Android Studio Dolphin Essentials

Java Edition

Android Studio Dolphin Essentials – Java Edition

ISBN-13: 978-1-951442-56-9

Rev: 1.0

Table of Contents

Chapter 1

1. Introduction

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

This book begins with the basics and outlines 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.

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

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

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

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

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

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

1.1 Downloading the Code Samples

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

https://www.ebookfrenzy.com/retail/dolphinjava/index.php

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

1. From the Welcome to Android Studio dialog, click on the Open button option.

2. In the project selection dialog, navigate to and select the folder containing the project to be imported and click on OK.

1.2 Feedback

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

1.3 Errata

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

https://www.ebookfrenzy.com/errata/dolphinjava.html

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

2. Setting up an Android Studio Development Environment

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

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

2.1 System requirements

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

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

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

2.2 Downloading the Android Studio package

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

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

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

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

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

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

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

2.3 Installing Android Studio

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

2.3.1 Installation on Windows

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

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

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

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

2.3.2 Installation on macOS

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

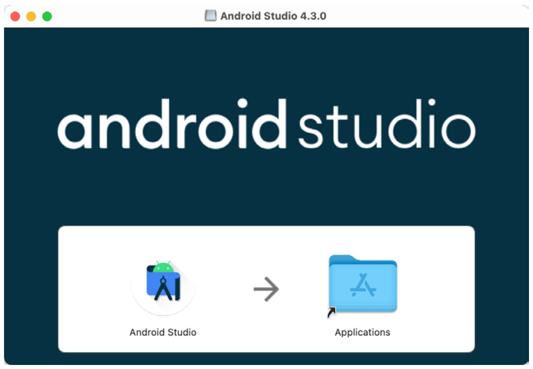


Figure 2-1

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

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

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

2.3.3 Installation on Linux

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

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

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

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

```
./studio.sh
```

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

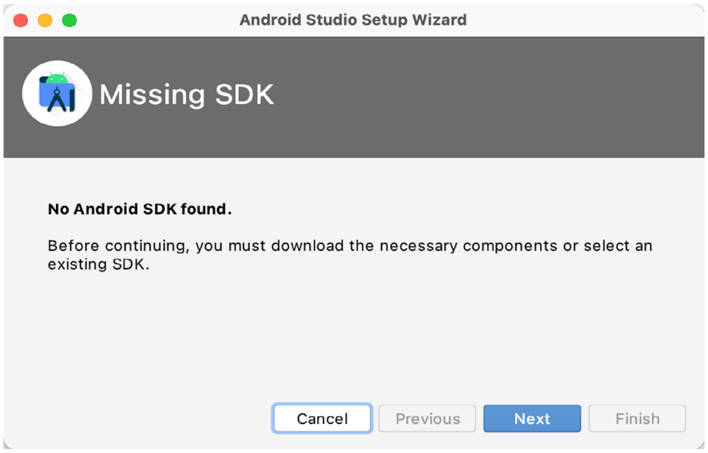
```
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386 lib32z1 libbz2-
1.0:i386
```

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

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

2.4 The Android Studio setup wizard

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

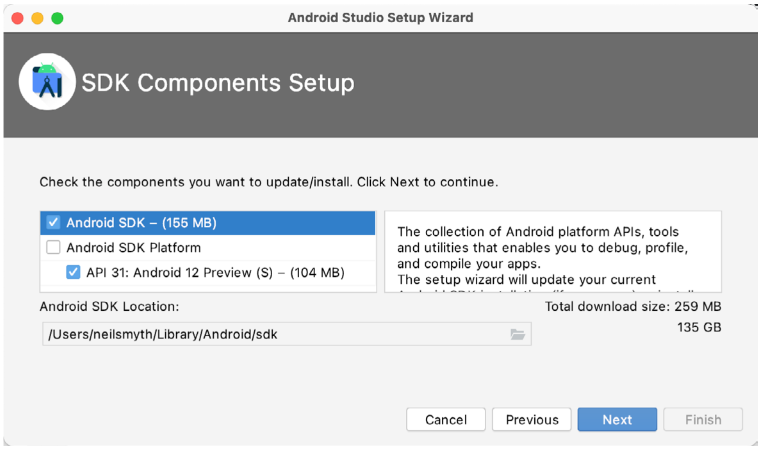


No Android SDK found.

Before continuing, you must download the necessary components or select an existing SDK.

Figure 2-2

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



Check the components you want to update/install. Click Next to continue.

- [✓] Android SDK – (155 MB)
- [ ] Android SDK Platform
  - [✓] API 31: Android 12 Preview (S) – (104 MB)

The collection of Android platform APIs, tools and utilities that enables you to debug, profile, and compile your apps.
The setup wizard will update your current

Android SDK Location: /Users/neilsmyth/Library/Android/sdk

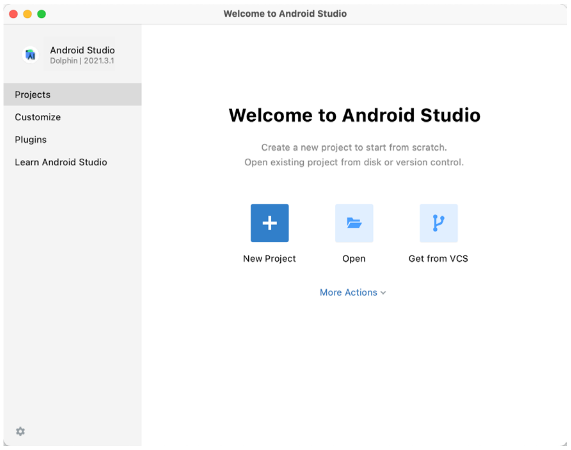
Total download size: 259 MB
135 GB

Figure 2-3

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

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

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



Welcome to Android Studio

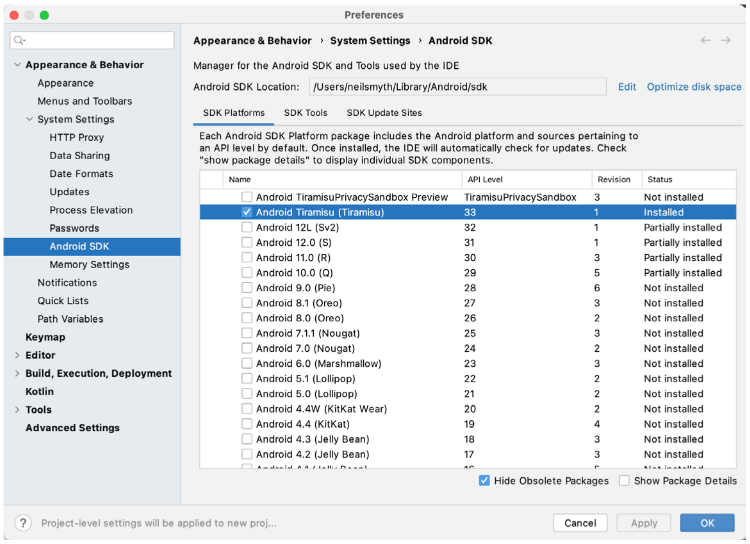
Create a new project to start from scratch.
Open existing project from disk or version control.

Figure 2-4

2.5 Installing additional Android SDK packages

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

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



Preferences

- Appearance & Behavior
  - Appearance
  - Menus and Toolbars
  - System Settings
    - HTTP Proxy
    - Data Sharing
    - Date Formats
    - Updates
    - Process Elevation
    - Passwords
    - Android SDK
    - Memory Settings
  - Notifications
  - Quick Lists
  - Path Variables
- Keymap
- Editor
- Build, Execution, Deployment
- Kotlin
- Tools
- Advanced Settings

Appearance & Behavior › System Settings › Android SDK

Manager for the Android SDK and Tools used by the IDE

Android SDK Location: /Users/neilsmyth/Library/Android/sdk Edit Optimize disk space

SDK Platforms SDK Tools SDK Update Sites

Each Android SDK Platform package includes the Android platform and sources pertaining to an API level by default. Once installed, the IDE will automatically check for updates. Check "show package details" to display individual SDK components.

Name	API Level	Revision	Status
☐ Android TiramisuPrivacySandbox Preview	TiramisuPrivacySandbox	3	Not installed
✓ Android Tiramisu (Tiramisu)	33	1	Installed
☐ Android 12L (Sv2)	32	1	Partially installed
☐ Android 12.0 (S)	31	1	Partially installed
☐ Android 11.0 (R)	30	3	Partially installed
☐ Android 10.0 (Q)	29	5	Partially installed
☐ Android 9.0 (Pie)	28	6	Not installed
☐ Android 8.1 (Oreo)	27	3	Not installed
☐ Android 8.0 (Oreo)	26	2	Not installed
☐ Android 7.1.1 (Nougat)	25	3	Not installed
☐ Android 7.0 (Nougat)	24	2	Not installed
☐ Android 6.0 (Marshmallow)	23	3	Not installed
☐ Android 5.1 (Lollipop)	22	2	Not installed
☐ Android 5.0 (Lollipop)	21	2	Not installed
☐ Android 4.4W (KitKat Wear)	20	2	Not installed
☐ Android 4.4 (KitKat)	19	4	Not installed
☐ Android 4.3 (Jelly Bean)	18	3	Not installed
☐ Android 4.2 (Jelly Bean)	17	3	Not installed

✓ Hide Obsolete Packages ☐ Show Package Details

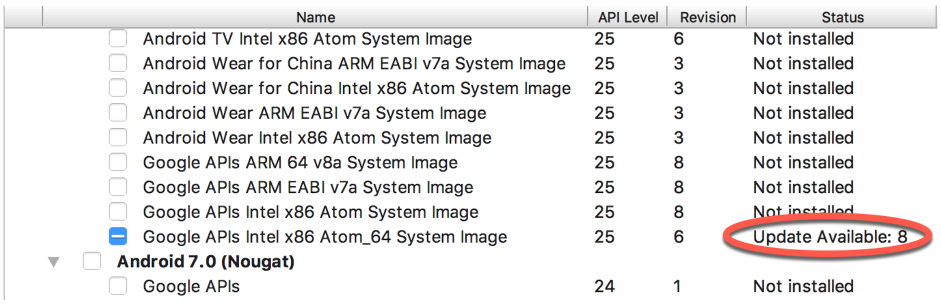
Project-level settings will be applied to new proj... Cancel Apply OK

Figure 2-5

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

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

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

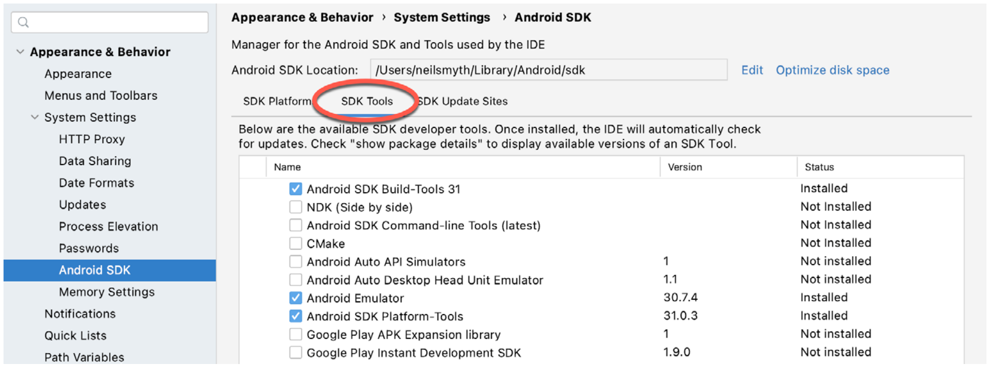


Name	API Level	Revision	Status
☐ Android TV Intel x86 Atom System Image	25	6	Not installed
☐ Android Wear for China ARM EABI v7a System Image	25	3	Not installed
☐ Android Wear for China Intel x86 Atom System Image	25	3	Not installed
☐ Android Wear ARM EABI v7a System Image	25	3	Not installed
☐ Android 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 2-6

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

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



Appearance & Behavior
Appearance
Menus and Toolbars
System Settings
HTTP Proxy
Data Sharing
Date Formats
Updates
Process Elevation
Passwords
Android SDK
Memory Settings
Notifications
Quick Lists
Path Variables

Appearance & Behavior › System Settings › Android SDK

Manager for the Android SDK and Tools used by the IDE

Android SDK Location: /Users/neilsmyth/Library/Android/sdk Edit Optimize disk space

SDK Platforms SDK Tools SDK Update Sites

Below are the available SDK developer tools. Once installed, the IDE will automatically check for updates. Check "show package details" to display available versions of an SDK Tool.

Name	Version	Status
✓ Android SDK Build-Tools 31		Installed
NDK (Side by side)		Not Installed
Android SDK Command-line Tools (latest)		Not Installed
CMake		Not Installed
Android Auto API Simulators	1	Not installed
Android Auto Desktop Head Unit Emulator	1.1	Not installed
✓ Android Emulator	30.7.4	Installed
✓ Android SDK Platform-Tools	31.0.3	Installed
Google Play APK Expansion library	1	Not installed
Google Play Instant Development SDK	1.9.0	Not installed

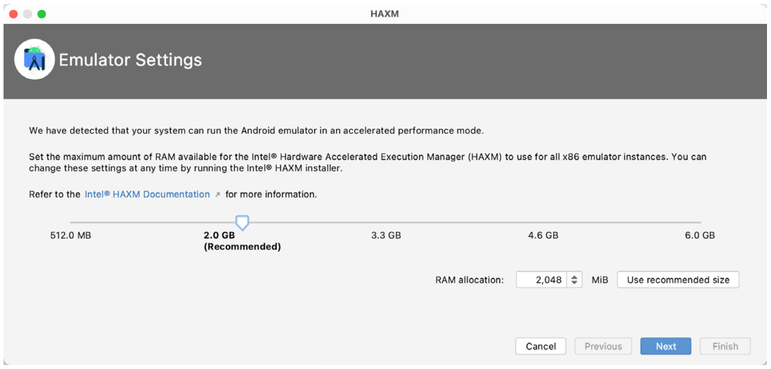
Figure 2-7

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

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

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

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



HAXM

Emulator Settings

We have detected that your system can run the Android emulator in an accelerated performance mode.

Set the maximum amount of RAM available for the Intel® Hardware Accelerated Execution Manager (HAXM) to use for all x86 emulator instances. You can change these settings at any time by running the Intel® HAXM installer.

Refer to the Intel® HAXM Documentation for more information.

512.0 MB 2.0 GB (Recommended) 3.3 GB 4.6 GB 6.0 GB

RAM allocation: 2,048 MiB Use recommended size

Cancel Previous Next Finish

Figure 2-8

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

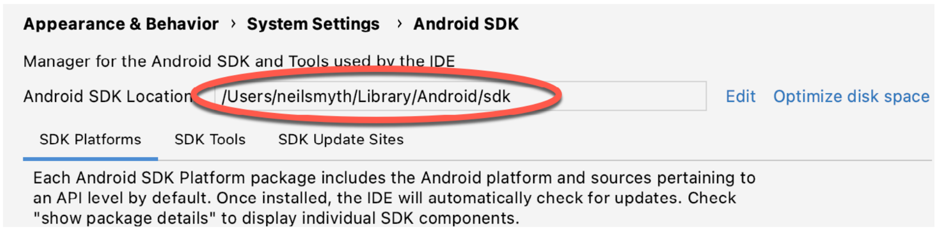
2.6 Making the Android SDK tools command-line accessible

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

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

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

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



Manager for the Android SDK and Tools used by the IDE

Android SDK Location: /Users/neilsmyth/Library/Android/sdk

Each Android SDK Platform package includes the Android platform and sources pertaining to an API level by default. Once installed, the IDE will automatically check for updates. Check "show package details" to display individual SDK components.

Figure 2-9

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

2.6.1 Windows 8.1

1. On the start screen, move the mouse to the bottom right-hand corner of the screen and select Search from the resulting menu. In the search box, enter Control Panel. When the Control Panel icon appears in the results area, click on it to launch the tool on the desktop.

2. Within the Control Panel, use the Category menu to change the display to Large Icons. From the list of icons select the one labeled System.

3. In the Environment Variables dialog, locate the Path variable in the System variables list, select it and click on the *Edit…* button. Using the *New* button in the edit dialog, add three new entries to the path. For example, assuming the Android SDK was installed into C:\Users\demo\AppData\Local\Android\Sdk, the following entries would need to be added:

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

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

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

```
echo %Path%
```

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

```
adb
```

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

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

```
avdmanager
```

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

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

2.6.2 Windows 10

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

2.6.3 Windows 11

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

2.6.4 Linux

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

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

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

2.6.5 macOS

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

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

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

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

2.7 Android Studio memory management

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

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

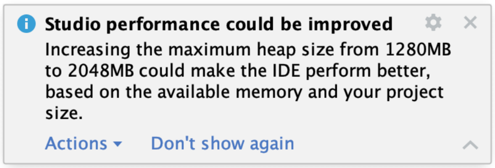
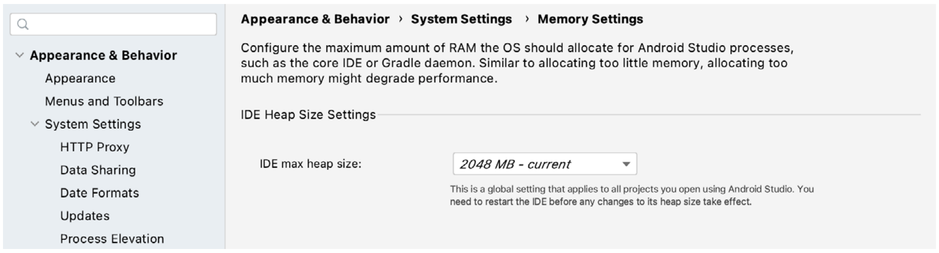

Studio performance could be improved
Increasing the maximum heap size from 1280MB to 2048MB could make the IDE perform better, based on the available memory and your project size.

Figure 2-10

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

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



Configure the maximum amount of RAM the OS should allocate for Android Studio processes, such as the core IDE or Gradle daemon. Similar to allocating too little memory, allocating too much memory might degrade performance.

IDE Heap Size Settings

IDE max heap size: 2048 MB - current

This is a global setting that applies to all projects you open using Android Studio. You need to restart the IDE before any changes to its heap size take effect.

Figure 2-11

2.8 Updating Android Studio and the SDK

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

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

2.9 Summary

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

Chapter 3

3. Creating an Example Android App in Android Studio

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

3.1 About the Project

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

3.2 Creating a New Android Project

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

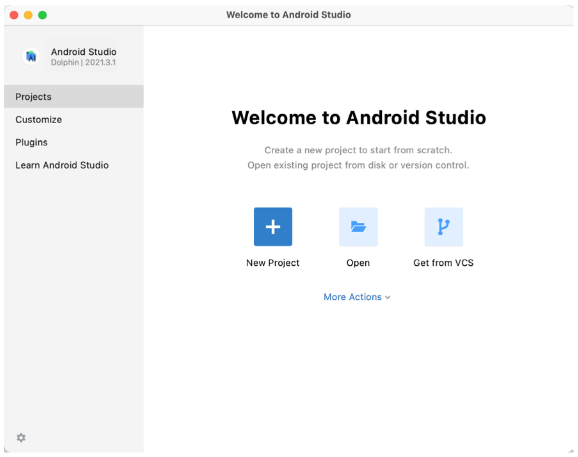

Welcome to Android Studio
Android Studio
Dolphin | 2021.3.1
Projects
Customize
Plugins
Learn Android Studio
Welcome to Android Studio
Create a new project to start from scratch.
Open existing project from disk or version control.
New Project
Open
Get from VCS
More Actions

Figure 3-1

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

3.3 Creating an Activity

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

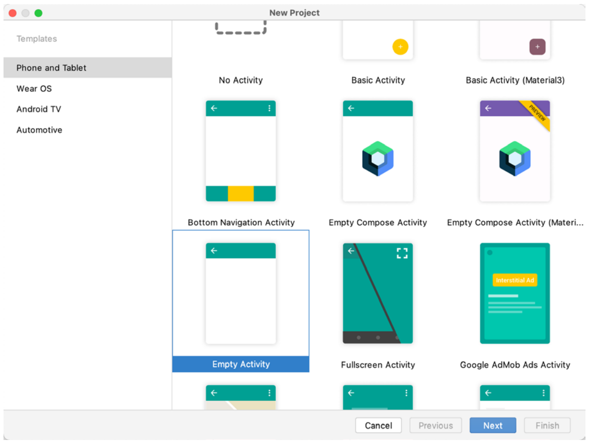


Figure 3-2

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

3.4 Defining the Project and SDK Settings

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

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

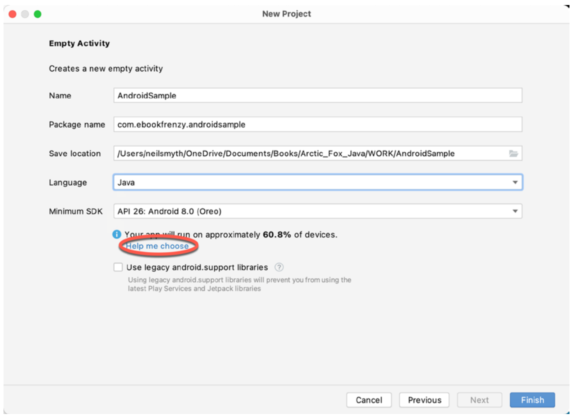
```
com.mycompany.androidsample
```

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

```
com.example.androidsample
```

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

Set the minimum SDK setting to API 26: Android 8.0 (Oreo). This is the minimum SDK that will be used in most of the projects created in this book unless a necessary feature is only available in a more recent version. The objective here is to build an app using the latest Android SDK, while also retaining compatibility with devices running older versions of Android (in this case as far back as Android 8.0). The text beneath the Minimum SDK setting will outline the percentage of Android devices currently in use on which the app will run. Click on the *Help me choose* button (highlighted in Figure 3-3) to see a full breakdown of the various Android versions still in use:


New Project

Empty Activity

Creates a new empty activity

Name: AndroidSample

Package name: com.ebookfrenzy.androidsample

Save location: /Users/neilsmyth/OneDrive/Documents/Books/Arctic_Fox_Java/WORK/AndroidSample

Language: Java

Minimum SDK: API 26: Android 8.0 (Oreo)

Your app will run on approximately **60.8%** of devices.

Help me choose

Use legacy android.support libraries

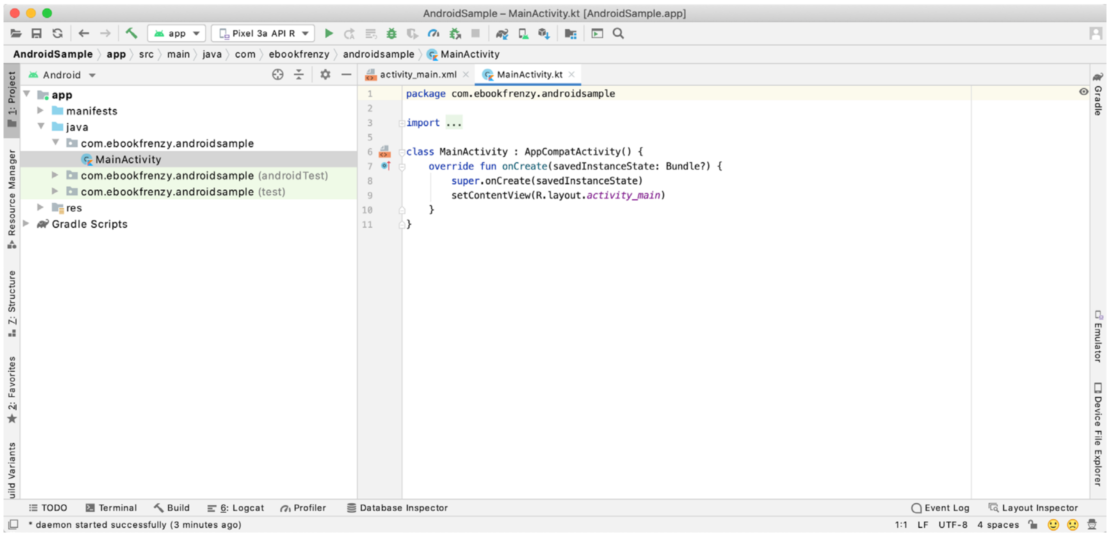
Using legacy android.support libraries will prevent you from using the latest Play Services and Jetpack libraries

Figure 3-3

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

3.5 Modifying the Example Application

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


AndroidSample – MainActivity.kt [AndroidSample.app]

AndroidSample 〉 app 〉 src 〉 main 〉 java 〉 com 〉 ebookfrenzy 〉 androidsample 〉 MainActivity

```
app
  manifests
  java
    com.ebookfrenzy.androidsample
      MainActivity
    com.ebookfrenzy.androidsample (androidTest)
    com.ebookfrenzy.androidsample (test)
  res
Gradle Scripts
```

```
package com.ebookfrenzy.androidsample

import ...

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

* daemon started successfully (3 minutes ago)

Figure 3-4

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

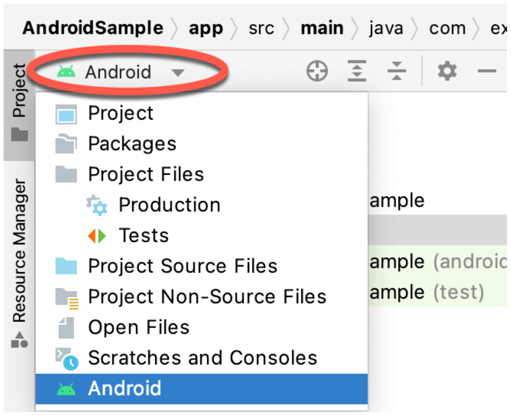


Figure 3-5

3.6 Modifying the User Interface

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

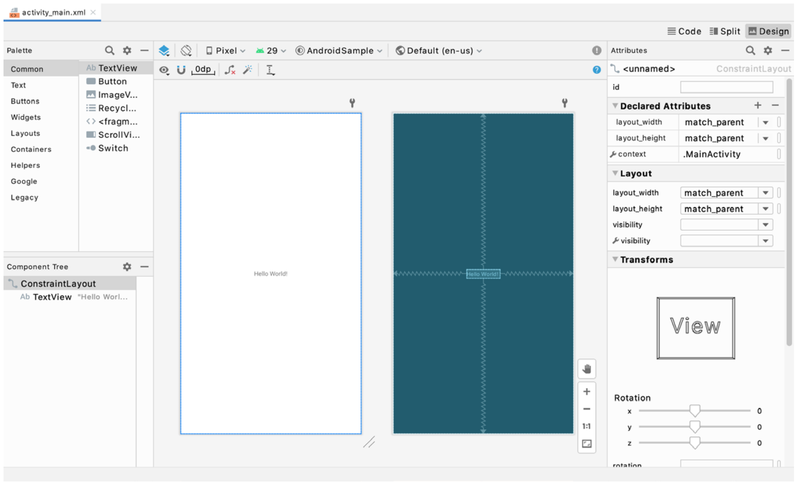


Figure 3-6

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

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

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

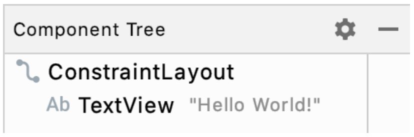


Figure 3-7

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

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

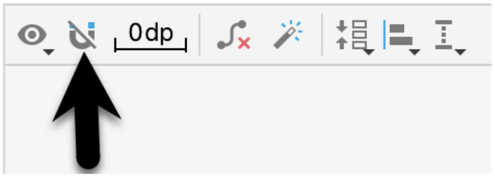


Figure 3-8

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

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

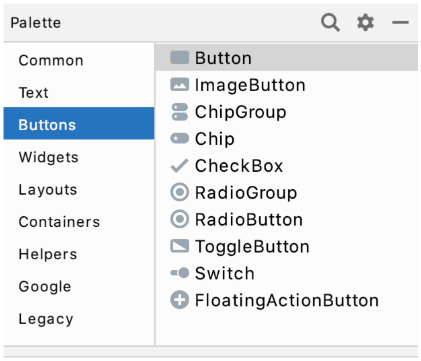


Figure 3-9

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

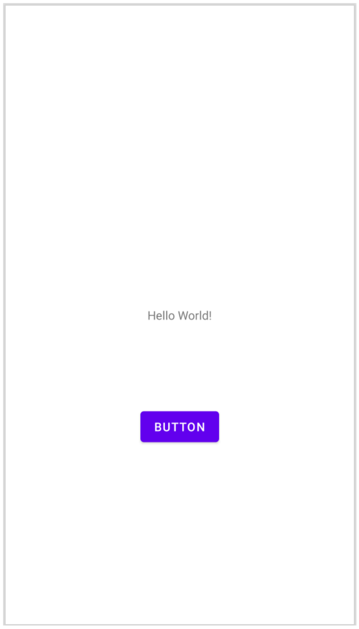
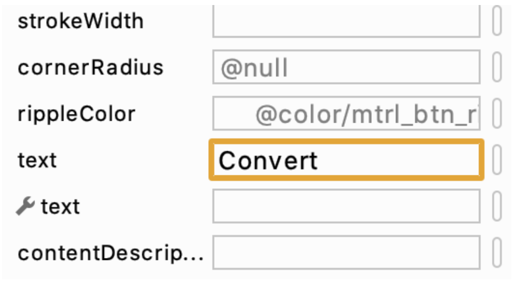


Figure 3-10

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



strokeWidth	
cornerRadius	@null
rippleColor	@color/mtrl_btn_r
text	Convert
text	
contentDescrip...	

Figure 3-11

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

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

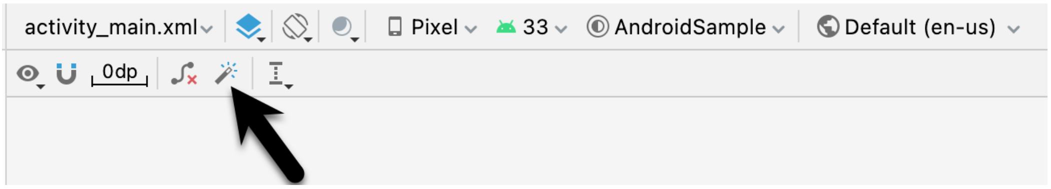


Figure 3-12

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

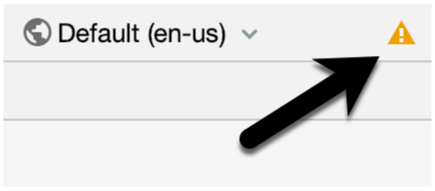
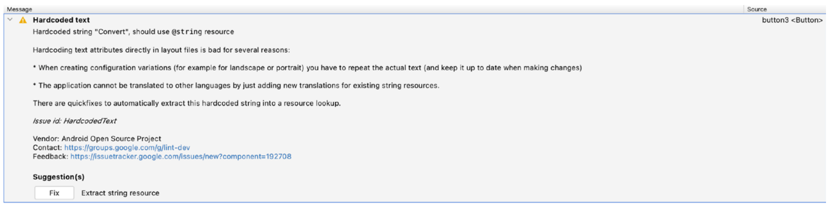


Figure 3-13

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



Message | Source

Hardcoded text button3 <Button>

Hardcoded string "Convert", should use `@string` resource

Hardcoding text attributes directly in layout files is bad for several reasons:

• When creating configuration variations (for example for landscape or portrait) you have to repeat the actual text (and keep it up to date when making changes)

• The application cannot be translated to other languages by just adding new translations for existing string resources.

There are quickfixes to automatically extract this hardcoded string into a resource lookup.

Issue id: HardcodedText

Vendor: Android Open Source Project
Contact: https://groups.google.com/g/lint-dev
Feedback: https://issuetracker.google.com/issues/new?component=192708

Suggestion(s)

Fix Extract string resource

Figure 3-14

Currently, the only warning listed reads as follows:

```
Hardcoded string "Convert", should use @string resource
```

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

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

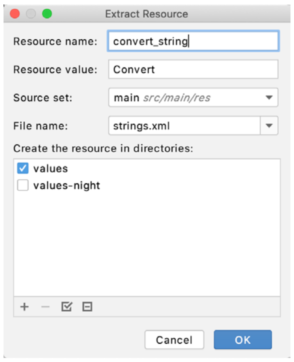

Extract Resource
Resource name: convert_string
Resource value: Convert
Source set: main src/main/res
File name: strings.xml
Create the resource in directories:
values
values-night
Cancel
OK

Figure 3-15

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

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

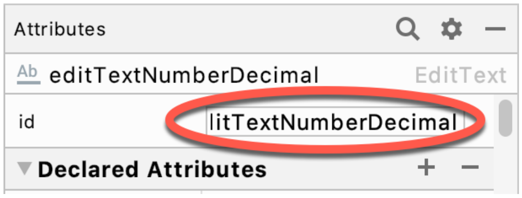


Figure 3-16

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

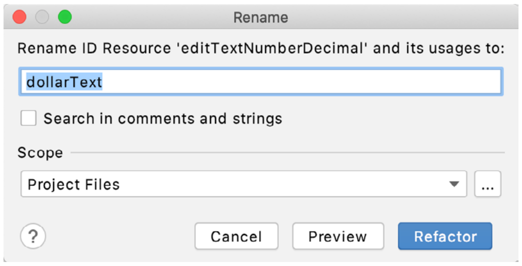

Rename
Rename ID Resource 'editTextNumberDecimal' and its usages to:
dollarText
Search in comments and strings
Scope
Project Files
Cancel
Preview
Refactor

Figure 3-17

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

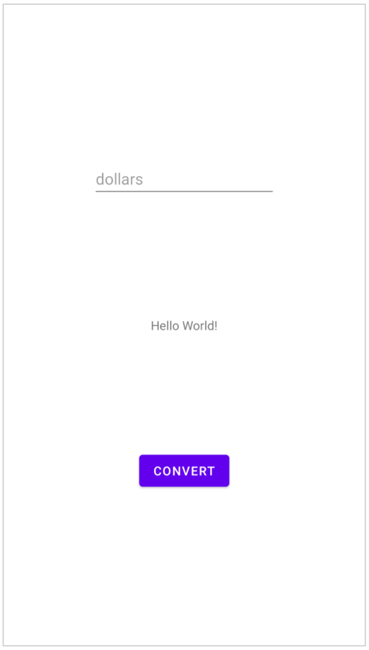


Figure 3-18

3.7 Reviewing the Layout and Resource Files

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

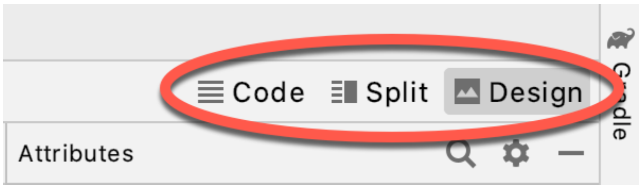
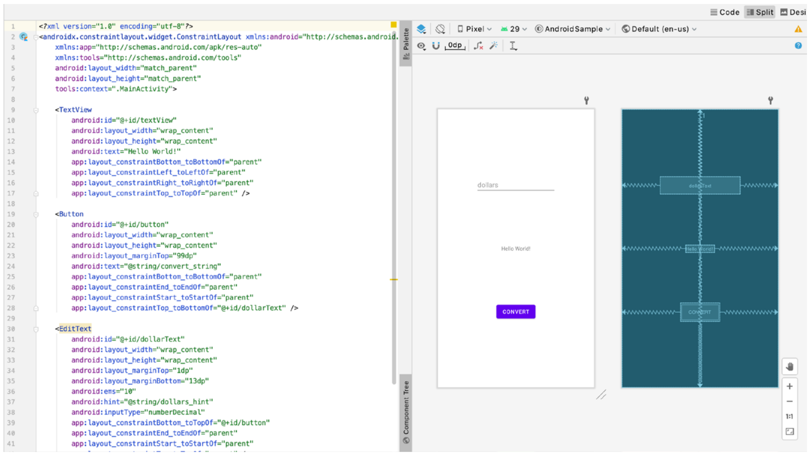


Figure 3-19

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



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

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="99dp"
        android:text="@string/convert_string"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/dollarText" />

    <EditText
        android:id="@+id/dollarText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="1dp"
        android:layout_marginBottom="13dp"
        android:ems="10"
        android:hint="@string/dollars_hint"
        android:inputType="numberDecimal"
        app:layout_constraintBottom_toTopOf="@+id/button"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
```

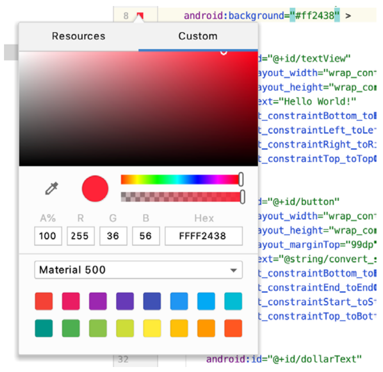
Figure 3-20

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

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

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

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



```
android:background="#ff2438" >

d="@+id/textView"
ayout_width="wrap_con
ayout_height="wrap_co
ext="Hello World!"
t_constraintBottom_to
t_constraintLeft_toLe
t_constraintRight_toR
t_constraintTop_toTop

d="@+id/button"
ayout_width="wrap_con
ayout_height="wrap_co
ayout_marginTop="99dp"
ext="@string/convert_
t_constraintBottom_to
t_constraintEnd_toEnd
t_constraintStart_toS
t_constraintTop_toBot

android:id="@+id/dollarText"
```

Figure 3-21

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

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

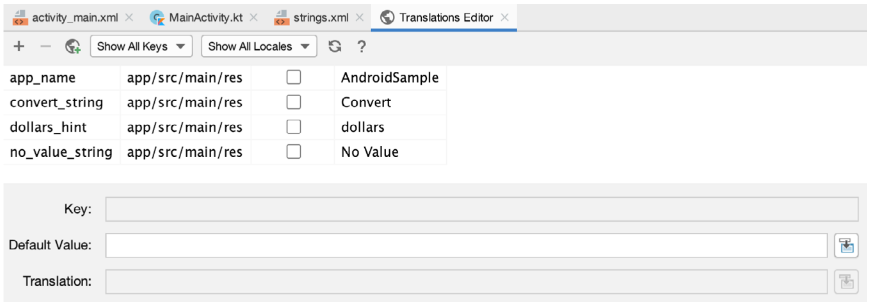
```
<resources>
    <string name="app_name">AndroidSample</string>
    <string name="convert_string">Convert</string>
    <string name="dollars_hint">dollars</string>
</resources>
```

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

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

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

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



app_name	app/src/main/res	☐	AndroidSample
convert_string	app/src/main/res	☐	Convert
dollars_hint	app/src/main/res	☐	dollars
no_value_string	app/src/main/res	☐	No Value

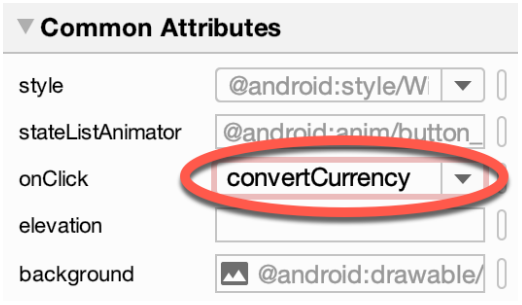
Key:
Default Value:
Translation:

Figure 3-22

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

3.8 Adding Interaction

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



style	@android:style/Wi
stateListAnimator	@android:anim/button_
onClick	convertCurrency
elevation	
background	@android:drawable/

Figure 3-23

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

```
package com.ebookfrenzy.androidsample;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;
.
```

```
.
import java.util.Locale;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void convertCurrency(View view) {

        EditText dollarText = findViewById(R.id.dollarText);
        TextView textView = findViewById(R.id.textView);

        if (!dollarText.getText().toString().equals("")) {

            float dollarValue = Float.parseFloat(dollarText.getText().toString());
            float euroValue = dollarValue * 0.85F;
            textView.setText(String.format(Locale.ENGLISH,"%.2f", euroValue));
        } else {
            textView.setText(R.string.no_value_string);
        }
    }
}
```

The method begins by obtaining references to the EditText and TextView objects by making a call to a method named *findViewById*, passing through the id assigned within the layout file. A check is then made to ensure that the user has entered a dollar value and if so, that value is extracted, converted from a String to a floating point value and converted to euros. Finally, the result is displayed on the TextView widget. If any of this is unclear, rest assured that these concepts will be covered in greater detail in later chapters.

3.9 Summary

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

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

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

Chapter 6

6. 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.

6.1 The Welcome Screen

The welcome screen (Figure 6-1) 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 bypass the welcome screen next time it is launched, automatically opening the previously active project.

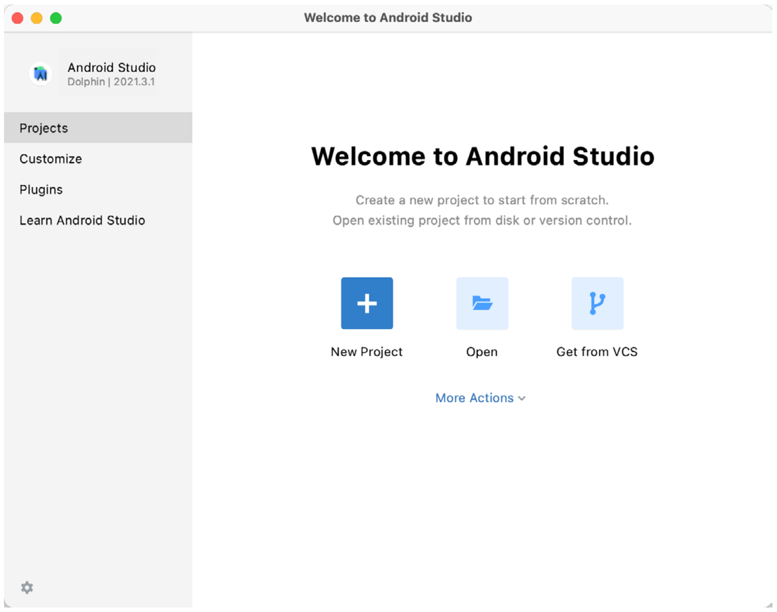

Welcome to Android Studio
Android Studio
Dolphin | 2021.3.1
Projects
Customize
Plugins
Learn Android Studio
Welcome to Android Studio
Create a new project to start from scratch.
Open existing project from disk or version control.
New Project
Open
Get from VCS
More Actions

Figure 6-1

In addition to a list of recent projects, the welcome screen provides options for performing tasks such as opening and creating projects along with access to projects currently under version control. In addition, the *Customize* screen provides options to change the theme and font settings used by both the IDE and the editor. Android Studio plugins may be viewed, installed, and managed using the *Plugins* option.

Additional options are available by clicking on the menu button as shown in Figure 6-2:

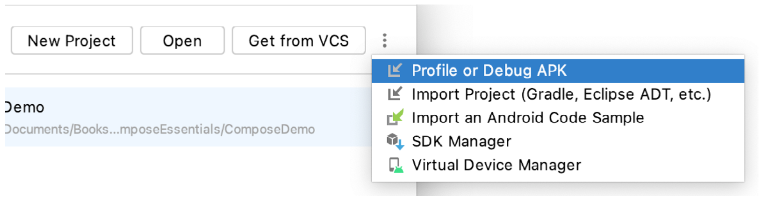
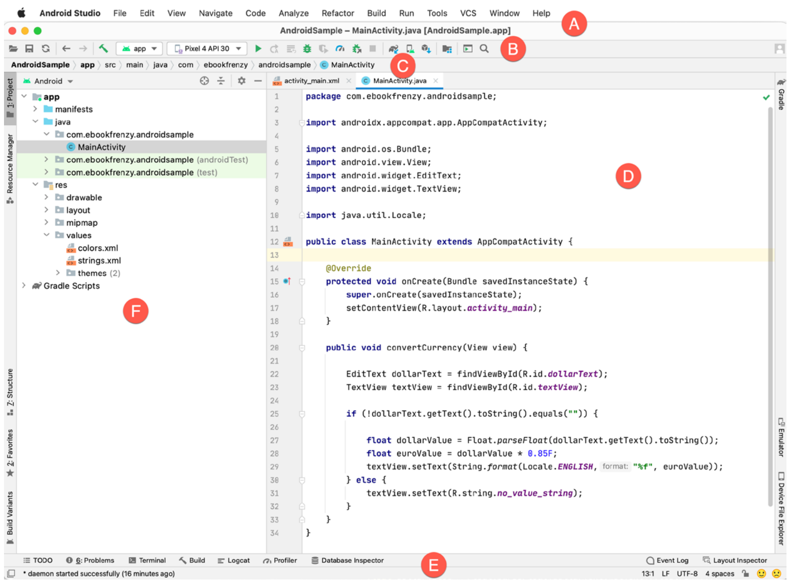

New Project
Open
Get from VCS
Demo
Documents/Books...mposeEssentials/ComposeDemo
Profile or Debug APK
Import Project (Gradle, Eclipse ADT, etc.)
Import an Android Code Sample
SDK Manager
Virtual Device Manager

Figure 6-2

6.2 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 6-3.


Android Studio File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
A
AndroidSample – MainActivity.java [AndroidSample.app]
B
AndroidSample app src main java com ebookfrenzy androidsample MainActivity
C
activity_main.xml
MainActivity.java
D
app
manifests
java
com.ebookfrenzy.androidsample
MainActivity
com.ebookfrenzy.androidsample (androidTest)
com.ebookfrenzy.androidsample (test)
res
drawable
layout
mipmap
values
colors.xml
strings.xml
themes (2)
Gradle Scripts
F
TODO 6: Problems Terminal Build Logcat Profiler Database Inspector
E
Event Log Layout Inspector
daemon started successfully (16 minutes ago)
13:1 LF UTF-8 4 spaces

```
package com.ebookfrenzy.androidsample;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

import java.util.Locale;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void convertCurrency(View view) {

        EditText dollarText = findViewById(R.id.dollarText);
        TextView textView = findViewById(R.id.textView);

        if (!dollarText.getText().toString().equals("")) {

            float dollarValue = Float.parseFloat(dollarText.getText().toString());
            float euroValue = dollarValue * 0.85F;
            textView.setText(String.format(Locale.ENGLISH, format: "%f", euroValue));
        } else {
            textView.setText(R.string.no_value_string);
        }
    }
}
```

Figure 6-3

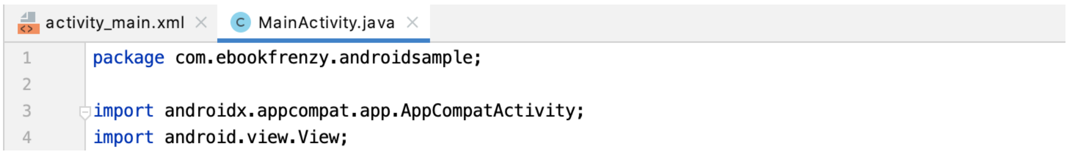
The various elements of the main window can be summarized as follows:

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 quick 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. If the toolbar is not visible, it can be displayed using the *View -> Appearance -> Toolbar* 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 sub-folders and files at that location ready for selection. Similarly, clicking on a class name displays a menu listing methods contained within that class. Select a method from the list to be taken to the corresponding location within the code editor. Hide and display this bar using the *View -> Appearance -> Navigation Bar* menu option.

D – Editor Window – The editor window displays the content of the file on which the developer is currently working. When multiple files are open, each file is represented by a tab located along the top edge of the editor as shown in Figure 6-4:



```
activity_main.xml ×   MainActivity.java ×
package com.ebookfrenzy.androidsample;

import androidx.appcompat.app.AppCompatActivity;
import android.view.View;
```

Figure 6-4

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 display 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 several 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 many tool windows available within the Android Studio environment.

6.3 The Tool Windows

In addition to the project view tool window, Android Studio also includes many other windows which, when enabled, are displayed along the bottom and sides of the main window. The tool window quick access menu can be displayed by hovering the mouse pointer over the button located in the far left-hand corner of the status bar (Figure 6-5) without clicking the mouse button.

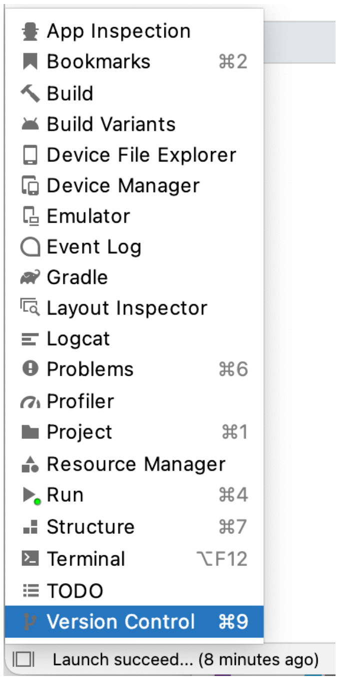


Figure 6-5

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 6-6) 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.

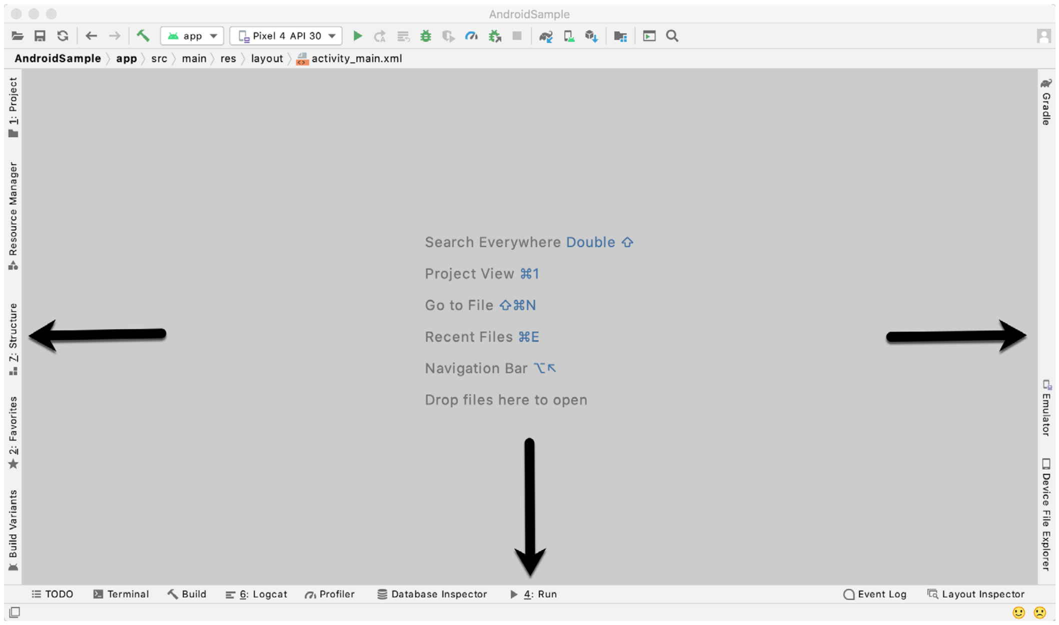


Figure 6-6

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 toolbars.

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 6-7 shows the settings menu for the Project 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.

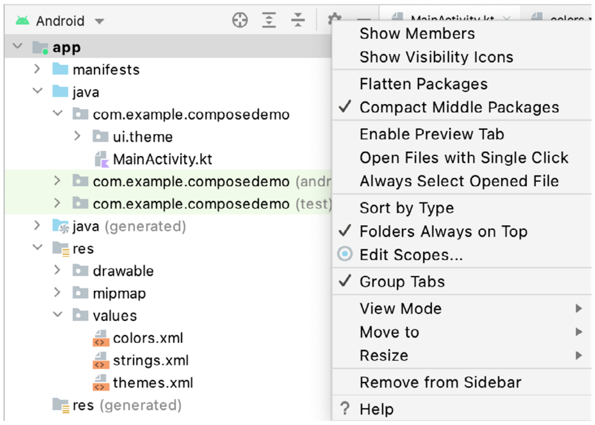


Figure 6-7

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:

- **App Inspector** - Provides access to the Database and Background Task inspectors. The Database Inspector allows you to inspect, query, and modify your app's databases while the app is running. The Background Task Inspector allows background worker tasks created using WorkManager to be monitored and managed.

- **Build** - The build tool window displays information about the build process while a project is being compiled and packaged and displays details of any errors encountered.

- **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).

- **Device File Explorer** – Available via the *View -> Tool Windows -> Device File Explorer* menu, this 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.

- **Device Manager** - Provides access to the Device Manager tool window where physical Android device connections and emulators may be added, removed, and managed.

- **Emulator** - Contains the AVD emulator if the option has been enabled to run the emulator in a tool window as outlined in the chapter entitled *"Creating an Android Virtual Device (AVD) in Android Studio"*.

- **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.

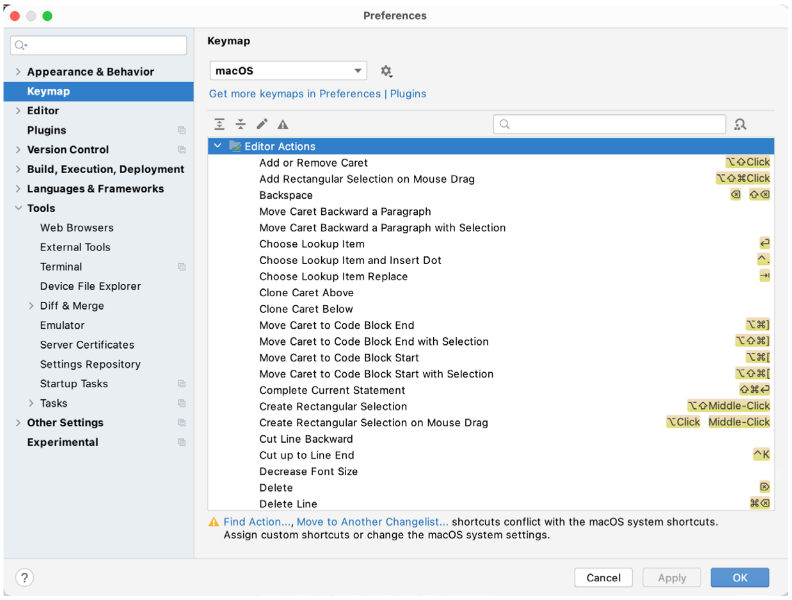
- **Favorites** – A variety of project items can be added to the favorites list. Right-clicking on a file in the project view, for example, provides access to an *Add to Favorites* menu option. Similarly, a method in a source file can be added as a favorite by right-clicking on it in the Structure tool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

- **Gradle** – The Gradle tool window provides a view of 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.

- **Layout Inspector** - Provides a visual 3D rendering of the hierarchy of components that make up a user interface layout.
- **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.
- **Problems** - A central location in which to view all of the current errors or warnings within the project. Double-clicking on an item in the problem list will take you to the problem file and location.
- **Profiler** – The Android Profiler tool window provides real-time monitoring and analysis tools for identifying performance issues within running apps, including CPU, memory and network usage. This option becomes available when an app is currently running.
- **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.
- **Resource Manager** - A tool for adding and managing resources and assets such as images, colors, and layout files contained with the project.
- **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.
- **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.
- **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.
- **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*.
- **Version Control** - This tool window is used when the project files are under source code version control, allowing access to Git repositories and code change history.

6.4 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. You may also list and modify the keyboard shortcuts by selecting the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS) and clicking on the Keymap entry as shown in Figure 6-8 below:



Preferences

Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Build, Execution, Deployment
Languages & Frameworks
Tools
Web Browsers
External Tools
Terminal
Device File Explorer
Diff & Merge
Emulator
Server Certificates
Settings Repository
Startup Tasks
Tasks
Other Settings
Experimental

Keymap

macOS

Get more keymaps in Preferences | Plugins

Editor Actions	
Add or Remove Caret	⌥⇧Click
Add Rectangular Selection on Mouse Drag	⌥⇧⌘Click
Backspace	⌫ ⇧⌫
Move Caret Backward a Paragraph	
Move Caret Backward a Paragraph with Selection	
Choose Lookup Item	↩
Choose Lookup Item and Insert Dot	^.
Choose Lookup Item Replace	⇥
Clone Caret Above	
Clone Caret Below	
Move Caret to Code Block End	⌥⌘]
Move Caret to Code Block End with Selection	⌥⇧⌘]
Move Caret to Code Block Start	⌥⌘[
Move Caret to Code Block Start with Selection	⌥⇧⌘[
Complete Current Statement	⇧⌘↩
Create Rectangular Selection	⌥⇧Middle-Click
Create Rectangular Selection on Mouse Drag	⌥Click Middle-Click
Cut Line Backward	
Cut up to Line End	^K
Decrease Font Size	
Delete	⌦
Delete Line	⌘⌫

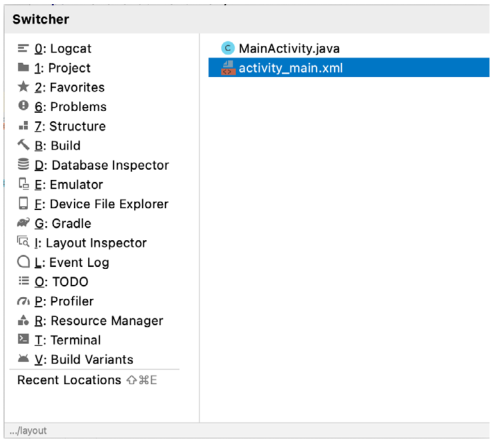
Find Action..., Move to Another Changelist... shortcuts conflict with the macOS system shortcuts. Assign custom shortcuts or change the macOS system settings.

Cancel Apply OK

Figure 6-8

6.5 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 6-9).



Switcher

0: Logcat
1: Project
2: Favorites
6: Problems
7: Structure
B: Build
D: Database Inspector
E: Emulator
F: Device File Explorer
G: Gradle
I: Layout Inspector
L: Event Log
O: TODO
P: Profiler
R: Resource Manager
T: Terminal
V: Build Variants
Recent Locations ⇧⌘E

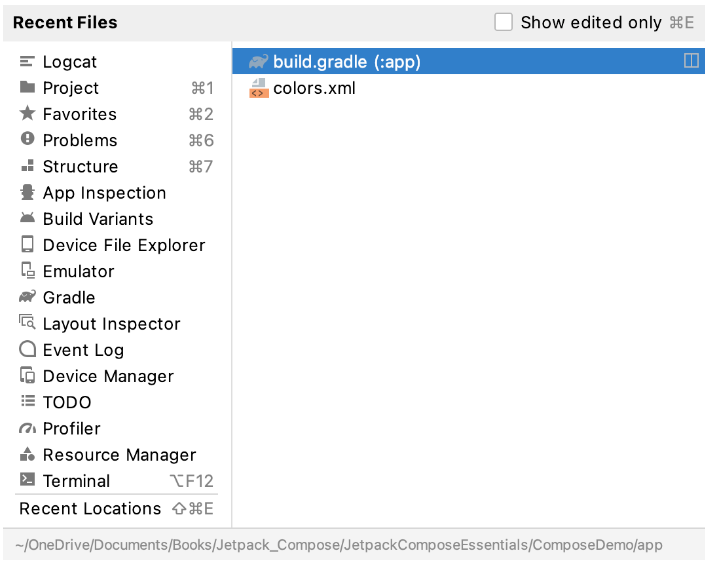
MainActivity.java
activity_main.xml

.../layout

Figure 6-9

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 6-10). 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 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.


Recent Files   Show edited only ⌘E

Logcat
Project ⌘1
Favorites ⌘2
Problems ⌘6
Structure ⌘7
App Inspection
Build Variants
Device File Explorer
Emulator
Gradle
Layout Inspector
Event Log
Device Manager
TODO
Profiler
Resource Manager
Terminal ⌥F12
Recent Locations ⇧⌘E

build.gradle (:app)
colors.xml

~/OneDrive/Documents/Books/Jetpack_Compose/JetpackComposeEssentials/ComposeDemo/app

Figure 6-10

6.6 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 & Behavior* option followed by *Appearance* 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 Light, IntelliJ, Windows, High Contrast, and Darcula. Figure 6-11 shows an example of the main window with the Darcula theme selected:

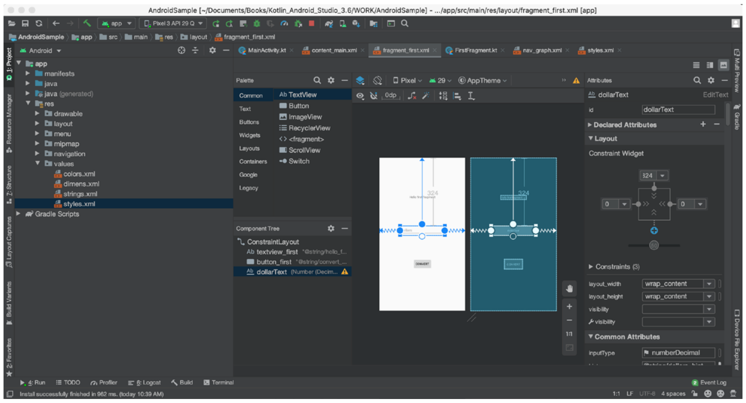
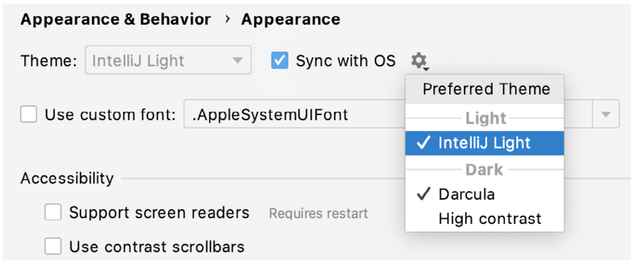


Figure 6-11

To synchronize the Android Studio theme with the operating system light and dark mode setting, enable the *Sync with OS* option and use the drop-down menu to control which theme to use for each mode:



Appearance & Behavior › Appearance

Theme: IntelliJ Light ✓ Sync with OS

Use custom font: .AppleSystemUIFont

Accessibility

Support screen readers Requires restart

Use contrast scrollbars

Figure 6-12

6.7 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 that 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.

Chapter 8

8. The Basics of the Android Studio Code Editor

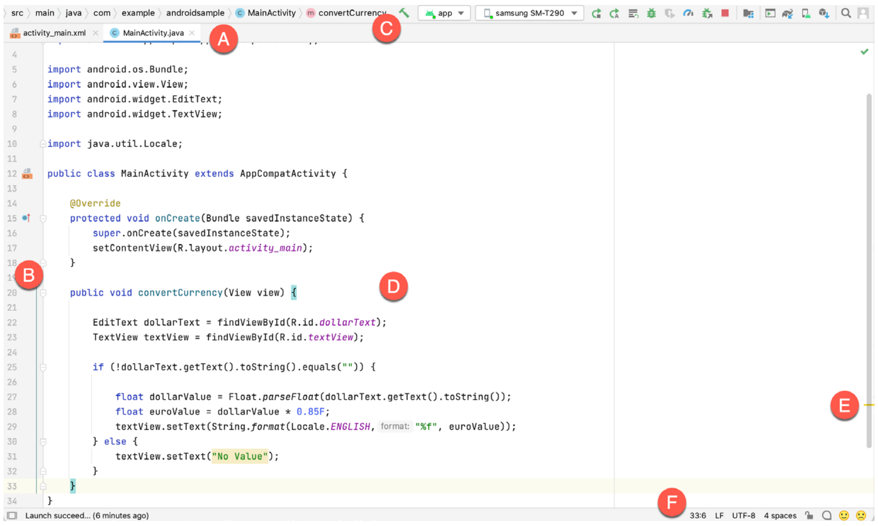
Developing applications for Android involves a considerable amount of programming work which, by definition, involves typing, reviewing and modifying lines of code. It should come as no surprise that the majority of a developer's time spent using Android Studio will typically involve editing code within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


src 〉 main 〉 java 〉 com 〉 example 〉 androidsample 〉 MainActivity 〉 convertCurrency
app
samsung SM-T290
activity_main.xml
MainActivity.java
A
B
C
D
E
F

```
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

import java.util.Locale;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void convertCurrency(View view) {

        EditText dollarText = findViewById(R.id.dollarText);
        TextView textView = findViewById(R.id.textView);

        if (!dollarText.getText().toString().equals("")) {

            float dollarValue = Float.parseFloat(dollarText.getText().toString());
            float euroValue = dollarValue * 0.85F;
            textView.setText(String.format(Locale.ENGLISH, format: "%f", euroValue));
        } else {
            textView.setText("No Value");
        }
    }
}
```

Launch succeed... (6 minutes ago)   33:6 LF UTF-8 4 spaces

Figure 8-1

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

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

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

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

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

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

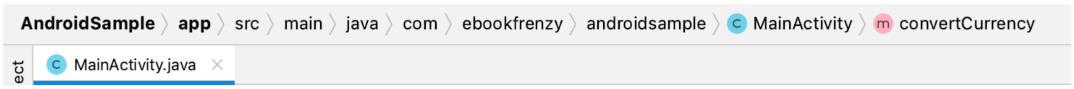
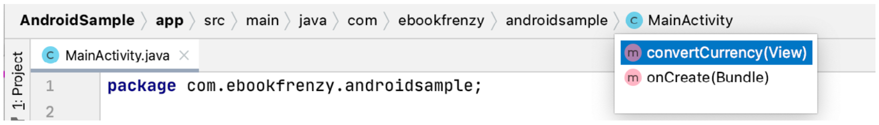

AndroidSample 〉 app 〉 src 〉 main 〉 java 〉 com 〉 ebookfrenzy 〉 androidsample 〉 MainActivity 〉 convertCurrency
ect
MainActivity.java

Figure 8-2

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

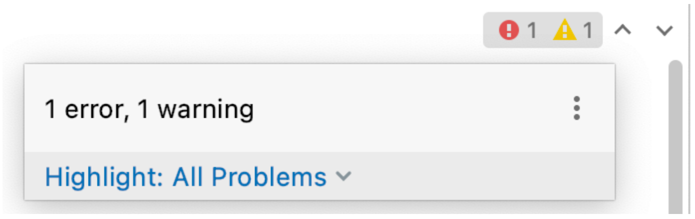

AndroidSample 〉 app 〉 src 〉 main 〉 java 〉 com 〉 ebookfrenzy 〉 androidsample 〉 MainActivity
1: Project
MainActivity.java
convertCurrency(View)
onCreate(Bundle)

```
package com.ebookfrenzy.androidsample;
```

Figure 8-3

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

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



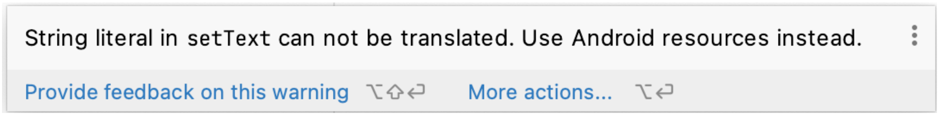
1 error, 1 warning

Highlight: All Problems

Figure 8-4

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

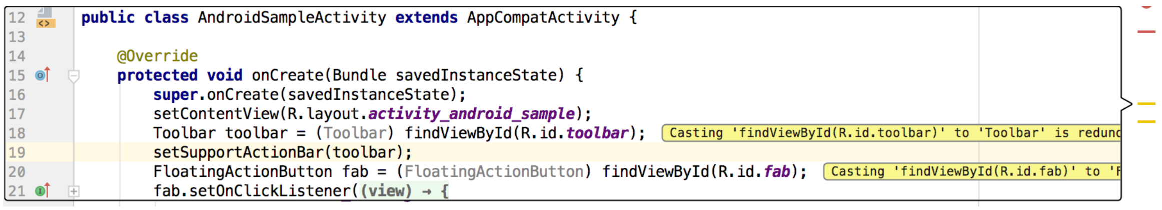
The sidebar also displays markers at the locations where issues have been detected using the same color coding. Hovering the mouse pointer over a marker when the line of code is visible in the editor area will display a popup containing a description of the issue:



String literal in setText can not be translated. Use Android resources instead.

Figure 8-5

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



```
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);  Casting 'findViewById(R.id.toolbar)' to 'Toolbar' is redund
        setSupportActionBar(toolbar);
        FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);  Casting 'findViewById(R.id.fab)' to 'F
        fab.setOnClickListener((view) → {
```

Figure 8-6

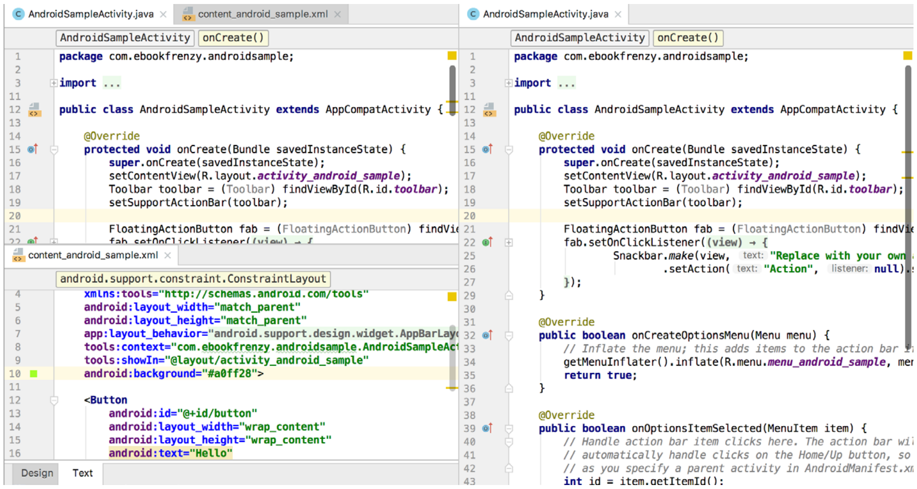
It is also worth noting that the lens overlay is not limited to warnings and errors in the sidebar. Hovering over any part of the sidebar will result in a lens appearing containing the code present at that location within the source file.

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

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

8.2 Splitting the Editor Window

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


AndroidSampleActivity.java
content_android_sample.xml
AndroidSampleActivity
onCreate()
android.support.constraint.ConstraintLayout
Design
Text

```
package com.ebookfrenzy.androidsample;

import ...

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) findVie
```

```
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="android.support.design.widget.AppBarLay
    tools:context="com.ebookfrenzy.androidsample.AndroidSampleAct
    tools:showIn="@layout/activity_android_sample"
    android:background="#a0ff28">

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

```
package com.ebookfrenzy.androidsample;

import ...

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) findVi
        fab.setOnClickListener((view) → {
                Snackbar.make(view, text: "Replace with your own
                        .setAction( text: "Action", listener: null).
        });
    }

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

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

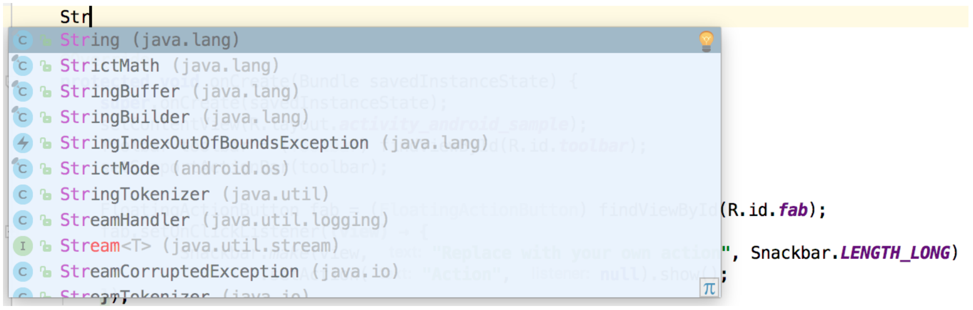
Figure 8-7

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

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

8.3 Code Completion

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



```
    Str
```

String (java.lang)
StrictMath (java.lang)
StringBuffer (java.lang)
StringBuilder (java.lang)
StringIndexOutOfBoundsException (java.lang)
StrictMode (android.os)
StringTokenizer (java.util)
StreamHandler (java.util.logging)
Stream<T> (java.util.stream)
StreamCorruptedException (java.io)
π

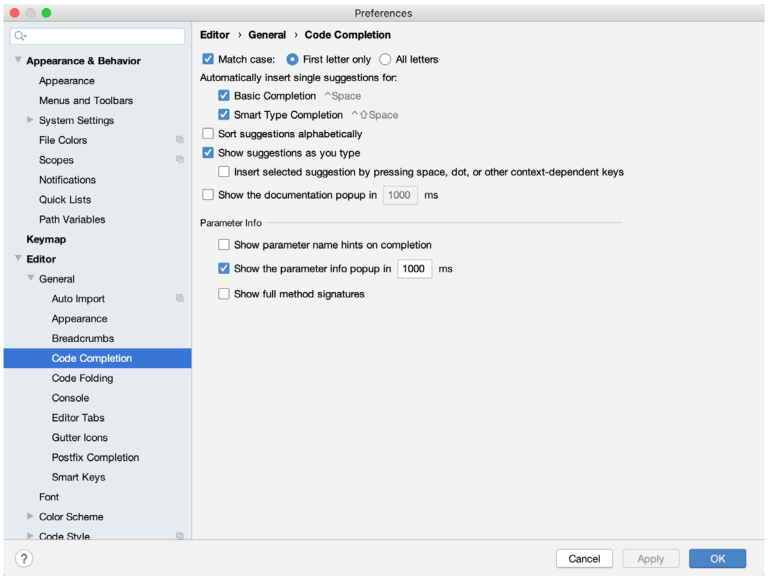
Figure 8-8

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

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

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

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



Editor › General › Code Completion

[✓] Match case: [● First letter only] ○ All letters

Automatically insert single suggestions for:

[✓] Basic Completion ^Space

[✓] Smart Type Completion ^⇧Space

[ ] Sort suggestions alphabetically

[✓] Show suggestions as you type

[ ] Insert selected suggestion by pressing space, dot, or other context-dependent keys

[ ] Show the documentation popup in 1000 ms

Parameter Info

[ ] Show parameter name hints on completion

[✓] Show the parameter info popup in 1000 ms

[ ] Show full method signatures

Figure 8-9

8.4 Statement Completion

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

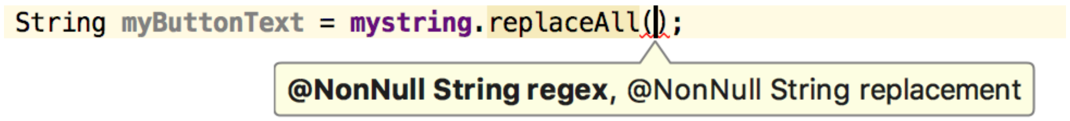
```
myMethod()
```

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

```
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:

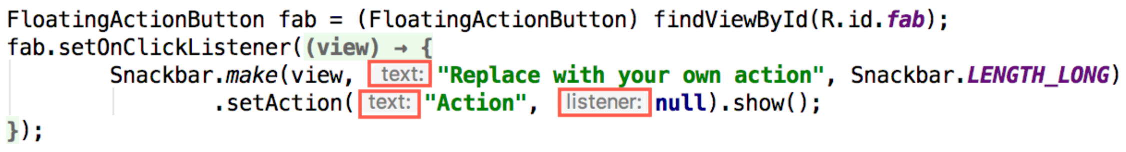


```
String myButtonText = mystring.replaceAll();
```

Figure 8-10

8.6 Parameter Name Hints

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



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

Figure 8-11

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

8.7 Code Generation

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

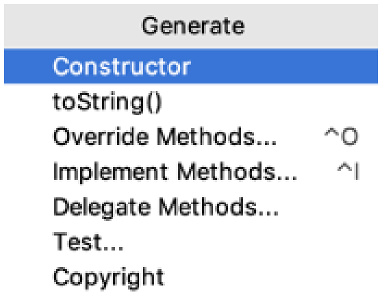
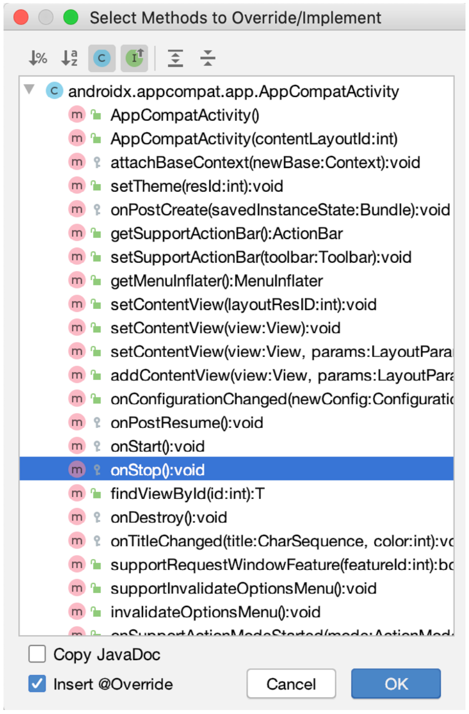


Figure 8-12

For the purposes of an example, consider a situation where we want to be notified when an Activity in our project is about to be destroyed by the operating system. As will be outlined in a later chapter of this book, this can be achieved by overriding the *onStop()* lifecycle method of the Activity superclass. To have Android Studio generate a stub method for this, simply select the *Override Methods…* option from the code generation list and select the *onStop()* method from the resulting list of available methods:



Select Methods to Override/Implement

```
androidx.appcompat.app.AppCompatActivity
    AppCompatActivity()
    AppCompatActivity(contentLayoutId:int)
    attachBaseContext(newBase:Context):void
    setTheme(resId:int):void
    onPostCreate(savedInstanceState:Bundle):void
    getSupportActionBar():ActionBar
    setSupportActionBar(toolbar:Toolbar):void
    getMenuInflater():MenuInflater
    setContentView(layoutResID:int):void
    setContentView(view:View):void
    setContentView(view:View, params:LayoutPara
    addContentView(view:View, params:LayoutPara
    onConfigurationChanged(newConfig:Configuratio
    onPostResume():void
    onStart():void
    onStop():void
    findViewById(id:int):T
    onDestroy():void
    onTitleChanged(title:CharSequence, color:int):vo
    supportRequestWindowFeature(featureId:int):bo
    supportInvalidateOptionsMenu():void
    invalidateOptionsMenu():void
```

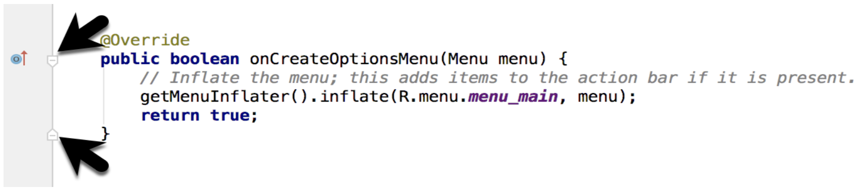
Figure 8-13

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

```
@Override
protected void onStop() {
    super.onStop();
}
```

8.8 Code Folding

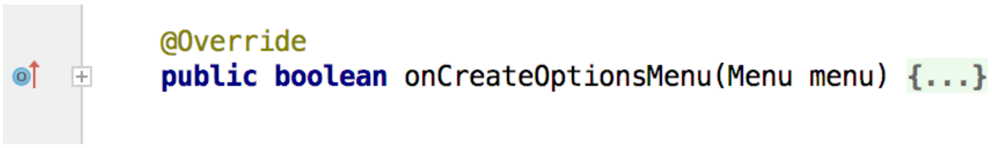
Once a source code file reaches a certain size, even the most carefully formatted and well organized code can become overwhelming and difficult to navigate. Android Studio takes the view that it is not always necessary to have the content of every code block visible at all times. Code navigation can 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 8-14, for example, highlights the start and end markers for a method declaration which is not currently folded:



```
@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_main, menu);
    return true;
}
```

Figure 8-14

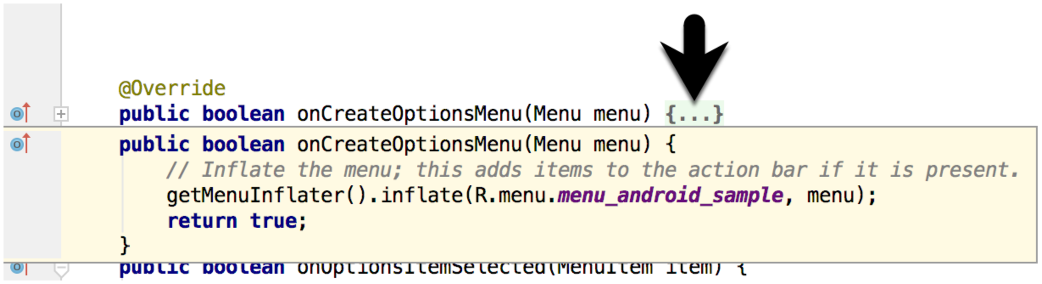
Clicking on either of these markers will fold the statement such that only the signature line is visible as shown in Figure 8-15:



```
@Override
public boolean onCreateOptionsMenu(Menu menu) {...}
```

Figure 8-15

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

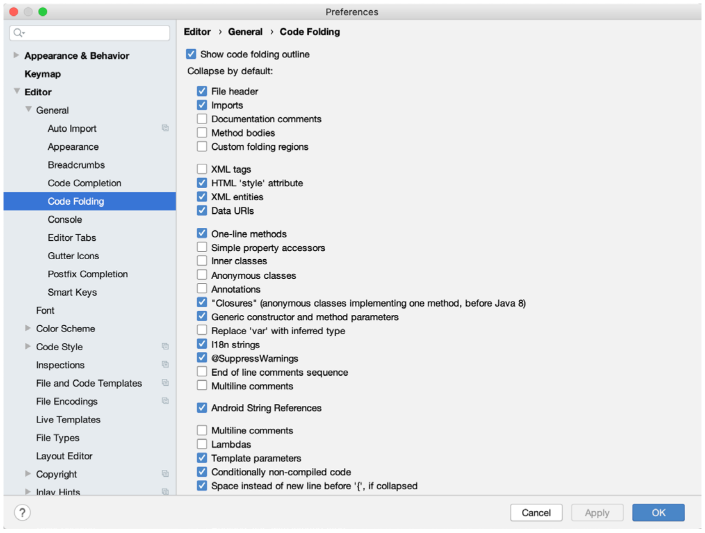


```
@Override
public boolean onCreateOptionsMenu(Menu menu) {...}
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_android_sample, menu);
    return true;
}
public boolean onOptionsItemSelected(MenuItem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

Figure 8-16

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

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



Editor › General › Code Folding

✓ Show code folding outline

Collapse by default:

- ✓ File header
- ✓ Imports
- ☐ Documentation comments
- ☐ Method bodies
- ☐ Custom folding regions

- ☐ XML tags
- ✓ HTML 'style' attribute
- ✓ XML entities
- ✓ Data URIs

- ✓ One-line methods
- ☐ Simple property accessors
- ☐ Inner classes
- ☐ Anonymous classes
- ☐ Annotations
- ✓ "Closures" (anonymous classes implementing one method, before Java 8)
- ✓ Generic constructor and method parameters
- ☐ Replace 'var' with inferred type
- ✓ I18n strings
- ✓ @SuppressWarnings
- ☐ End of line comments sequence
- ☐ Multiline comments

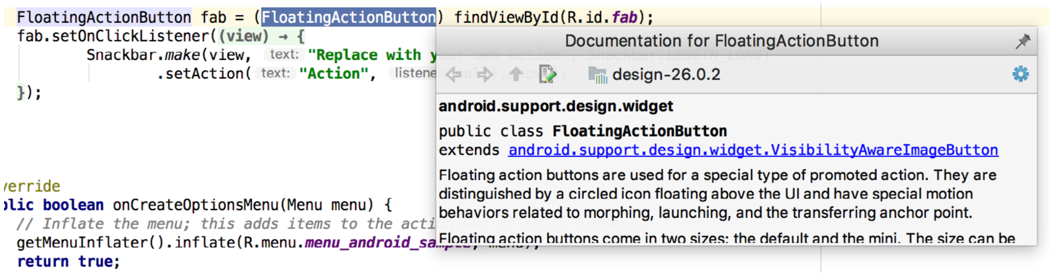
- ✓ Android String References

- ☐ Multiline comments
- ☐ Lambdas
- ✓ Template parameters
- ✓ Conditionally non-compiled code
- ✓ Space instead of new line before '{', if collapsed

Figure 8-17

8.9 Quick Documentation Lookup

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



```
FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener((view) → {
        Snackbar.make(view,  text: "Replace with y
                .setAction( text: "Action",  listene
});


/erride
)lic boolean onCreateOptionsMenu(Menu menu) {
 // Inflate the menu; this adds items to the acti
 getMenuInflater().inflate(R.menu.menu_android_sa
 return true;
```

Documentation for FloatingActionButton

design-26.0.2

android.support.design.widget

`public class` **`FloatingActionButton`**
`extends` `android.support.design.widget.VisibilityAwareImageButton`

Floating action buttons are used for a special type of promoted action. They are distinguished by a circled icon floating above the UI and have special motion behaviors related to morphing, launching, and the transferring anchor point.

Floating action buttons come in two sizes: the default and the mini. The size can be

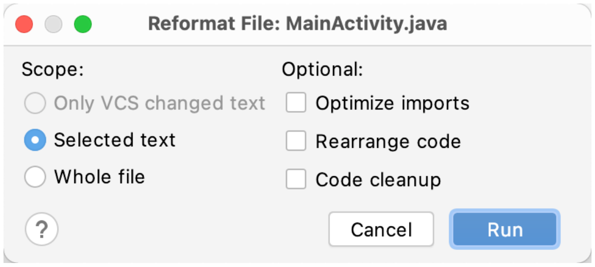
Figure 8-18

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

8.10 Code Reformatting

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

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



Reformat File: MainActivity.java

Scope:
- Only VCS changed text
- [✓ Selected text]
- Whole file

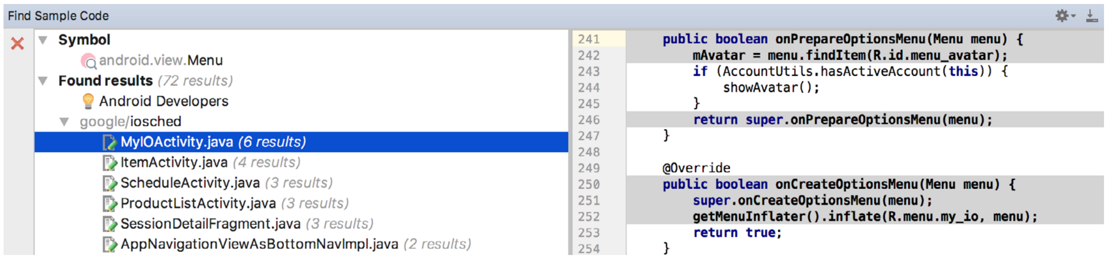
Optional:
- Optimize imports
- Rearrange code
- Code cleanup

Figure 8-19

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

8.11 Finding Sample Code

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


Find Sample Code

- Symbol
  - android.view.Menu
- Found results (72 results)
  - Android Developers
  - google/iosched
    - MyIOActivity.java (6 results)
    - ItemActivity.java (4 results)
    - ScheduleActivity.java (3 results)
    - ProductListActivity.java (3 results)
    - SessionDetailFragment.java (3 results)
    - AppNavigationViewAsBottomNavImpl.java (2 results)

```
    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;
    }
```

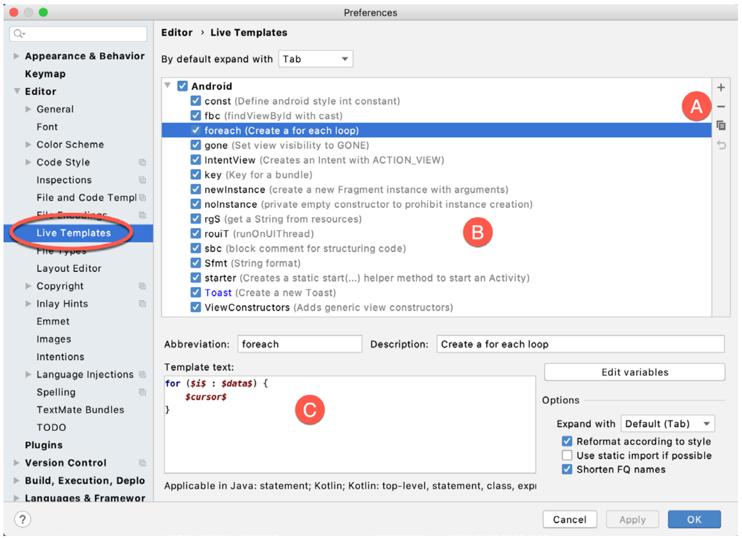
Figure 8-20

8.12 Live Templates

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

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

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



Editor › Live Templates

By default expand with: Tab

- ✓ Android
  - ✓ const (Define android style int constant)
  - ✓ fbc (findViewById with cast)
  - ✓ foreach (Create a for each loop)
  - ✓ gone (Set view visibility to GONE)
  - ✓ IntentView (Creates an Intent with ACTION_VIEW)
  - ✓ key (Key for a bundle)
  - ✓ newInstance (create a new Fragment instance with arguments)
  - ✓ noInstance (private empty constructor to prohibit instance creation)
  - ✓ rgS (get a String from resources)
  - ✓ rouiT (runOnUIThread)
  - ✓ sbc (block comment for structuring code)
  - ✓ Sfmt (String format)
  - ✓ starter (Creates a static start(...) helper method to start an Activity)
  - ✓ Toast (Create a new Toast)
  - ✓ ViewConstructors (Adds generic view constructors)

Abbreviation: foreach   Description: Create a for each loop

Template text:

```
for ($i$ : $data$) {
    $cursor$
}
```

Applicable in Java: statement; Kotlin; Kotlin: top-level, statement, class, expı

Options

Expand with: Default (Tab)
- ✓ Reformat according to style
- ☐ Use static import if possible
- ✓ Shorten FQ names

Figure 8-21

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

8.13 Summary

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

Chapter 12

12. Understanding Android Application and Activity Lifecycles

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

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

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

12.1 Android Applications and Resource Management

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

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

12.2 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 12-1, a process can be in one of the following five states at any given time:

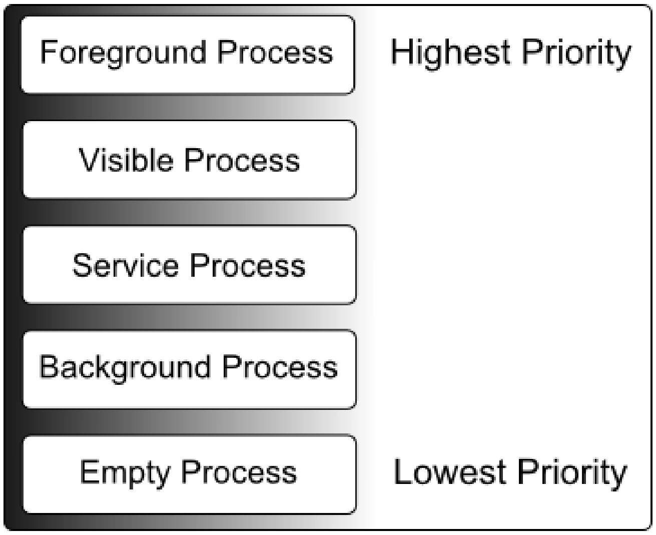


Figure 12-1

12.2.1 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.

12.2.2 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.

12.2.3 Service Process

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

12.2.4 Background Process

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

12.2.5 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.

12.3 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.

12.4 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*.

12.5 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 12-2.

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

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

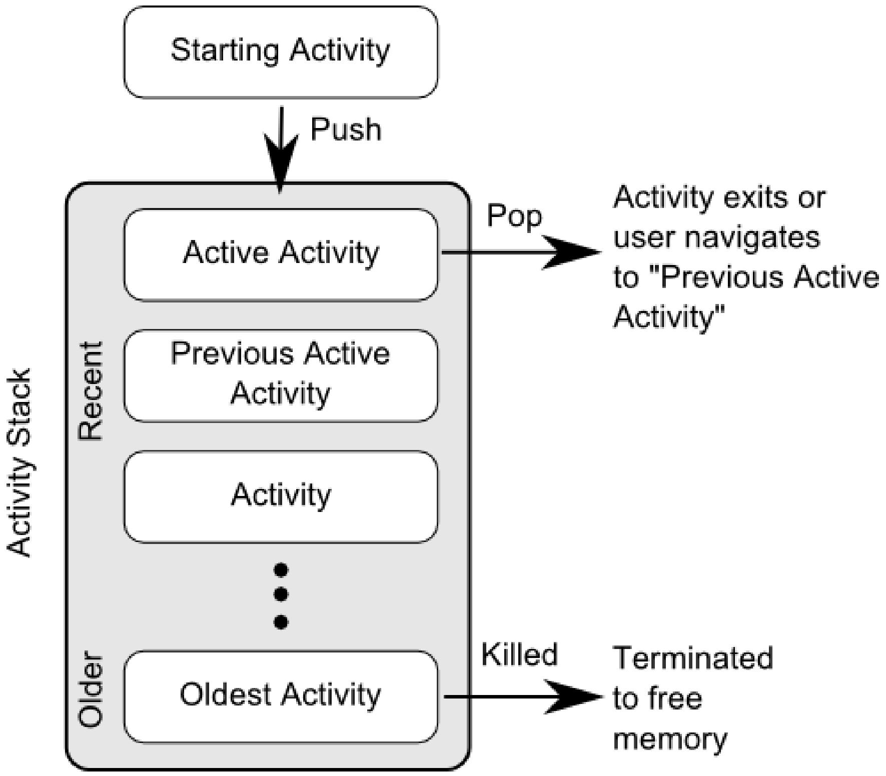


Figure 12-2

12.6 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.

12.7 Configuration Changes

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

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

12.8 Handling State Change

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

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

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

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

12.9 Summary

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

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

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

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

Chapter 13

13. 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 and fragments 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 does not, however, mean that the app cannot react to those changes and take appropriate actions.

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*.

13.1 New vs. Old Lifecycle Techniques

Up until recently, there was a standard way to build lifecycle awareness into an app. This is the approach covered in this chapter and involves implementing a set of methods (one for each lifecycle state) within an activity or fragment instance that get called by the operating system when the lifecycle status of that object changes. This approach has remained unchanged since the early years of the Android operating system and, while still a viable option today, it does have some limitations which will be explained later in this chapter.

With the introduction of the lifecycle classes with the Jetpack Android Architecture Components, a better approach to lifecycle handling is now available. This modern approach to lifecycle management (together with the Jetpack components and architecture guidelines) will be covered in detail in later chapters. It is still important, however, to understand the traditional lifecycle methods for a couple of reasons. First, as an Android developer you will not be completely insulated from the traditional lifecycle methods and will still make use of some of them. More importantly, understanding the older way of handling lifecycles will provide a good knowledge foundation on which to begin learning the new approach later in the book.

13.2 The Activity and Fragment Classes

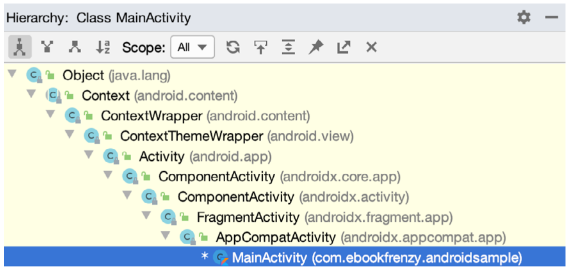
With few exceptions, activities and fragments in an application are created as subclasses of the Android AppCompatActivity class and Fragment classes respectively.

Consider, for example, the *AndroidSample* project created in *"Creating an Example Android App in Android Studio"* and subsequently converted to use view binding. Load this project into the Android Studio environment and locate the *MainActivity.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.example.androidsample;

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

import java.util.Locale;
import com.example.androidsample.databinding.ActivityMainBinding;

public class MainActivity extends AppCompatActivity {

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        View view = binding.getRoot();
        setContentView(view);
    }

    public void convertCurrency(View view) {

        if (!binding.dollarText.getText().toString().equals("")) {

            float dollarValue = Float.parseFloat(
                          binding.dollarText.getText().toString());
            float euroValue = dollarValue * 0.85F;
            binding.textView.setText(
                          String.format(Locale.ENGLISH,"%f", euroValue));
        } else {
            binding.textView.setText(R.string.no_value_string);
        }
    }
}
```

When the project was created, we instructed Android Studio also to create an initial activity named *MainActivity.java* As is evident from the above code, the MainActivity 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 *MainActivity.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 13-1, AppCompatActivity is clearly subclassed from the FragmentActivity class which is itself ultimately a subclass of the Activity class:


Hierarchy: Class MainActivity

Scope: All

```
Object (java.lang)
  Context (android.content)
    ContextWrapper (android.content)
      ContextThemeWrapper (android.view)
        Activity (android.app)
          ComponentActivity (androidx.core.app)
            ComponentActivity (androidx.activity)
              FragmentActivity (androidx.fragment.app)
                AppCompatActivity (androidx.appcompat.app)
                  * MainActivity (com.ebookfrenzy.androidsample)
```

Figure 13-1

The Activity and Fragment classes contain a range of methods that are intended to be called by the Android runtime to notify the object when its state is changing. For the purposes of this chapter, we will refer to these as the *lifecycle methods.* An activity or fragment class simply needs to *override* these methods and implement the necessary functionality within them 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 *MainActivity* class. In a later section we will explore in detail both *onCreate()* and the other relevant lifecycle methods of the Activity and Fragment classes.

13.3 Dynamic State vs. Persistent State

A key objective of 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.

13.4 The Android Lifecycle Methods

As previously explained, the Activity and Fragment classes contain a number of lifecycle methods which act as event handlers when the state of an instance changes. The primary methods supported by the Android Activity and Fragment 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.

The following lifecycle methods only apply to the Fragment class:

- **onAttach()** - Called when the fragment is assigned to an activity.
- **onCreateView()** - Called to create and return the fragment's user interface layout view hierarchy.
- **onViewCreated()** - Called after *onCreateView()* returns.
- **onViewStatusRestored()** - The fragment's saved view hierarchy has been restored.

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, 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 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. 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"*.

13.5 Lifetimes

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

- **Entire Lifetime** –The term "entire lifetime" is used to describe everything that takes place between the initial call to the *onCreate()* method and the call to *onDestroy()* prior to the object terminating.

- **Visible Lifetime** – Covers the periods of execution between the call to *onStart()* and *onStop()*. During this period the activity or fragment is visible to the user though may not be the object 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 or fragment 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 13-2:

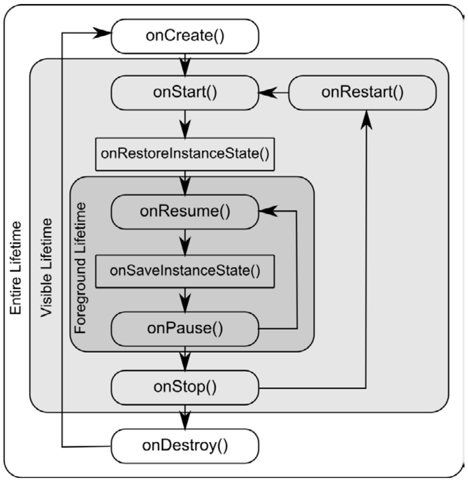


Figure 13-2

13.6 Foldable Devices and Multi-Resume

As discussed previously, an activity is considered to be in the resumed state when it has moved to the foreground and is the activity with which the user is currently interacting. On standard devices an app can have one activity in the resumed state at any one time and all other activities are likely to be in the paused or stopped state.

For some time now, Android has included multi-window support, allowing multiple activities to appear simultaneously in either split-screen or freeform configurations. Although originally used primarily on large screen tablet devices, this feature is likely to become more popular with the introduction of foldable devices.

On devices running Android 10 and on which multi-window support is enabled (as will be the case for most foldables), it will be possible for multiple app activities to be in the resumed state at the same time (a concept referred to as *multi-resume*) allowing those visible activities to continue functioning (for example streaming content or updating visual data) even when another activity currently has focus. Although multiple activities can be in the resumed state, only one of these activities will be considered to the *topmost resumed activity* (in other words, the activity with which the user most recently interacted).

An activity can receive notification that it has gained or lost the topmost resumed status by implementing the *onTopResumedActivityChanged()* callback method.

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

13.8 Lifecycle Method Limitations

As discussed at the start of this chapter, lifecycle methods have been in use for many years and, until recently, were the only mechanism available for handling lifecycle state changes for activities and fragments. There are, however, shortcomings to this approach.

One issue with the lifecycle methods is that they do not provide an easy way for an activity or fragment to find out its current lifecycle state at any given point during app execution. Instead the object would need to track the state internally, or wait for the next lifecycle method call.

Also, the methods do not provide a simple way for one object to observe the lifecycle state changes of other objects within an app. This is a serious consideration since many other objects within an app can potentially be impacted by a lifecycle state change in a given activity or fragment.

The lifecycle methods are also only available on subclasses of the Fragment and Activity classes. It is not possible, therefore, to build custom classes that are truly lifecycle aware.

Finally, the lifecycle methods result in most of the lifecycle handling code being written within the activity or fragment which can lead to complex and error prone code. Ideally, much of this code should reside in the other classes that are impacted by the state change. An app that streams video, for example, might include a class designed specifically to manage the incoming stream. If the app needs to pause the stream when the main activity is stopped, the code to do so should reside in the streaming class, not the main activity.

All of these problems and more are resolved by using *lifecycle-aware* components, a topic which will be covered starting with the chapter entitled *"Modern Android App Architecture with Jetpack"*.

13.9 Summary

All activities are derived from the Android *Activity* class which, in turn, contains a number of lifecycle methods that are designed to be called by the runtime system when the state of an activity changes. Similarly, the Fragment class contains a number of comparable methods. By overriding these methods, activities and fragments can respond to state changes and, where necessary, take steps to save and restore the current state of both the activity and the application. Lifecycle 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.

Although lifecycle methods have a number of limitations that can be avoided by making use of lifecycle-aware components, an understanding of these methods is important to fully understand the new approaches to lifecycle management covered later in this book.

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.

Chapter 16

16. 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.

16.1 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.

16.2 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 *widgets* 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.

16.3 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 value 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.

16.4 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 16-1:

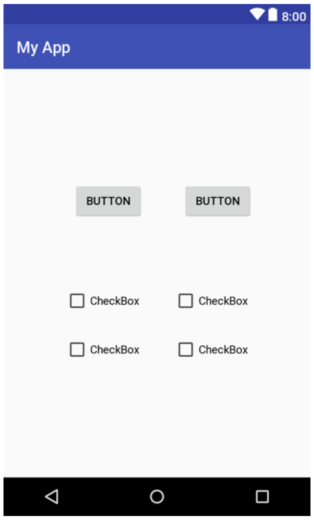


Figure 16-1

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 16-2 shows an alternative view of the user interface, this time highlighting the presence of the layout views in relation to the child views:

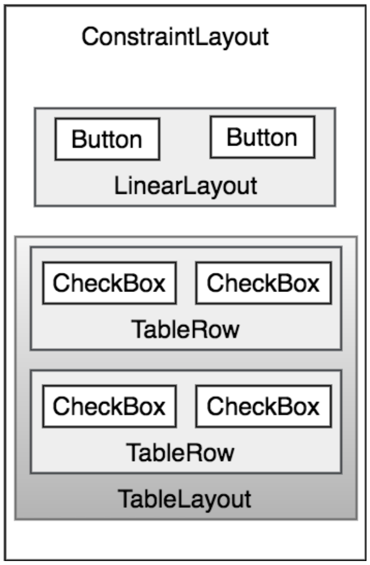


Figure 16-2

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 16-3:

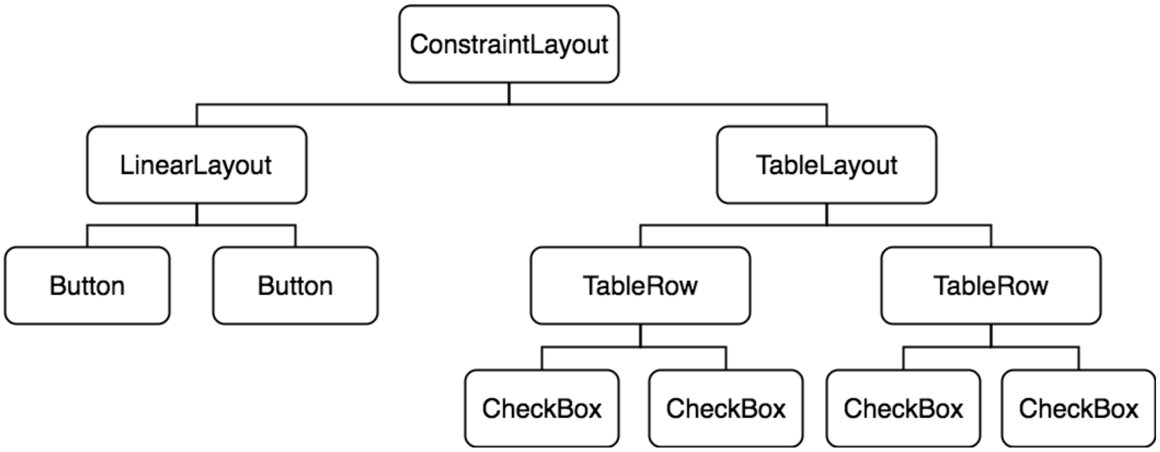


Figure 16-3

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 16-1. 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.

16.5 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.

16.6 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. If 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.

Chapter 17

17. 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.

17.1 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. To see these differences within the layout editor, use the View Options menu to enable Show System UI as shown in Figure 17-1 below:

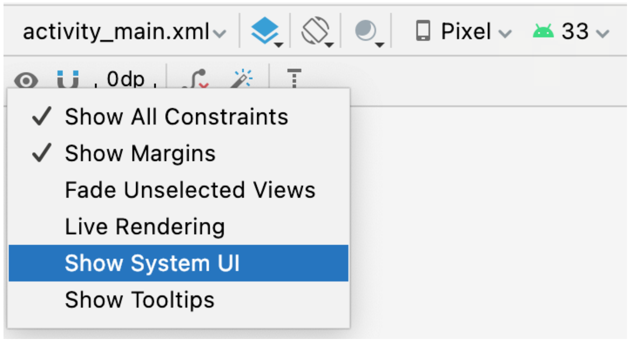


Figure 17-1

The Empty Activity template creates a single layout file consisting of a ConstraintLayout manager instance containing a TextView object as shown in Figure 17-2:

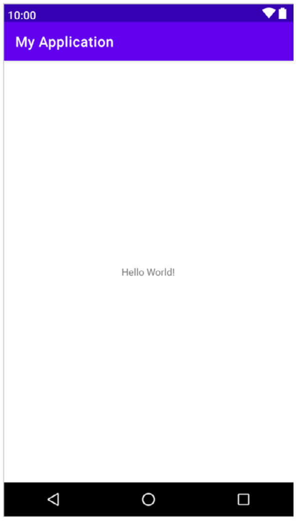
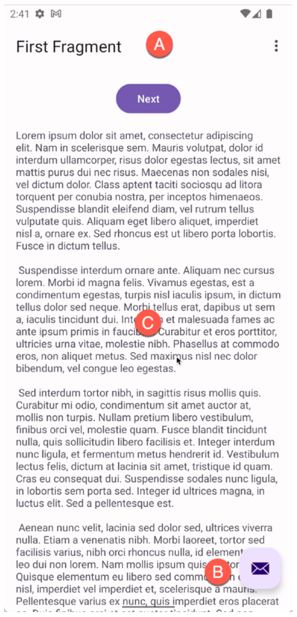


Figure 17-2

The Basic Activity, on the other hand, consists of multiple layout files. The top level layout file has a CoordinatorLayout as the root view, a configurable app bar (which contains a tool bar) that appears across the top of the device screen (marked A in Figure 17-3) and a floating action button (the email button marked B). In addition to these items, the *activity_main.xml* layout file contains a reference to a second file named *content_main.xml* containing the content layout (marked C):



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam in scelerisque sem. Mauris volutpat, dolor id interdum ullamcorper, risus dolor egestas lectus, sit amet mattis purus dui nec risus. Maecenas non sodales nisi, vel dictum dolor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Suspendisse blandit eleifend diam, vel rutrum tellus vulputate quis. Aliquam eget libero aliquet, imperdiet nisl a, ornare ex. Sed rhoncus est ut libero porta lobortis. Fusce in dictum tellus.

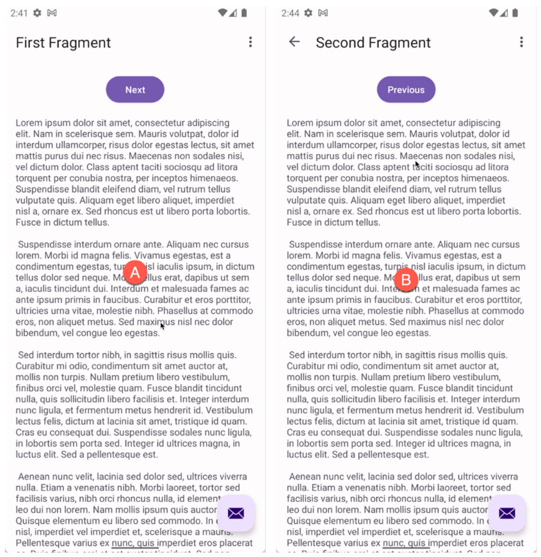
Suspendisse interdum ornare ante. Aliquam nec cursus lorem. Morbi id magna felis. Vivamus egestas, est a condimentum egestas, turpis nisl iaculis ipsum, in dictum tellus dolor sed neque. Morbi tellus erat, dapibus ut sem a, iaculis tincidunt dui. Inte n et malesuada fames ac ante ipsum primis in faucib Curabitur et eros porttitor, ultricies urna vitae, molestie nibh. Phasellus at commodo eros, non aliquet metus. Sed maximus nisl nec dolor bibendum, vel congue leo egestas.

Sed interdum tortor nibh, in sagittis risus mollis quis. Curabitur mi odio, condimentum sit amet auctor at, mollis non turpis. Nullam pretium libero vestibulum, finibus orci vel, molestie quam. Fusce blandit tincidunt nulla, quis sollicitudin libero facilisis et. Integer interdum nunc ligula, et fermentum metus hendrerit id. Vestibulum lectus felis, dictum at lacinia sit amet, tristique id quam. Cras eu consequat dui. Suspendisse sodales nunc ligula, in lobortis sem porta sed. Integer id ultrices magna, in luctus elit. Sed a pellentesque est.

Aenean nunc velit, lacinia sed dolor sed, ultrices viverra nulla. Etiam a venenatis nibh. Morbi laoreet, tortor sed facilisis varius, nibh orci rhoncus nulla, id element leo dui non lorem. Nam mollis ipsum quis tor Quisque elementum eu libero sed commo n nisl, imperdiet vel imperdiet et, scelerisque a mauris. Pellentesque varius ex nunc, quis imperdiet eros placerat

Figure 17-3

The Basic Activity contains layouts for two screens, both containing a button and a text view. The purpose of this template is to demonstrate how to implement navigation between multiple screens within an app. If an unmodified app using the Basic Activity template were to be run, the first of these two screens would appear (marked A in Figure 17-4). Pressing the Next button, would navigate to the second screen (B) which, in turn, contains a button to return to the first screen:



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam in scelerisque sem. Mauris volutpat, dolor id interdum ullamcorper, risus dolor egestas lectus, sit amet mattis purus dui nec risus. Maecenas non sodales nisi, vel dictum dolor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Suspendisse blandit eleifend diam, vel rutrum tellus vulputate quis. Aliquam eget libero aliquet, imperdiet nisl a, ornare ex. Sed rhoncus est ut libero porta lobortis. Fusce in dictum tellus.

Suspendisse interdum ornare ante. Aliquam nec cursus lorem. Morbi id magna felis. Vivamus egestas, est a condimentum egestas, turpis nisl iaculis ipsum, in dictum tellus dolor sed neque. Mo[illegible]llus erat, dapibus ut sem a, iaculis tincidunt dui. Interdum et malesuada fames ac ante ipsum primis in faucibus. Curabitur et eros porttitor, ultricies urna vitae, molestie nibh. Phasellus at commodo eros, non aliquet metus. Sed maximus nisl nec dolor bibendum, vel congue leo egestas.

Sed interdum tortor nibh, in sagittis risus mollis quis. Curabitur mi odio, condimentum sit amet auctor at, mollis non turpis. Nullam pretium libero vestibulum, finibus orci vel, molestie quam. Fusce blandit tincidunt nulla, quis sollicitudin libero facilisis et. Integer interdum nunc ligula, et fermentum metus hendrerit id. Vestibulum lectus felis, dictum at lacinia sit amet, tristique id quam. Cras eu consequat dui. Suspendisse sodales nunc ligula, in lobortis sem porta sed. Integer id ultrices magna, in luctus elit. Sed a pellentesque est.

Aenean nunc velit, lacinia sed dolor sed, ultrices viverra nulla. Etiam a venenatis nibh. Morbi laoreet, tortor sed facilisis varius, nibh orci rhoncus nulla, id element leo dui non lorem. Nam mollis ipsum quis auctor Quisque elementum eu libero sed commodo. In e nisl, imperdiet vel imperdiet et, scelerisque a mauris. Pellentesque varius ex nunc, quis imperdiet eros placerat

Figure 17-4

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

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

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_main.xml* layout file located in the Project tool window under *app -> res -> layout* to load it into the Layout Editor. With the layout loaded into the Layout Editor tool, select the floating action button and tap the keyboard *Delete* key to remove the object from the layout.

2. Locate and edit the 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);

    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    setSupportActionBar(binding.toolbar);

    NavController navController = Navigation.findNavController(this, R.id.nav_
host_fragment_content_main);
    appBarConfiguration = new AppBarConfiguration.Builder(navController.
getGraph()).build();
    NavigationUI.setupActionBarWithNavController(this, navController,
appBarConfiguration);

~~    binding.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.

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

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

```
public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
    binding.buttonFirst.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
~~            NavHostFragment.findNavController(FirstFragment.this)~~
~~                    .navigate(R.id.action_FirstFragment_to_SecondFragment);~~
        }
    });
}
```

17.2 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 three distinct modes referred to as *Design*, *Code* and *Split* modes.

17.3 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 17-5 highlights the key areas of the Android Studio Layout Editor tool in design mode:

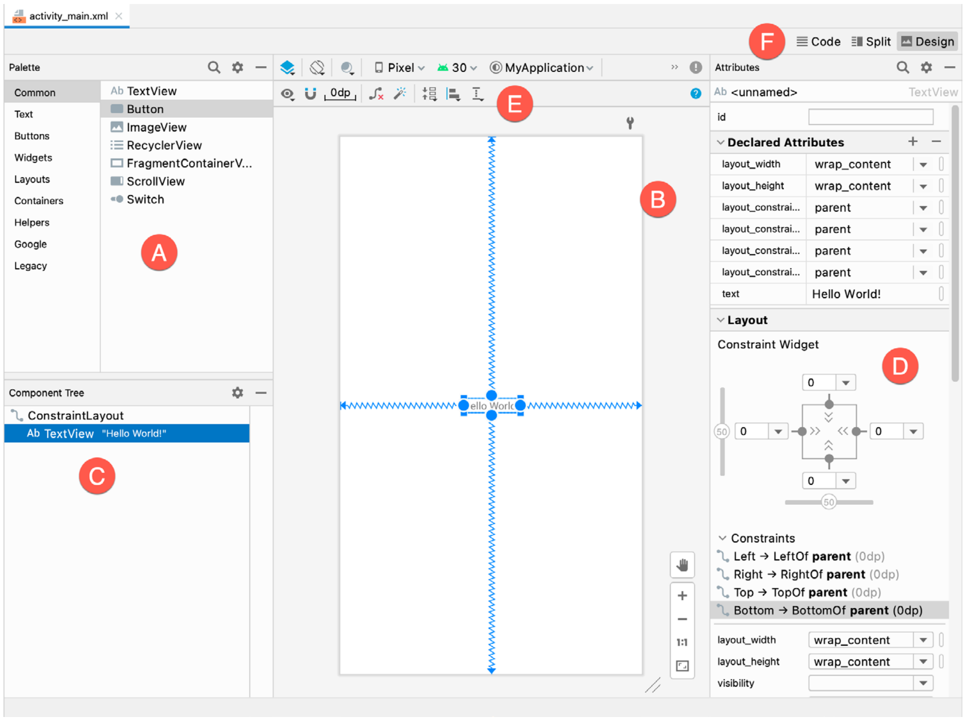


Figure 17-5

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

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

17.4 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 17-6) lists the different categories of view components supported by the Android SDK. When a category is selected from the list, the second panel (B) updates to display a list of the components that fall into that category:

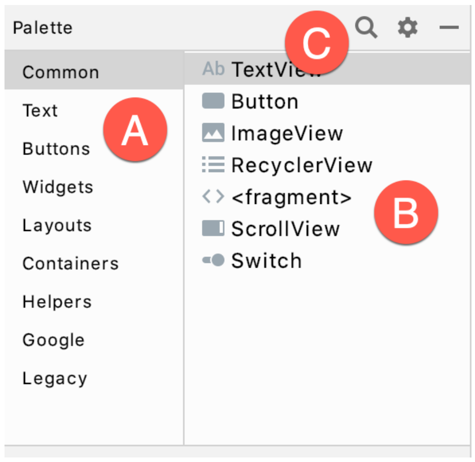


Figure 17-6

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 17-6 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.

17.5 Design Mode and Layout Views

By default, the layout editor will appear in Design mode as is the case in Figure 17-5 above. This mode provides a visual representation of the user interface. Design mode can be selected at any time by clicking on the rightmost mode switching control has shown in Figure 17-7:

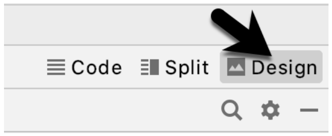


Figure 17-7

When the Layout Editor tool is in Design mode, the layout can be viewed in two different ways. The view shown in Figure 17-5 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 the Blueprint view can be shown either instead of, or concurrently with the Design view. The toolbar menu shown in Figure 17-8 provides options to display the Design, Blueprint, or both views. Settings are also available to adjust for color blindness. A fifth option, *Force Refresh Layout*, causes the layout to rebuild and redraw. This can be useful when the layout enters an unexpected state or is not accurately reflecting the current design settings:

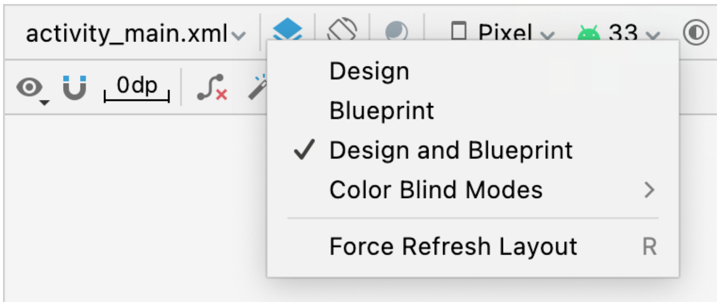


Figure 17-8

Whether to display the layout view, design view or both is a matter of personal preference. A good approach is to begin with both displayed as shown in Figure 17-9:

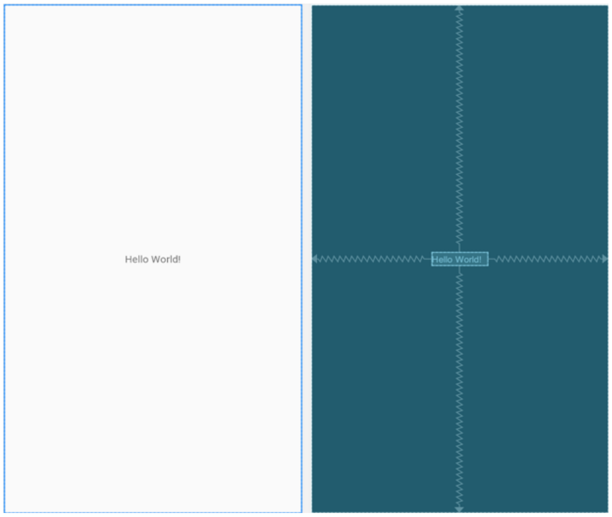


Figure 17-9

17.6 Night Mode

To view the layout in night mode during the design work, select the menu shown in Figure 17-10 below and change the setting to *Night*:

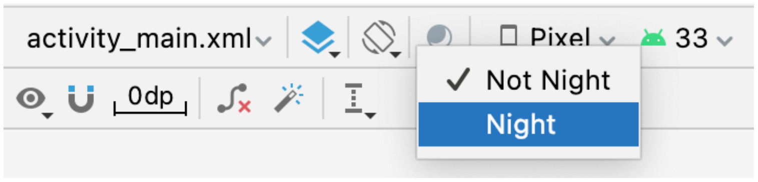


Figure 17-10

17.7 Code 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 *Code* button located in the top right-hand corner of the Layout Editor tool panel as shown in Figure 17-11:

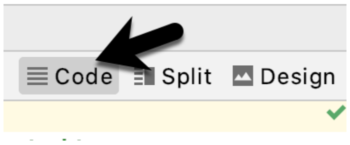
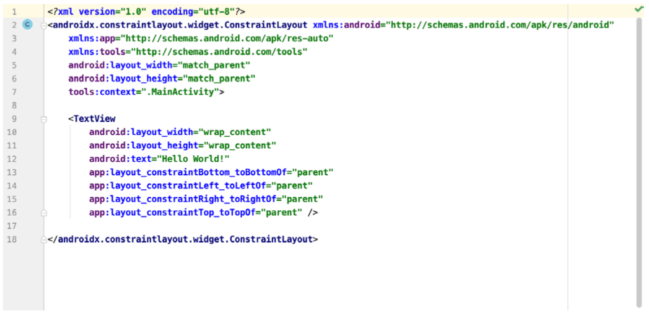


Figure 17-11

Figure 17-12 shows the Android Studio Layout Editor tool in Code mode, allowing changes to be made to the user interface declaration by making changes to the XML:



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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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

Figure 17-12

17.8 Split Mode

In Split mode, the editor shows the Design and Code views side-by-side allowing the user interface to be modified both visually using the design canvas and by making changes directly to the XML declarations. To enter Split mode, click on the middle button shown in Figure 17-13 below:

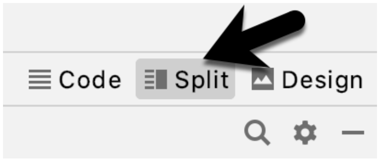
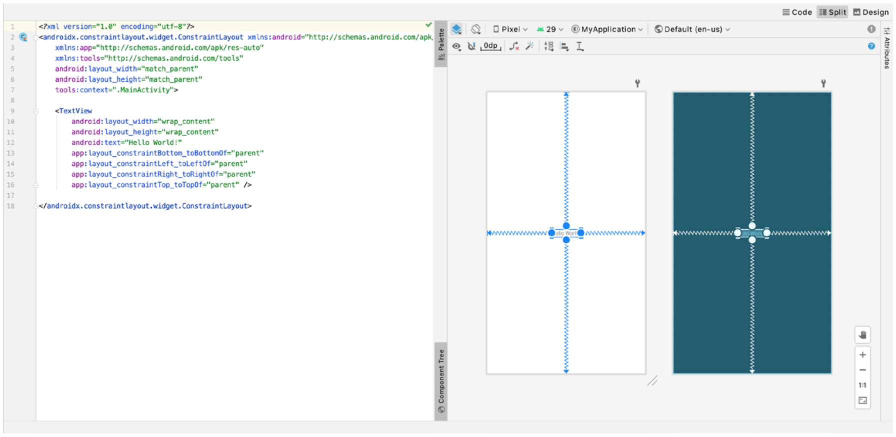


Figure 17-13

Any changes to the XML are automatically reflected in the design canvas and vice versa. Figure 17-14 shows the editor in Split mode:

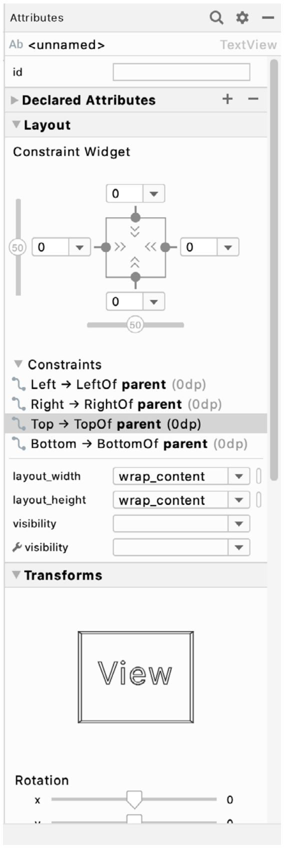


```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk,
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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

Figure 17-14

17.9 Setting Attributes

The Attributes panel provides access to all of the available settings for the currently selected component. Figure 17-15, for example, shows the attributes for the TextView widget:



id

Left → LeftOf **parent** (0dp)
Right → RightOf **parent** (0dp)
Top → TopOf **parent** (0dp)
Bottom → BottomOf **parent** (0dp)

layout_width wrap_content
layout_height wrap_content
visibility
visibility

x 0

Figure 17-15

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

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

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

Some attributes contain a narrow button to the right of the value field. This indicates that the Resources dialog is available to assist in selecting a suitable property value. To display the dialog, simply click on the button. The appearance of this button changes to reflect whether or not the corresponding property value is stored in a resource file or hard-coded. If the value is stored in a resource file, the button to the right of the text property field will be filled in to indicate that the value is not hard coded as highlighted in Figure 17-16 below:

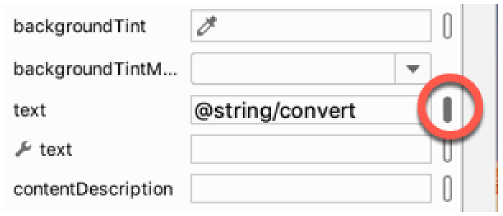

backgroundTint
backgroundTintM...
text @string/convert
text
contentDescription

Figure 17-16

Attributes for which a finite number of valid options are available will present a drop down menu (Figure 17-17) from which a selection may be made.

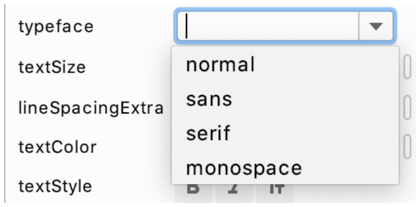
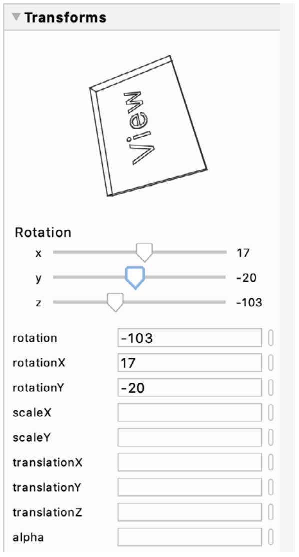


Figure 17-17

A dropper icon (as shown in the backgroundTint field in Figure 17-16 above) can be clicked to display the color selection palette. Similarly, when a flag icon appears in this position it can clicked to display a list of options available for the attribute, while an image icon opens the resource manager panel allowing images and other resource types to be selected for the attribute.

17.10 Transforms

The transforms panel within the Attributes tool window (Figure 17-18) provides a set of controls and properties which control visual aspects of the currently selected object in terms of rotation, alpha (used to fade a view in and out), scale (size), and translation (offset from current position):



rotation -103
rotationX 17
rotationY -20
scaleX
scaleY
translationX
translationY
translationZ
alpha

Figure 17-18

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

17.11 Tools Visibility Toggles

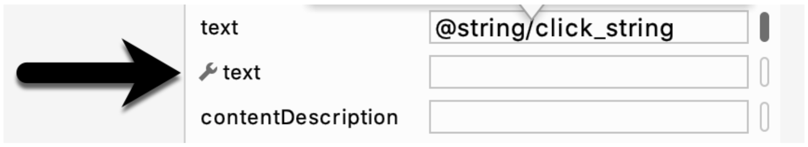
When reviewing the content of an Android Studio XML layout file in Code mode you will notice that many of the attributes that define how a view is to appear and behave begin with the *android:* prefix. This indicates that the attributes are set within the *android* namespace and will take effect when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

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

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

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

A tool attribute of this type is set in the Attributes tool window by entering the value into the property fields marked by the wrench icon as shown in Figure 17-19:



text	@string/click_string
text	
contentDescription	

Figure 17-19

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

```
tools:visibility="invisible"
```

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

```
tools:visibility="gone"
```

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

```
android:visibility="gone"
```

While these visibility tools attributes are useful, having to manually edit the XML layout file is a cumbersome process. To make it easier to change these settings, Android Studio provides a set of toggles within the layout editor Component Tree panel. To access these controls, click in the margin to the right of the corresponding view in the panel. Figure 17-20, for example, shows the tools visibility toggle controls for a Button view named myButton:

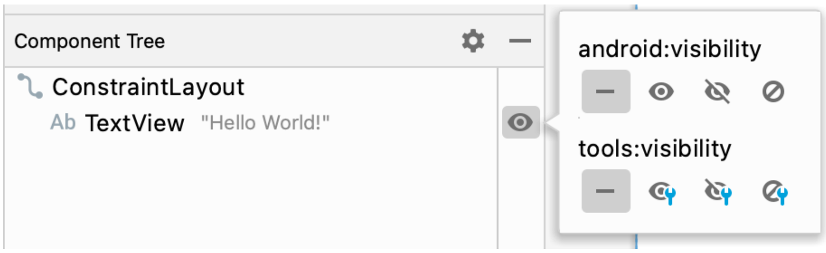


Figure 17-20

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

In addition to the visibility toggles in the Component Tree panel, the layout editor also includes the *tools visibility and position* toggle button shown highlighted in Figure 17-21 below:

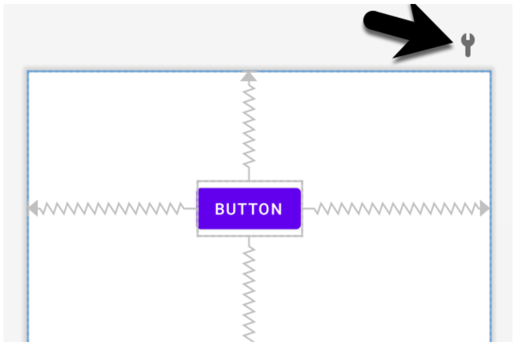


Figure 17-21

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

17.12 Converting Views

Changing a view in a layout from one type to another (such as converting a TextView to an EditText) can be performed easily within the Android Studio layout editor simply by right-clicking on the view either within the screen layout or Component tree window and selecting the *Convert view...* menu option (Figure 17-22):

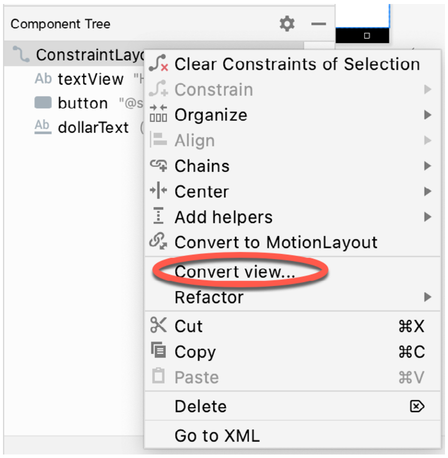


Figure 17-22

Once selected, a dialog will appear containing a list of compatible view types to which the selected object is eligible for conversion. Figure 17-23, for example shows the types to which an existing TextView view may be converted:

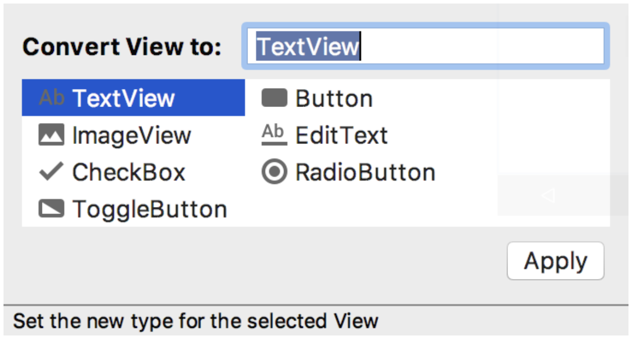
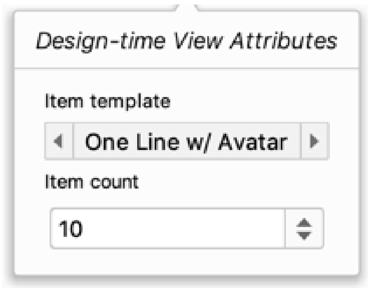


Figure 17-23

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

17.13 Displaying Sample Data

When designing layouts in Android Studio situations will arise where the content to be displayed within the user interface will not be available until the app is completed and running. This can sometimes make it difficult to assess from within the layout editor how the layout will appear at app runtime. To address this issue, the layout editor allows sample data to be specified that will populate views within the layout editor with sample images and data. This sample data only appears within the layout editor and is not displayed when the app runs. Sample data may be configured either by directly editing the XML for the layout, or visually using the design-time helper by right-clicking on the widget in the design area and selecting the *Set Sample Data* menu option. The design-time helper panel will display a range of preconfigured options for sample data to be displayed on the selected view item including combinations of text and images in a variety of configurations. Figure 17-24, for example, shows the sample data options displayed when selecting sample data to appear in a RecyclerView list:



Design-time View Attributes

Item template

One Line w/ Avatar

Item count

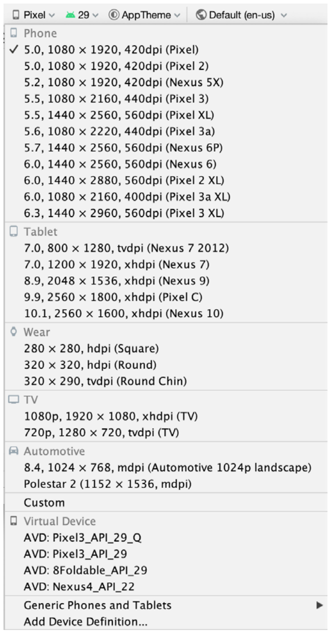
10

Figure 17-24

Alternatively, custom text and images may be provided for display during the layout design process. An example of using sample data within the layout editor is included in a later chapter entitled *"A Layout Editor Sample Data Tutorial"*. Since sample data is implemented as a *tools* attribute, the visibility of the data within the preview can be controlled using the toggle button highlighted in Figure 17-21 above.

17.14 Creating a Custom Device Definition

The device menu in the Layout Editor toolbar (Figure 17-25) provides a list of pre-configured device types which, when selected, will appear as the device screen canvas. In addition to the pre-configured device types, any AVD instances 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"*.



Pixel 29 AppTheme Default (en-us)

Phone
✓ 5.0, 1080 × 1920, 420dpi (Pixel)
5.0, 1080 × 1920, 420dpi (Pixel 2)
5.2, 1080 × 1920, 420dpi (Nexus 5X)
5.5, 1080 × 2160, 440dpi (Pixel 3)
5.5, 1440 × 2560, 560dpi (Pixel XL)
5.6, 1080 × 2220, 440dpi (Pixel 3a)
5.7, 1440 × 2560, 560dpi (Nexus 6P)
6.0, 1440 × 2560, 560dpi (Nexus 6)
6.0, 1440 × 2880, 560dpi (Pixel 2 XL)
6.0, 1080 × 2160, 400dpi (Pixel 3a XL)
6.3, 1440 × 2960, 560dpi (Pixel 3 XL)

Tablet
7.0, 800 × 1280, tvdpi (Nexus 7 2012)
7.0, 1200 × 1920, xhdpi (Nexus 7)
8.9, 2048 × 1536, xhdpi (Nexus 9)
9.9, 2560 × 1800, xhdpi (Pixel C)
10.1, 2560 × 1600, xhdpi (Nexus 10)

Wear
280 × 280, hdpi (Square)
320 × 320, hdpi (Round)
320 × 290, tvdpi (Round Chin)

TV
1080p, 1920 × 1080, xhdpi (TV)
720p, 1280 × 720, tvdpi (TV)

Automotive
8.4, 1024 × 768, mdpi (Automotive 1024p landscape)
Polestar 2 (1152 × 1536, mdpi)

Custom

Virtual Device
AVD: Pixel3_API_29_Q
AVD: Pixel3_API_29
AVD: 8Foldable_API_29
AVD: Nexus4_API_22

Generic Phones and Tablets
Add Device Definition...

Figure 17-25

17.15 Changing the Current Device

As an alternative to the device selection menu, the current device format may be changed by selecting the *Custom* option from the device menu, clicking on the resize handle located next to the bottom right-hand corner of the device screen (Figure 17-26) 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:

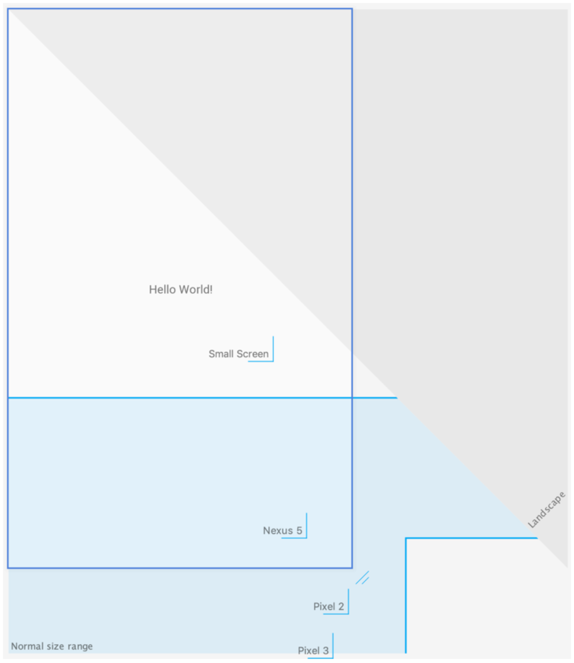


Figure 17-26

17.16 Layout Validation (Multi Preview)

The layout validation (also referred to as multi preview) option allows the user interface layout to be previewed on a range of Pixel-sized screens simultaneously. To access multi preview, click on the tab located near the top right-hand corner of the Android Studio main window as indicated in Figure 17-27:

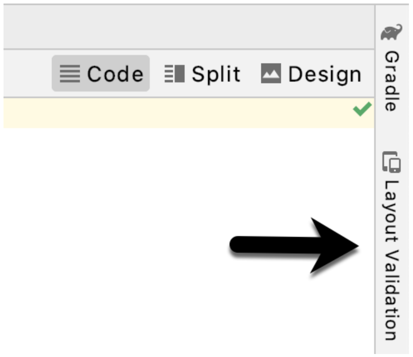


Figure 17-27

Once loaded, the panel will appear as shown in Figure 17-28 with the layout rendered on multiple device screen configurations:

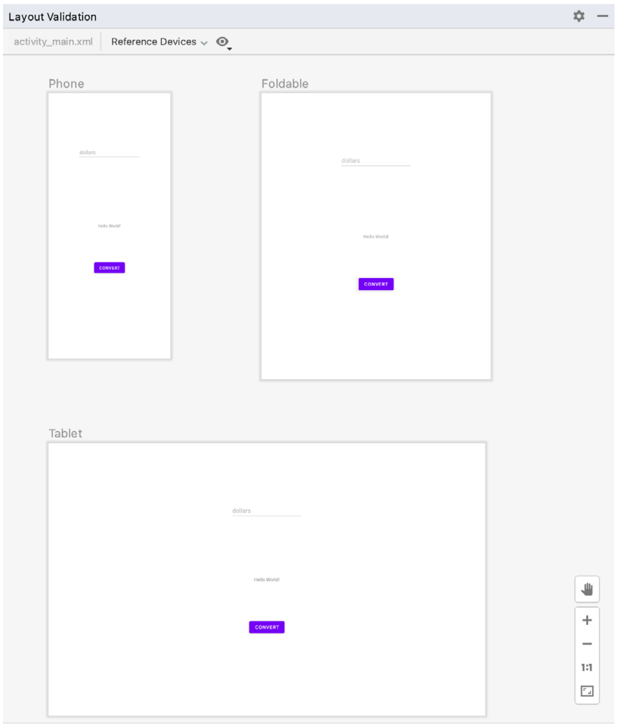


Figure 17-28

17.17 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 three 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 Code mode, the underlying XML that represents the user interface layout can be directly edited. Split mode, on the other hand allows the layout to be created and modified both visually and via direct XML editing. These modes combine to provide an extensive and intuitive user interface design environment.

The layout validation panel allows user interface layouts to be quickly previewed on a range of different device screen sizes.

Chapter 21

21. 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. This chapter will 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.

21.1 An Android Studio Layout Editor Tool Example

The first step in this phase of the example is to create a new Android Studio project. Start by launching Android Studio and closing any previously opened projects by selecting the *File -> Close Project* menu option.

Select the *New Project* option from the welcome screen. 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 make sure that the *No Activity* option is selected before clicking on the Next button

Enter *LayoutSample* into the Name field and specify *com.ebookfrenzy.layoutsample* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.

21.2 Creating a New Activity

Once the project creation process is complete, the Android Studio main window should appear and we are ready 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 if it is not already visible using the Alt-1/Cmd-1 keyboard shortcut. Once the Android hierarchy is displayed, unfold it 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 21-1. Alternatively, select the *New -> Activity -> Gallery...* option to browse the available templates and make a selection using the New Android Activity dialog.

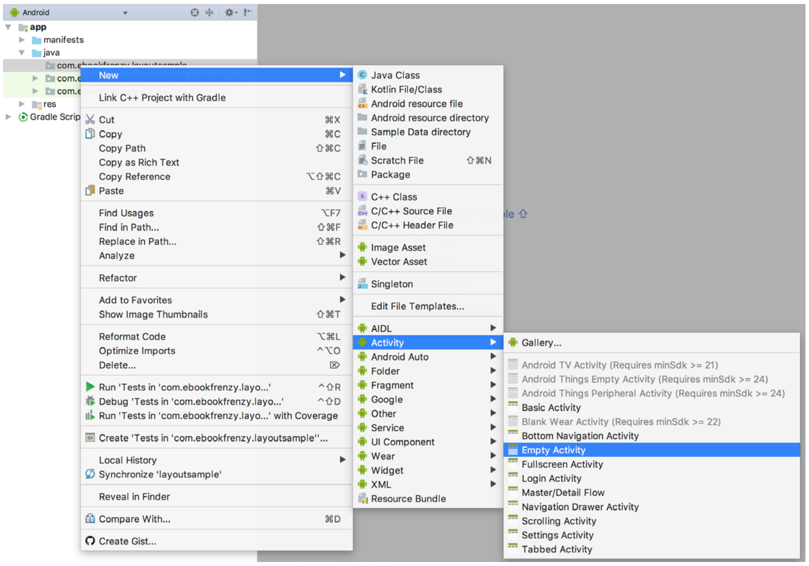


Figure 21-1

In the resulting *New Android Activity* dialog, name the new activity *MainActivity* and the layout *activity_main*. The activity will, of course, need a layout resource file so make sure that the *Generate a 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_main.xml* file and there is no separate content layout file. Also, since we will not be writing any code to access views in the user interface layout, it is not necessary to convert the project to support view binding.

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"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.ebookfrenzy.layoutsample">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.LayoutSample"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

21.3 Preparing the Layout Editor Environment

Locate and double-click on the *activity_main.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 21-2:

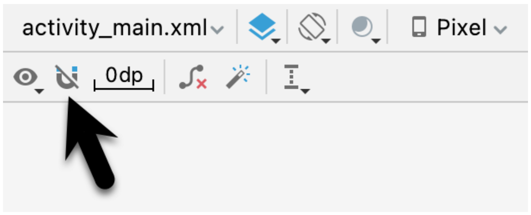


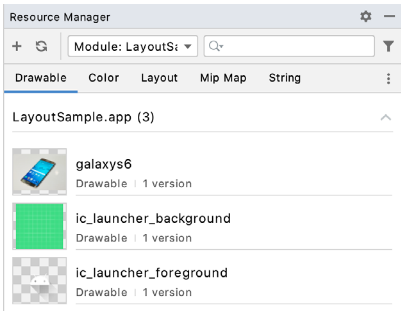
Figure 21-2

If the default margin value to the right of the Autoconnect button is not set to 8dp, click on it and select 8dp from the resulting panel.

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:

https://www.ebookfrenzy.com/retail/dolphinjava/index.php

Within Android Studio, display the Resource Manager tool window (*View* -> *Tool Windows* -> *Resource Manager*). Locate the *galaxy6s.png* image in the file system navigator for your operating system and drag and drop the image onto the Resource Manager tool window. In the resulting dialog, click *Next* followed by the *Import* button to add the image to project. The image should now appear in the Resource Manager as shown in Figure 21-3 below:



LayoutSample.app (3)

galaxys6
Drawable | 1 version

ic_launcher_background
Drawable | 1 version

ic_launcher_foreground
Drawable | 1 version

Figure 21-3

The image will also appear in the *res -> drawables* section of the Project tool window:

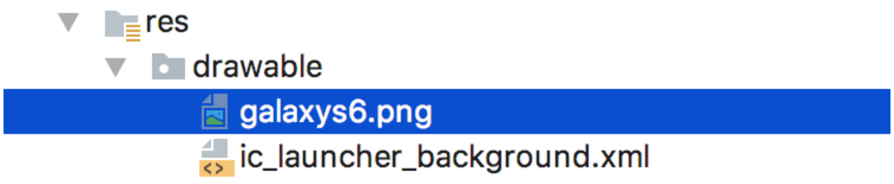
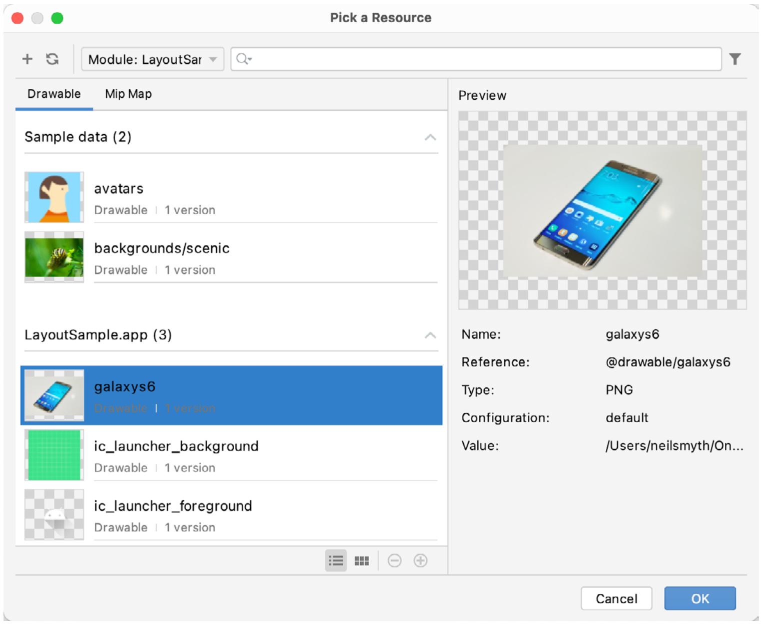


Figure 21-4

21.4 Adding the Widgets to the User Interface

From within the *Common* 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 21-5.



Sample data (2)

avatars
Drawable | 1 version

backgrounds/scenic
Drawable | 1 version

LayoutSample.app (3)

galaxys6
Drawable | 1 version

ic_launcher_background
Drawable | 1 version

ic_launcher_foreground
Drawable | 1 version

Name:	galaxys6
Reference:	@drawable/galaxys6
Type:	PNG
Configuration:	default
Value:	/Users/neilsmyth/On...

Figure 21-5

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 21-6:

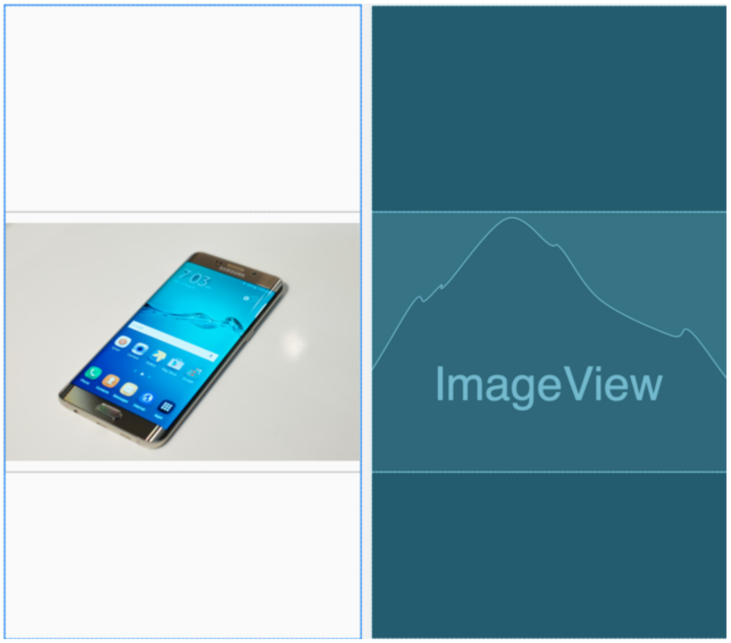


Figure 21-6

Click and drag a TextView object from the *Common* section of the palette and position it so that it appears above the ImageView as illustrated in Figure 21-7.

Using the Attributes panel, unfold the *textAppearance* attribute entry in the Common Attributes section, change the *textSize* property to 24sp, the *textAlignment* setting to center and the text to "Samsung Galaxy S6".

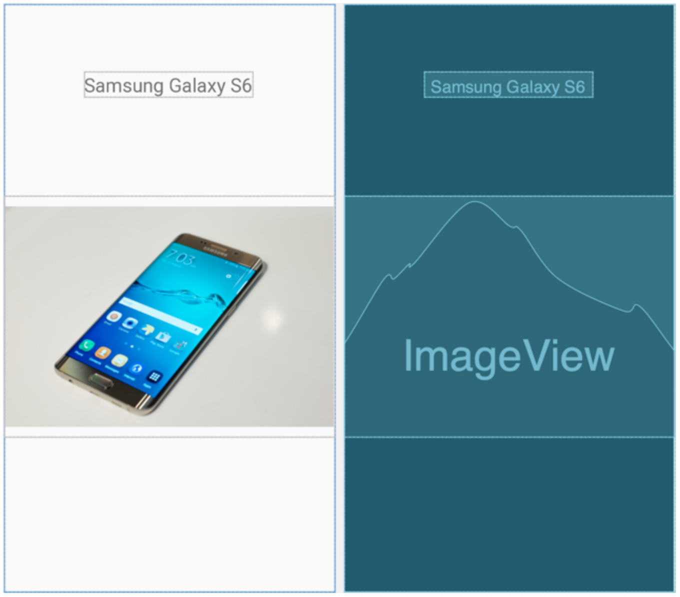


Figure 21-7

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 21-8:

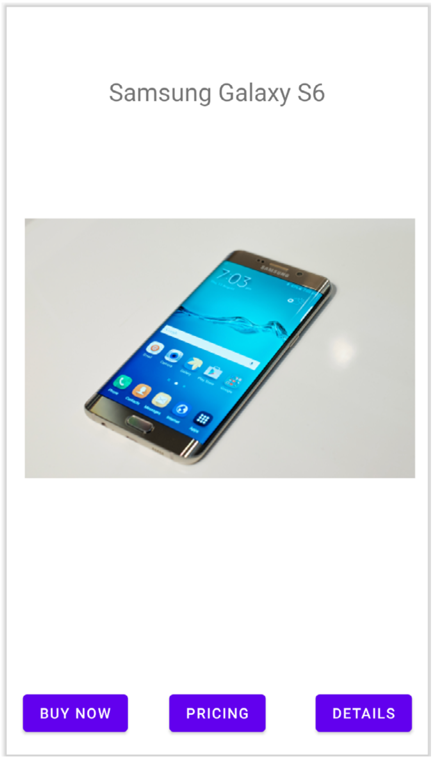


Figure 21-8

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 21-9) to view the user interface in landscape orientation:

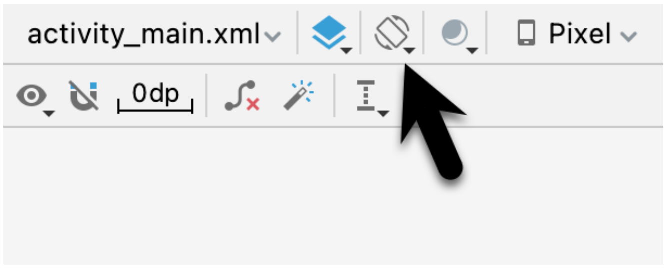


Figure 21-9

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.

21.5 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 21-10:

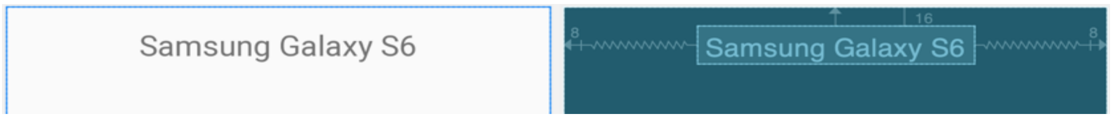


Figure 21-10

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:

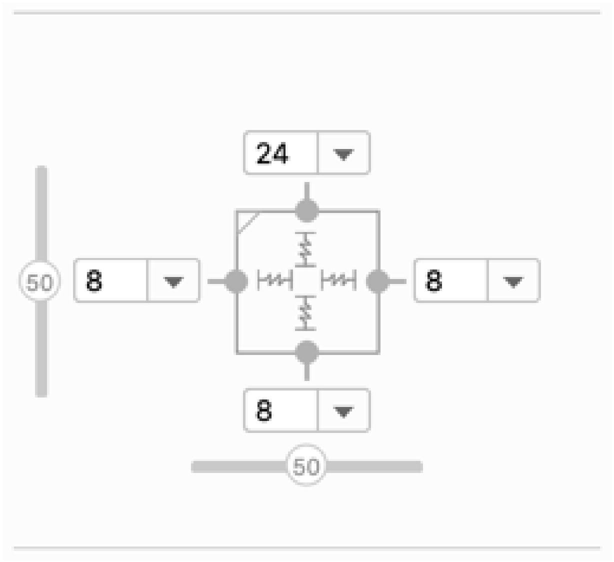


Figure 21-11

Figure 21-12, shows the currently implemented constraints for the ImageView in relation to the other elements in the layout:

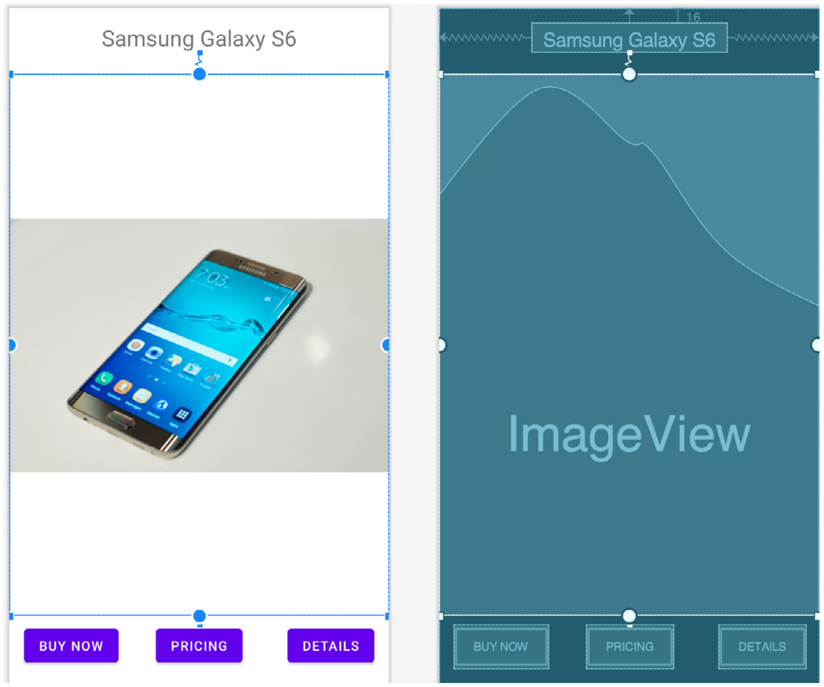


Figure 21-12

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 21-2.

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 *Chains -> 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 21-13:

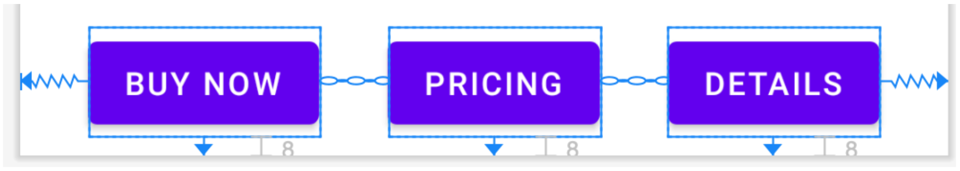


Figure 21-13

21.6 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 21-14, for example, shows the running app in landscape orientation:

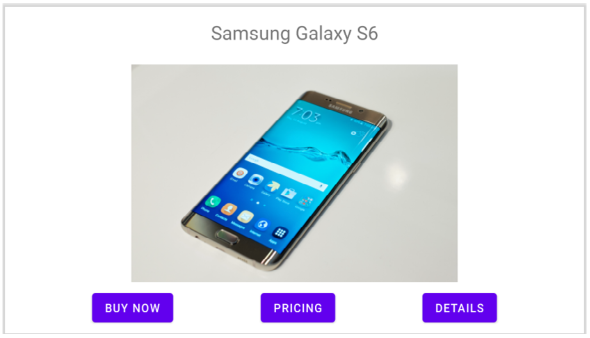


Figure 21-14

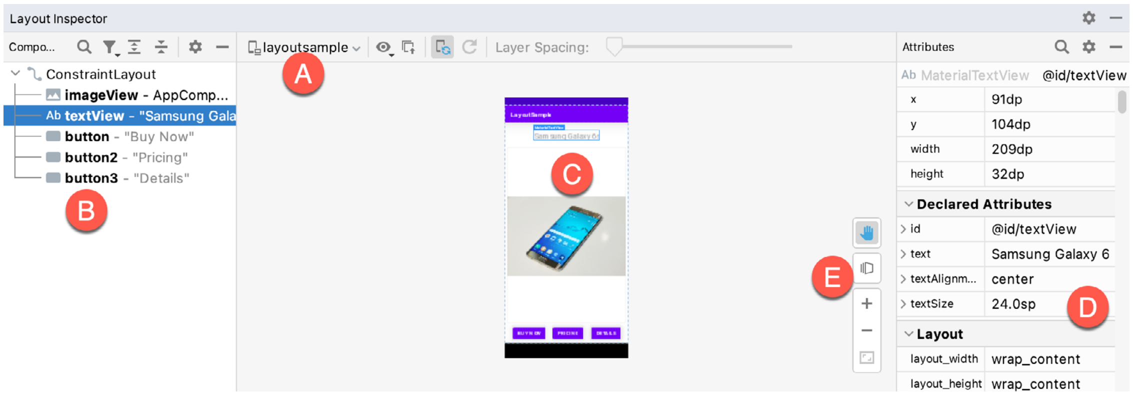
The user interface design is now complete. Designing a more complex user interface layout is a continuation of the steps outlined above. Simply drag and drop views onto the display, position, constrain and set properties as needed.

21.7 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. To access this information the app must be running on a device or emulator running Android API 29 or later. Once the app is running, select the *Tools -> Layout Inspector* menu option followed by the process to be inspected using the menu marked A in Figure 21-15 below).

Once the inspector loads, the left most panel (B) shows the hierarchy of components that make up the user interface layout. The center panel (C) 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 right-most panel (marked D in Figure 21-15) contains all of the property settings for the currently selected component, allowing for in-depth analysis of the component's internal configuration. Where appropriate, the value cell will contain a link to the location of the property setting within the project source code.



ConstraintLayout
imageView - AppComp...
textView - "Samsung Gala
button - "Buy Now"
button2 - "Pricing"
button3 - "Details"

MaterialTextView @id/textView

x	91dp
y	104dp
width	209dp
height	32dp

Declared Attributes

id	@id/textView
text	Samsung Galaxy 6
textAlignm...	center
textSize	24.0sp

Layout

layout_width	wrap_content
layout_height	wrap_content

Figure 21-15

To view the layout in 3D, click on the button labeled E. This displays an "exploded" representation of the hierarchy so that it can be rotated and inspected. This can be useful for tasks such as identifying obscured views:

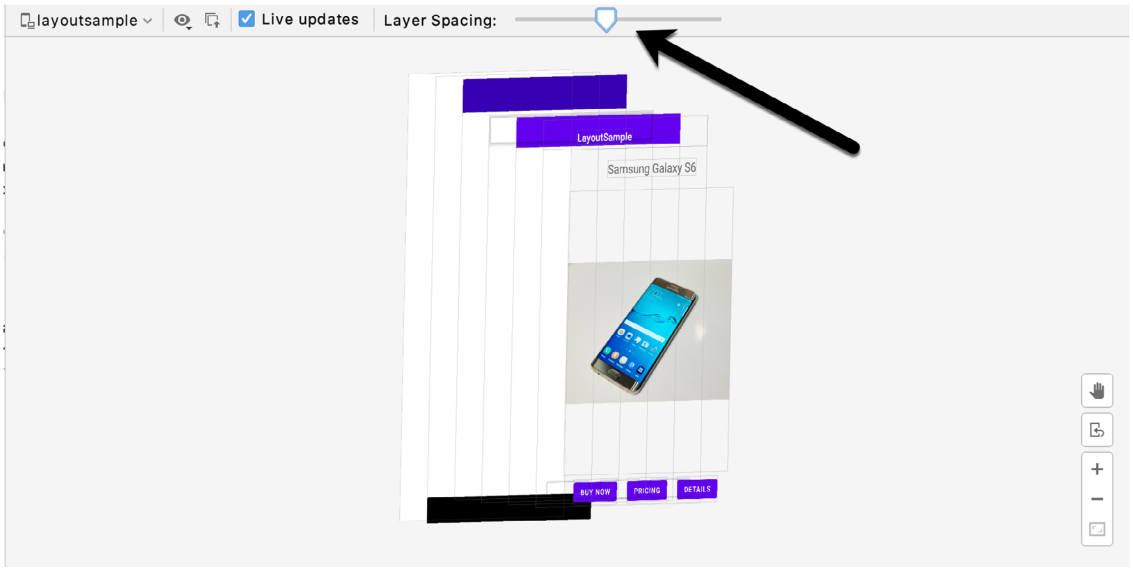


Figure 21-16

Click and drag the rendering to rotate it in three dimensions, using the slider indicated by the arrow in the above figure to increase the spacing between the layers. Click the button marked E again to return to the 2D view.

21.8 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 Live Layout Inspector tool which is useful for analyzing the structural composition of a user interface layout.

Chapter 26

26. 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.

26.1 Understanding Android Events

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

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

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

26.2 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. As will be outlined in later chapters, the onClick property also has limitations in layouts involving fragments. When working within Android Studio Layout Editor, the onClick property can be found and configured in the Attributes panel when a suitable view type is selected in the device screen layout.

26.3 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.

26.4 An Event Handling Example

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

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

Enter *EventExample* into the Name field and specify *com.ebookfrenzy.eventexample* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java. Using the steps outlined in section *11.8 Migrating a Project to View Binding*, convert the project to use view binding.

26.5 Designing the User Interface

The user interface layout for the *MainActivity* class in this example is to consist of a ConstraintLayout, a Button and a TextView as illustrated in Figure 26-1.

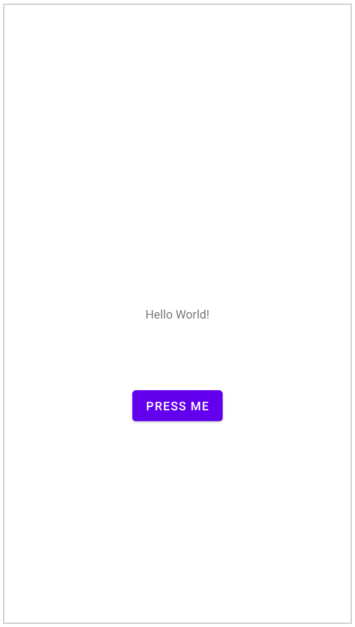


Figure 26-1

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

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

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

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

With the Button widget selected, use the Attributes panel to set the text property to Press Me. Using the yellow warning button located in the top right-hand corner of the Layout Editor (Figure 26-2), display the warnings list and click on the *Fix* button to extract the text string on the button to a resource named *press_me*:

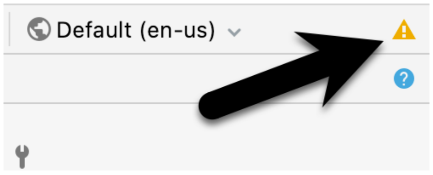


Figure 26-2

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

26.6 The Event Listener and Callback Method

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

If the *MainActivity.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 -> MainActivity*) and double-click on it to load it into the code editor. Once loaded, locate the template *onCreate()* method and modify it to obtain a reference to the button view, register the event listener and implement the *onClick()* callback method:

```
package com.ebookfrenzy.eventexample;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        View view = binding.getRoot();
        setContentView(view);

        binding.myButton.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);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        View view = binding.getRoot();
        setContentView(view);

       binding.myButton.setOnClickListener(
                new Button.OnClickListener() {
                    public void onClick(View v) {
                       binding.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.

26.7 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) {
.
.
        binding.myButton.setOnLongClickListener(
                 new Button.OnLongClickListener() {
                      public boolean onLongClick(View v) {
                        binding.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 so that the onLongClick listener now returns a *false* value:

```
button.setOnLongClickListener(
    new Button.OnLongClickListener() {
        public boolean onLongClick(View v) {
            TextView myTextView = 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 clicked". This is because the *false* value returned by the *onLongClick* listener code indicated to the Android framework that the event was not consumed by the method and was eligible to be passed on to the next registered listener on the view. In this case, the runtime ascertained that the onClickListener on the button was also interested in events of this type and subsequently called the *onClick* listener code.

26.8 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.

Chapter 27

27. 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.

This chapter will explain the handling of touches that involve motion and explore the concept of intercepting multiple concurrent touches. The topic of identifying distinct gestures will be covered in the next chapter.

27.1 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:

```
binding.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*.

27.2 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.

27.3 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. To identify the index of the pointer that triggered the event, the *getActionIndex()* callback method of the MotionEvent object must be called.

27.4 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 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.

27.5 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.

Select the *New Project* option from the welcome screen and, within the resulting new project dialog, choose the

Empty Activity template before clicking on the Next button.

Enter *MotionEvent* into the Name field and specify *com.ebookfrenzy.motionevent* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.

Adapt the project to use view binding as outlined in section *11.8 Migrating a Project to View Binding*.

27.6 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_main.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:

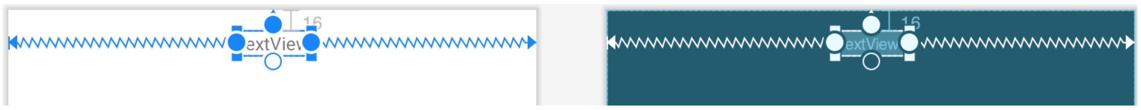


Figure 27-1

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:

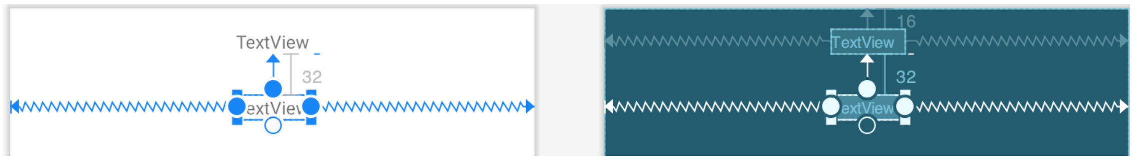


Figure 27-2

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.

27.7 Implementing the Touch Event Listener

To receive touch event notifications it will be necessary to register a touch listener on the layout view within the *onCreate()* method of the *MainActivity* activity class. Select the *MainActivity.java* tab from the Android Studio editor panel to display the source code. Within the *onCreate()* method, add code to 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 androidx.appcompat.app.AppCompatActivity;
import androidx.constraintlayout.widget.ConstraintLayout;

import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
import com.ebookfrenzy.motionevent.databinding.ActivityMainBinding;

public class MainActivity extends AppCompatActivity {

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        View view = binding.getRoot();
        setContentView(view);

        binding.activityMain.setOnTouchListener(
                new ConstraintLayout.OnTouchListener() {
                    public boolean onTouch(View v, MotionEvent m) {
                        handleTouch(m);
                        return true;
                    }
                }
        );
    }
```

When we designed the user interface, the parent ConstraintLayout was not assigned an ID that would allow us to access it via the view binding mechanism. Since this layout component is the top-most component in the UI layout hierarchy, we have been able to reference it using the *root* binding property in the code above.

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) {

    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;
            case MotionEvent.ACTION_MOVE:
                actionString = "MOVE";
                break;
            default:
                actionString = "";
        }

        String touchStatus = "Action: " + actionString + " Index: " + actionIndex
+ " ID: " + id + " X: " + x + " Y: " + y;

        if (id == 0)
            binding.textView1.setText(touchStatus);
        else
            binding.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 = findViewById(R.id.textView1);
TextView textView2 = 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:

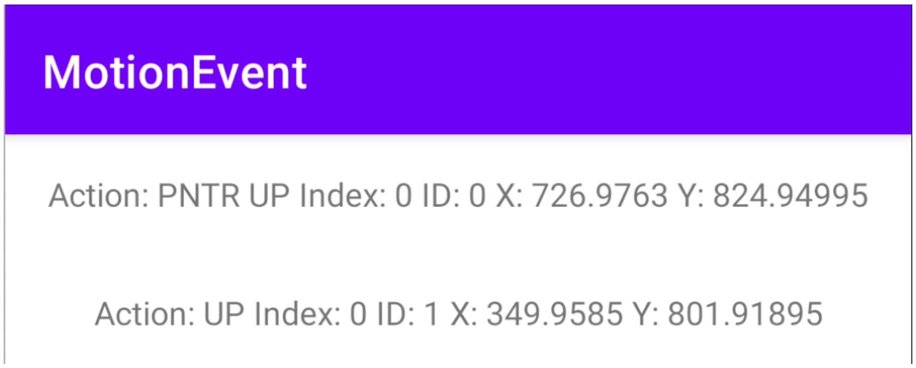
```
switch (action)
{
    case MotionEvent.ACTION_DOWN:
        actionString = "DOWN";
        break;
    case MotionEvent.ACTION_UP:
        actionString = "UP";
        break;
    case MotionEvent.ACTION_POINTER_DOWN:
        actionString = "PNTR DOWN";
        break;
    case MotionEvent.ACTION_POINTER_UP:
        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)
     binding.textView1.setText(touchStatus);
else
     binding.textView2.setText(touchStatus);
```

27.8 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 27-3. When running on an emulator, multiple touches may be simulated by holding down the Ctrl (Cmd on macOS) key while clicking the mouse button (note that simulating multiple touches may not work if the emulator is running in a tool window):



Action: PNTR UP Index: 0 ID: 0 X: 726.9763 Y: 824.94995

Action: UP Index: 0 ID: 1 X: 349.9585 Y: 801.91895

Figure 27-3

27.9 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.

Chapter 30

30. 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.

30.1 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.

30.2 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 of a ConstraintLayout with a red background containing a single TextView with a white foreground:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.
android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/constraintLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/holo_red_dark"
    tools:context=".FragmentOne">
    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="My First Fragment"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        android:textColor="@color/white"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The corresponding class to go with the layout must be a subclass of the Android *Fragment* class. This 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.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import androidx.fragment.app.Fragment;

public class FragmentOne extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
             ViewGroup container,
              Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        binding = FragmentTextBinding.inflate(inflater, container, false);
        return binding.getRoot();
    }
}
```

In addition to the *onCreateView()* method, the class may also override the standard lifecycle methods.

Once the fragment layout and class have been created, the fragment is ready to be used within application activities.

30.3 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 androidx.fragment.app.FragmentActivity;
import android.view.Menu;

public class MainActivity extends FragmentActivity {
.
.
```

Fragments are embedded into activity layout files using the FragmentContainerView class. The following example layout embeds the fragment created in the previous section of this chapter into an activity layout:

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

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment2"
        android:name="com.ebookfrenzy.myfragmentdemo.FragmentOne"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="32dp"
        android:layout_marginEnd="32dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:layout="@layout/fragment_one" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

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 30-1, for example, shows the above layout with the embedded fragment within the Android Studio Layout Editor:

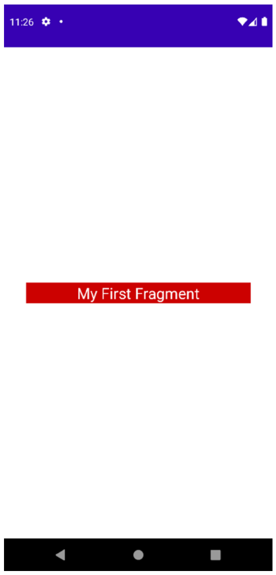


Figure 30-1

30.4 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. 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 instance.
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 if 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();
```

30.5 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:

```
binding.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.

30.6 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 MainActivity 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.

30.7 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.

Chapter 31

31. 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.

31.1 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 ConstraintLayout view. The second fragment will consist solely of a TextView object, also contained within a ConstraintLayout 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.

31.2 Creating the Example Project

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

Enter *FragmentExample* into the Name field and specify *com.ebookfrenzy.fragmentexample* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java. Using the steps outlined in section *11.8 Migrating a Project to View Binding*, modify the project to use view binding.

Return to the *Gradle Scripts -> build.gradle (Module: FragmentExample.app)* file and add the following directive to the *dependencies* section (keeping in mind that a more recent version of the library may now be available):

```
implementation 'androidx.navigation:navigation-fragment:2.5.2'
```

31.3 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 consist of an XML layout file and a fragment class. While these could be added manually, it is quicker to ask Android Studio to create them for us. Within the project tool window, locate the *app -> java -> com -> ebookfrenzy -> fragmentexample* entry and right click on it. From the resulting menu, select the *New*

-> *Fragment* -> *Gallery...* option to display the dialog shown in Figure 31-1 below:

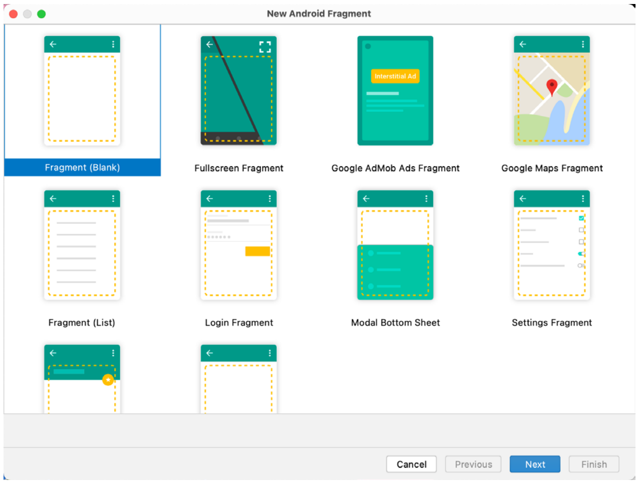
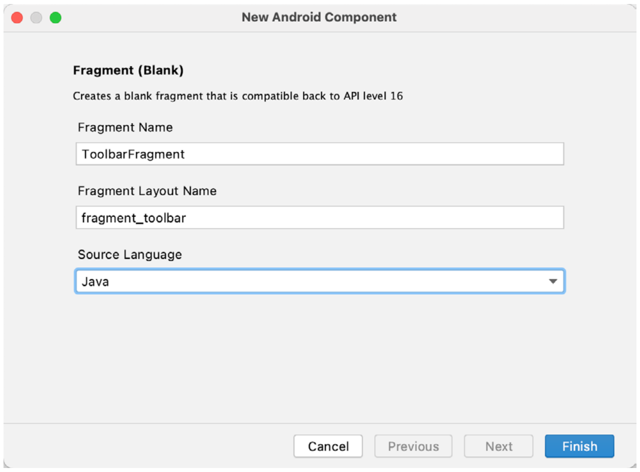


Figure 31-1

Select the *Fragment (Blank)* template before clicking the Next button. On the subsequent screen, name the fragment *ToolbarFragment* with a layout file named *fragment_toolbar*:



Fragment (Blank)

Creates a blank fragment that is compatible back to API level 16

Fragment Name: ToolbarFragment

Fragment Layout Name: fragment_toolbar

Source Language: Java

Figure 31-2

Load the *fragment_toolbar.xml* file into the layout editor using Design mode, right-click on the FrameLayout entry in the Component Tree panel and select the *Convert FrameLayout to ConstraintLayout* menu option, accepting the default settings in the confirmation dialog. Change the id from *frameLayout* to *constraintLayout*. Select and delete the default TextView and add a Plain EditText, Seekbar and Button to the layout and change the view ids to *editText1*, *button1* and *seekBar1* respectively.

Change the text on the button to read "Change Text", extract the text to a string resource named *change_text* and remove the Name text from the EditText view. Finally, set the *layout_width* property of the Seekbar to *match_constraint* with margins set to 16dp on the left and right edges.

Use the *Infer constraints* toolbar button to add any missing constraints, at which point the layout should match that shown in Figure 31-3 below:

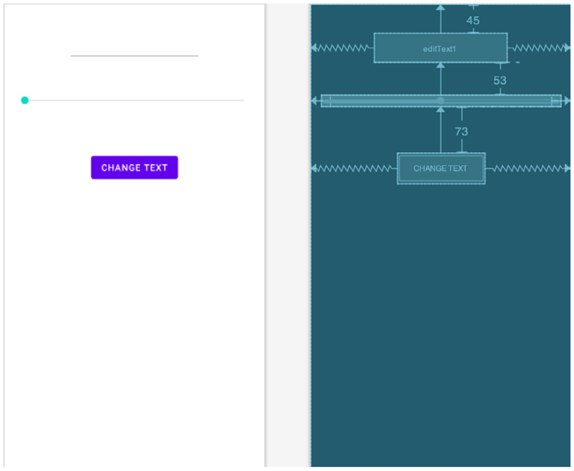


Figure 31-3

31.4 Migrating a Fragment to View Binding

As with the Empty Activity template, Android Studio does not enable view binding support when new fragments are added to a project. Before moving to the next step of this tutorial, therefore, we will need to perform this migration. Begin by editing the *ToolbarFragment.java* file and importing the binding for the fragment as follows:

```
import com.ebookfrenzy.fragmentexample.databinding.FragmentToolbarBinding;
```

Next, locate the *onCreateView()* method and make the following declarations and changes (which also include adding the *onDestroyView()* method to ensure that the binding reference is removed when the fragment is destroyed):

```
.
.
private FragmentToolbarBinding binding;

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    ~~return inflater.inflate(R.layout.fragment_toolbar, container, false);~~
    binding = FragmentToolbarBinding.inflate(inflater, container, false);
    return binding.getRoot();
}

@Override
public void onDestroyView() {
    super.onDestroyView();
    binding = null;
}
```

Once these changes are complete, the fragment is ready to use view binding.

31.5 Adding the Second Fragment

Repeating the steps used to create the toolbar fragment, add another empty fragment named TextFragment with a layout file named *fragment_text*. Once again, convert the FrameLayout container to a ConstraintLayout (changing the id to *constraintLayout2*) and remove the default TextView.

Drag a drop a TextView widget from the palette and position it in the center of the layout, using the *Infer constraints* button to add any missing constraints. Change the id of the TextView to *textView2*, the text to read "Fragment Two" and modify the *textAppearance* attribute to *Large*.

On completion, the layout should match that shown in Figure 31-4:

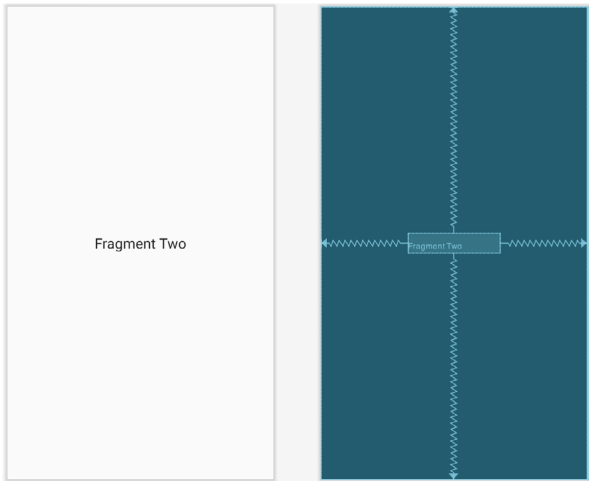


Figure 31-4

Repeat the steps performed in the previous section to migrate the TextFragment class to use view binding as follows:

```
.
.
import com.ebookfrenzy.fragmentexample.databinding.FragmentTextBinding;
.
.
private FragmentTextBinding binding;

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    ~~return inflater.inflate(R.layout.fragment_text, container, false);~~
    binding = FragmentTextBinding.inflate(inflater, container, false);
    return binding.getRoot();
}

@Override
public void onDestroyView() {
    super.onDestroyView();
    binding = null;
}
```

31.6 Adding the Fragments to the Activity

The main activity for the application has associated with it an XML layout file named *activity_main.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_main.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 *Common* category in the palette. Drag the *FragmentContainerView* component from the list of views 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:

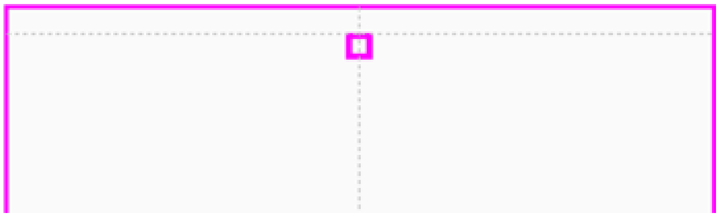
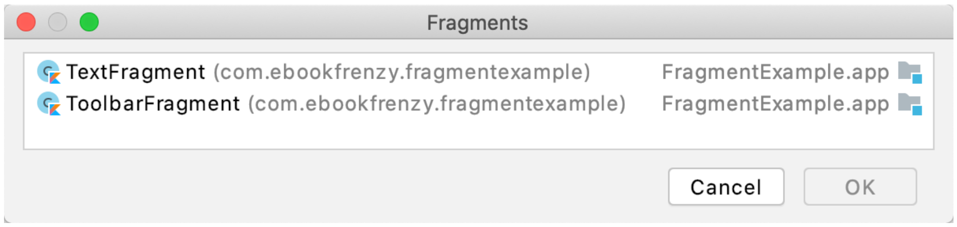


Figure 31-5

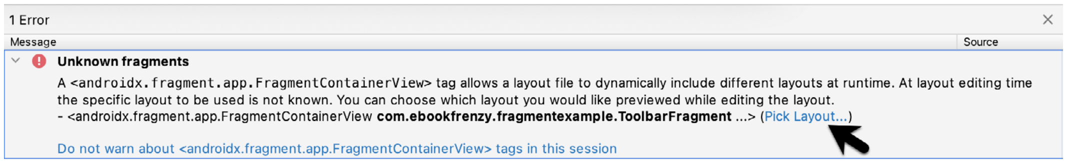
On dropping the fragment onto the layout, a dialog will appear displaying a list of Fragments available within the current project as illustrated in Figure 31-6:



Fragments
TextFragment (com.ebookfrenzy.fragmentexample) FragmentExample.app
ToolbarFragment (com.ebookfrenzy.fragmentexample) FragmentExample.app
Cancel
OK

Figure 31-6

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 will be listed indicating that the Layout Editor tool needs to know which fragment to display during the preview session. Click on the *Pick Layout...* link in the error message as indicated in Figure 31-7:



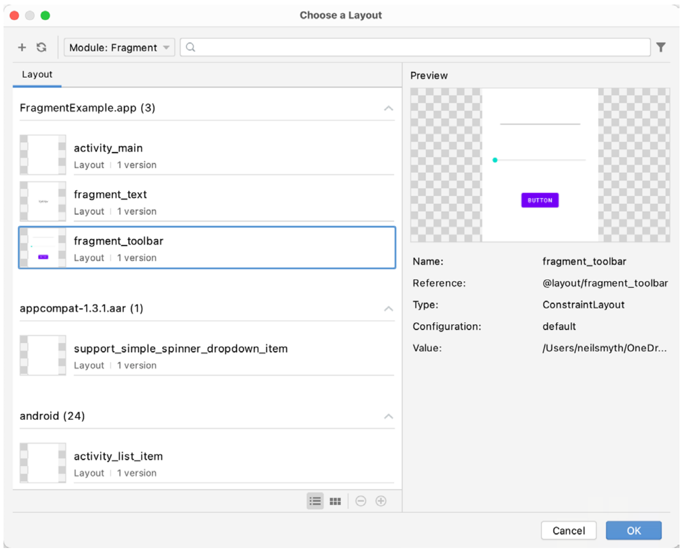
Unknown fragments

A <androidx.fragment.app.FragmentContainerView> tag allows a layout file to dynamically include different layouts at runtime. At layout editing time the specific layout to be used is not known. You can choose which layout you would like previewed while editing the layout.
- <androidx.fragment.app.FragmentContainerView **com.ebookfrenzy.fragmentexample.ToolbarFragment** ...> (Pick Layout...)

Do not warn about <androidx.fragment.app.FragmentContainerView> tags in this session

Figure 31-7

In the resulting dialog (Figure 31-8) select the *fragment_toolbar* entry and before clicking the OK button:



Name: fragment_toolbar
Reference: @layout/fragment_toolbar
Type: ConstraintLayout
Configuration: default
Value: /Users/neilsmyth/OneDr...

Figure 31-8

With the fragment selected, change the *layout_width* property to *match_constraint* so that it occupies the full width of the screen. Click and drag another *FragmentContainerView* entry from the palette and position it so that it is centered horizontally and located beneath the bottom edge of the first fragment. When prompted, select the *TextFragment* entry from the fragment dialog before clicking on the OK button. Display the error panel once again and click on the *Use @layout/fragment_text* option. Use the *Infer constraints* button to establish any missing layout constraints.

Note that the fragments are now visible in the layout as demonstrated in Figure 31-9:

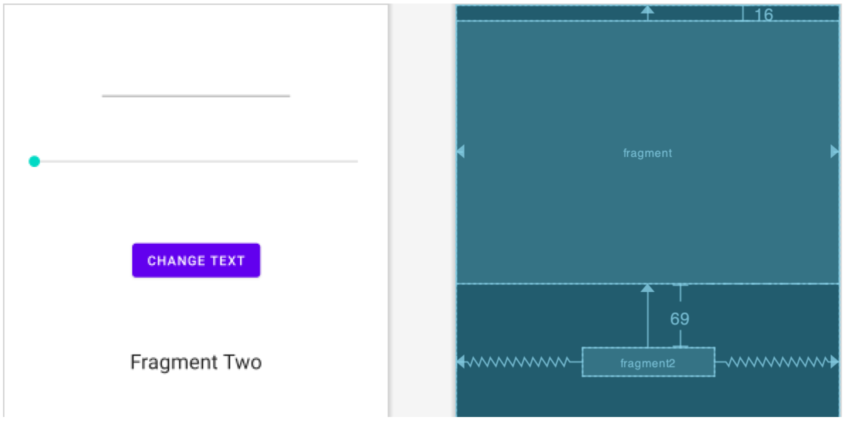


Figure 31-9

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*.

31.7 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 EditText view 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 androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import android.content.Context;
import android.widget.SeekBar;

public class ToolbarFragment extends Fragment implements
                                 SeekBar.OnSeekBarChangeListener {

    private static int seekvalue = 10;
.
.
@Override
    public void onViewCreated(@NonNull View view,
                   @Nullable Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);

        binding.seekBar1.setOnSeekBarChangeListener(this);
        binding.button1.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                buttonClicked(v);
            }
        });
    }

    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 *onViewCreated()* method has been added 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 SeekBar.OnSeekBarChangeListener {

        private static int seekvalue = 10;
        private FragmentToolbarBinding binding;
        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");
              }
         }
.
.
         public void buttonClicked (View view) {
                activityCallback.onButtonClick(seekvalue,
                        binding.editText1.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 *MainActivity.java* file:

```
package com.ebookfrenzy.fragmentexample;

~~import androidx.appcompat.app.AppCompatActivity;~~
import androidx.fragment.app.FragmentActivity;
import android.os.Bundle;

public class MainActivity extends FragmentActivity implements ToolbarFragment.
ToolbarListener {
.
.
    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.

31.8 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:

```
package com.ebookfrenzy.fragmentexample;
.
.

public class TextFragment extends Fragment {
.
.
```

```
    public void changeTextProperties(int fontsize, String text)
    {
        binding.textView2.setTextSize(fontsize);
        binding.textView2.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 *MainActivity.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);
}
```

31.9 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:

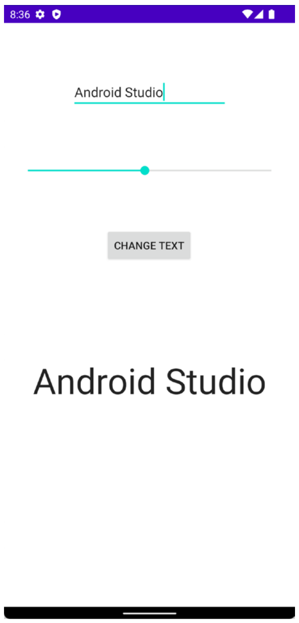


Figure 31-10

31.10 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.

Chapter 38

38. Working with Android Lifecycle-Aware Components

The earlier chapter entitled *"Understanding Android Application and Activity Lifecycles"* described the use of lifecycle methods to track lifecycle state changes within a UI controller such as an activity or fragment. One of the main problems with these methods is that they place the burden of handling lifecycle changes onto the UI controller. On the surface this might seem like the logical approach since the UI controller is, after all, the object going through the state change. The fact is, however, that the code that is typically impacted by the state change invariably resides in other classes within the app. This led to complex code appearing in the UI controller that needed to manage and manipulate other objects in response to changes in lifecycle state. Clearly this is a scenario best avoided when following the Android architectural guidelines.

A much cleaner and logical approach would be for the objects within an app to be able to observe the lifecycle state of other objects and to be responsible for taking any necessary actions in response to the changes. The class responsible for tracking a user's location, for example, could observe the lifecycle state of a UI controller and suspend location updates when the controller enters a paused state. Tracking would then be restarted when the controller enters the resumed state. This is made possible by the classes and interfaces provided by the Lifecycle package bundled with the Android architecture components.

This chapter will introduce the terminology and key components that enable lifecycle awareness to be built into Android apps.

38.1 Lifecycle Awareness

An object is said to be *lifecycle-aware* if it is able to detect and respond to changes in the lifecycle state of other objects within an app. Some Android components, LiveData being a prime example, are already lifecycle-aware. It is also possible to configure any class to be lifecycle-aware by implementing the LifecycleObserver interface within the class.

38.2 Lifecycle Owners

Lifecycle-aware components can only observe the status of objects that are *lifecycle owners*. Lifecycle owners implement the LifecycleOwner interface and are assigned a companion *Lifecycle object* which is responsible for storing the current state of the component and providing state information to *lifecycle observers*. Most standard Android Framework components (such as activity and fragment classes) are lifecycle owners. Custom classes may also be configured as lifecycle owners by using the LifecycleRegistry class and implementing the LifecycleObserver interface. For example:

```
public class SampleOwner implements LifecycleOwner {

    private LifecycleRegistry lifecycleRegistry;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        lifecycleRegistry = new LifecycleRegistry(this);
    }

    @NonNull
    @Override
    public Lifecycle getLifecycle() {
        return lifecycleRegistry;
    }
}
```

Unless the lifecycle owner is a subclass of another lifecycle-aware component, the class will need to trigger lifecycle state changes itself via calls to methods of the LifecycleRegistry class. The *markState()* method can be used to trigger a lifecycle state change passing through the new state value:

```
public void resuming() {
    lifecycleRegistry.markState(Lifecycle.State.RESUMED);
}
```

The above call will also result in a call to the corresponding event handler. Alternatively, the LifecycleRegistry *handleLifecycleEvent()* method may be called and passed the lifecycle event to be triggered (which will also result in the lifecycle state changing). For example:

```
lifecycleRegistry.handleLifecycleEvent(Lifecycle.Event.ON_START);
```

38.3 Lifecycle Observers

In order for a lifecycle-aware component to observe the state of a lifecycle owner it must implement the DefaultLifecycleObserver interface and override methods for any lifecycle change events it needs to observe.

```
public class SampleObserver implements DefaultLifecycleObserver {
    // Lifecycle event methods overrides go here
}
```

An instance of this observer class is then created and added to the list of observers maintained by the Lifecycle object.

```
getLifecycle().addObserver(new SampleObserver());
```

An observer may also be removed from the Lifecycle object at any time if it no longer needs to track the lifecycle state.

Figure 38-1 illustrates the relationship between the key elements that provide lifecycle awareness:

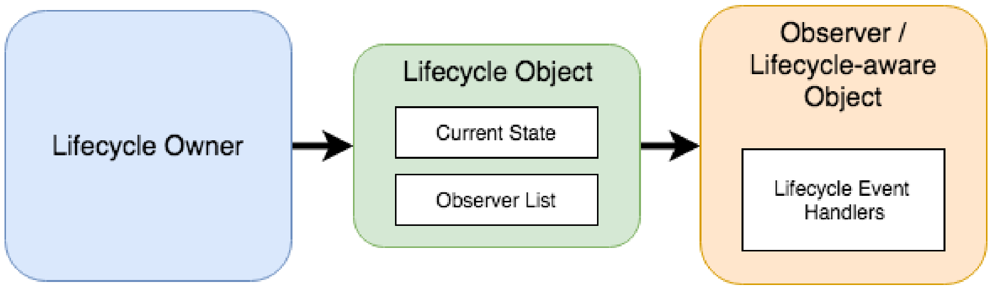


Figure 38-1

38.4 Lifecycle States and Events

When the status of a lifecycle owner changes, the assigned Lifecycle object will be updated with the new state. At any given time, a lifecycle owner will be in one of the following five states:

- Lifecycle.State.INITIALIZED
- Lifecycle.State.CREATED
- Lifecycle.State.STARTED
- Lifecycle.State.RESUMED
- Lifecycle.State.DESTROYED

As the component transitions through the different states, the Lifecycle object will trigger events on any observers that have been added to the list. The following event methods are available to be overridden within the lifecycle observer:

- onCreate()
- onResume()
- onPause()
- onStop()
- onStart()
- onDestroy()

The following code, for example, overrides the DefaultLifecycleObserver *onResume()* method:

```
@Override
public void onResume(@NonNull LifecycleOwner owner) {
    // Perform tasks in response to Resume status event
}
```

The flowchart in Figure 38-2 illustrates the sequence of state changes for a lifecycle owner and the lifecycle events that will be triggered on observers between each state transition:

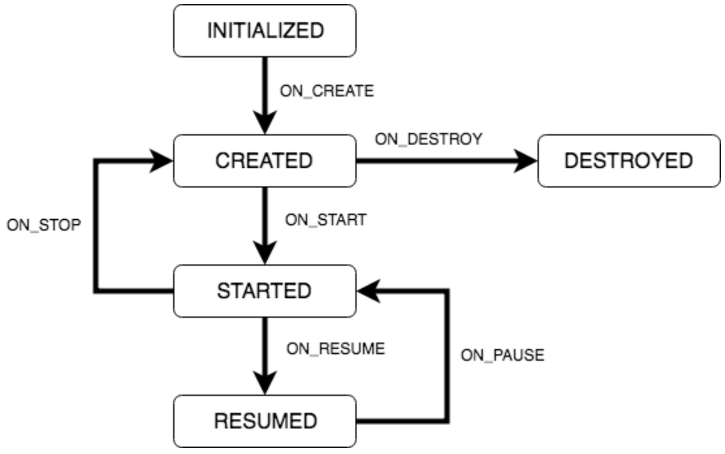


Figure 38-2

38.5 Summary

This chapter has introduced the basics of lifecycle awareness and the classes and interfaces of the Android Lifecycle package included with Android Jetpack. The package contains a number of classes and interfaces that are used to create lifecycle owners, lifecycle observers and lifecycle-aware components. A lifecycle owner has assigned to it a Lifecycle object that maintains a record of the owners state and a list of subscribed observers. When the owner's state changes, the observer is notified via lifecycle event methods so that it can respond to the change.

The next chapter will create an Android Studio project that demonstrates how to work with and create lifecycle-aware components including the creation of both lifecycle observers and owners, and the handling of lifecycle state changes and events.

Chapter 42

42. An Introduction to MotionLayout

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

42.1 An Overview of MotionLayout

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

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

MotionLayout animations are declared entirely in XML and do not typically require that any code be written. These XML declarations may be implemented manually in the Android Studio code editor, visually using the MotionLayout editor, or using a combination of both approaches.

42.2 MotionLayout

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

Unsurprisingly since it is a subclass of ConstraintLayout, MotionLayout supports all of the layout features of the ConstraintLayout. A user interface layout can, therefore, be designed in exactly the same way when using MotionLayout for any views that do not require animation.

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

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

42.3 MotionScene

As we have seen in earlier chapters, an XML layout file contains the information necessary to configure the appearance and layout behavior of the static views presented to the user, and this is still the case when using MotionLayout. For non-static views (in other words the views that will be animated) those views are still declared within the layout file, but the start, end and transition declarations related to those views are stored in a separate XML file referred to as the MotionScene file (so called because all of the declarations are defined within a MotionScene element). This file is imported into the layout XML file and contains the start and end ConstraintSets and Transition declarations (a single file can contain multiple ConstraintSet pairs and Transition declarations allowing different animations to be targeted to specific views within the user interface layout).

The following listing shows a template for a MotionScene file:

```
<?xml version="1.0" encoding="utf-8"?>
<MotionScene
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:motion="http://schemas.android.com/apk/res-auto">

    <Transition
        motion:constraintSetEnd="@+id/end"
        motion:constraintSetStart="@id/start"
        motion:duration="1000">
       <KeyFrameSet>
       </KeyFrameSet>
    </Transition>

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

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

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

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

42.4 Configuring ConstraintSets

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

- alpha
- visibility
- elevation
- rotation
- rotationX
- rotationY

- translationX
- translationY
- translationZ
- scaleX
- scaleY

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

```
<ConstraintSet android:id="@+id/start">
    <Constraint
.
.
        motion:layout_constraintStart_toStartOf="parent"
        android:rotation="0">
    </Constraint>
</ConstraintSet>

<ConstraintSet android:id="@+id/end">
    <Constraint
.
.
        motion:layout_constraintBottom_toBottomOf="parent"
        android:rotation="180">
    </Constraint>
</ConstraintSet>
```

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

42.5 Custom Attributes

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

```
myButton.setBackgroundColor(Color.RED)
```

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

- **motion:customBoolean** - Boolean attribute values.
- **motion:customColorValue** - Color attribute values.
- **motion:customDimension** - Dimension attribute values.

- **motion:customFloatValue** - Floating point attribute values.
- **motion:customIntegerValue** - Integer attribute values.
- **motion:customStringValue** - String attribute values

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

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

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

```
.
.
   <ConstraintSet android:id="@+id/start">
        <Constraint
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            motion:layout_editor_absoluteX="21dp"
            android:id="@+id/button"
            motion:layout_constraintTop_toTopOf="parent"
            motion:layout_constraintStart_toStartOf="parent" >
            <CustomAttribute
                motion:attributeName="backgroundColor"
                motion:customColorValue="#33CC33" />
        </Constraint>
    </ConstraintSet>

    <ConstraintSet android:id="@+id/end">
        <Constraint
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            motion:layout_editor_absoluteY="21dp"
            android:id="@+id/button"
            motion:layout_constraintEnd_toEndOf="parent"
            motion:layout_constraintBottom_toBottomOf="parent" >
            <CustomAttribute
                motion:attributeName="backgroundColor"
                motion:customColorValue="#F80A1F" />
        </Constraint>
    </ConstraintSet>
.
.
```

42.6 Triggering an Animation

Without some form of event to tell MotionLayout to start the animation, none of the settings in the MotionScene file will have any effect on the layout (with the exception that the view will be positioned based on the setting in the *start* ConstraintSet).

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

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

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

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

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

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

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

42.7 Arc Motion

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

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

Figure 42-1 illustrates startVertical and startHorizontal arcs in comparison to the default straight line motion:

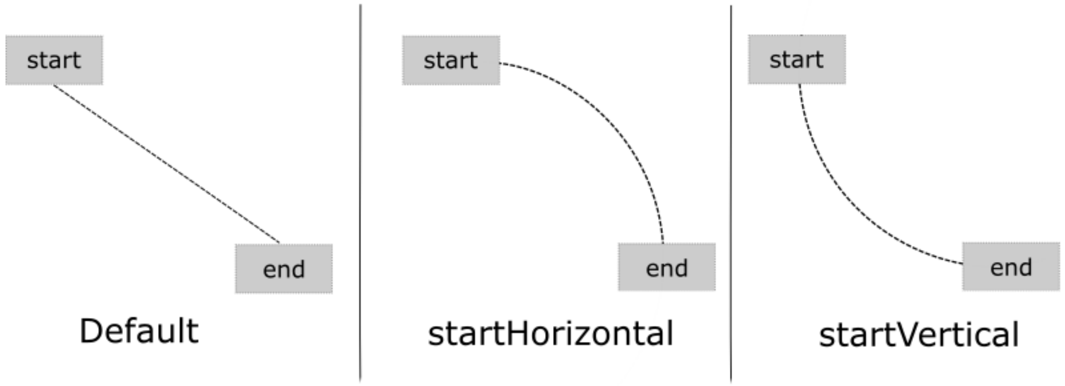


Figure 42-1

42.8 Keyframes

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

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

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

42.8.1 Attribute Keyframes

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

```
<Transition
    motion:constraintSetEnd="@+id/end"
    motion:constraintSetStart="@id/start"
    motion:duration="1000">
   <KeyFrameSet>
       <KeyAttribute
           motion:motionTarget="@+id/button"
           motion:framePosition="50"
           android:scaleX="2.0" />
       <KeyAttribute
           motion:motionTarget="@+id/button"
           motion:framePosition="50"
           android:scaleY="2.0" />
   </KeyFrameSet>
```

42.8.2 Position Keyframes

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

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

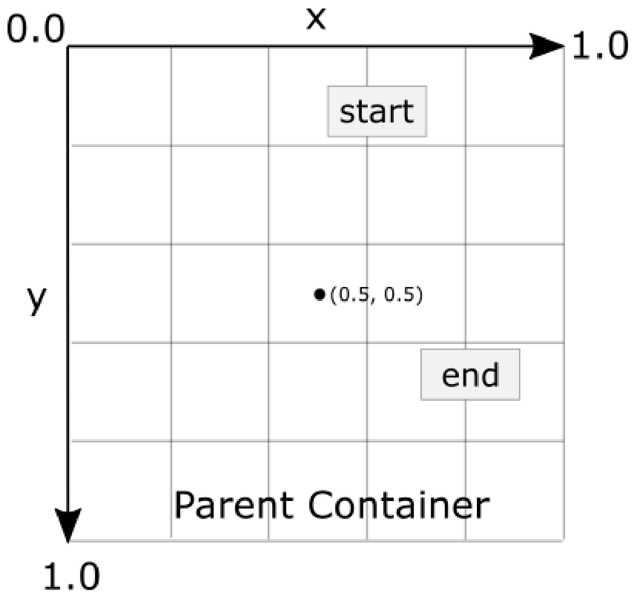


Figure 42-2

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

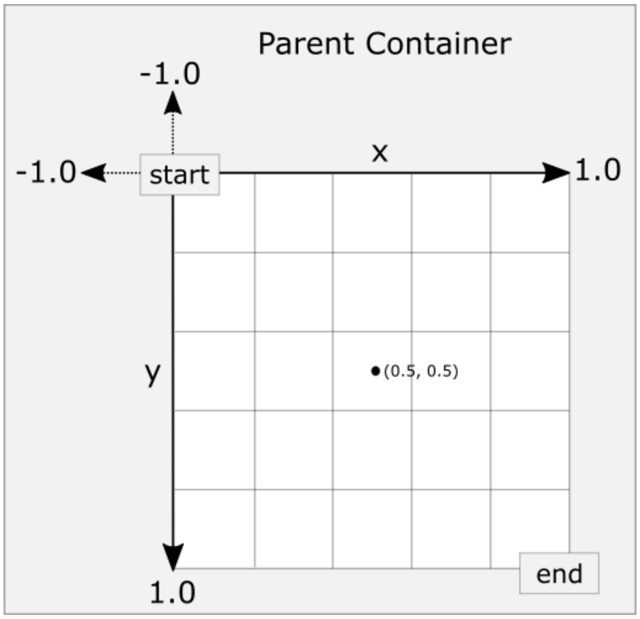


Figure 42-3

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

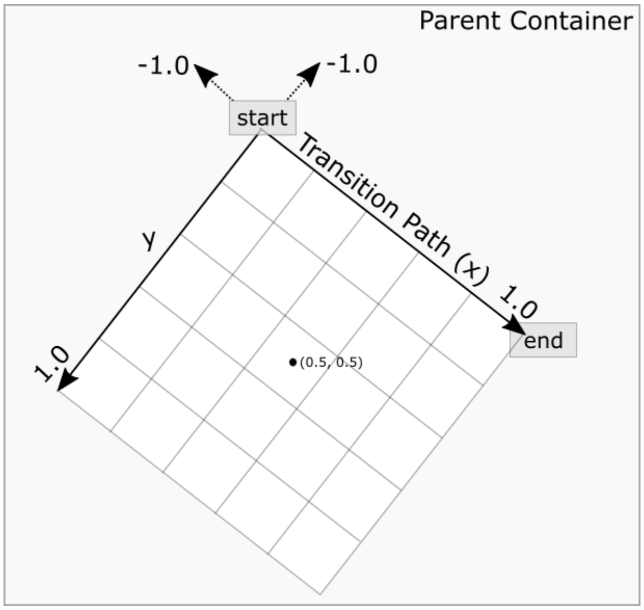


Figure 42-4

As an example, the following ConstraintSets declare start and end points on either side of a device screen. By default, a view transition using these points would move in a straight line across the screen as illustrated in Figure 42-5:

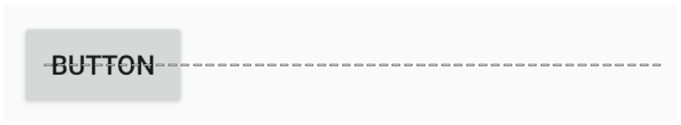


Figure 42-5

Suppose, however, that the view is required to follow a path similar to that shown in Figure 42-6 below:

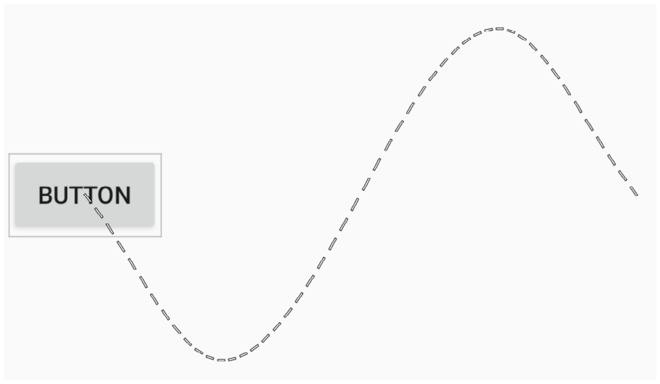


Figure 42-6

To achieve this, key frame position points could be declared within the transition as follows:

```
<KeyPosition
   motion:motionTarget="@+id/button"
   motion:framePosition="25"
   motion:keyPositionType="pathRelative"
   motion:percentY="0.3"
   motion:percentX="0.25"/>

<KeyPosition
   motion:motionTarget="@+id/button"
   motion:framePosition="75"
   motion:keyPositionType="pathRelative"
   motion:percentY="-0.3"
   motion:percentX="0.75"/>
```

The above elements create key frame position points 25% and 75% through the path using the pathRelative coordinate system. The first position is placed at coordinates (0.25, 0.3) and the second at (0.75, -0.3). These position key frames can be visualized as illustrated in Figure 42-7 below:

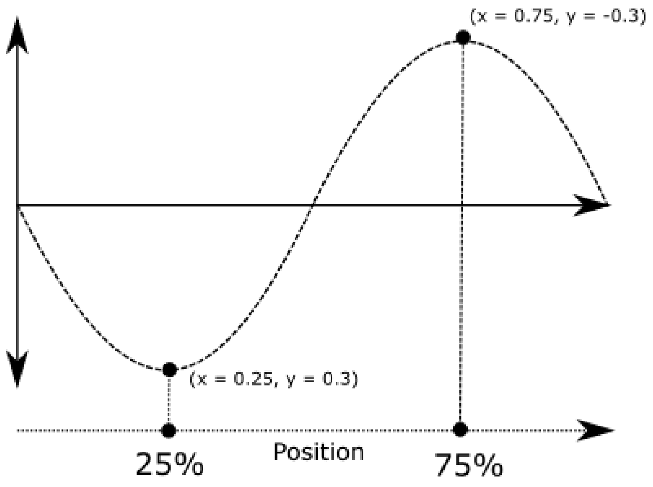


Figure 42-7

42.9 Time Linearity

In the absence of any additional settings, the animations outlined above will be performed at a constant speed. To vary the speed of an animation (for example so that it accelerates and the decelerates) the transition easing attribute (transitionEasing) can be used either within a ConstraintSet or Keyframe.

For complex easing requirements, the linearity can be defined by plotting points on a cubic Bézier curve, for example:

```
.
.
            motion:layout_constraintBottom_toBottomOf="parent"
            motion:transitionEasing="cubic(0.2, 0.7, 0.3, 1)"
            android:rotation="360">
.
.
```

If you are unfamiliar with Bézier curves, consider using the curve generator online at the following URL:

https://cubic-bezier.com/

For most requirements, however, easing can be specified using the built-in *standard*, *accelerate* and *decelerate* values:

```
.
.
            motion:layout_constraintBottom_toBottomOf="parent"
            motion:transitionEasing="decelerate"
            android:rotation="360">
.
.
```

42.10 KeyTrigger

The trigger keyframe (KeyTrigger) allows a method on a view to be called when the animation reaches a specified frame position within the animation timeline. This also takes into consideration the direction of the animations. For example, different methods can be called depending on whether the animation is running forward or backward. Consider a button that is to be made visible when the animation moves beyond 20% of the timeline. The KeyTrigger would be implemented within the KeyFrameSet of the Transition element as follows using the *onPositiveCross* property:

```
.
.
    <KeyFrameSet>
           <KeyTrigger
               motion:framePosition="20"
               motion:onPositiveCross="show"
               motion:motionTarget="@id/button"/>
.
.
```

Similarly, if the same button is to be hidden when the animation is reversed and drops below the 10%, a second key trigger could be added using the *onNegativeCross* property:

```
<KeyTrigger
     motion:framePosition="20"
     motion:onNegativeCross="show"
     motion:motionTarget="@id/button2"/>
```

If the animation is using toggle action, simply use the *onCross* property:

```
<KeyTrigger
     motion:framePosition="20"
     motion:onCross="show"
     motion:motionTarget="@id/button2"/>
```

42.11 Cycle and Time Cycle Keyframes

While position keyframes can be used to add intermediate state changes into the animation this would quickly become cumbersome if large numbers of repetitive positions and changes needed to be implemented. For situations where state changes need to be performed repetitively with predictable changes, MotionLayout includes the Cycle and Time Cycle keyframes, a topic which will be covered in detail in the chapter entitled *"A MotionLayout KeyCycle Tutorial"*.

42.12 Starting an Animation from Code

So far in this chapter we have only looked at controlling an animation using the OnSwipe and OnClick handlers. It is also possible to start an animation from within code by calling methods on the MotionLayout instance. The following code, for example, runs the transition from start to end with a duration of 2000ms for a layout named *motionLayout*:

```
motionLayout.setTransitionDuration(2000);
motionLayout.transitionToEnd();
```

In the absence of addition settings, the start and end states used for the animation will be those declared in the Transition declaration of the MotionScene file. To use specific start and end constraint sets, simply reference them by id in a call to the *setTransition()* method of the MotionLayout instance:

```
motionLayout.setTransitionDuration(2000);
motionLayout.transitionToEnd();
```

To monitor the state of an animation while it is running, add a transition listener to the MotionLayout instance as follows:

```
motionLayout.setTransitionListener(transitionListener);

MotionLayout.TransitionListener transitionListener =
                        new MotionLayout.TransitionListener() {
    @Override
    public void onTransitionStarted(MotionLayout motionLayout,
                                  int startId, int endId) {
            // Called when the transition starts
    }

    @Override
    public void onTransitionChange(MotionLayout motionLayout, int startId,
                                  int endId, float progress) {
            // Called each time a preoperty changes. Track progress value to find
            // current position
    }

    @Override
    public void onTransitionCompleted(MotionLayout motionLayout, int currentId) {
            // Called when the transition is complete
    }

    @Override
    public void onTransitionTrigger(MotionLayout motionLayout, int triggerId,
                                 boolean positive, float progress) {
            // Called when a trigger keyframe threshold is crossed
    }
};
```

42.13 Summary

MotionLayout is a subclass of ConstraintLayout designed specifically to add animation effects to the views in user interface layouts. MotionLayout works by animating the transition of a view between two states defined by start and end constraint sets. Additional animation effects may be added between these start and end points by making use of keyframes.

Animations may be triggered either via OnClick or OnSwipe handlers or programmatically via method calls on the MotionLayout instance.

Chapter 47

47. 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.

47.1 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 if 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.

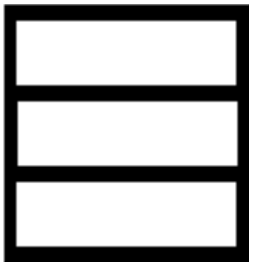


Figure 47-1

- **GridLayoutManager** – The list items are presented in grid format. This manager is best used when the list items are of uniform size.

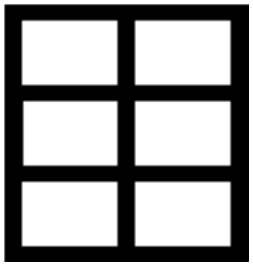


Figure 47-2

- **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.

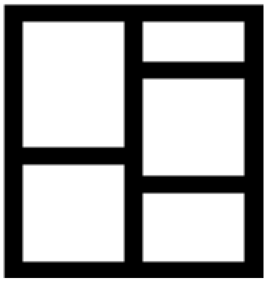


Figure 47-3

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:

- **getItemCount()** – 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 content layout file of the activity in which it is to appear. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context=".MainActivity"
    tools:showIn="@layout/activity_card_demo">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:listItem="@layout/card_layout" />

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

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"*).

47.2 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 47-4, for example, shows three CardView instances configured to display a layout consisting of an ImageView and two TextViews:

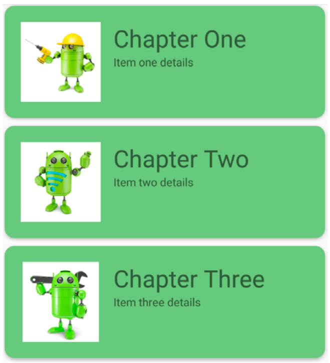


Figure 47-4

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"?>
    <androidx.cardview.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>
</androidx.cardview.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.

47.3 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.

Chapter 81

81. An Overview of Android In-App Billing

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

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

81.1 Preparing a Project for In-App Purchasing

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

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

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

```
dependencies {
.
.
    implementation 'com.android.billingclient:billing:<latest version>'
.
.
}
```

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

81.2 Creating In-App Products and Subscriptions

Products and subscriptions are created and managed using the options listed beneath the Monetize section of the Play Console navigation panel as highlighted in Figure 81-1 below:

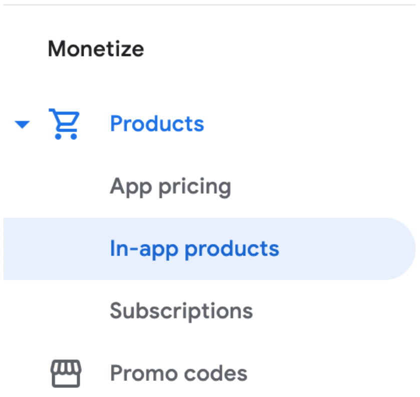


Figure 81-1

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

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

81.3 Billing Client Initialization

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

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

```
private final PurchasesUpdatedListener purchasesUpdatedListener =
                                     new PurchasesUpdatedListener() {
    @Override
    public void onPurchasesUpdated(BillingResult billingResult,
                                   List<Purchase> purchases) {

        if (billingResult.getResponseCode() ==
                BillingClient.BillingResponseCode.OK
                && purchases != null) {

            // Purchase(s) successful

            for (Purchase purchase : purchases) {
                // Process purchases
            }
        } else if (billingResult.getResponseCode() ==
                BillingClient.BillingResponseCode.USER_CANCELED) {
            // User cancelled purchase
        } else {
            // handle errors here
        }
    }
};

private BillingClient billingClient = BillingClient.newBuilder(context)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases()
    .build();
```

81.4 Connecting to the Google Play Billing Library

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

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

```
billingClient.startConnection(new BillingClientStateListener() {

    @Override
    public void onBillingSetupFinished(
            @NonNull BillingResult billingResult) {

        if (billingResult.getResponseCode() ==
                BillingClient.BillingResponseCode.OK) {
            // Connection successful
        } else {
            // Connection failed
        }
    }

    @Override
    public void onBillingServiceDisconnected() {
        // Existing connection lost
    }
});
```

81.5 Querying Available Products

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

```
QueryProductDetailsParams queryProductDetailsParams =
    QueryProductDetailsParams.newBuilder()
        .setProductList(
            ImmutableList.of(
                QueryProductDetailsParams.Product.newBuilder()
                    .setProductId("one_button_click")
                    .setProductType(BillingClient.ProductType.INAPP)
                    .build()))
        .build();

billingClient.queryProductDetailsAsync(queryProductDetailsParams,
    new ProductDetailsResponseListener() {
        public void onProductDetailsResponse(
                @NonNull BillingResult billingResult,
                @NonNull List<ProductDetails> productDetailsList) {

            if (!productDetailsList.isEmpty()) {
                // Process list of matching products
            } else {
                // No product matches found
            }
        }
    }
);
```

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

81.6 Starting the Purchase Process

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

```
BillingFlowParams billingFlowParams =
    BillingFlowParams.newBuilder()
        .setProductDetailsParamsList(
            ImmutableList.of(
                BillingFlowParams.ProductDetailsParams.newBuilder()
```

```
                        .setProductDetails(productDetails)
                        .build()
            )
        )
        .build();

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

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

81.7 Completing the Purchase

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

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

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

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

```
AcknowledgePurchaseParams acknowledgePurchaseParams =
        AcknowledgePurchaseParams.newBuilder()
                .setPurchaseToken(purchase.getPurchaseToken())
                .build();

AcknowledgePurchaseResponseListener acknowledgePurchaseResponseListener =
                   new AcknowledgePurchaseResponseListener() {

    @Override
    public void onAcknowledgePurchaseResponse(
                      @NonNull BillingResult billingResult) {
        // Check acknowledgement result
    }
};
```

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

```
ConsumeParams consumeParams =
    ConsumeParams.newBuilder()
            .setPurchaseToken(purchase.getPurchaseToken())
            .build();

ConsumeResponseListener listener = new ConsumeResponseListener() {
    @Override
    public void onConsumeResponse(BillingResult billingResult,
                                  @NonNull String purchaseToken) {
        if (billingResult.getResponseCode() ==
                BillingClient.BillingResponseCode.OK) {
            // Purchase consumed successfully
        }
    }
};

billingClient.consumeAsync(consumeParams, listener);
```

81.8 Querying Previous Purchases

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

```
QueryPurchasesParams queryPurchasesParams =
        QueryPurchasesParams.newBuilder()
                .setProductType(BillingClient.ProductType.INAPP)
                .build();

billingClient.queryPurchasesAsync(queryPurchasesParams,
                       new PurchasesResponseListener() {
    @Override
    public void onQueryPurchasesResponse(@NonNull BillingResult billingResult,
              @NonNull List<Purchase> list) {
        // Process list of purchases
    }
});
```

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

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

```
QueryPurchaseHistoryParams queryPurchaseHistoryParams =
        QueryPurchaseHistoryParams.newBuilder()
                .setProductType(BillingClient.ProductType.INAPP)
                .build();

billingClient.queryPurchaseHistoryAsync(queryPurchaseHistoryParams,
               new PurchaseHistoryResponseListener() {
```

```
    @Override
    public void onPurchaseHistoryResponse(@NonNull BillingResult billingResult,
                @NonNull List<PurchaseHistoryRecord> list) {
        // Process purchase history
    }
});
```

81.9 Summary

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

Index