was created.

36.8 Performing the Initialization Tasks

The remaining tasks involve initializing the TabLayout, ViewPager and TabPagerAdapter instances. All of these tasks will be performed in the `onCreate()` method of the `TabLayoutDemoActivity.java` file. Edit this file and modify the `onCreate()` method so that it reads as follows:

```
package com.ebookfrenzy.tablayoutdemo;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.support.design.widget.TabLayout;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;

.

.

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_tab_layout_demo);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    FloatingActionButton fab =
        (FloatingActionButton) findViewById(R.id.fab);
    fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action",
                Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
        }
    });
}
```

```
        });

    configureTabLayout();

}

private void configureTabLayout() {
    TabLayout tabLayout =
        (TabLayout) findViewById(R.id.tab_layout);

    tabLayout.addTab(tabLayout.newTab().setText("Tab 1 Item"));
    tabLayout.addTab(tabLayout.newTab().setText("Tab 2 Item"));
    tabLayout.addTab(tabLayout.newTab().setText("Tab 3 Item"));
    tabLayout.addTab(tabLayout.newTab().setText("Tab 4 Item"));

    final ViewPager viewPager =
        (ViewPager) findViewById(R.id.pager);
    final PagerAdapter adapter = new TabPagerAdapter
        (getSupportFragmentManager(),
         tabLayout.getTabCount());
    viewPager.setAdapter(adapter);

    viewPager.addOnPageChangeListener(new
        TabLayout.TabLayoutOnPageChangeListener(tabLayout));
    tabLayout.addOnTabSelectedListener(new
        TabLayout.OnTabSelectedListener() {
            @Override
            public void onTabSelected(TabLayout.Tab tab) {
                viewPager.setCurrentItem(tab.getPosition());
            }
        }

        @Override
        public void onTabUnselected(TabLayout.Tab tab) {

        }

        @Override
        public void onTabReselected(TabLayout.Tab tab) {

        }
    );
}
```

```
}
```

The code begins by obtaining a reference to the TabLayout object that was added to the *activity_tab_layout_demo.xml* file and creating four tabs, assigning the text to appear on each:

```
tabLayout.addTab(tabLayout.newTab().setText("Tab 1 Item"));
tabLayout.addTab(tabLayout.newTab().setText("Tab 2 Item"));
tabLayout.addTab(tabLayout.newTab().setText("Tab 3 Item"));
tabLayout.addTab(tabLayout.newTab().setText("Tab 4 Item"));
```

A reference to the ViewPager instance in the layout file is then obtained and an instance of the TabPagerAdapter class created. Note that the code to create the TabPagerAdapter instance passes through the number of tabs that have been assigned to the TabLayout component. The TabPagerAdapter instance is then assigned as the adapter for the ViewPager and the TabLayout component added to the page change listener:

```
final ViewPager viewPager = (ViewPager) findViewById(R.id.pager);
final PagerAdapter adapter = new TabPagerAdapter
    (getSupportFragmentManager(),
     tabLayout.getTabCount());
viewPager.setAdapter(adapter);

viewPager.addOnPageChangeListener(new
    TabLayout.TabLayoutOnPageChangeListener(tabLayout));
```

Finally, the *onTabSelectedListener* is configured on the TabLayout instance and the *onTabSelected()* method implemented to set the current page on the ViewPager based on the currently selected tab number. For the sake of completeness the other listener methods are added as stubs:

```
tabLayout.setOnTabSelectedListener(new
    TabLayout.OnTabSelectedListener()
{
    @Override
    public void onTabSelected(TabLayout.Tab tab) {
        viewPager.setCurrentItem(tab.getPosition());
    }

    @Override
    public void onTabUnselected(TabLayout.Tab tab) {

    }
}
```

```
    @Override  
    public void onTabReselected(TabLayout.Tab tab) {  
  
    }  
});
```

36.9 Testing the Application

Compile and run the app on a device or emulator and make sure that selecting a tab causes the corresponding fragment to appear in the content area of the screen:



Figure 36-6

36.10 Customizing the TabLayout

The TabLayout in this example project is configured using *fixed* mode. This mode works well for a limited number of tabs with short titles. A greater number of tabs or longer titles can quickly become a problem when using fixed mode as illustrated by [Figure 36-7](#):

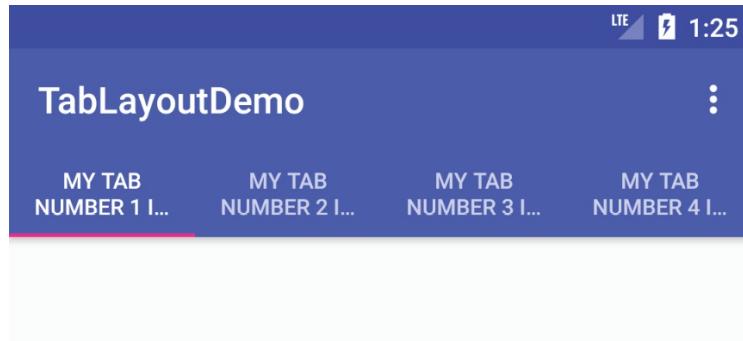


Figure 36-7

In an effort to fit the tabs into the available display width the TabLayout has used multiple lines of text. Even so, the second line is clearly truncated making it impossible to see the full title. The best solution to this problem is to switch the TabLayout to *scrollable* mode. In this mode the titles appear in full length, single line format allowing the user to swipe to scroll horizontally through the available items as demonstrated in [Figure 36-8](#):

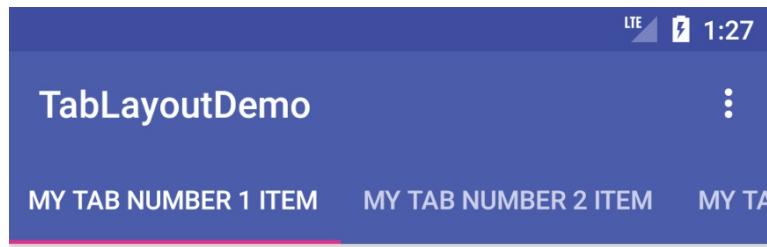


Figure 36-8

To switch a TabLayout to scrollable mode, simply change the *app:tabMode* property in the *activity_tab_layout_demo.xml* layout resource file from “fixed” to “scrollable”:

```
<android.support.design.widget.TabLayout  
    android:id="@+id/tab_layout"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    app:tabMode="scrollable"  
    app:tabGravity="fill"/>  
</android.support.design.widget.AppBarLayout>
```

When in fixed mode, the TabLayout may be configured to control how the tab items are displayed to take up the available space on the screen. This is controlled via the *app:tabGravity* property, the results of which are more noticeable on wider displays such as tablets in landscape orientation. When

set to “fill”, for example, the items will be distributed evenly across the width of the TabLayout as shown in [Figure 36-9](#):

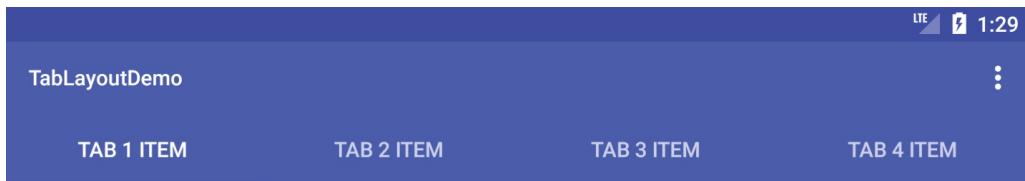


Figure 36-9

Changing the property value to “center” will cause the items to be positioned relative to the center of the tab bar:

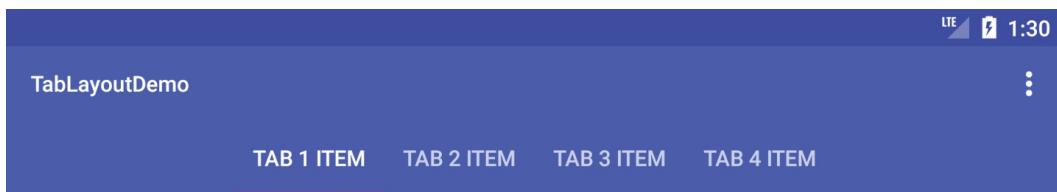


Figure 36-
10

Before proceeding to the final step in this chapter, revert the tabMode and tabGravity attributes in the *activity_tab_layout_demo.xml* file to “fixed” and “fill” respectively.

36.1 Displaying Icon Tab Items

The last step in this tutorial is to replace the text based tabs with icons. To achieve this, modify the *onCreate()* method in the *TabLayoutDemoActivity.java* file to assign some built-in drawable icons to the tab items:

```
private void configureTabLayout() {  
  
    TabLayout tabLayout = (TabLayout) findViewById(R.id.tab_layout);  
  
    tabLayout.addTab(tabLayout.newTab().setIcon(  
        android.R.drawable.ic_dialog_email));  
    tabLayout.addTab(tabLayout.newTab().setIcon(  
        android.R.drawable.ic_dialog_dialer));  
    tabLayout.addTab(tabLayout.newTab().setIcon(  
        android.R.drawable.ic_dialog_map));  
    tabLayout.addTab(tabLayout.newTab().setIcon(  
        android.R.drawable.ic_dialog_info));
```

```
final ViewPager viewPager =  
    (ViewPager) findViewById(R.id.pager);  
.  
.  
.  
}
```

Instead of using the *setText()* method of the tab item, the code is now calling the *setIcon()* method and passing through a drawable icon reference. When compiled and run, the tab bar should now appear as shown in [Figure 36-11](#). Note if using Instant Run that it will be necessary to trigger a warm swap using Ctrl-Shift-R for the changes to take effect:

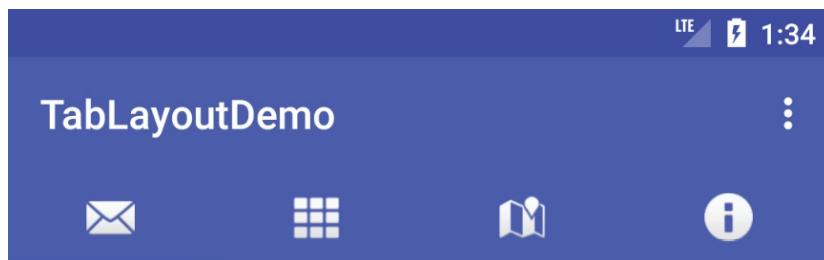


Figure 36-
11

36.1 Summary

TabLayout is one of the components introduced as part of the Android material design implementation. The purpose of the TabLayout component is to present a series of tab items which, when selected, display different content to the user. The tab items can display text, images or a combination of both. When combined with the ViewPager class and fragments, tab layouts can be created with relative ease, with each tab item selection displaying a different fragment.

37. Working with the RecyclerView and CardView Widgets

The RecyclerView and CardView widgets work together to provide scrollable lists of information to the user in which the information is presented in the form of individual cards. Details of both classes will be covered in this chapter before working through the design and implementation of an example project.

37.1 An Overview of the RecyclerView

Much like the ListView class outlined in the chapter entitled [*“Working with the Floating Action Button and Snackbar”*](#), the purpose of the RecyclerView is to allow information to be presented to the user in the form of a scrollable list. The RecyclerView, however, provides a number of advantages over the ListView. In particular, the RecyclerView is significantly more efficient in the way it manages the views that make up a list, essentially reusing existing views that make up list items as they scroll off the screen instead of creating new ones (hence the name “recycler”). This both increases the performance and reduces the resources used by a list, a feature that is of particular benefit when presenting large amounts of data to the user.

Unlike the ListView, the RecyclerView also provides a choice of three built-in layout managers to control the way in which the list items are presented to the user:

- **LinearLayoutManager** – The list items are presented as either a horizontal or vertical scrolling list.



Figure 37-1

- **GridLayoutManager** – The list items are presented in grid format. This manager is best used when the list items of are of uniform size.



Figure 37-2

- **StaggeredGridLayoutManager** - The list items are presented in a staggered grid format. This manager is best used when the list items are not of uniform size.



Figure 37-3

For situations where none of the three built-in managers provide the necessary layout, custom layout managers may be implemented by subclassing the `RecyclerView.LayoutManager` class.

Each list item displayed in a `RecyclerView` is created as an instance of the `ViewHolder` class. The `ViewHolder` instance contains everything necessary for the `RecyclerView` to display the list item, including the information to be displayed and the view layout used to display the item.

As with the `ListView`, the `RecyclerView` depends on an adapter to act as the intermediary between the `RecyclerView` instance and the data that is to be displayed to the user. The adapter is created as a subclass of the `RecyclerView.Adapter` class and must, at a minimum, implement the

following methods, which will be called at various points by the RecyclerView object to which the adapter is assigned:

- **getCount()** – This method must return a count of the number of items that are to be displayed in the list.
- **onCreateViewHolder()** – This method creates and returns a ViewHolder object initialized with the view that is to be used to display the data. This view is typically created by inflating the XML layout file.
- **onBindViewHolder()** – This method is passed the ViewHolder object created by the *onCreateViewHolder()* method together with an integer value indicating the list item that is about to be displayed. Contained within the ViewHolder object is the layout assigned by the *onCreateViewHolder()* method. It is the responsibility of the *onBindViewHolder()* method to populate the views in the layout with the text and graphics corresponding to the specified item and to return the object to the RecyclerView where it will be presented to the user.

Adding a RecyclerView to a layout is simply a matter of adding the appropriate element to the XML layout file of the activity in which it is to appear. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    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" android:fitsSystemWindows="true"
    tools:context=".CardStuffActivity">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior"/>

    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
```

```
    android:theme="@style/AppTheme.AppBarOverlay">

    <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay" />

</android.support.design.widget.AppBarLayout>
.
.
.
}
```

In the above example the RecyclerView has been embedded into the CoordinatorLayout of a main activity layout file along with the AppBar and Toolbar. This provides some additional features, such as configuring the Toolbar and AppBar to scroll off the screen when the user scrolls up within the RecyclerView (a topic covered in more detail in the chapter entitled [“Working with the AppBar and Collapsing Toolbar Layouts”](#)).

37.2 An Overview of the CardView

The CardView class is a user interface view that allows information to be presented in groups using a card metaphor. Cards are usually presented in lists using a RecyclerView instance and may be configured to appear with shadow effects and rounded corners. [Figure 37-4](#), for example, shows three CardView instances configured to display a layout consisting of an ImageView and two TextViews:

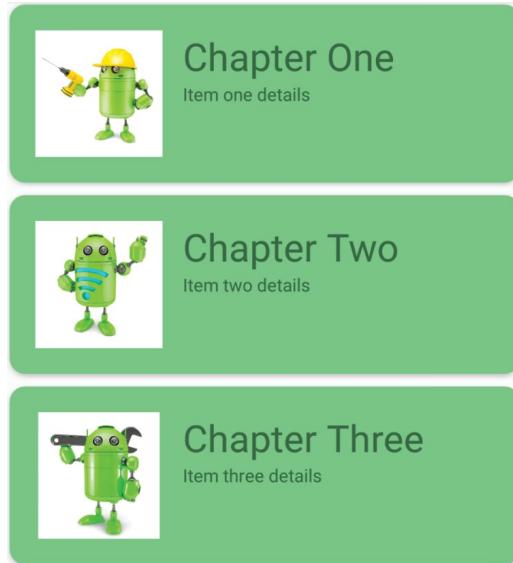


Figure 37-4

The user interface layout to be presented with a CardView instance is defined within an XML layout resource file and loaded into the CardView at runtime. The CardView layout can contain a layout of any complexity using the standard layout managers such as RelativeLayout and LinearLayout. The following XML layout file represents a card view layout consisting of a RelativeLayout and a single ImageView. The card is configured to be elevated to create shadowing effect and to appear with rounded corners:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/card_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="5dp"
    card_view:cardCornerRadius="12dp"
    card_view:cardElevation="3dp"
    card_view:contentPadding="4dp">

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp" >

        <ImageView
```

```
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:id="@+id/item_image"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:layout_marginRight="16dp" />
    </RelativeLayout>
</android.support.v7.widget.CardView>
```

When combined with the RecyclerView to create a scrollable list of cards, the `onCreateViewHolder()` method of the recycler view inflates the layout resource file for the card, assigns it to the ViewHolder instance and returns it to the RecyclerView instance.

37.3 Adding the Libraries to the Project

In order to use the RecyclerView and CardView components, the corresponding libraries must be added to the Gradle build dependencies for the project. Within the module level `build.gradle` file, therefore, the following lines need to be added to the `dependencies` section:

```
dependencies {
    ...
    ...
    implementation 'com.android.support:recyclerview-v7:26.1.0'
    implementation 'com.android.support:cardview-v7:26.1.0'
}
```

37.4 Summary

This chapter has introduced the Android RecyclerView and CardView components. The RecyclerView provides a resource efficient way to display scrollable lists of views within an Android app. The CardView is useful when presenting groups of data (such as a list of names and addresses) in the form of cards. As previously outlined, and demonstrated in the tutorial contained in the next chapter, the RecyclerView and CardView are particularly useful when combined.

38. An Android RecyclerView and CardView Tutorial

In this chapter an example project will be created that makes use of both the CardView and RecyclerView components to create a scrollable list of cards. The completed app will display a list of cards containing images and text. In addition to displaying the list of cards, the project will be implemented such that selecting a card causes a message to be displayed to the user indicating which card was tapped.

38.1 Creating the CardDemo Project

Create a new project in Android Studio, entering *CardDemo* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich) and continue to proceed through the screens.

In the next chapter, the scroll handling features of the AppBar, Toolbar and CoordinatorLayout layout will be demonstrated using this project. On the activity selection screen, therefore, request the creation of a Basic Activity named *CardDemoActivity* with a corresponding layout file named *activity_card_demo*. Click on the *Finish* button to initiate the project creation process.

Once the project has been created, load the *content_card_demo.xml* file into the Layout Editor tool and select and delete the “Hello World” TextView object.

38.2 Removing the Floating Action Button

Since the Basic Activity was selected, the layout includes a floating action button which is not required for this project. Load the *activity_card_demo.xml* layout file into the Layout Editor tool, select the floating action button and tap the keyboard delete key to remove the object from the layout. Edit the *CardDemoActivity.java* file and remove the floating action button code from the onCreate method as follows:

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_card_demo);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    FloatingActionButton fab =
        (FloatingActionButton) findViewById(R.id.fab);
    fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action",
                Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
        }
    });
}

```

38.3 Adding the RecyclerView and CardView Libraries

Within the Project tool window locate and select the module level *build.gradle* file and modify the dependencies section of the file to add the support library dependencies for the RecyclerView and CardView:

```

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:26.1.0'
    implementation 'com.android.support.constraint:constraint-
layout:1.0.2'
    implementation 'com.android.support:design:26.1.0'
    implementation 'com.android.support:recyclerview-v7:26.1.0'
    implementation 'com.android.support:cardview-v7:26.1.0'
    .
    .
}

```

When prompted to do so, resync the new Gradle build configuration by clicking on the *Sync Now* link in the warning bar.

38.4 Designing the CardView Layout

The layout of the views contained within the cards will be defined within a separate XML layout file. Within the Project tool window right click on the *app -> res -> layout* entry and select the *New -> Layout resource file* menu

option. In the New Resource Dialog enter *card_layout* into the *File name:* field and *android.support.v7.widget.CardView* into the root element field before clicking on the OK button.

Load the *card_layout.xml* file into the Layout Editor tool, switch to Text mode and modify the layout so that it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/card_view"
    android:layout_margin="5dp"
    card_view:cardBackgroundColor="#81C784"
    card_view:cardCornerRadius="12dp"
    card_view:cardElevation="3dp"
    card_view:contentPadding="4dp" >

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp" >

        <ImageView
            android:layout_width="100dp"
            android:layout_height="100dp"
            android:id="@+id/item_image"
            android:layout_alignParentStart="true"
            android:layout_alignParentLeft="true"
            android:layout_alignParentTop="true"
            android:layout_marginEnd="16dp"
            android:layout_marginRight="16dp" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/item_title"
            android:layout_toEndOf="@+id/item_image"
            android:layout_toRightOf="@+id/item_image"
            android:layout_alignParentTop="true"
            android:textSize="30sp" />
    
```

```

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/item_detail"
    android:layout_toEndOf="@+id/item_image"
    android:layout_toRightOf="@+id/item_image"
    android:layout_below="@+id/item_title" />

</RelativeLayout>
</android.support.v7.widget.CardView>
```

38.5 Adding the RecyclerView

Select the *activity_card_demo.xml* layout file and modify it to add the RecyclerView component immediately before the AppBarLayout:

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    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"
    android:fitsSystemWindows="true"
    tools:context="com.ebookfrenzy.carddemo.CardDemoActivity">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior" />

    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">

        .
        .
        .

    </AppBarLayout>
</CoordinatorLayout>
```

38.6 Creating the RecyclerView Adapter

As outlined in the previous chapter, the RecyclerView needs to have an adapter to handle the creation of the list items. Add this new class to the