Android Studio Electric Eel Essentials

Kotlin Edition

Android Studio Electric Eel Essentials – Kotlin Edition

Rev: 1.0

Contents

Table of Contents

Table of Contents

1. Introduction

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

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

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

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

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

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

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

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

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

1.1 Downloading the Code Samples

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

https://www.ebookfrenzy.com/retail/eelkotlin/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/eelkotlin.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), the Kotlin plug-in and OpenJDK Java development environment.

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

2.1 System requirements

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

- Windows 8/10/11 64-bit

- macOS 10.14 or later running on Intel or Apple silicon

- Chrome OS device with Intel i5 or higher

- Linux systems with version 2.31 or later of the GNU C Library (glibc)

- Minimum of 8GB of RAM

- Approximately 8GB of available disk space

- 1280 x 800 minimum screen resolution

2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio Electric Eel 2022.1.1 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 Electric Eel" 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 Electric Eel 2022.1.1 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:
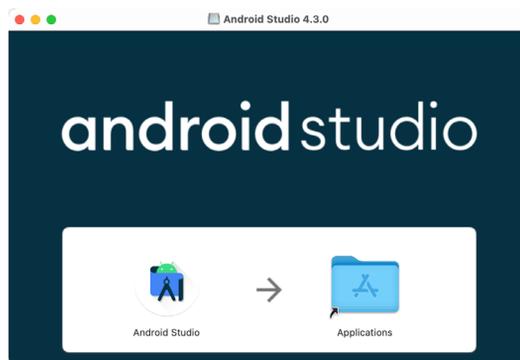
Figure 2-1

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 have previously installed an earlier version of Android Studio, the first time 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.

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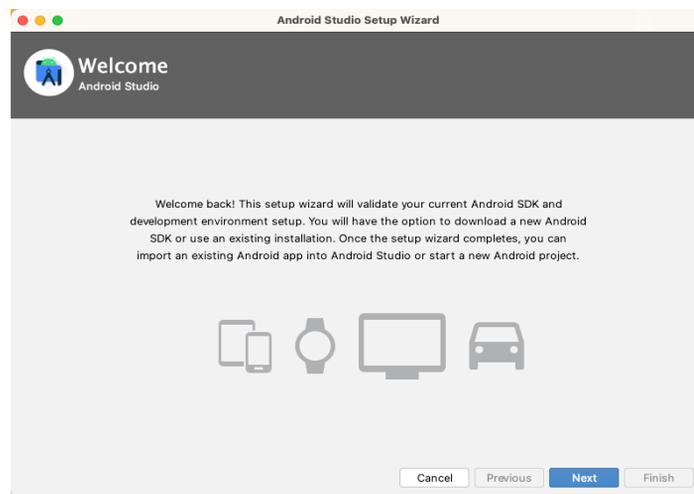
Figure 2-2

If this dialog appears, click the Next button to display the Install Type screen (Figure 2-3). On this screen, select the Standard installation option before clicking Next.
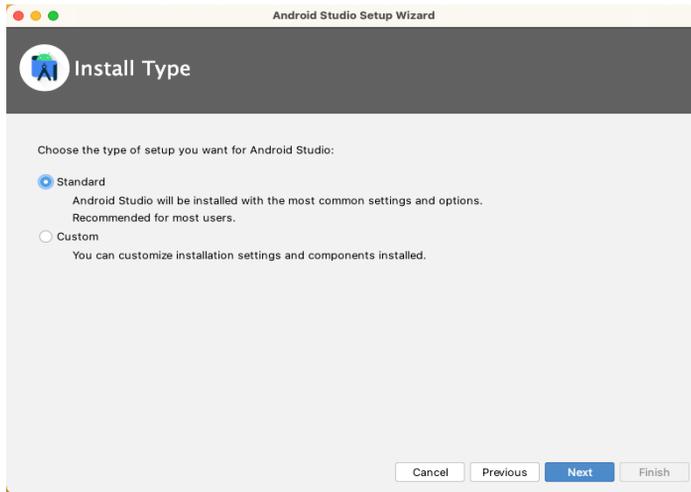
Figure 2-3

On the Select UI Theme screen, select either the Darcula or Light theme based on your preferences. After making a choice, click Next, and review the options in the Verify Settings screen before proceeding to the License Agreement screen. Select each license category and enable the Accept checkbox. Finally, click on the Finish button to initiate the installation.

After these initial setup steps have been taken, click the Finish button to display the Welcome to Android Studio screen using your chosen UI theme:
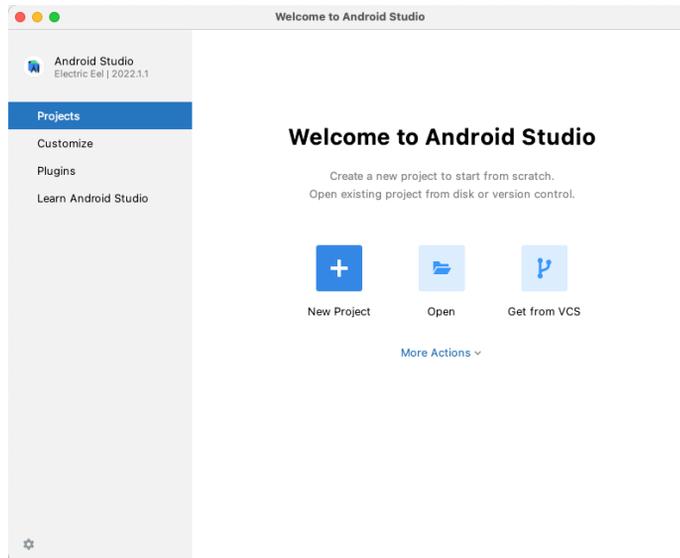
Figure 2-4

2.5 Installing additional Android SDK packages

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

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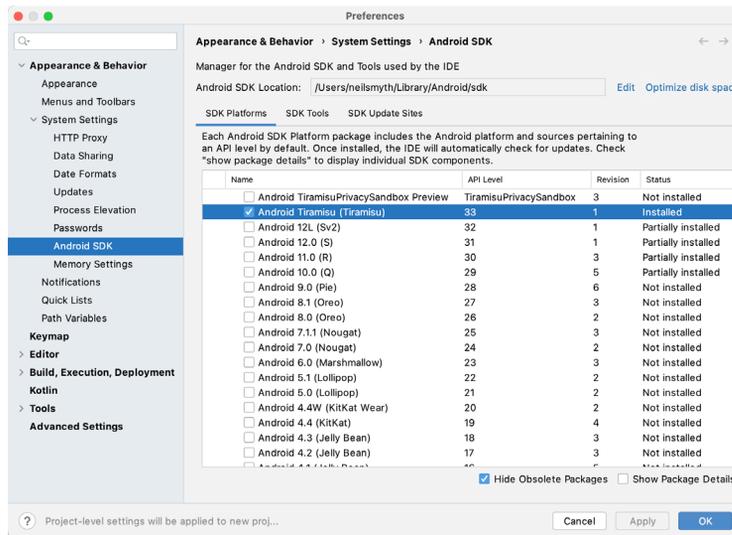
Figure 2-5

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

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

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

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.

Setting up an Android Studio Development Environment

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:
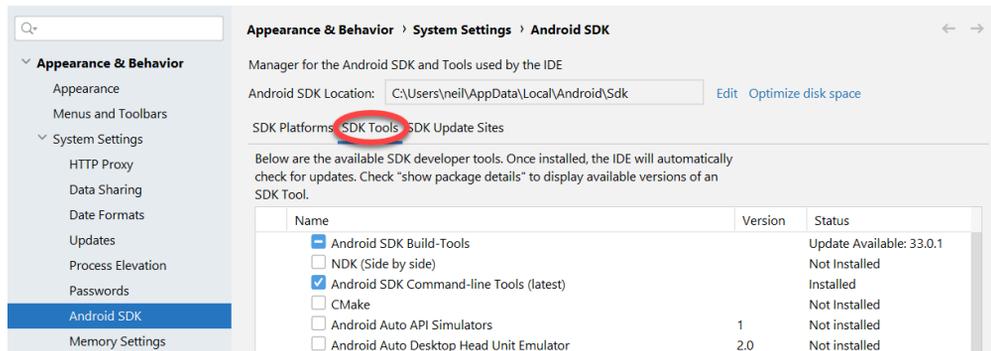
Figure 2-7

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

- Android SDK Build-tools

- Android Emulator

- Android SDK Platform-tools

- Google Play Services

- Intel x86 Emulator Accelerator (HAXM installer)[*]

- Google USB Driver (Windows only)

- Layout Inspector image server for API 31 and T

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

If any of the above packages are listed as *Not Installed* or requiring an update, simply select the checkboxes next to those packages and click on the *Apply* button to initiate the installation process. If the HAXM emulator settings dialog appears, select the recommended memory allocation:
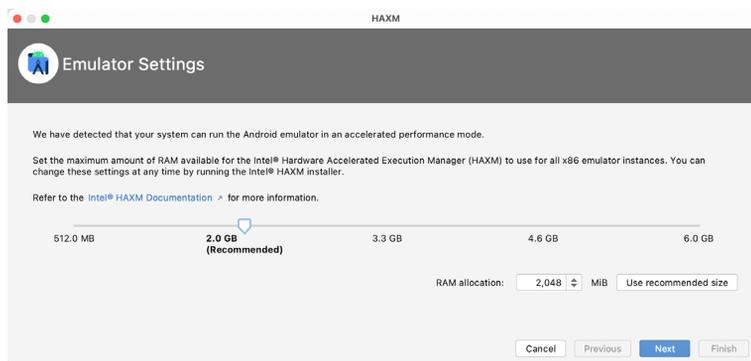
Figure 2-8

Once the installation is complete, review the package list and make sure that the selected packages are now listed

as *Installed* in the *Status* column. If any are listed as *Not installed,* make sure they are selected and click on the *Apply* button again.

2.6 Installing the Android SDK Command-line Tools

Android Studio includes a set of tools that allow some tasks to be performed from your operating system command line. To install these tools on your system, open the SDK Manager, select the SDK Tools tab and enable the *Show Package Details* option in the bottom left-hand corner of the window. Next, scroll down the list of packages and, when the *Android SDK Command-line Tools (latest)* package comes into view, enable it as shown in Figure 2-9:
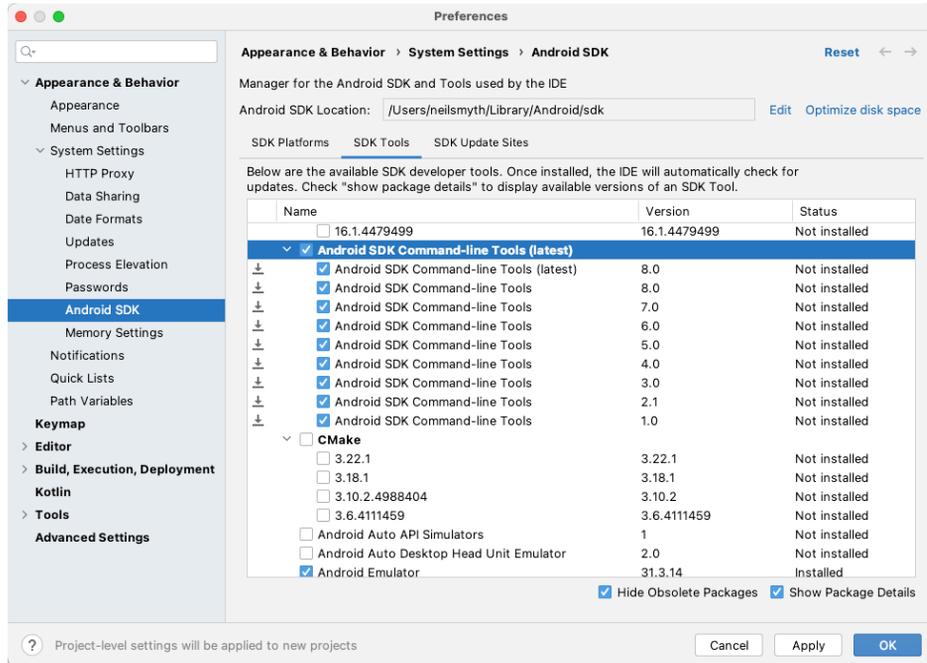
Figure 2-9

After you have selected the command-line tools package, click on Apply followed by OK to complete the installation. When the installation completes, click Finish and close the SDK Manager dialog.

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 your operating system, you will need to configure the PATH environment variable to include the following paths (where *<path_to_android_sdk_installation>* represents the file system location into which you installed the Android SDK):

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

You can identify the location of the SDK on your system 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 2-10:
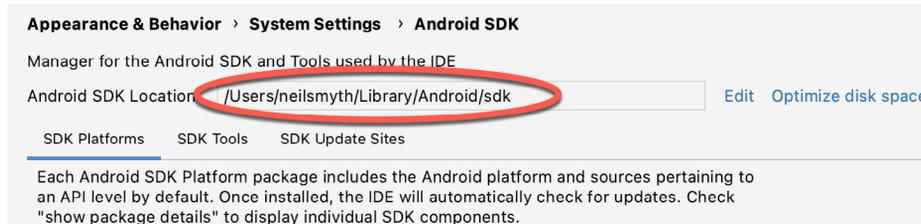
Figure 2-10

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

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

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

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

2.7 Android Studio memory management

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

If you are running Android Studio on a system with sufficient unused RAM to increase these values (this feature is only available on 64-bit systems with 5GB or more of RAM) and find that Android Studio performance appears to be degraded it may be worth experimenting with these memory settings. Android Studio may also notify you that performance can be increased via a dialog similar to the one shown below:
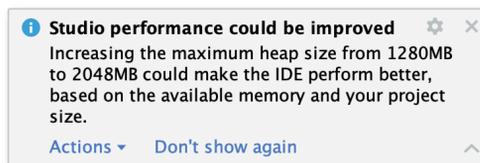
Figure 2-11

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 *Appearance & Behavior* followed by the *Memory Settings* option listed under *System Settings* in the left-hand navigation panel, as illustrated in Figure 2-12 below.

When changing the memory allocation, be sure not to allocate more memory than necessary or than your system can spare without slowing down other processes.
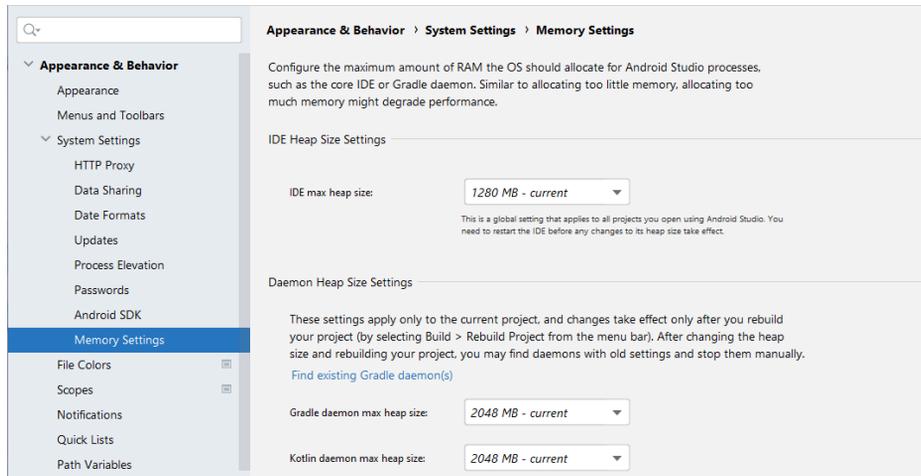
Figure 2-12

The IDE heap size setting adjusts the memory allocated to Android Studio and applies regardless of the currently loaded project. On the other hand, when a project is built and run from within Android Studio, a number of background processes (referred to as daemons) perform the task of compiling and running the app. When compiling and running large and complex projects, build time could be improved by adjusting the daemon heap settings. Unlike the IDE heap settings, these daemon settings apply only to the current project and can only be accessed when a project is open in Android Studio.

2.8 Updating Android Studio and the SDK

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

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

2.9 Summary

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

3. Creating an Example Android App in Android Studio

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

3.1 About the Project

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

3.2 Creating a New Android Project

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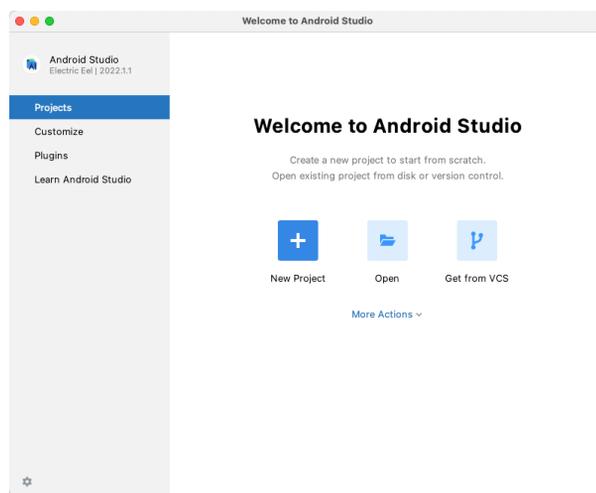
Figure 3-1

Once this window appears, Android Studio is ready for a new project to be created. To create the new project,

simply click on the *New Project* option to display the first screen of the *New Project* wizard.

3.3 Creating an Activity

The first step is to define the type of initial activity that is to be created for the application. Options are available to create projects for Phone and Tablet, Wear OS, TV, Android Audio or Android Things. A range of different activity types is available when developing Android applications, many of which will be covered extensively in later chapters. For the purposes of this example, however, simply select the *Phone and Tablet* option from the Templates panel followed by the option to create an *Empty Activity.* The Empty Activity option creates a template user interface consisting of a single TextView object.
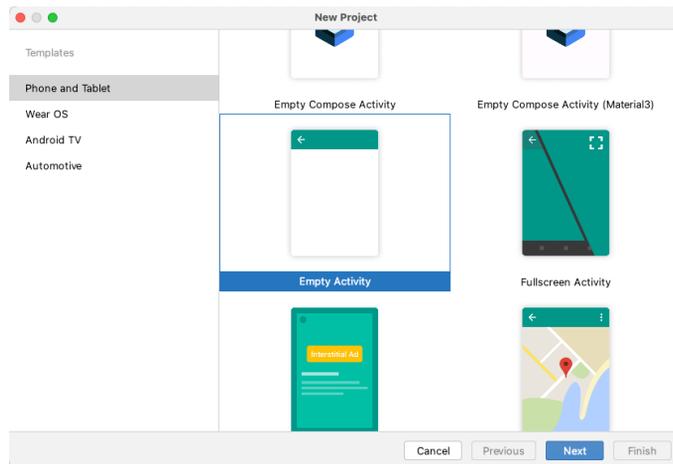
Figure 3-2

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

3.4 Defining the Project and SDK Settings

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

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

```
com.mycompany.androidsample
```

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

```
com.example.androidsample
```

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

Set the minimum SDK setting to API 26: Android 8.0 (Oreo). This is the minimum SDK that will be used in

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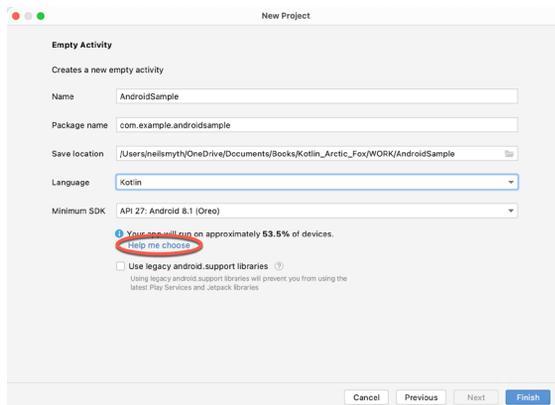
Figure 3-3

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

3.5 Modifying the Example Application

At this point, Android Studio has created a minimal example application project and opened the main window.
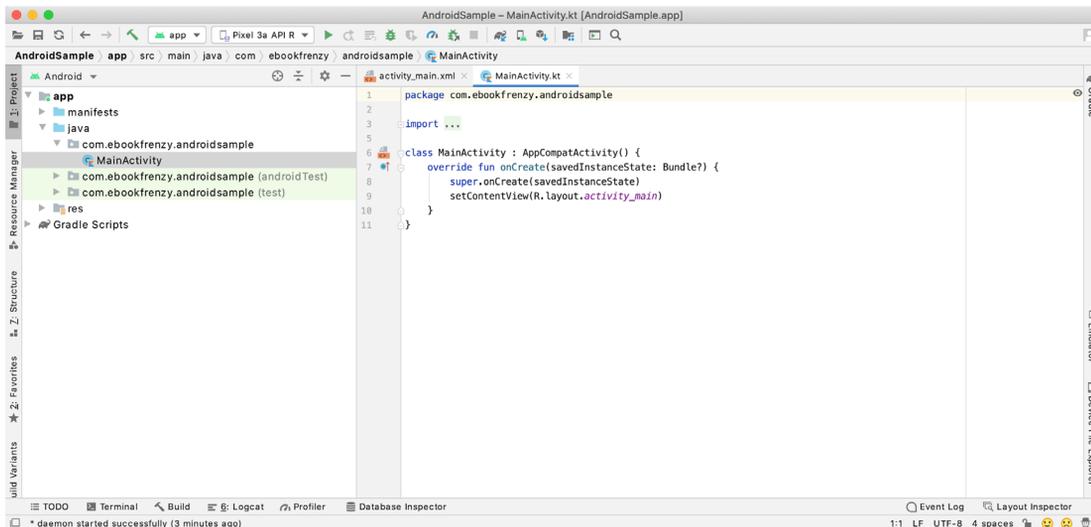
Figure 3-4

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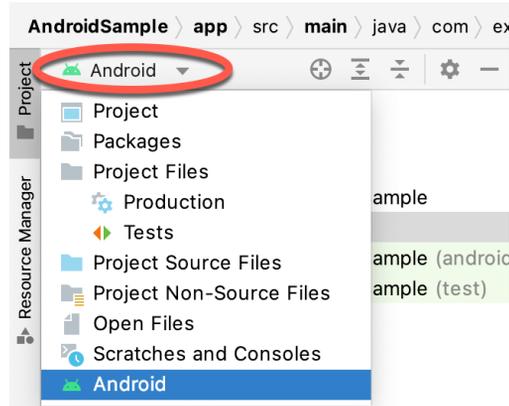
Figure 3-5

3.6 Modifying the User Interface

The user interface design for our activity is stored in a file named *activity_main.xml* which, in turn, is located under *app -> res -> layout* in the project file hierarchy. Once located in the Project tool window, double-click on the file to load it into the user interface Layout Editor tool which will appear in the center panel of the Android Studio main window:
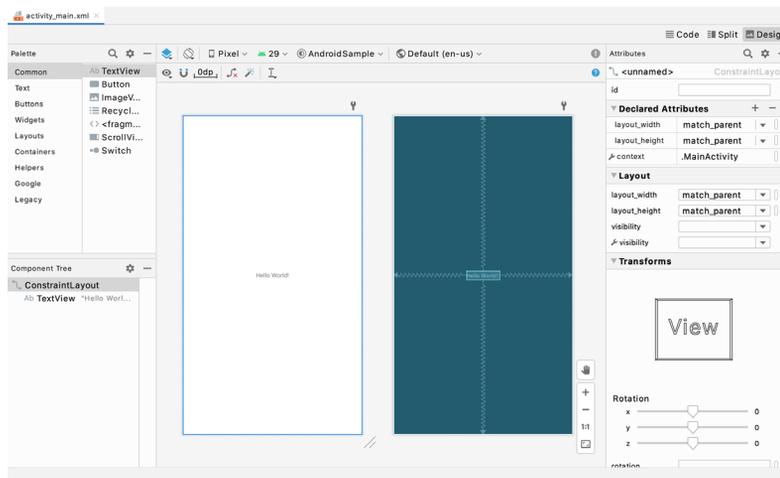
Figure 3-6

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

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

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

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

Figure 3-7

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

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

Figure 3-8

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

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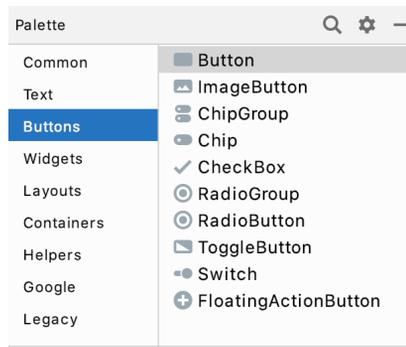
Figure 3-9

Click and drag the *Button* object from the Buttons list and drop it in the horizontal center of the user interface design so that it is positioned beneath the existing TextView widget:
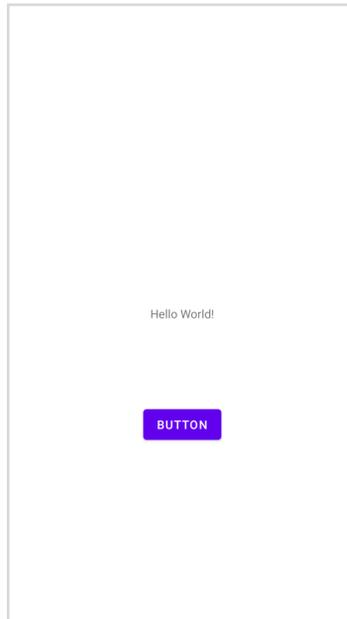
Figure 3-10

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

Figure 3-11

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

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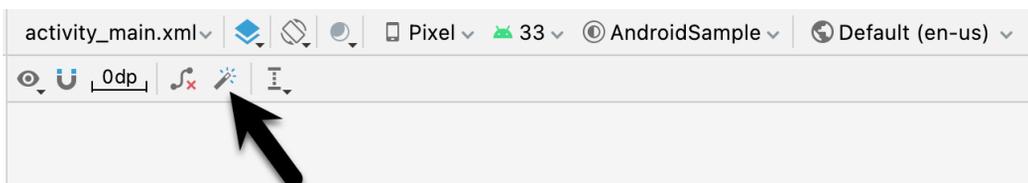
Figure 3-12

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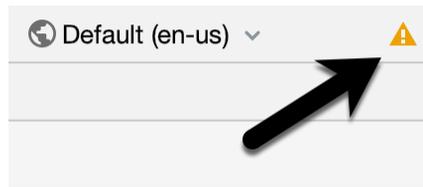
Figure 3-13

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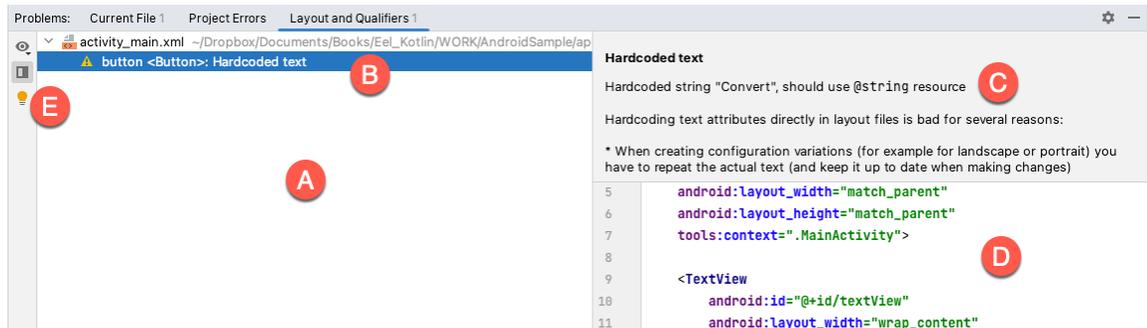
Figure 3-14

This tool window is divided into two panels. The left panel (marked A in the above figure) lists issues detected within the layout file. In our example, only the following problem is listed:

```
button <Button>: Hardcoded text
```

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

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

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

This I18N message 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".

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

Figure 3-15

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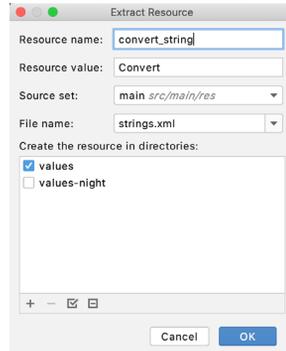
Figure 3-16

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 3-17:
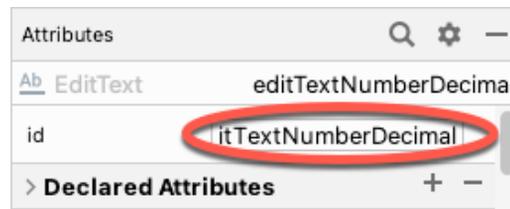
Figure 3-17

Change the id to *dollarText* and, in the Rename dialog, click on the *Refactor* button. This ensures that any references elsewhere within the project to the old id are automatically updated to use the new id:
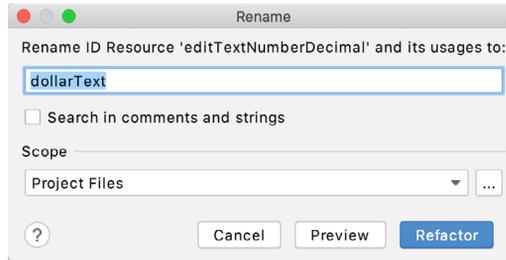
Figure 3-18

Repeat the steps to set the id of the TextView widget to *textView*.

Add any missing layout constraints by clicking on the *Infer constraints* button. At this point the layout should resemble that shown in Figure 3-19:
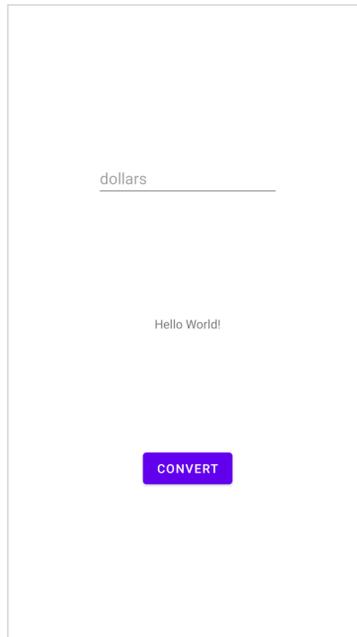
Figure 3-19

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 3-20 below:

Creating an Example Android App in Android Studio

Figure 3-20

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 3-21:
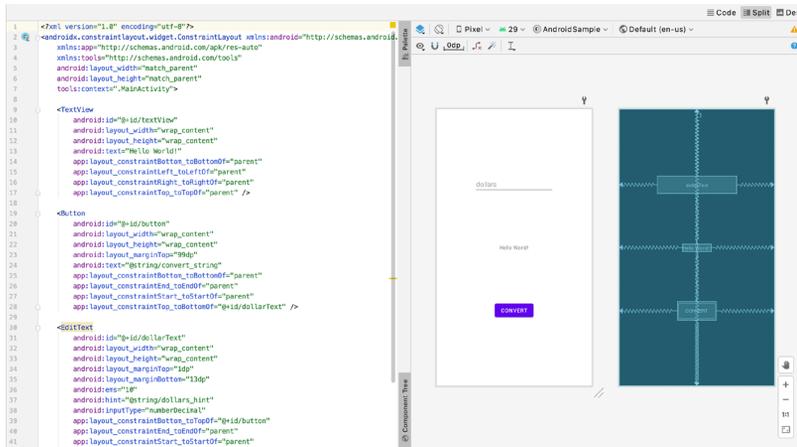
Figure 3-21

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

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

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

Note that the color of the layout changes in real-time to match the new setting in the XML file. Note also that a small red square appears in the left-hand margin (also referred to as the *gutter*) of the XML editor next to the line containing the color setting. This is a visual cue to the fact that the color red has been set on a property. Clicking on the red square will display a color chooser allowing a different color to be selected:
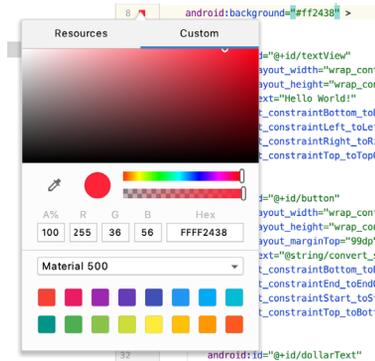
Figure 3-22

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 by clicking on the *Open editor* link in the top left-hand corner of the editor window. This will display the Translation Editor in the main panel of the Android Studio window:
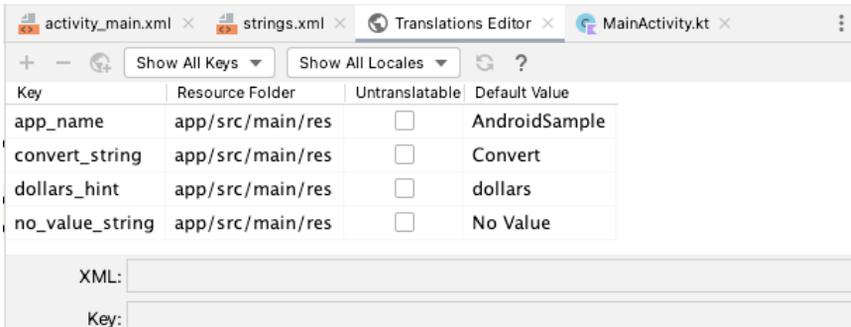
Figure 3-23

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

3.8 Adding the Kotlin Extensions Plugin

The next section will add some code to the project so that a currency conversion occurs when the button is tapped and the result displayed to the user. Before adding this code, however, we first need to add a plugin to the project build configuration which will make it easier for us to reference the user interface widgets from within the Kotlin code. To do this, begin by opening the module level *build.gradle* file located in the project tool window (*app -> Gradle Scripts - > build.gradle (Module :app)*) as shown in Figure 3-24:
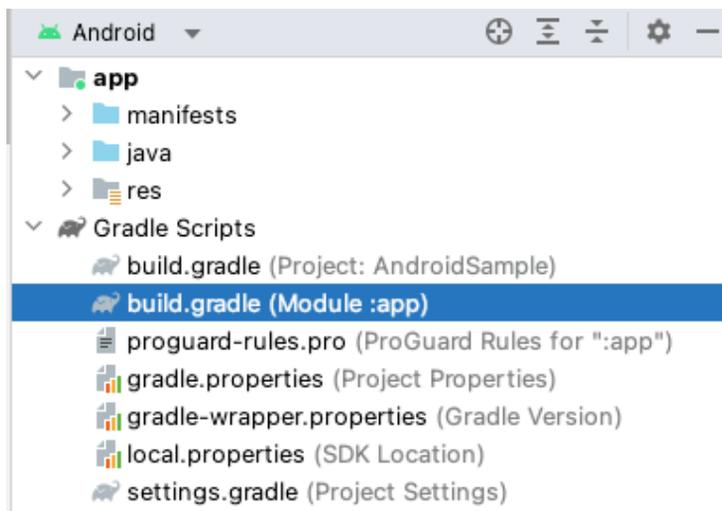
Figure 3-24

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

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

Finally, click on the *Sync Now* link highlighted in below to commit the change and update the project:
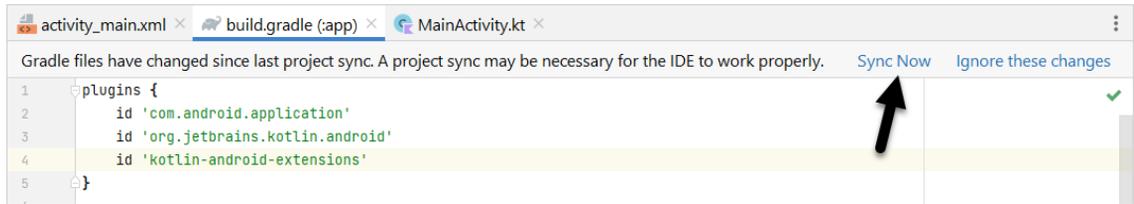
Figure 3-25

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

3.9 Adding Interaction

The final step in this example project is to make the app interactive so that when the user enters a dollar value into the EditText field and clicks the convert button the converted euro value appears on the TextView. This involves the implementation of some event handling on the Button widget. Specifically, the Button needs to be configured so that a method in the app code is called when an *onClick* event is triggered. Event handling can be implemented in a number of different ways and is covered in detail in a later chapter entitled *"An Overview and Example of Android Event Handling"*. Return the layout editor to Design mode, select the Button widget in the layout editor, refer to the Attributes tool window and specify a method named *convertCurrency* as shown below:
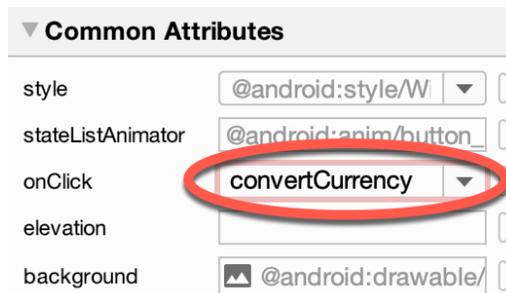
Figure 3-26

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

```
package com.example.androidsample

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

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

class MainActivity : AppCompatActivity() {

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

```
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    fun convertCurrency(view: View) {

        if (dollarText.text.isNotEmpty()) {

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

            val euroValue = dollarValue * 0.85f

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

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

3.10 Summary

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

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

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

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

Although the Android Studio Preview panel allows us to see the layout we are designing and test basic functionality using interactive mode, it be will necessary to compile and run an entire app to fully test that it works. An Android application may be tested by installing and running it either on a physical device or in an *Android Virtual Device (AVD)* emulator environment. Before an AVD can be used, it must first be created and configured to match the specifications of a particular device model. In this chapter, we will work through the creation of such a virtual device using the Pixel 4 phone as a reference example.

4.1 About Android Virtual Devices

AVDs are essentially emulators that allow Android applications to be tested without the necessity to install the application on a physical Android-based device. An AVD may be configured to emulate a variety of hardware features including options such as screen size, memory capacity, and the presence or otherwise of features such as a camera, GPS navigation support, or an accelerometer. As part of the standard Android Studio installation, several emulator templates are installed allowing AVDs to be configured for a range of different devices. Custom configurations may be created to match any physical Android device by specifying properties such as processor type, memory capacity, and the size and pixel density of the screen.

An AVD session can appear either as a separate window or embedded within the Android Studio window.

New AVDs are created and managed using the Android Virtual Device Manager, which may be used either in command-line mode or with a more user-friendly graphical user interface. To create a new AVD, the first step is to launch the AVD Manager. This can be achieved from within the Android Studio environment by selecting the *Tools -> Device Manager* menu option from within the main window.

Once launched, the manager will appear as a tool window as shown in Figure 4-1:
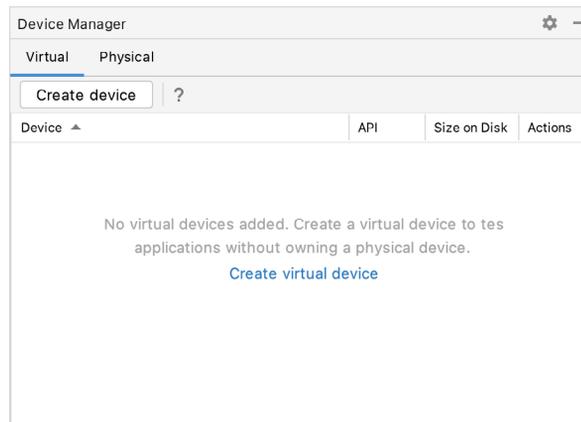
Figure 4-1

If you installed Android Studio for the first time on a computer (as opposed to upgrading an existing Android

Creating an Android Virtual Device (AVD) in Android Studio

Studio installation), the installer might have created an initial AVD instance ready for use, as shown in Figure 4-2:
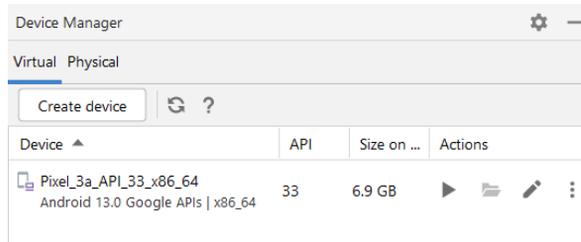
Figure 4-2

If this AVD is present on your system, you can use it to test apps. If no AVD was created, or you would like to create AVDs for different device types, follow the steps in the rest of this chapter.

To add a new AVD, begin by making sure that the Virtual tab is selected before clicking on the *Create device* button to open the *Virtual Device Configuration* dialog:
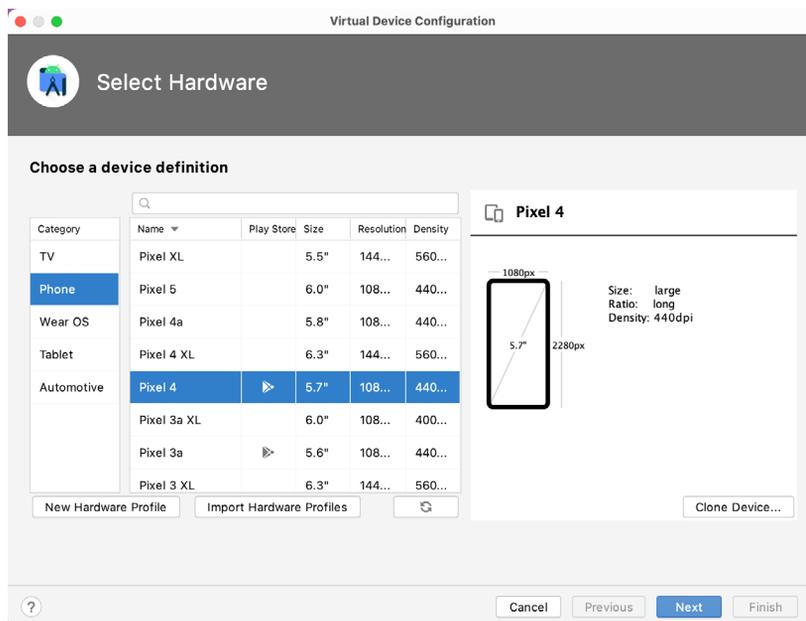
Figure 4-3

Within the dialog, perform the following steps to create a Pixel 4 compatible emulator:

1. From the *Category* panel, select the *Phone* option to display the list of available Android phone AVD templates.

2. Select the *Pixel 4* device option and click *Next*.

3. On the System Image screen, select the latest version of Android. Note that if the system image has not yet been installed a *Download* link will be provided next to the Release Name. Click this link to download and install the system image before selecting it. If the image you need is not listed, click on the *x86 Images* (or *ARM images* if you are running a Mac with Apple Silicon) and *Other images* tabs to view alternative lists.

4. Click *Next* to proceed and enter a descriptive name (for example *Pixel 4 API 33*) into the name field or simply accept the default name.

5. Click *Finish* to create the AVD.

6. With the AVD created, the Device Manager may now be closed. If future modifications to the AVD are necessary, simply re-open the Device Manager, select the AVD from the list, and click on the pencil icon in the Actions column to edit the settings.

4.2 Starting the Emulator

To perform a test run of the newly created AVD emulator, simply select the emulator from the Device Manager and click on the launch button (the triangle in the Actions column). The emulator will appear embedded into the main Android Studio window and begin the startup process. The amount of time it takes for the emulator to start will depend on the configuration of both the AVD and the system on which it is running:
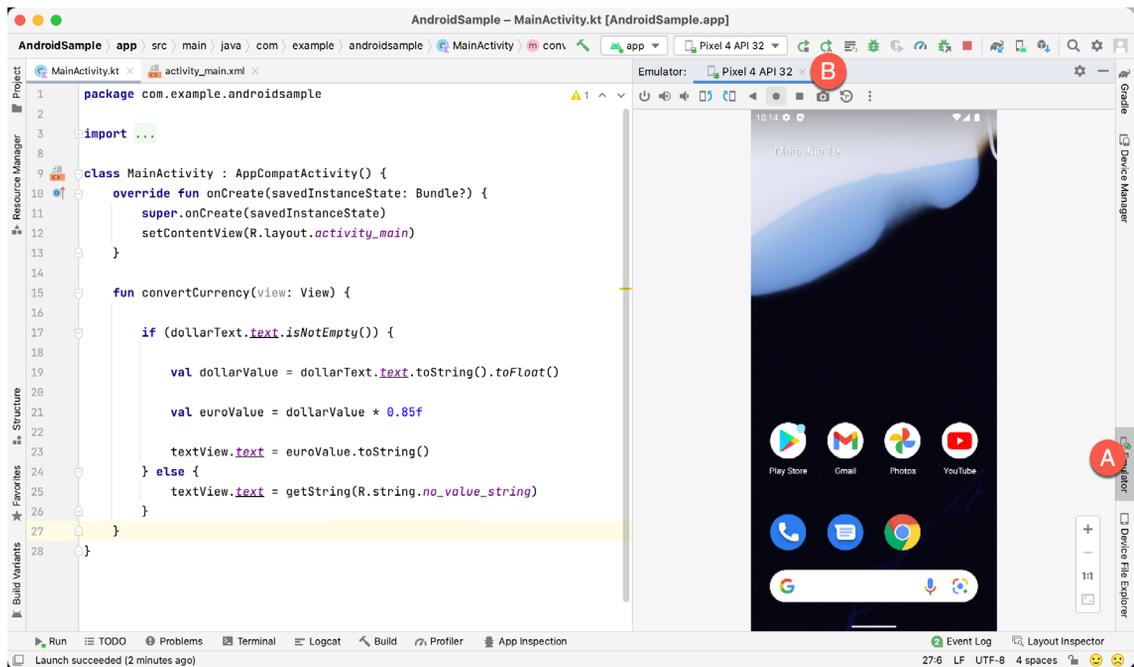
Figure 4-4

To hide and show the emulator tool window, click on the Emulator tool window button (marked A above). Click on the "x" close button next to the tab (B) to exit the emulator. The emulator tool window can accommodate multiple emulator sessions, with each session represented by a tab. Figure 4-5, for example, shows a tool window with two emulator sessions:
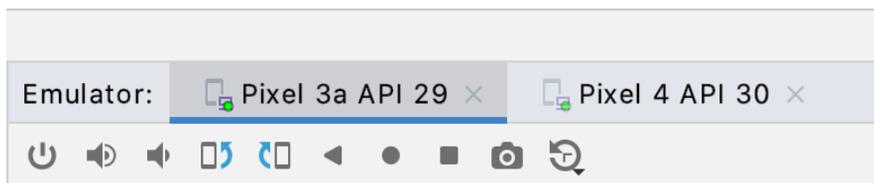
Figure 4-5

To switch between sessions, simply click on the corresponding tab.

Although the emulator probably defaulted to appearing in portrait orientation, this and other default options can be changed. Within the Device Manager, select the new Pixel 4 entry and click on the pencil icon in the *Actions* column of the device row. In the configuration screen, locate the *Startup orientation* section and change the orientation setting. Exit and restart the emulator session to see this change take effect. More details on the emulator are covered in the next chapter *"Using and Configuring the Android Studio AVD Emulator"*).

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

4.3 Running the Application in the AVD

With an AVD emulator configured, the example AndroidSample application created in the earlier chapter now can be compiled and run. With the AndroidSample project loaded into Android Studio, make sure that the newly created Pixel 4 AVD is displayed in the device menu (marked A in Figure 4-6 below), then either click on the run button represented by a green triangle (B), select the *Run -> Run 'app'* menu option or use the Ctrl-R keyboard shortcut:
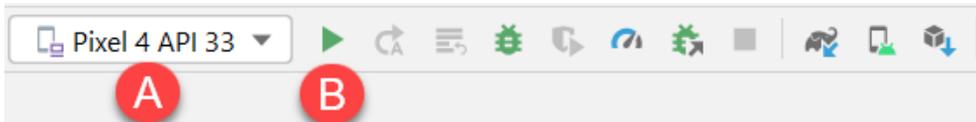
Figure 4-6

The device menu (A) may be used to select a different AVD instance or physical device as the run target, and also to run the app on multiple devices. The menu also provides access to the Device Manager as well as device connection configuration and troubleshooting options:
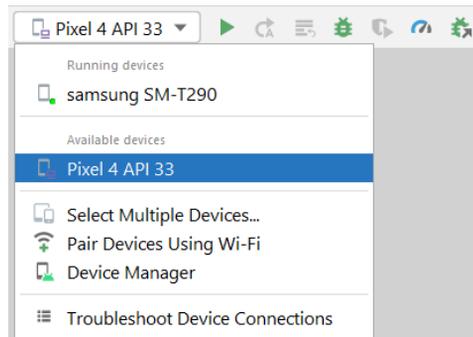
Figure 4-7

Once the application is installed and running, the user interface for the first fragment will appear within the emulator (a fragment is a reusable section of an Android project typically consisting of a user interface layout and some code, a topic which will be covered later in the chapter entitled *"An Introduction to Android Fragments"*):
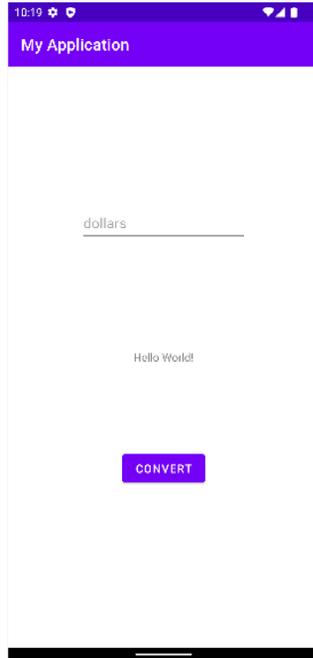
Figure 4-8

If the activity does not automatically launch, check to see if the launch icon has appeared among the apps on the emulator. If it has, simply click on it to launch the application. Once the run process begins, the Run tool window will become available. The Run tool window will display diagnostic information as the application package is installed and launched. Figure 4-9 shows the Run tool window output from a typical successful application launch:

Figure 4-9

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

Assuming that the application loads into the emulator and runs as expected, we have safely verified that the Android development environment is correctly installed and configured. With the app now running, try performing a currency conversion to verify that the app works as intended.

4.4 Running on Multiple Devices

The run menu shown in Figure 4-7 above includes an option to run the app on multiple emulators and devices in parallel. When selected, this option displays the dialog shown in Figure 4-10 providing a list of both the AVDs configured on the system and any attached physical devices. Enable the checkboxes next to the emulators or devices to be targeted before clicking on the Run button:

Creating an Android Virtual Device (AVD) in Android Studio
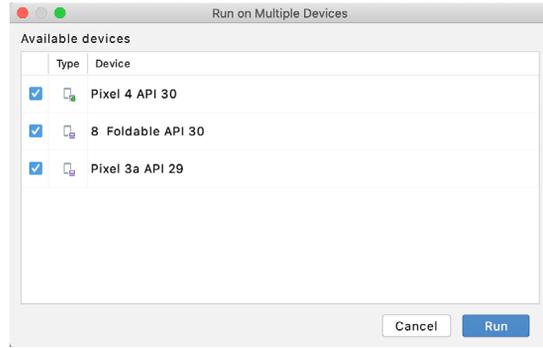
Figure 4-10

After the Run button is clicked, Android Studio will launch the app on the selected emulators and devices.

4.5 Stopping a Running Application

To stop a running application, simply click on the stop button located in the main toolbar as shown in Figure 4-11:
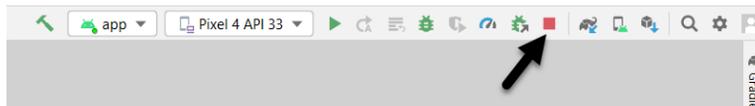
Figure 4-11

An app may also be terminated using the Run tool window. Begin by displaying the *Run* tool window using the window bar button that becomes available when the app is running. Once the Run tool window appears, click the stop button highlighted in Figure 4-12 below:

Figure 4-12

4.6 Supporting Dark Theme

Android 10 introduced the much-awaited dark theme, support for which is enabled by default in Android Studio app projects. To test dark theme in the AVD emulator, open the Settings app within the running Android instance in the emulator. Within the Settings app, choose the *Display* category and enable the *Dark theme* option as shown in Figure 4-13 so that the screen background turns black:
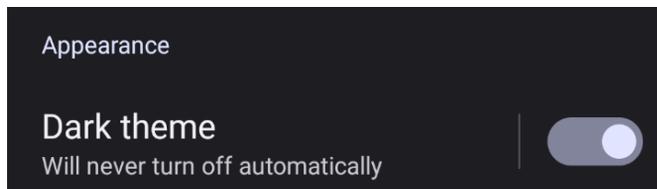
Figure 4-13

With dark theme enabled, run the AndroidSample app and note that it appears using a dark theme including a black background and a purple background color on the button as shown in Figure 4-14:
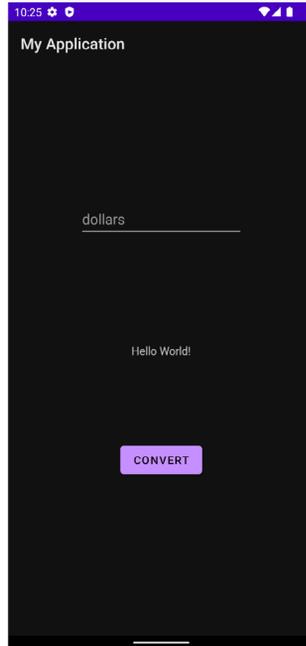
Figure 4-14

Return to the Settings app and turn off Dark theme mode before continuing.

4.7 Running the Emulator in a Separate Window

So far in this chapter, we have only used the emulator as a tool window embedded within the main Android Studio window. To run the emulator in a separate window, select the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS), navigate to *Tools -> Emulator* in the left-hand navigation panel of the preferences dialog, and disable the *Launch in a tool window* option:
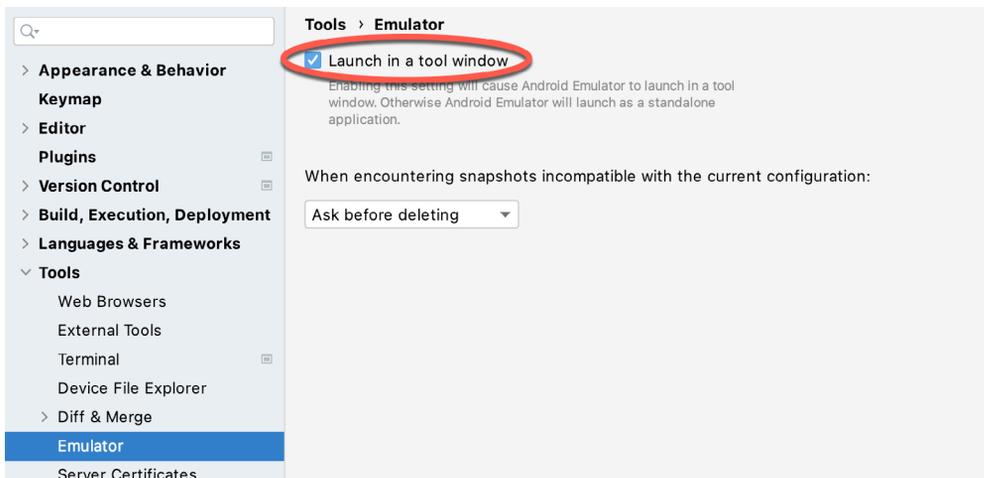
Figure 4-15

With the option disabled, click the Apply button followed by OK to commit the change, then exit the current emulator session by clicking on the close button on the tab marked B in Figure 4-4 above.

Creating an Android Virtual Device (AVD) in Android Studio

Run the sample app once again, at which point the emulator will appear as a separate window as shown below:
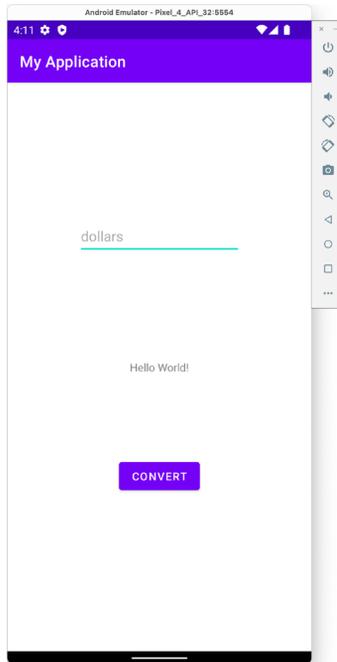
Figure 4-16

The choice of standalone or tool window mode is a matter of personal preference. If you prefer the emulator running in a tool window, return to the settings screen and re-enable the *Launch in a tool window* option. Before committing to standalone mode, however, keep in mind that the emulator tool window may also be detached from the main Android Studio window by clicking on the settings button (represented by the gear icon) in the tool emulator toolbar and selecting the *View Mode -> Float* menu option:
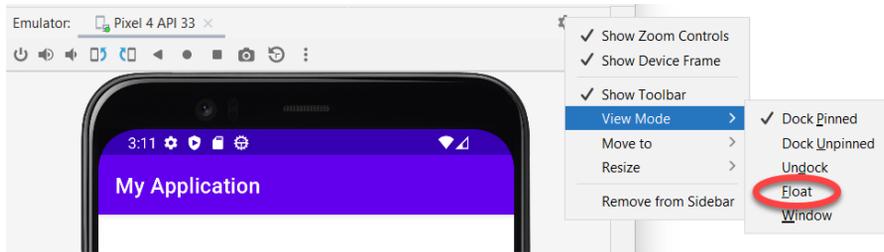
Figure 4-17

4.8 Enabling the Device Frame

The emulator can be configured to appear with (Figure 4-18) or without the device frame (Figure 4-16).
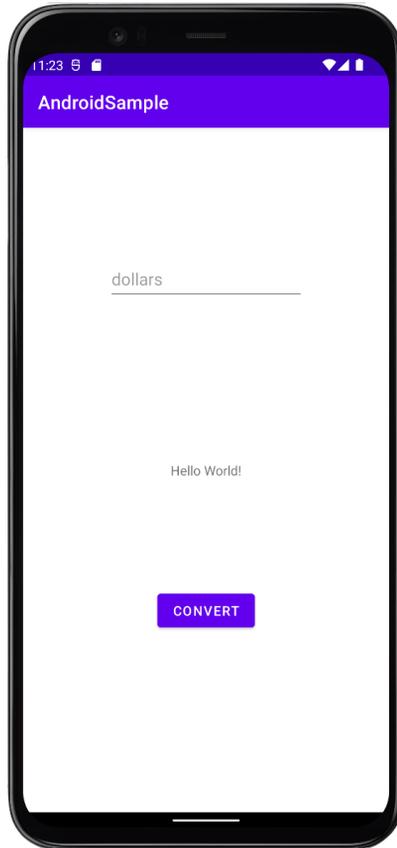
Figure 4-18

To change the setting, open the Device Manager, select the AVD from the list, and click on the pencil icon in the Actions column to edit the settings. In the settings screen, locate and change the Enable Device Frame option:
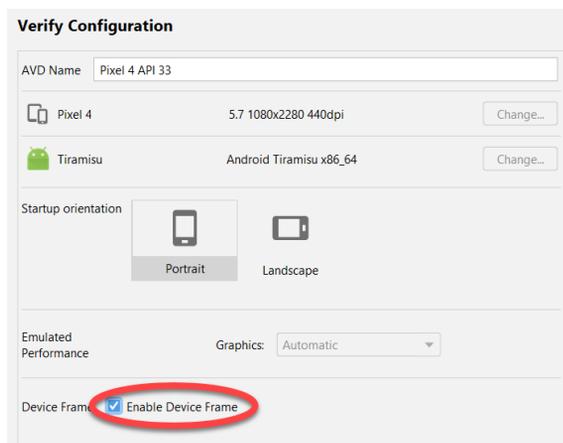
Figure 4-19

4.9 AVD Command-line Creation

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

The avdmanager tool requires access to the Java Runtime Environment (JRE) to run. If, when attempting to run avdmanager, an error message appears indicating that the 'java' command cannot be found, the command prompt or terminal window within which you are running the command can be configured to use the OpenJDK environment bundled with Android Studio. Begin by identifying the location of the OpenJDK JRE as follows:

1. Launch Android Studio and open the ComposeDemo project created earlier in the book.

2. Select the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS).

3. Navigate to the Build, Execution, Deployment section and select the Gradle option listed under the Build Tools category.

4. Click on the *Gradle JDK* setting and make a note of the path for *Android Studio java home*:
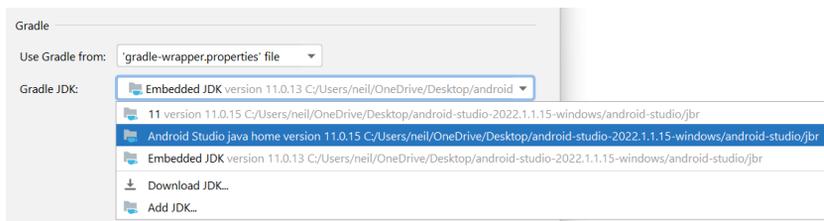
Figure 4-20

On Windows, execute the following command within the command prompt window from which avdmanager is to be run (where *<path to jre>* is replaced by the path copied from the Project Structure dialog above):

```
set JAVA_HOME=<path to jre>
```

On macOS or Linux, execute the following command:

```
export JAVA_HOME="<path to jre>"
```

If you expect to use the avdmanager tool frequently, follow the environment variable steps for your operating system outlined in the chapter entitled *"Setting up an Android Studio Development Environment"* to configure JAVA_HOME on a system-wide basis.

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

```
avdmanager list targets
```

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

```
Available Android targets:
----------
id: 1 or "android-29"
     Name: Android API 29
     Type: Platform
     API level: 29
```

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

The avdmanager tool also allows new AVD instances to be created from the command-line. For example, to create a new AVD named *myAVD* using the target ID for the Android API level 29 device using the x86 ABI, the following command may be used:

```
avdmanager create avd -n myAVD -k "system-images;android-29;google_apis_
playstore;x86"
```

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

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

```
avdmanager list avd

Available Android Virtual Devices:
    Name: Pixel_XL_API_28_No_Play
  Device: pixel_xl (Google)
    Path: /Users/neilsmyth/.android/avd/Pixel_XL_API_28_No_Play.avd
  Target: Google APIs (Google Inc.)
         Based on: Android API 28 Tag/ABI: google_apis/x86
    Skin: pixel_xl_silver
  Sdcard: 512M
```

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

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

4.10 Android Virtual Device Configuration Files

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

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

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

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

4.11 Moving and Renaming an Android Virtual Device

The current name or the location of the AVD files may be altered from the command-line using the *avdmanager* tool's *move avd* argument. For example, to rename an AVD named Pixel4 to Pixel4a, the following command may be executed:

```
avdmanager move avd -n Pixel4 -r Pixel4a
```

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

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

For example, to move an AVD from its current file system location to /tmp/Pixel4Test:

```
avdmanager move avd -n Pixel4 -p /tmp/Pixel4Test
```

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

4.12 Summary

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

The AVD emulator session may be displayed as a standalone window or embedded into the main Android Studio user interface.

5. Using and Configuring the Android Studio AVD Emulator

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

5.1 The Emulator Environment

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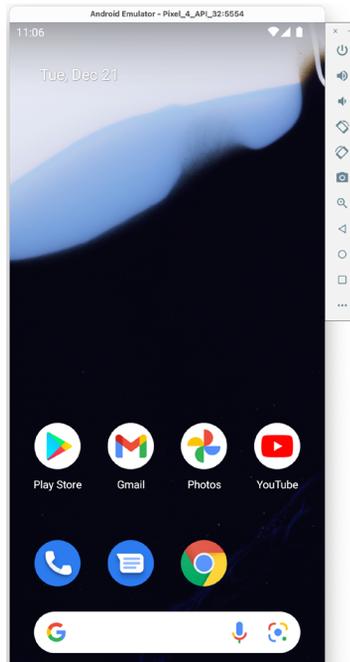
Figure 5-1

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

5.2 Emulator Toolbar Options

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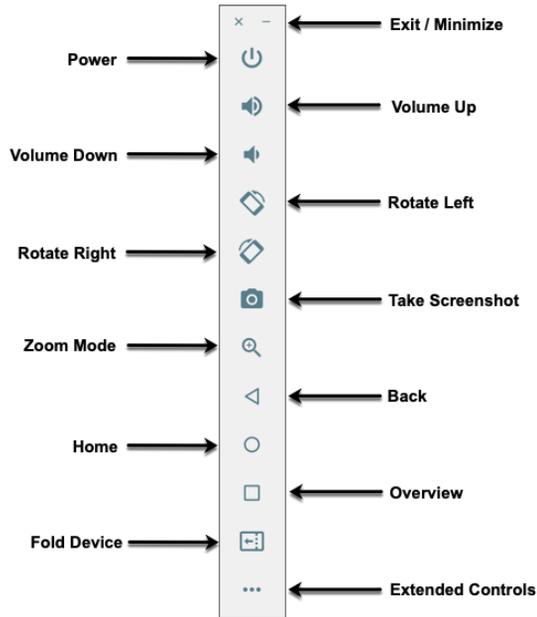
Figure 5-2

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

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

- **Exit / Minimize** – The uppermost 'x' button in the toolbar exits the emulator session when selected while the '-' option minimizes the entire window.

- **Power** – The Power button simulates the hardware power button on a physical Android device. Clicking and releasing this button will lock the device and turn off the screen. Clicking and holding this button will initiate the device "Power off" request sequence.

- **Volume Up / Down** – Two buttons that control the audio volume of playback within the simulator environment.

- **Rotate Left/Right** – Rotates the emulated device between portrait and landscape orientations.

- **Take Screenshot** – Takes a screenshot of the content currently displayed on the device screen. The captured image is stored at the location specified in the Settings screen of the extended controls panel as outlined later in this chapter.

- **Zoom Mode** – This button toggles in and out of zoom mode, details of which will be covered later in this chapter.

- **Back** – Performs the standard Android "Back" navigation to return to a previous screen.

- **Home** – Displays the device home screen.

- **Overview** – Simulates selection of the standard Android "Overview" navigation which displays the currently running apps on the device.

- **Fold Device** – Simulates the folding and unfolding of a foldable device. This option is only available if the emulator is running a foldable device system image.

- **Extended Controls** – Displays the extended controls panel, allowing for the configuration of options such as simulated location and telephony activity, battery strength, cellular network type, and fingerprint identification.

5.3 Working in Zoom Mode

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

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

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

5.4 Resizing the Emulator Window

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

5.5 Extended Control Options

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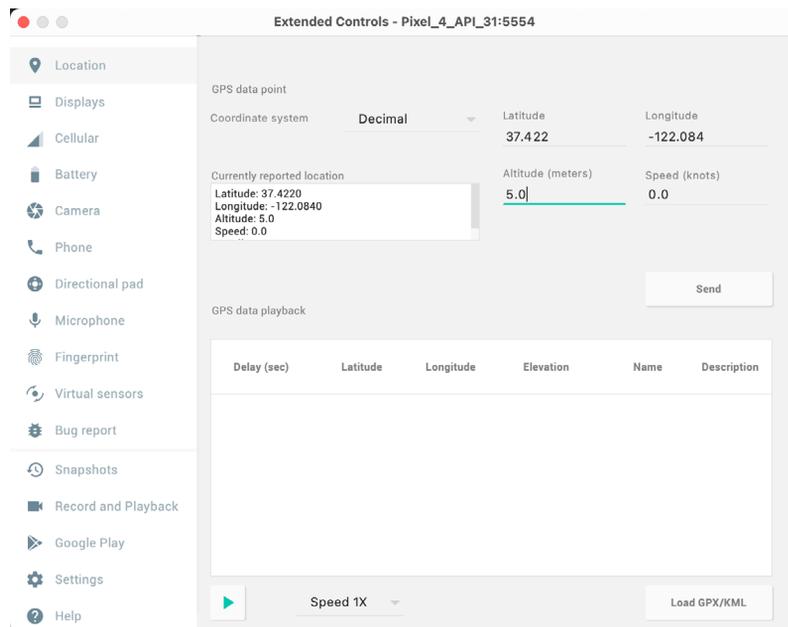
Figure 5-3

5.5.1 Location

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

5.5.2 Displays

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

5.5.3 Cellular

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

5.5.4 Battery

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

5.5.5 Camera

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

5.5.6 Phone

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

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

5.5.7 Directional Pad

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

5.5.8 Microphone

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

5.5.9 Fingerprint

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

5.5.10 Virtual Sensors

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

5.5.11 Snapshots

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

5.5.12 Record and Playback

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

5.5.13 Google Play

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

5.5.14 Settings

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

5.5.15 Help

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

5.6 Working with Snapshots

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

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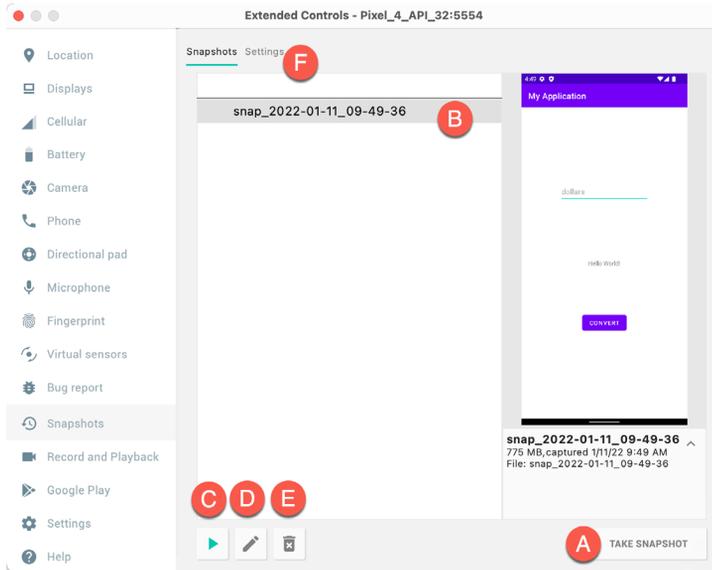
Figure 5-4

The Settings option (F) provides the option to configure the automatic saving of quick-boot snapshots (by default the emulator will ask whether to save the quick boot snapshot each time the emulator exits) and to reload the most recent snapshot. To force an emulator session to perform a cold boot instead of using a previous quick-boot snapshot, open the AVD Manager (*Tools -> AVD Manager*), click on the down arrow in the Actions column for the emulator and select the Cold Boot Now menu option.
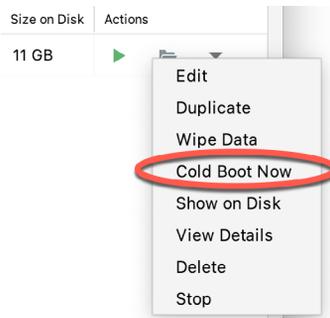
Figure 5-5

5.7 Configuring Fingerprint Emulation

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

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

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

44

Finger 1 is selected in the main settings panel:
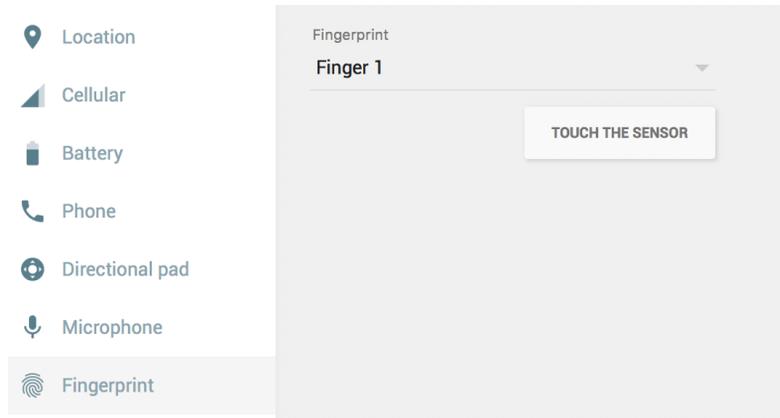
Figure 5-6

Click on the *Touch the Sensor* button to simulate Finger 1 touching the fingerprint sensor. The emulator will report the successful addition of the fingerprint:
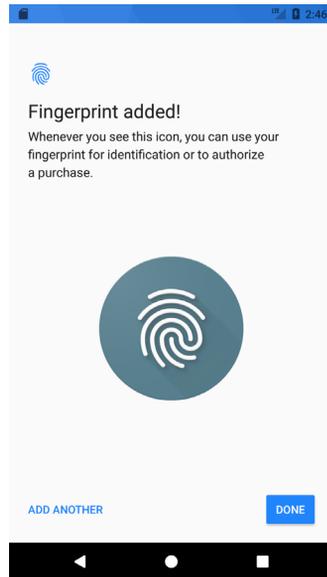
Figure 5-7

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

5.8 The Emulator in Tool Window Mode

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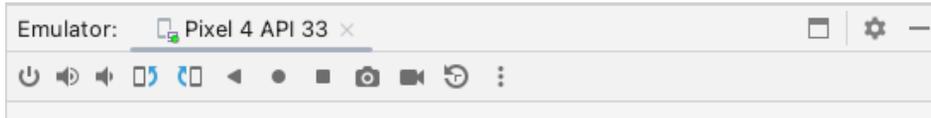
Figure 5-8

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

- Power

- Volume Up

- Volume Down

- Rotate Left

- Rotate Right

- Back

- Home

- Overview

- Screenshot

- Snapshots

- Extended Controls

5.9 Creating a Resizable Emulator

In addition to emulators configured to match specific Android device models, Android Studio also provides a resizable AVD that allows you to switch between phone, tablet and foldable device sizes. To create a resizable emulator, open the Device Manager and click the *Create device* button. Next, select the Resizable device definition illustrated in Figure 5-9, and follow the usual steps to create a new AVD:
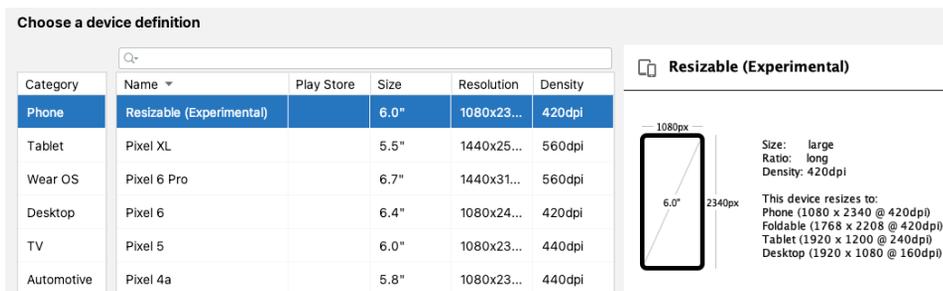
Figure 5-9

When you run an app on the new emulator within a tool window, the *Display mode* option will appear in the toolbar, allowing you to switch between emulator configurations as shown in Figure 5-10:
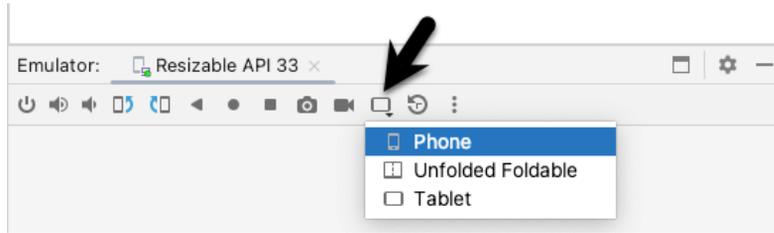
Figure 5-10

If the emulator is running in standalone mode, the Display mode option can be found in the side toolbar as shown below:
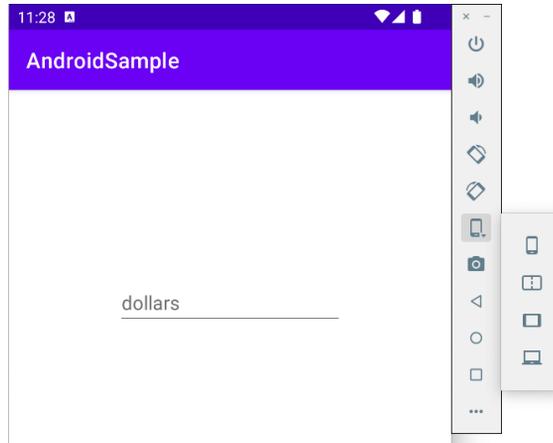
Figure 5-11

5.10 Summary

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

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.
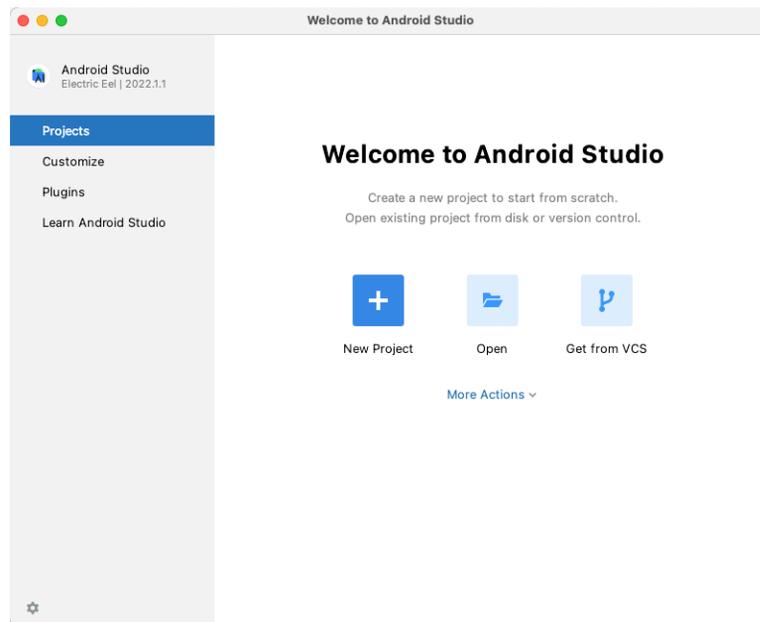
Figure 6-1

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:
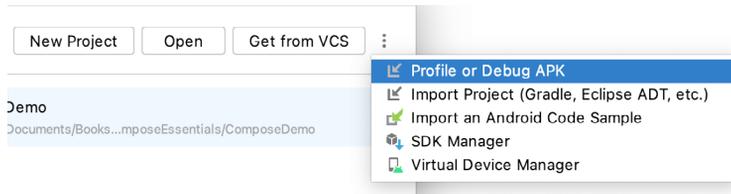
Figure 6-2

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.
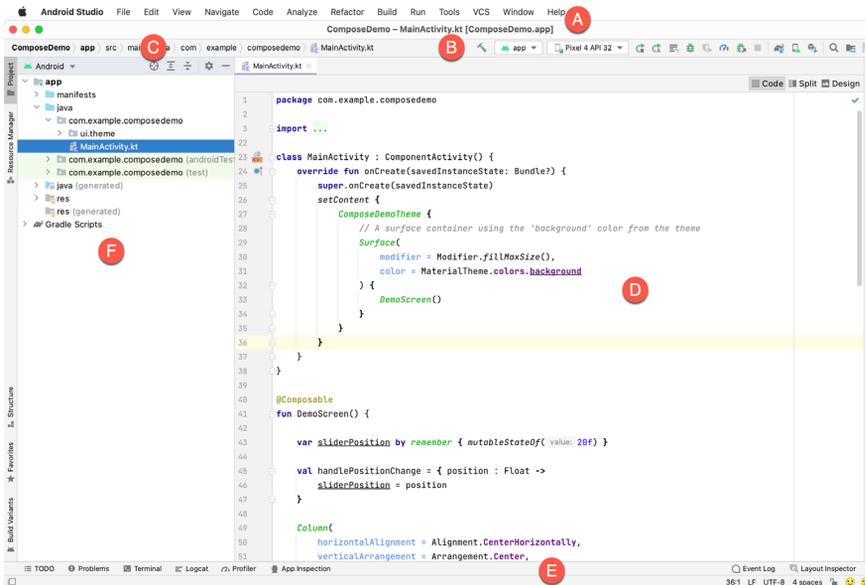
Figure 6-3

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:

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.
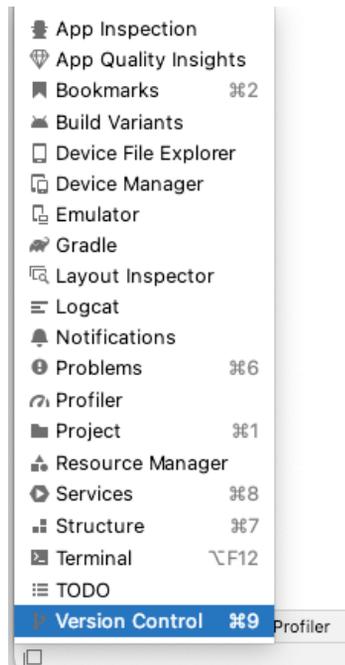
Figure 6-5

A Tour of the Android Studio User Interface

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.
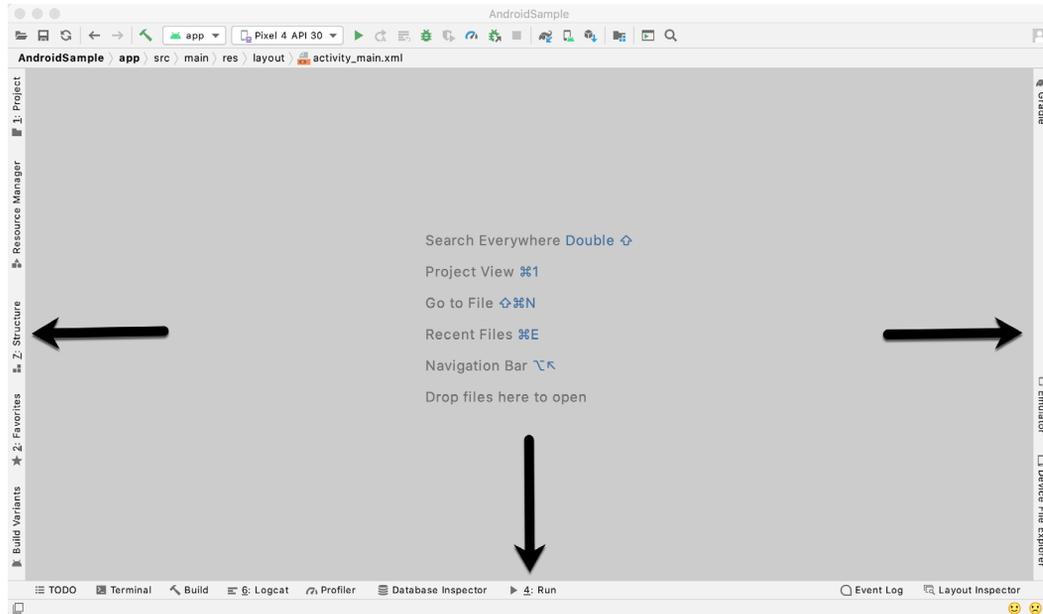
Figure 6-6

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.
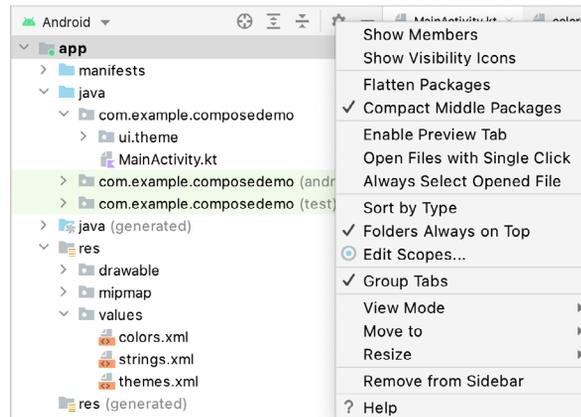
Figure 6-7

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 Inspection** - 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"*.

- **Bookmarks** – The Bookmarks tool window provides quick access to bookmarked files and code lines. For example, right-clicking on a file in the project view allows access to an Add to Bookmarks menu option. Similarly, you can bookmark a line of code in a source file by moving the cursor to that line and pressing the F11 key (F3 on macOS). All bookmarked items can be accessed through this 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:
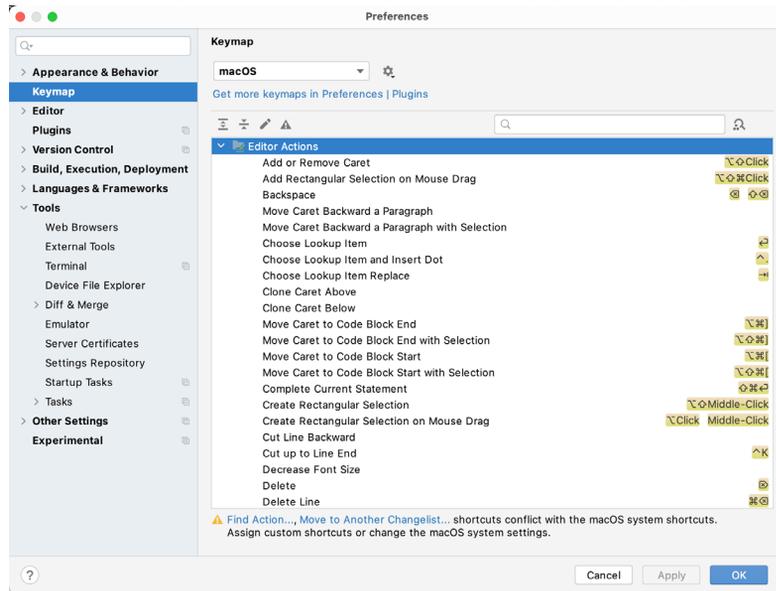
Figure 6-8

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).
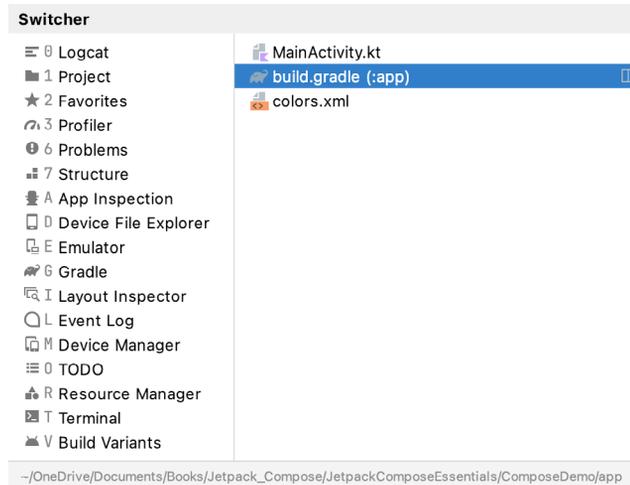
Figure 6-9

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.
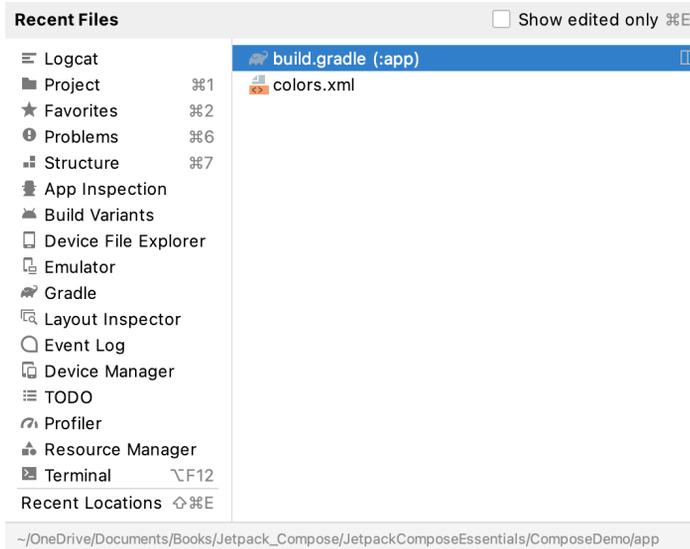
Figure 6-10

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:
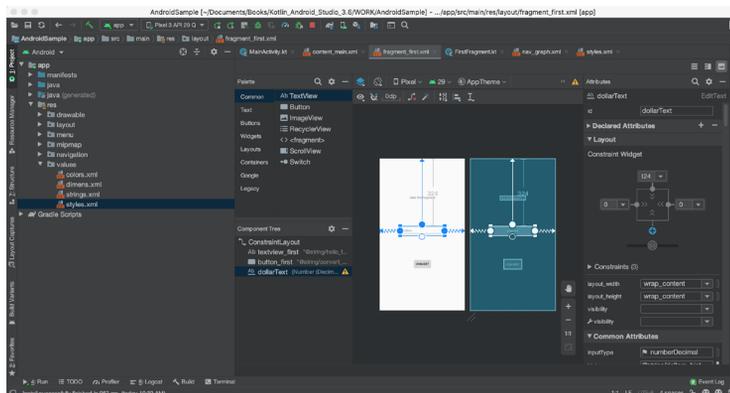
Figure 6-11

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:
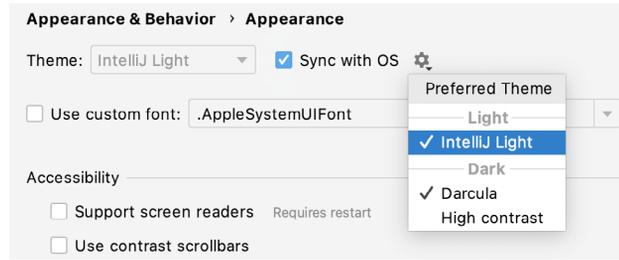
Figure 6-12

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.

7. Testing Android Studio Apps on a Physical Android Device

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

Communication with both AVD instances and connected Android devices is handled by the *Android Debug Bridge (ADB)*. In this chapter, we explain how to configure the adb environment to enable application testing on an Android device with macOS, Windows, and Linux-based systems.

7.1 An Overview of the Android Debug Bridge (ADB)

The primary purpose of the ADB is to facilitate interaction between a development system, in this case, Android Studio, and both AVD emulators and Android devices to run and debug applications. ADB allows you to connect to devices either over a WiFi network or directly using a USB cable.

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

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

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

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

7.2 Enabling USB Debugging ADB on Android Devices

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

1. Open the Settings app on the device and select the *About tablet* or *About phone* option (on some versions of Android this can be found on the *System* page of the Settings app).

2. On the *About* screen, scroll down to the *Build number* field (Figure 7-1) and tap on it seven times until a message appears indicating that developer mode has been enabled. If the Build number is not listed on the About screen it may be available via the *Software information* option. Alternatively, unfold the Advanced section of the list if available.
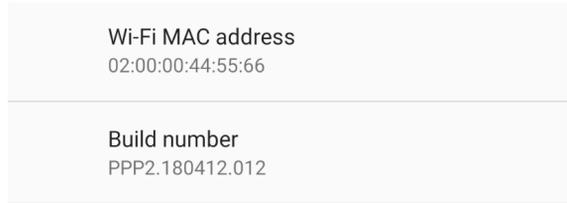
Figure 7-1

3. Return to the main Settings screen and note the appearance of a new option titled Developer options (on newer versions of Android this option is listed on the System settings screen). Select this option and on the resulting screen, locate the USB debugging option as illustrated in Figure 7-2:
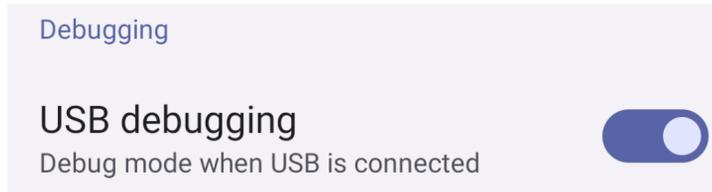
Figure 7-2

4. Enable the USB debugging option and tap the Allow button when confirmation is requested.

At this point, the device is now configured to accept debugging connections from adb on the development system over a USB connection. All that remains is to configure the development system to detect the device when it is attached. While this is a relatively straightforward process, the steps involved differ depending on whether the development system is running Windows, macOS, or Linux. Note that the following steps assume that the Android SDK *platform-tools* directory is included in the operating system PATH environment variable as described in the chapter entitled *"Setting up an Android Studio Development Environment"*.

7.2.1 macOS ADB Configuration

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

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

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

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

If the device is listed as *offline*, go to the Android device and check for the presence of the dialog shown in Figure 7-3 seeking permission to *Allow USB debugging*. Enable the checkbox next to the option that reads *Always allow from this computer*, before clicking on *OK*.

Allow USB debugging?

The computer's RSA key fingerprint is:
6E:BF:56:13:95:F8:9B:7E:12:CF:C5:67

☐ Always allow from this computer

CANCEL   OK

Figure 7-3

Repeating the *adb devices* command should now list the device as being available:

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

If the device is not listed, try logging out and then back into the macOS desktop and, if the problem persists, rebooting the system.

7.2.2 Windows ADB Configuration

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

https://developer.android.com/sdk/win-usb.html

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

https://developer.android.com/tools/extras/oem-usb.html

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

```
adb devices
```

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

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

If the device is listed as *offline* or *unauthorized*, go to the device display and check for the dialog shown in Figure 7-3 seeking permission to *Allow USB debugging*. Enable the checkbox next to the option that reads *Always allow from this computer*, before clicking on *OK*. Repeating the *adb devices* command should now list the device as being ready:
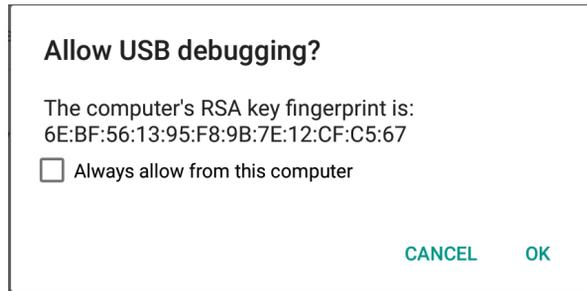
```
List of devices attached
HT4CTJT01906    device
```

If the device is not listed, execute the following commands to restart the ADB server:

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

Testing Android Studio Apps on a Physical Android Device

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

```
android update adb
```

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

7.2.3 Linux adb Configuration

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

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

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

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

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

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

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

If the device is listed as *offline* or *unauthorized*, go to the Android device and check for the dialog shown in Figure 7-3 seeking permission to *Allow USB debugging*.

7.3 Resolving USB Connection Issues

If you are unable to successfully connect to the device using the above steps, display the run target menu (Figure 7-4) and select the *Troubleshoot Device Connections* option:
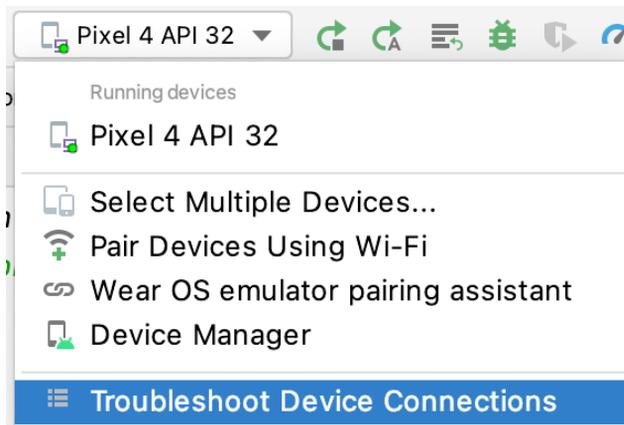
Figure 7-4

The connection assistant will scan for devices and report problems and possible solutions.

7.4 Enabling Wireless Debugging on Android Devices

Follow steps 1 through 3 from section 7.2 above, this time enabling the Wireless Debugging option as shown in Figure 7-5:

Figure 7-5

Next, tap the above Wireless debugging entry to display the screen shown in Figure 7-6:
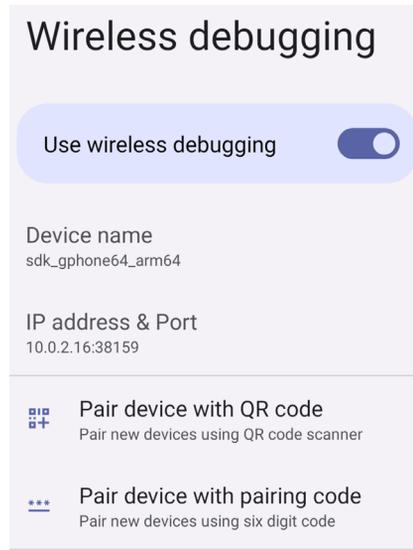
Figure 7-6

If the device you are using has a camera, select *Pair device with QR code*, otherwise select the *Pair device with pairing code* option. Depending on your selection, the Settings app will either start a camera session or display a pairing code as shown in Figure 7-7:
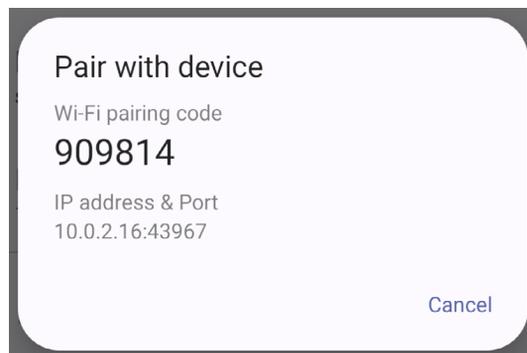
Figure 7-7

With an option selected, return to Android Studio and select the *Pair Devices Using WiFi* option from the run target menu as illustrated in Figure 7-8:

Testing Android Studio Apps on a Physical Android Device
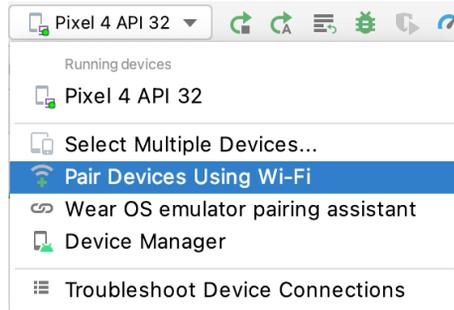
Figure 7-8

In the pairing dialog, select either *Pair using QR code* or *Pair using pairing code* depending on your previous selection in the Settings app on the device:

Figure 7-9

Either scan the QR code using the Android device or enter the pairing code displayed on the device screen into the Android Studio dialog (Figure 7-10) to complete the pairing process:
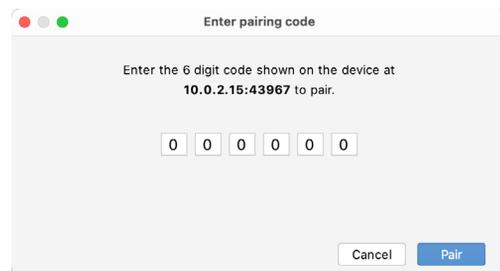
Figure 7-10

If the pairing process fails, try rebooting both the development system and Android device and try again.

7.5 Testing the adb Connection

Assuming that the adb configuration has been successful on your chosen development platform, the next step is to try running the test application created in the chapter entitled *"Creating an Example Android App in Android Studio"* on the device. Launch Android Studio, open the AndroidSample project, and verify that the device appears in the device selection menu as highlighted in Figure 7-11:
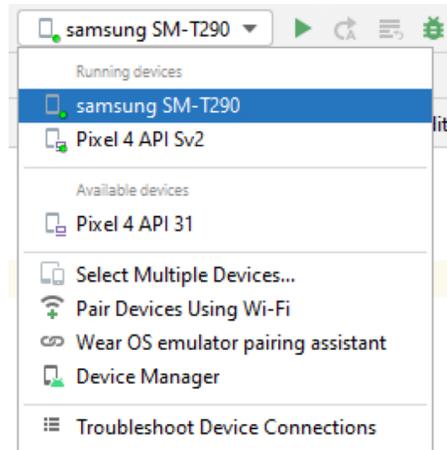
Figure 7-11

Select the device from the list and click on the run button (the green arrow button located immediately to the right of the device menu) to install and run the app.

7.6 Device Mirroring

Device mirroring allows you to run an app on a physical device while viewing the device display within Android Studio's Running Devices tool window. In other words, although your app is running on a physical device, it appears within Android Studio in the same way as an AVD instance.

Device Mirroring is an experimental feature in Android Studio Electric Eel and must be manually enabled. To enable device mirroring, select *File -> Settings...* (*Android Studio -> Preferences...* on macOS), select the Experimental entry in the left-hand panel, and turn on the *Enable mirroring of physical Android devices* as shown in Figure 7-12:
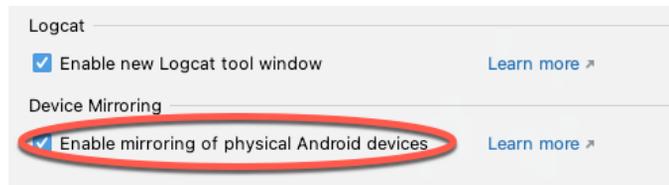
Figure 7-12

With the device mirroring option enabled and a device connected to Android Studio, display the *Running Devices* tool window (*View -> Tool Windows -> Running Devices*). Android Studio will mirror the display of the physical device in the tool window.

7.7 Summary

While the Android Virtual Device emulator provides an excellent testing environment, it is important to keep in mind that there is no real substitute for making sure an application functions correctly on a physical Android

device. This, after all, is where the application will be used in the real world.

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

8. The Basics of the Android Studio Code Editor

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

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

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

8.1 The Android Studio Editor

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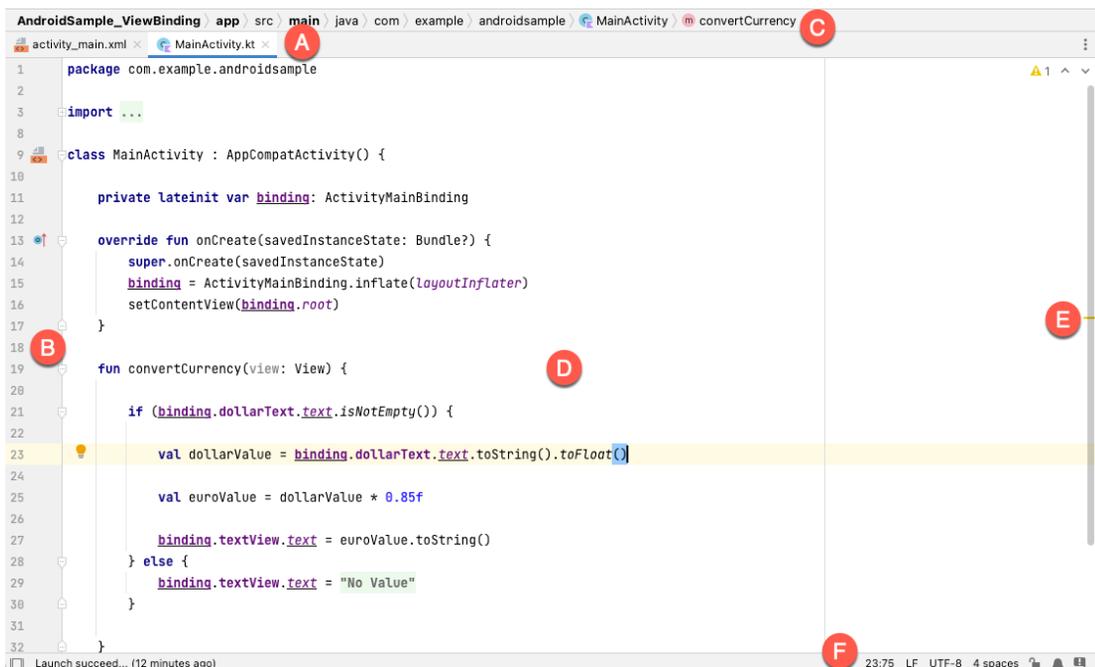

Figure 8-1

The Basics of the Android Studio Code Editor

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(view: View)
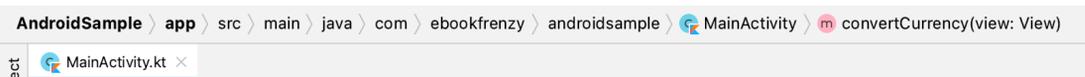
MainActivity.kt ✕

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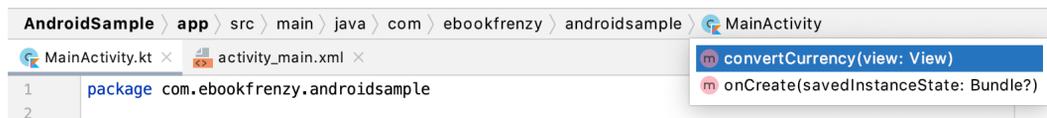
AndroidSample 〉 **app** 〉 src 〉 main 〉 java 〉 com 〉 ebookfrenzy 〉 androidsample 〉 MainActivity

MainActivity.kt ✕   activity_main.xml ✕

| 1 | package com.ebookfrenzy.androidsample |
| 2 | |

convertCurrency(view: View)
onCreate(savedInstanceState: Bundle?)

Figure 8-3

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

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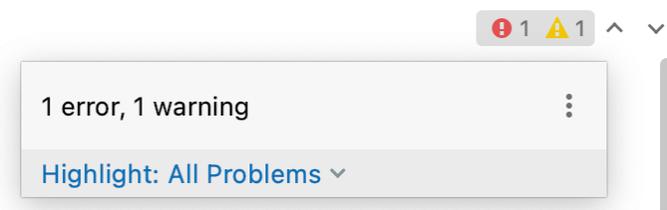
Figure 8-4

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

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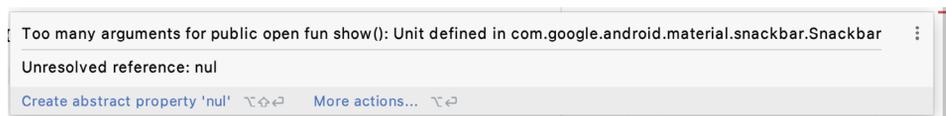

Figure 8-5

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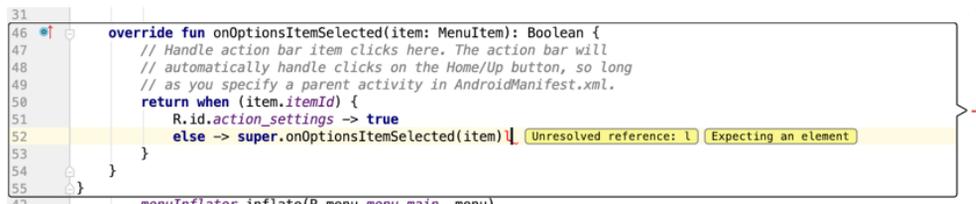

Figure 8-6

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

F – The Status Bar – Though the status bar is actually part of the main window, as opposed to the editor, it does contain some information about the currently active editing session. This information includes the current position of the cursor in terms of lines and characters and the encoding format of the file (UTF-8, ASCII etc.). Clicking on these values in the status bar allows the corresponding setting to be changed. Clicking on the line number, for example, displays the Go to Line:Column dialog. Use the *View -> Appearance -> Status Bar Widgets* menu option to add and remove widgets. For example, the Memory Indicator is a helpful widget if you are experiencing performance problems with Android Studio.

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

8.2 Splitting the Editor Window

By default, the editor will display a single panel showing the content of the currently selected file. A particularly useful feature when working simultaneously with multiple source code files is the ability to split the editor into

multiple panes. To split the editor, right-click on a file tab within the editor window and select either the Split Vertically or Split Horizontally menu option. Figure 8-7, for example, shows the splitter in action with the editor split into three panels:
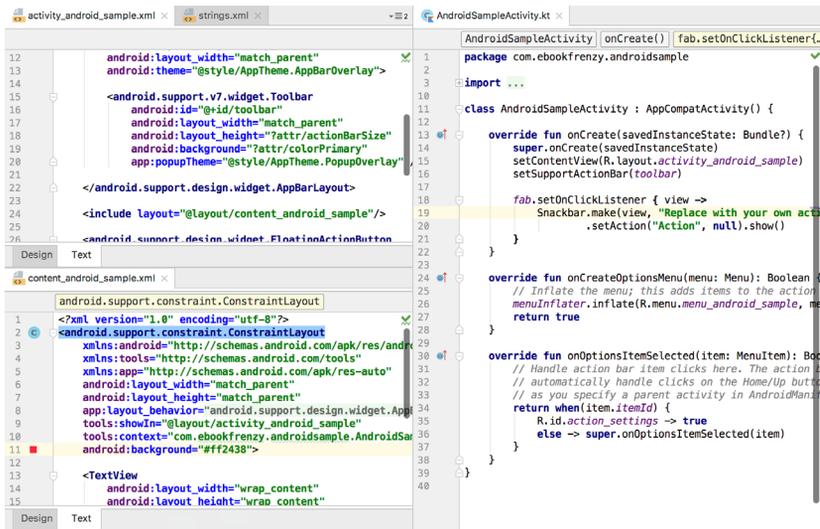

Figure 8-7

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

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

8.3 Code Completion

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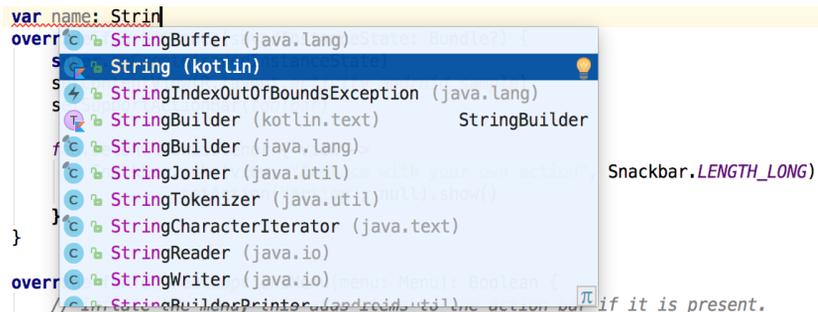
Figure 8-8

If none of the auto completion suggestions are correct, simply keep typing and the editor will continue to refine

the suggestions where appropriate. To accept the top most suggestion, simply press the Enter or Tab key on the keyboard. To select a different suggestion, use the arrow keys to move up and down the list, once again using the Enter or Tab key to select the highlighted item.

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

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

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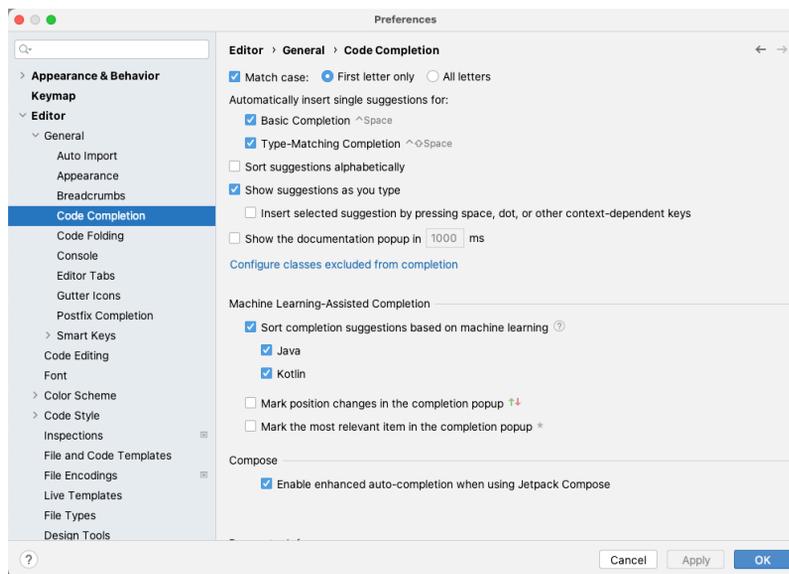
Figure 8-9

8.4 Statement Completion

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

```
fun myMethod()
```

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

```
fun myMethod() {


}
```

8.5 Parameter Information

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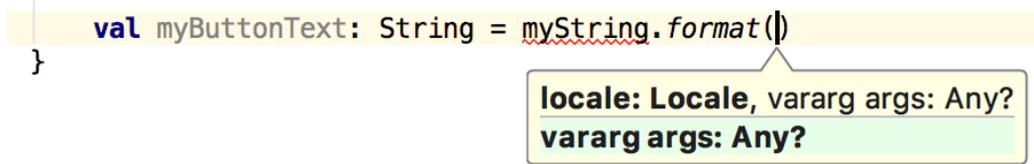

Figure 8-10

8.6 Parameter Name Hints

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

Figure 8-11

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

8.7 Code Generation

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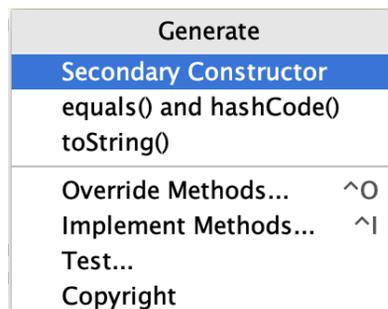
Figure 8-12

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

generate a stub method for this, simply select the *Override Methods…* option from the code generation list and select the *onStop()* method from the resulting list of available methods:
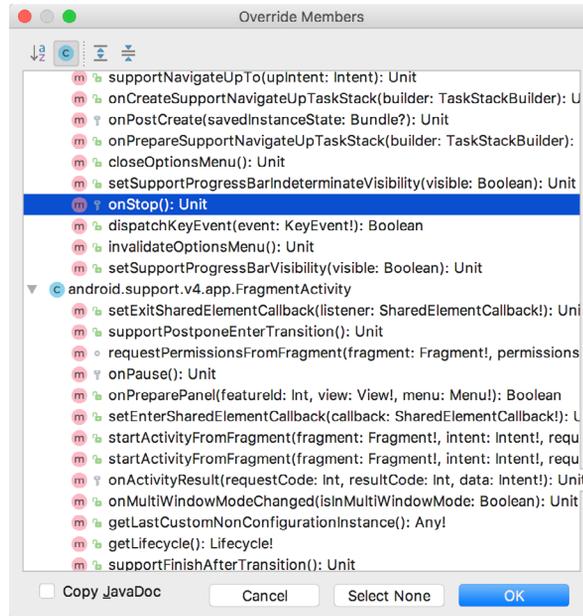
Figure 8-13

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

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

8.8 Code Folding

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

Figure 8-14

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

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {...}
```
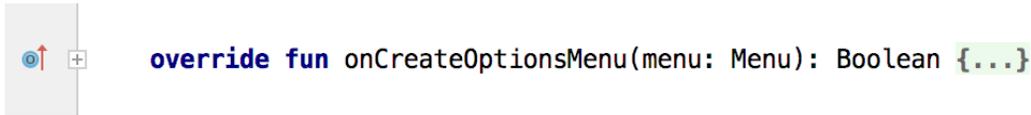
Figure 8-15

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

Figure 8-16

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

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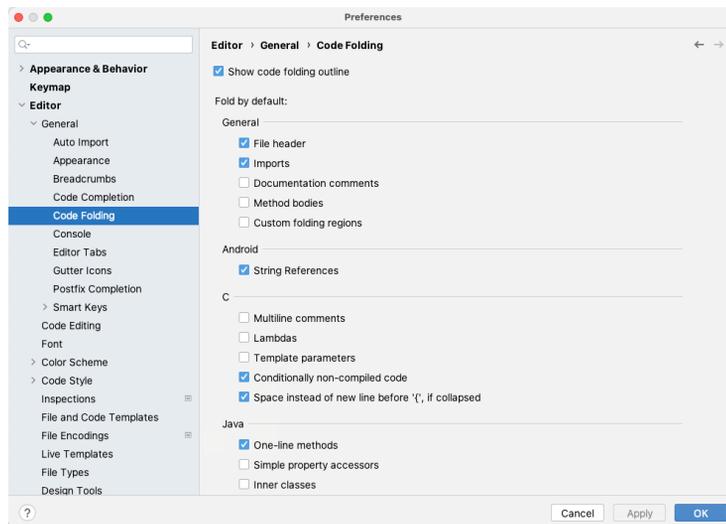
Figure 8-17

8.9 Quick Documentation Lookup

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

```
]   override fun onCreateOptionsMenu(menu: Menu): Boolean {
        // Inflate the menu; this adds items  android.view
        menuInflater.inflate(R.menu.menu_mai  public interface Menu
        return true                          Interface for managing the items in a menu.
]   }                                        By default, every Activity supports an options menu of
                                             actions or options. You can add items to this menu
                                             and handle clicks on your additions. The easiest way
]   override fun onOptionsItemSelected(item:  of adding menu items is inflating an XML file into the
        // Handle action bar item clicks her  Menu via MenuInflater. The easiest way of attaching
```

Figure 8-18

8.10 Code Reformatting

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

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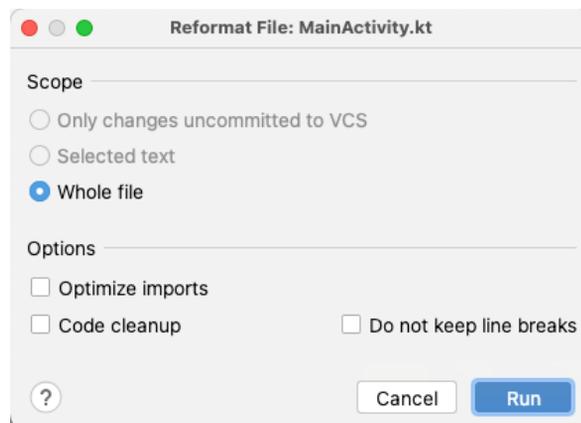
Figure 8-19

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

8.11 Finding Sample Code

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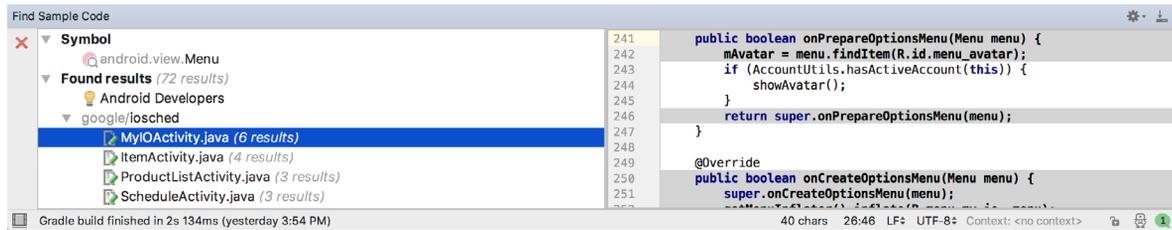
Figure 8-20

8.12 Live Templates

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

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

To list and edit existing templates, change the special key, or add your own templates, open the Preferences dialog and select Live Templates from the Editor section of the left-hand navigation panel:
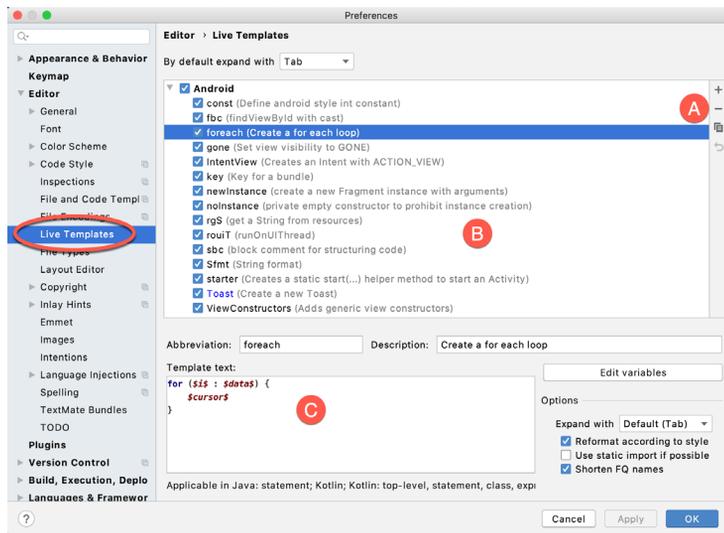
Figure 8-21

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

8.13 Summary

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

9. An Overview of the Android Architecture

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

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

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

9.1 The Android Software Stack

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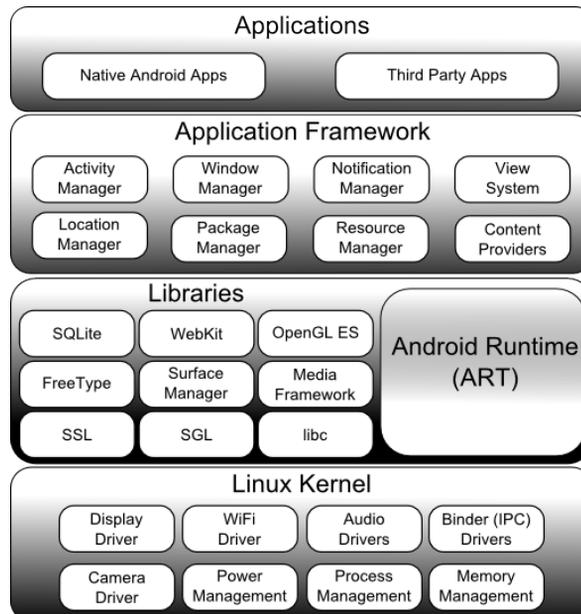
Figure 9-1

9.2 The Linux Kernel

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

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

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

9.3 Android Runtime – ART

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

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

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

9.4 Android Libraries

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

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

- **android.app** – Provides access to the application model and is the cornerstone of all Android applications.

- **android.content** – Facilitates content access, publishing and messaging between applications and application components.

- **android.database** – Used to access data published by content providers and includes SQLite database management classes.

- **android.graphics** – A low-level 2D graphics drawing API including colors, points, filters, rectangles and canvases.

- **android.hardware** – Presents an API providing access to hardware such as the accelerometer and light sensor.

- **android.opengl** – A Java interface to the OpenGL ES 3D graphics rendering API.

- **android.os** – Provides applications with access to standard operating system services including messages, system services and inter-process communication.

- **android.media** – Provides classes to enable playback of audio and video.

- **android.net** – A set of APIs providing access to the network stack. Includes *android.net.wifi*, which provides access to the device's wireless stack.

- **android.print** – Includes a set of classes that enable content to be sent to configured printers from within Android applications.

- **android.provider** – A set of convenience classes that provide access to standard Android content provider databases such as those maintained by the calendar and contact applications.

- **android.text** – Used to render and manipulate text on a device display.

- **android.util** – A set of utility classes for performing tasks such as string and number conversion, XML handling and date and time manipulation.

- **android.view** – The fundamental building blocks of application user interfaces.

- **android.widget** - A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.

- **android.webkit** – A set of classes intended to allow web-browsing capabilities to be built into applications.

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

9.4.1 C/C++ Libraries

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

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

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

9.5 Application Framework

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

found and reused by other applications.

The Android framework includes the following key services:

- **Activity Manager** – Controls all aspects of the application lifecycle and activity stack.

- **Content Providers** – Allows applications to publish and share data with other applications.

- **Resource Manager** – Provides access to non-code embedded resources such as strings, color settings and user interface layouts.

- **Notifications Manager** – Allows applications to display alerts and notifications to the user.

- **View System** – An extensible set of views used to create application user interfaces.

- **Package Manager** – The system by which applications are able to find out information about other applications currently installed on the device.

- **Telephony Manager** – Provides information to the application about the telephony services available on the device such as status and subscriber information.

- **Location Manager** – Provides access to the location services allowing an application to receive updates about location changes.

9.6 Applications

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

9.7 Summary

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

10. The Anatomy of an Android Application

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

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

10.1 Android Activities

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

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

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

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

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

10.2 Android Fragments

An activity, as described above, typically represents a single user interface screen within an app. One option is to construct the activity using a single user interface layout and one corresponding activity class file. A better alternative, however, is to break the activity into different sections. Each of these sections is referred to as a fragment, each of which consists of part of the user interface layout and a matching class file (declared as a subclass of the Android *Fragment* class). In this scenario, an activity simply becomes a container into which one or more fragments are embedded.

In fact, fragments provide an efficient alternative to having each user interface screen represented by a separate activity. Instead, an app can consist of a single activity that switches between different fragments, each representing a different app screen.

10.3 Android Intents

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

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

10.4 Broadcast Intents

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

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

10.5 Broadcast Receivers

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

10.6 Android Services

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

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

Example situations where a Service might be a practical solution include, as previously mentioned, the streaming of audio that should continue when the application is no longer active, or a stock market tracking application

that needs to notify the user when a share hits a specified price.

10.7 Content Providers

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

The native Android applications include a number of standard Content Providers allowing applications to access data such as contacts and media files. The Content Providers currently available on an Android system may be located using a *Content Resolver*.

10.8 The Application Manifest

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

10.9 Application Resources

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

10.10 Application Context

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

10.11 Summary

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

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

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

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

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

11. An Introduction to Kotlin

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

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

11.1 What is Kotlin?

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

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

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

11.2 Kotlin and Java

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

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

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

11.3 Converting from Java to Kotlin

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

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

11.4 Kotlin and Android Studio

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

11.5 Experimenting with Kotlin

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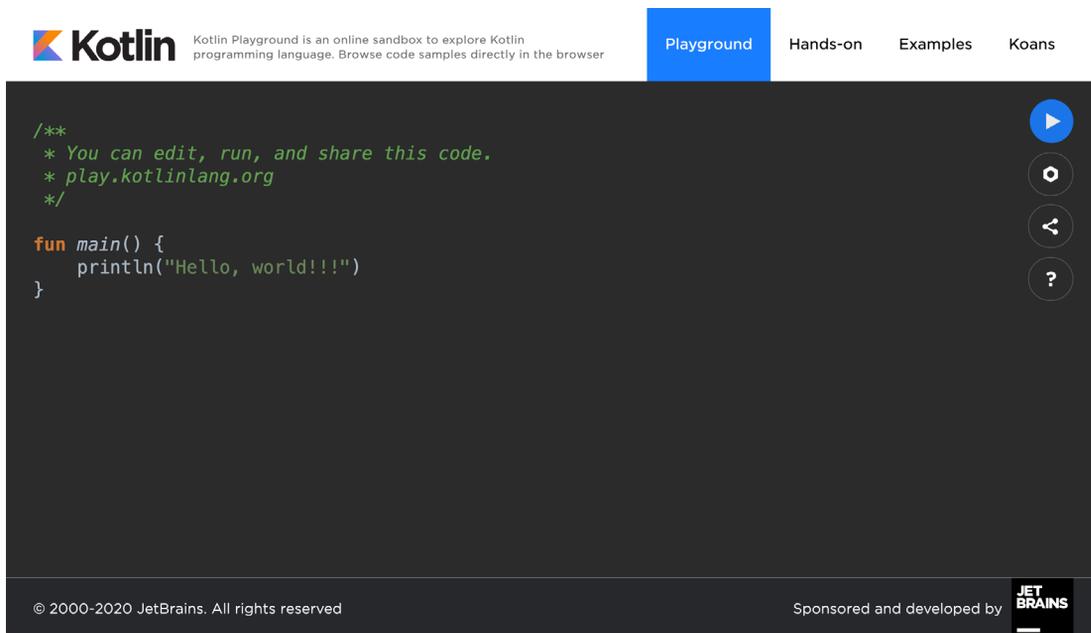
Figure 11-1

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

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

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

    println("Welcome to Kotlin")

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

```
}
```

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

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

Figure 11-2

11.6 Semi-colons in Kotlin

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

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

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

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

11.7 Summary

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

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

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

12. Kotlin Data Types, Variables, and Nullability

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

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

12.1 Kotlin Data Types

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

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

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

```
val mynumber = 10
```

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

```
1010
```

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

```
val myletter = 'c'
```

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

converted to binary, it is stored as:

```
10101100011
```

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

12.1.1 Integer Data Types

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

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

12.1.2 Floating-Point Data Types

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

12.1.3 Boolean Data Type

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

12.1.4 Character Data Type

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

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

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

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

```
val myChar4 = '\u0058'
```

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

12.1.5 String Data Type

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

```
val message = "You have 10 new messages."
```

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

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

90

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

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

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

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

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


println(message)
```

When executed, the code will output the following message:

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

12.1.6 Escape Sequences

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

```
var newline = '\n'
```

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

```
var backslash = '\\'
```

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

- \n - Newline

- \r - Carriage return

- \t - Horizontal tab

- \\ - Backslash

- \" - Double quote (used when placing a double quote into a string declaration)

- \' - Single quote (used when placing a single quote into a string declaration)

- \$ - Used when a character sequence containing a $ is misinterpreted as a variable in a string template.

- \unnnn – Double byte Unicode scalar where nnnn is replaced by four hexadecimal digits representing the Unicode character.

12.2 Mutable Variables

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

12.3 Immutable Variables

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

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

12.4 Declaring Mutable and Immutable Variables

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

```
var userCount = 10
```

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

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

Immutable variables are declared using the *val* keyword.

```
val maxUserCount = 20
```

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

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

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

12.5 Data Types are Objects

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

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

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

```
val uppercase = myString.toUpperCase()
```

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

```
val length = myString.length
```

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

```
val result = myString.contains("fox")
```

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

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

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

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

12.6 Type Annotations and Type Inference

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

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

```
val userCount: Int = 10
```

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

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

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

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

```
val bookTitle = "Android Studio Development Essentials"
```

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

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

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

12.7 Nullable Type

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

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

```
val username: String = null
```

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

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

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

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

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

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

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

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

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

```
val username: String? = null

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

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

12.8 The Safe Call Operator

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

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

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

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

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

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

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

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

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

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

```
val uppercase = username?.length
```

12.9 Not-Null Assertion

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

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

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

```
Exception in thread "main" kotlin.KotlinNullPointerException
```

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

12.10 Nullable Types and the let Function

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

```
val firstNumber = 10
val secondNumber = 20

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

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

Kotlin Data Types, Variables, and Nullability

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


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

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

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

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

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


if (secondNumber != null) {
    val result = firstNumber.times(secondNumber)
    print(result)
}
```

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

```
secondNumber?.let {
    val result = firstNumber.times(it)
    print(result)
}
```

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

12.11 Late Initialization (lateinit)

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

```
var myName: String
```

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

```
lateinit var myName: String
```

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

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

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

```
lateinit var myName: String

print("My Name is " + myName)

Exception in thread "main" kotlin.UninitializedPropertyAccessException: lateinit
property myName has not been initialized
```

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

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

12.12 The Elvis Operator

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

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

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

```
return myString ?: "String is null"
```

12.13 Type Casting and Type Checking

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

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

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

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

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

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

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

12.14 Summary

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

Index

Index

Index

Index

Index

G

Index

Index

Index

Index