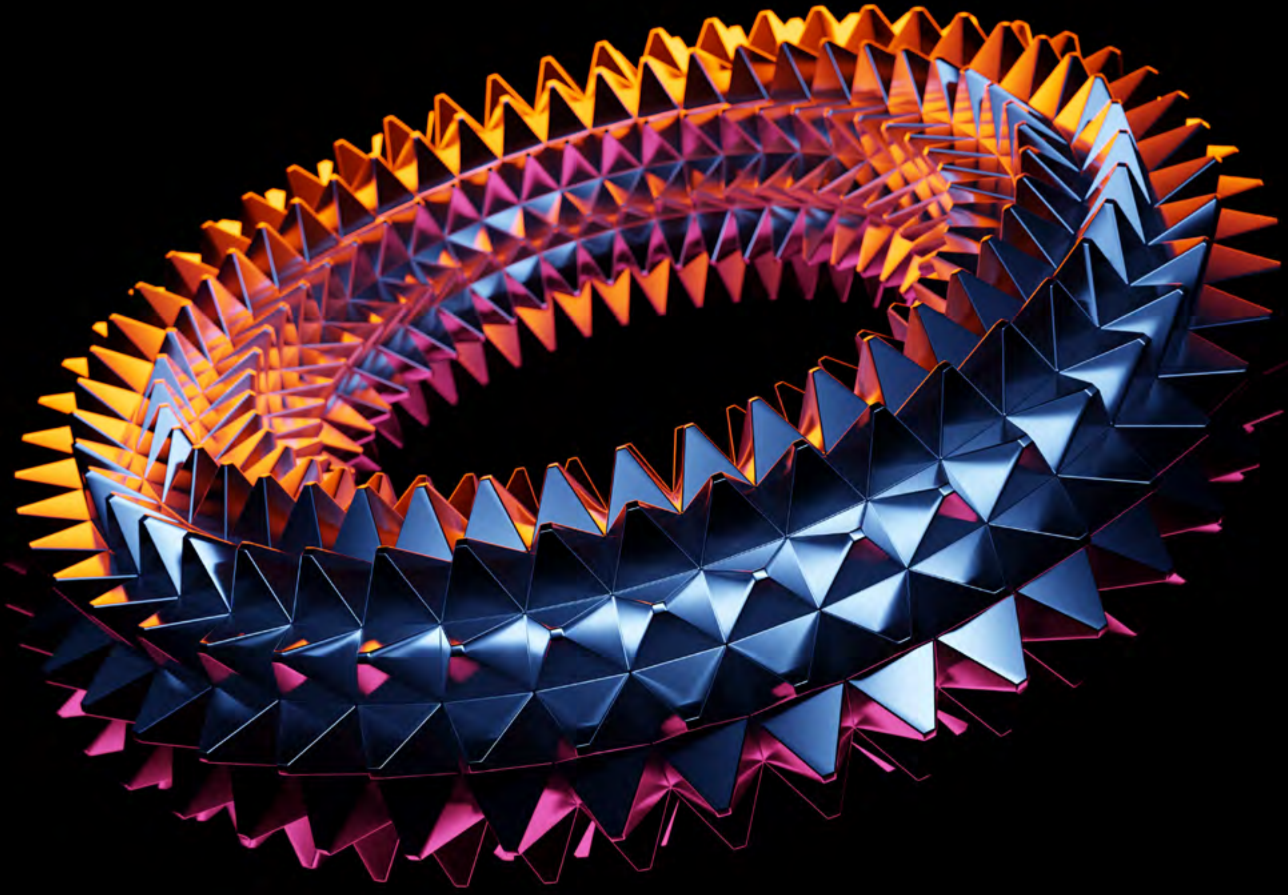
Android Studio Hedgehog Essentials



Java Edition

Payload publishing

Android Studio Hedgehog Essentials

Java Edition

Android Studio Hedgehog Essentials – Java Edition

ISBN: 978-1-951442-82-8

Rev: 1.0

Contents

Table of Contents

Chapter 1

1. Introduction

Fully updated for Android Studio Hedgehog (2023.1.1) and the new UI, this book teaches you how to develop Android-based applications using the Java programming language.

This book begins with the basics and outlines how to set up an Android development and testing environment, followed by an overview of 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 also cover 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, Gradle build configuration, in-app billing, and submitting apps to the Google Play Developer Console.

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

1.1 Downloading the Code Samples

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

https://www.payloadbooks.com/product/hedgehogjava

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 *info@payloadbooks.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.payloadbooks.com/hedgehogjava

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

Chapter 2

2. Setting up an Android Studio Development Environment

Before any work can begin on developing 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), including the Android Software Development Kit (SDK) and the 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 Hedgehog 2023.1.1 using the Android API 34 SDK (UpsideDownCake), which, at the time of writing, are the latest stable releases.

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, there may be differences between this book and the software. A web search for "Android Studio Hedgehog" 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 Hedgehog 2023.1.1 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 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 system users. When prompted to select the components to install, ensure 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 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 as 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:

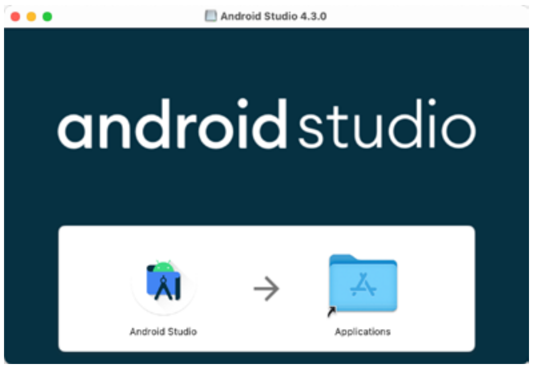


Figure 2-1

To install the package, 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:

```
tar xvfz /<path to package>/android-studio-<version>-linux.tar.gz
```

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

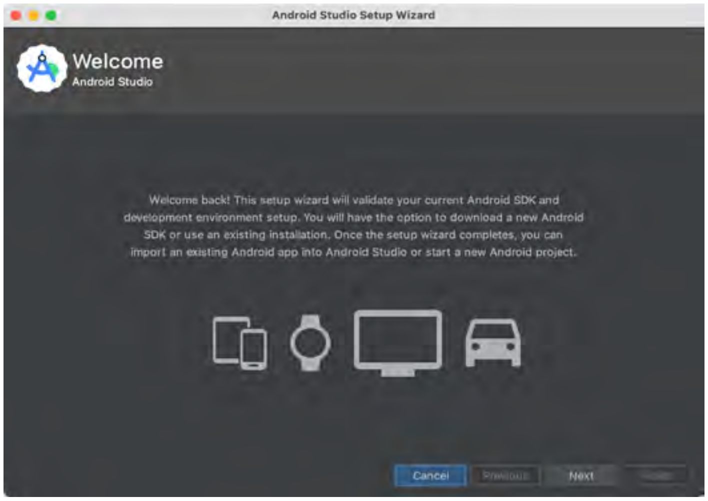
To launch Android Studio, open a terminal window, change directory to the *android-studio/bin* sub-directory, and execute the following command:

```
./studio.sh
```

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 the OK button to proceed.

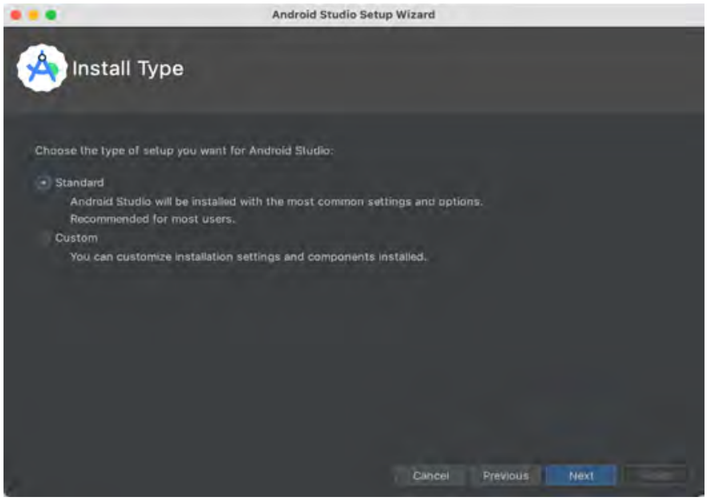
If you are installing Android Studio for the first time, the initial dialog that appears once the setup process starts may resemble that shown in Figure 2-2 below:



Welcome back! This setup wizard will validate your current Android SDK and development environment setup. You will have the option to download a new Android SDK or use an existing installation. Once the setup wizard completes, you can import an existing Android app into Android Studio or start a new Android project.

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.



Choose the type of setup you want for Android Studio:

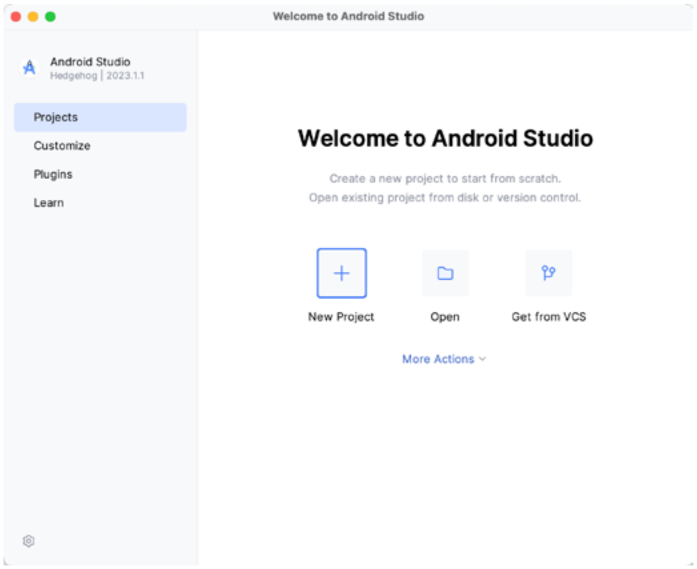
Standard
Android Studio will be installed with the most common settings and options.
Recommended for most users.

Custom
You can customize installation settings and components installed.

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 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:



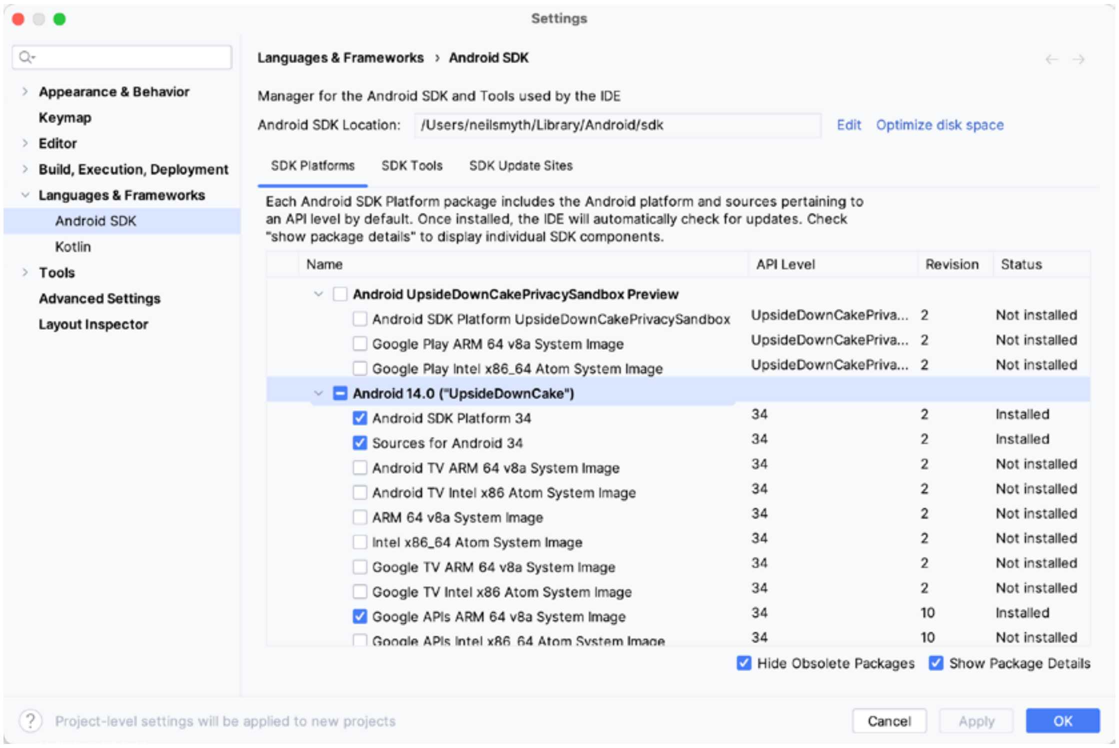
Create a new project to start from scratch.
Open existing project from disk or version control.

Figure 2-4

2.5 Installing additional Android SDK packages

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

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



Manager for the Android SDK and Tools used by the IDE

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

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

Name	API Level	Revision	Status
☐ Android UpsideDownCakePrivacySandbox Preview			
☐ Android SDK Platform UpsideDownCakePrivacySandbox	UpsideDownCakePriva...	2	Not installed
☐ Google Play ARM 64 v8a System Image	UpsideDownCakePriva...	2	Not installed
☐ Google Play Intel x86_64 Atom System Image	UpsideDownCakePriva...	2	Not installed
Android 14.0 ("UpsideDownCake")			
☑ Android SDK Platform 34	34	2	Installed
☑ Sources for Android 34	34	2	Installed
☐ Android TV ARM 64 v8a System Image	34	2	Not installed
☐ Android TV Intel x86 Atom System Image	34	2	Not installed
☐ ARM 64 v8a System Image	34	2	Not installed
☐ Intel x86_64 Atom System Image	34	2	Not installed
☐ Google TV ARM 64 v8a System Image	34	2	Not installed
☐ Google TV Intel x86 Atom System Image	34	2	Not installed
☑ Google APIs ARM 64 v8a System Image	34	10	Installed
☐ Google APIs Intel x86_64 Atom System Image	34	10	Not installed

☑ Hide Obsolete Packages ☑ Show Package Details

Project-level settings will be applied to new projects

Figure 2-5

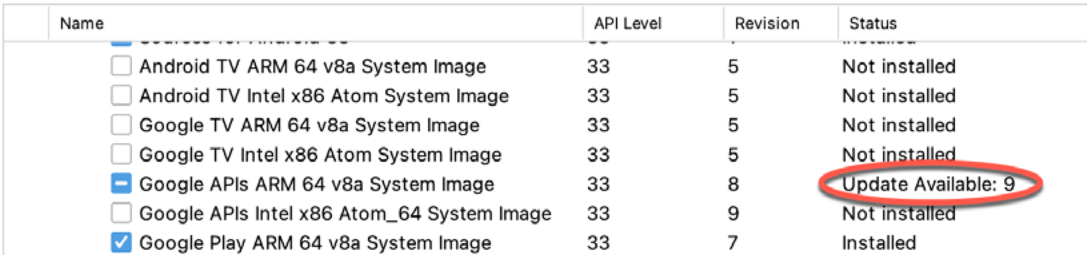
Google pairs each release of Android Studio with a maximum supported Application Programming Interface (API) level of the Android SDK. In the case of Android Studio Hedgehog, this is Android UpsideDownCake (API Level 34). This information can be confirmed using the following link:

https://developer.android.com/studio/releases#api-level-support

Immediately after installing Android Studio for the first time, it is likely that only the latest supported version of the Android SDK has been installed. To install older versions of the Android SDK, select the checkboxes corresponding to the versions and click the *Apply* button. The rest of this book assumes that the Android UpsideDownCake (API Level 34) 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 ensures 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 the Apply button. Click the OK button to install the SDK in the resulting confirmation dialog. Subsequent dialogs will seek the acceptance of licenses and terms before performing the installation. Click Finish once the installation is complete.

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

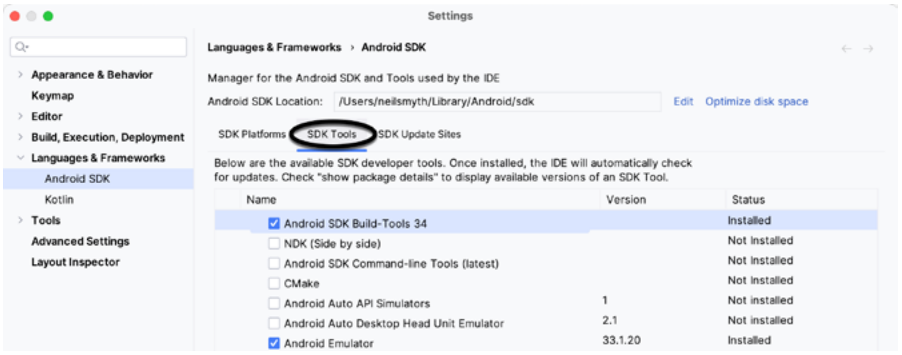


	Name	API Level	Revision	Status
☐	Android TV ARM 64 v8a System Image	33	5	Not installed
☐	Android TV Intel x86 Atom System Image	33	5	Not installed
☐	Google TV ARM 64 v8a System Image	33	5	Not installed
☐	Google TV Intel x86 Atom System Image	33	5	Not installed
▬	Google APIs ARM 64 v8a System Image	33	8	Update Available: 9
☐	Google APIs Intel x86 Atom_64 System Image	33	9	Not installed
☑	Google Play ARM 64 v8a System Image	33	7	Installed

Figure 2-6

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

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



Manager for the Android SDK and Tools used by the IDE

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

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

	Name	Version	Status
☑	Android SDK Build-Tools 34		Installed
☐	NDK (Side by side)		Not Installed
☐	Android SDK Command-line Tools (latest)		Not Installed
☐	CMake		Not Installed
☐	Android Auto API Simulators	1	Not installed
☐	Android Auto Desktop Head Unit Emulator	2.1	Not installed
☑	Android Emulator	33.1.20	Installed

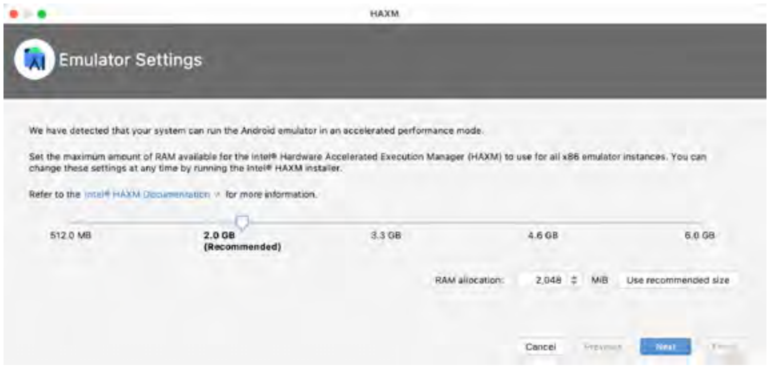
Figure 2-7

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

- Android SDK Build-tools
- Android Emulator
- Android SDK Platform-tools
- Google Play Services
- Intel x86 Emulator Accelerator (HAXM installer)*
- Google USB Driver (Windows only)
- Layout Inspector image server for API 31-34

*Note that 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, select the checkboxes next to those packages and click the *Apply* button to initiate the installation process. If the HAXM emulator settings dialog appears, select the recommended memory allocation:



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

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

Refer to the Intel® HAXM Documentation for more information.

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

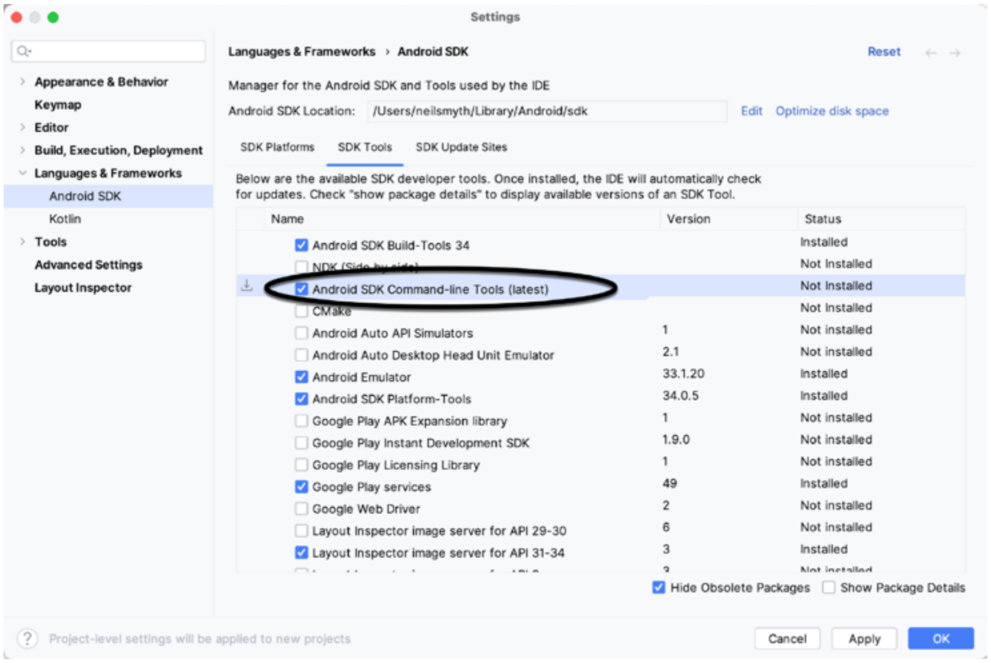
RAM allocation: 2,048 MiB | Use recommended size

Figure 2-8

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

2.6 Installing the Android SDK Command-line Tools

Android Studio includes 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 locate the *Android SDK Command-line Tools (latest)* package as shown in Figure 2-9:



Languages & Frameworks › Android SDK

Manager for the Android SDK and Tools used by the IDE

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

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

Name	Version	Status
✓ Android SDK Build-Tools 34		Installed
NDK (Side by side)		Not Installed
✓ Android SDK Command-line Tools (latest)		Not Installed
CMake		Not Installed
Android Auto API Simulators	1	Not installed
Android Auto Desktop Head Unit Emulator	2.1	Not installed
✓ Android Emulator	33.1.20	Installed
✓ Android SDK Platform-Tools	34.0.5	Installed
Google Play APK Expansion library	1	Not installed
Google Play Instant Development SDK	1.9.0	Not installed
Google Play Licensing Library	1	Not installed
✓ Google Play services	49	Installed
Google Web Driver	2	Not installed
Layout Inspector image server for API 29-30	6	Not installed
✓ Layout Inspector image server for API 31-34	3	Installed
[illegible]	3	Not installed

✓ Hide Obsolete Packages ☐ Show Package Details

Project-level settings will be applied to new projects

Figure 2-9

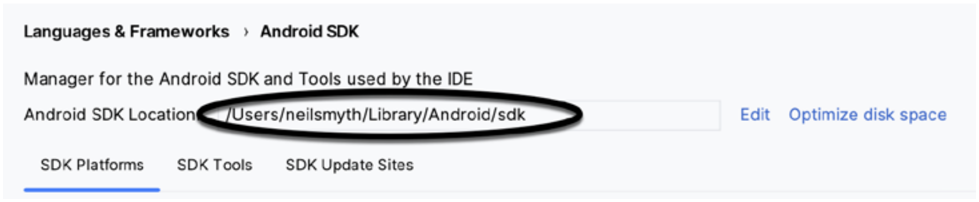
If the command-line tools package is not already installed, enable it and click Apply, followed by OK to complete the installation. When the installation completes, click Finish and close the SDK Manager dialog.

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:


Languages & Frameworks › Android SDK

Manager for the Android SDK and Tools used by the IDE

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

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 the *Edit…* button. Using the *New* button in the edit dialog, add two 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 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 the *Environment Variables...* button. Follow the steps outlined for Windows 8.1 starting from step 3.

2.6.4 Linux

This configuration can be achieved on Linux 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
```

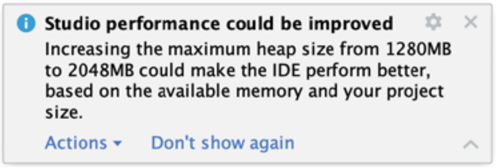
Note that since this is a system directory, it will be necessary to use the *sudo* command when creating the file. For example:

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

2.7 Android Studio memory management

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

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

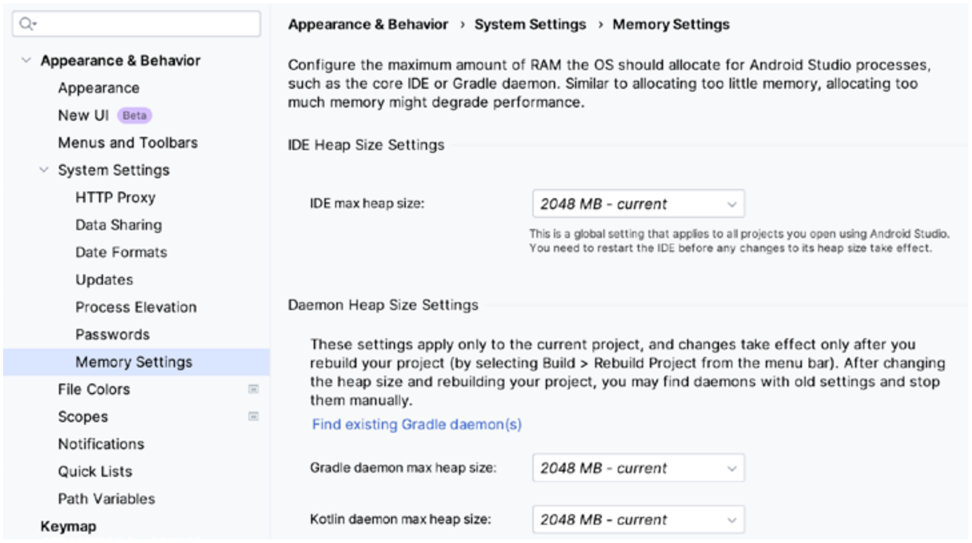


Studio performance could be improved

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

Figure 2-11

To view and modify the current memory configuration, select the *File -> Settings...* main menu option (*Android Studio -> Settings...* on macOS) 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:



Appearance & Behavior › System Settings › Memory Settings

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

IDE Heap Size Settings

IDE max heap size: 2048 MB - current

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

Daemon Heap Size Settings

These settings apply only to the current project, and changes take effect only after you rebuild your project (by selecting Build > Rebuild Project from the menu bar). After changing the heap size and rebuilding your project, you may find daemons with old settings and stop them manually.

Find existing Gradle daemon(s)

Gradle daemon max heap size: 2048 MB - current

Kotlin daemon max heap size: 2048 MB - current

Figure 2-12

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.

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, several 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. To display the SDK Manager from within an open project, select the *Tools -> SDK Manager...* menu option from the main menu.

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). This chapter covers the steps necessary to install these packages on Windows, macOS, and Linux.

Chapter 3

3. Creating an Example Android App in Android Studio

The preceding chapters of this book have explained how to configure an environment suitable for developing Android applications using the Android Studio IDE. Before moving on to slightly more advanced topics, now is a good time to validate that all 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 creating an Android application project using Android Studio. Once the project has been created, a later chapter will explore using 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 use one of the most basic Android Studio project templates. This simplicity allows us to introduce some key aspects of Android app development without overwhelming the beginner by introducing 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 the techniques and code used in this initial example project will be covered in much greater detail later.

3.2 Creating a New Android Project

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

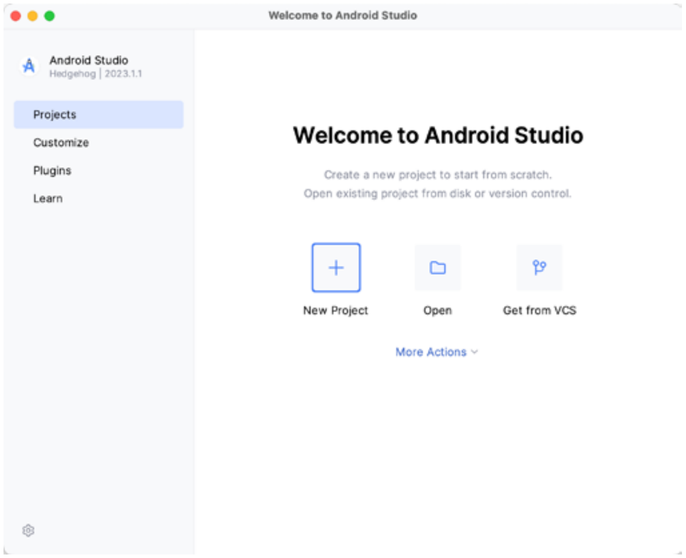

Welcome to Android Studio
Android Studio
Hedgehog | 2023.1.1
Projects
Customize
Plugins
Learn
Welcome to Android Studio
Create a new project to start from scratch.
Open existing project from disk or version control.
New Project
Open
Get from VCS
More Actions

Figure 3-1

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

3.3 Creating an Activity

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

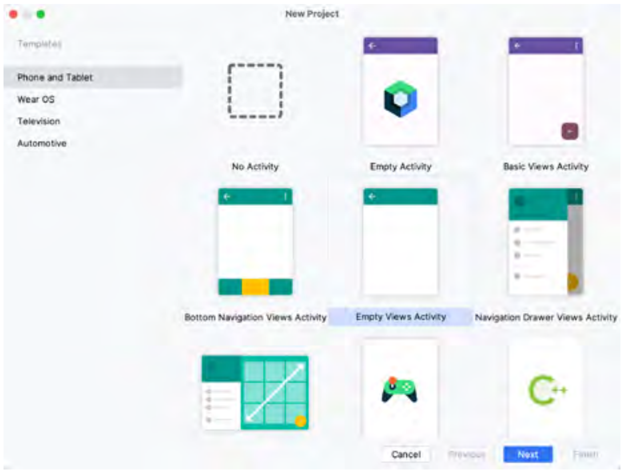


Figure 3-2

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

3.4 Defining the Project and SDK Settings

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

The *Package name* uniquely identifies 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 application's name. 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:

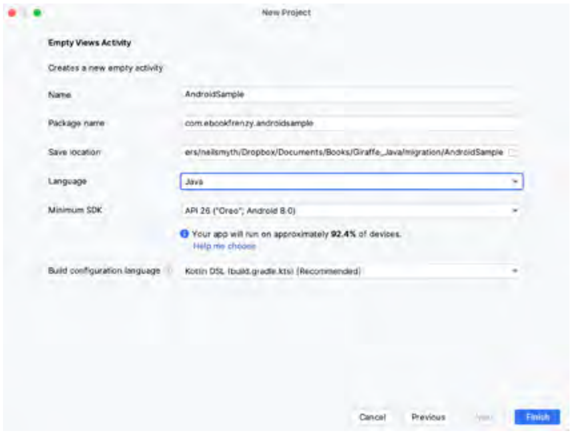
```
com.mycompany.androidsample
```

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 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 (Oreo; Android 8.0). This minimum SDK will be used in most projects created in this book unless a necessary feature is only available in a more recent version. The objective here is to build an app using the latest Android SDK while 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:



Empty Views Activity

Creates a new empty activity

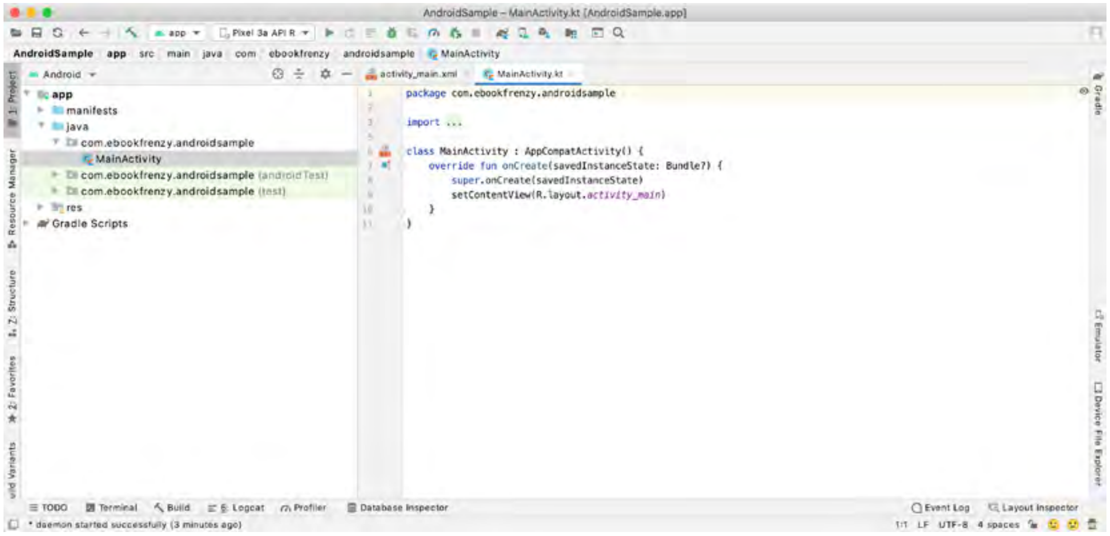
Name: AndroidSample
Package name: com.ebookfrenzy.androidsample
Save location: ers/neilsmyth/Dropbox/Documents/Books/Giraffe_Java/migration/AndroidSample
Language: Java
Minimum SDK: API 26 ("Oreo"; Android 8.0)
Your app will run on approximately **92.4%** of devices.
Help me choose
Build configuration language: Kotlin DSL (build.gradle.kts) (Recommended)

Figure 3-3

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

3.5 Enabling the New Android Studio UI

Android Studio is transitioning to a new, modern user interface that is not enabled by default in the Hedgehog version. If your installation of Android Studio resembles Figure 3-4 below, then you will need to enable the new UI before proceeding:

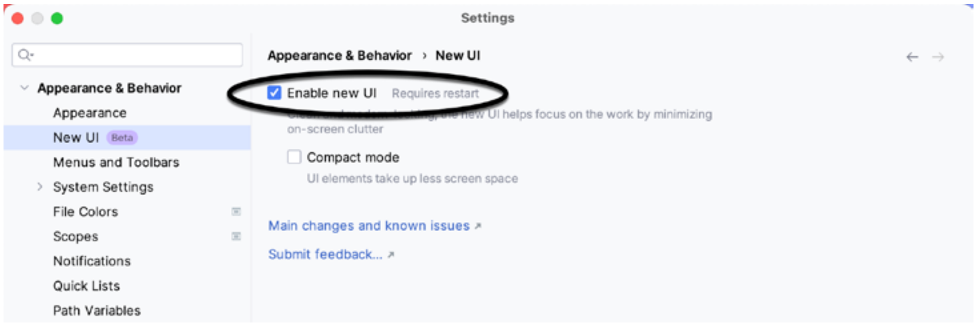


```
package com.ebookfrenzy.androidsample

import ...

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

* daemon started successfully (3 minutes ago)

Figure 3-4

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



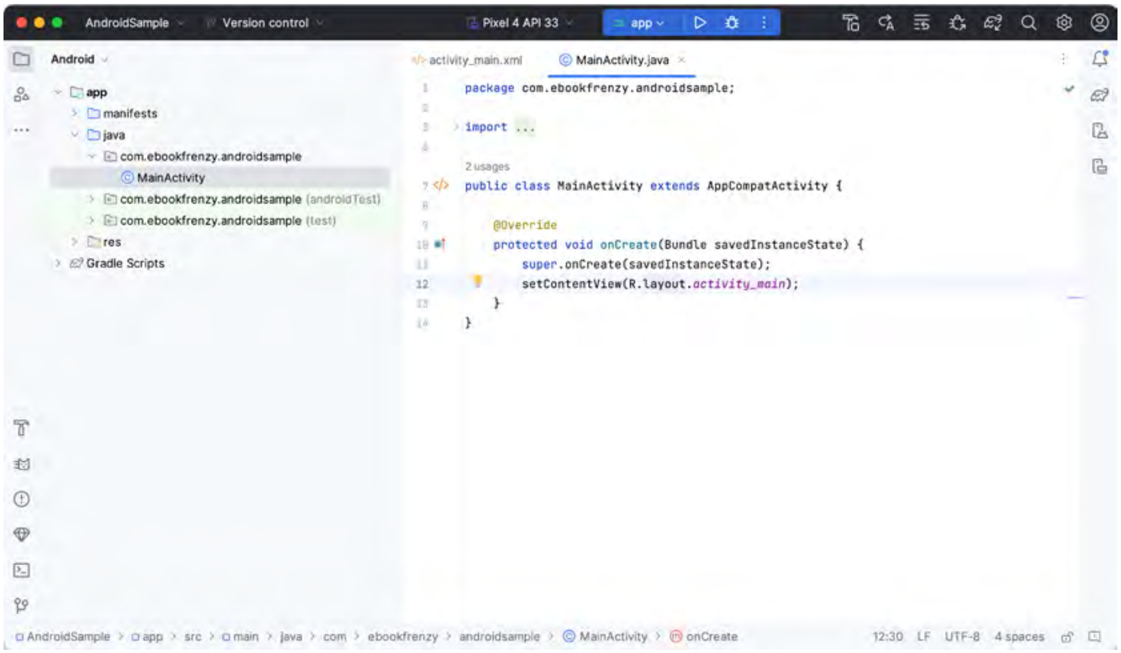
☑ Enable new UI Requires restart
the new UI helps focus on the work by minimizing on-screen clutter
☐ Compact mode
UI elements take up less screen space

Figure 3-5

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

3.6 Modifying the Example Application

Once Android Studio has restarted, the main window will reappear using the new UI and containing our AndroidSample project as illustrated in Figure 3-6 below:


AndroidSample
Version control
Pixel 4 API 33
app
Android
app
manifests
java
com.ebookfrenzy.androidsample
MainActivity
com.ebookfrenzy.androidsample (androidTest)
com.ebookfrenzy.androidsample (test)
res
Gradle Scripts
activity_main.xml
MainActivity.java
AndroidSample › app › src › main › java › com › ebookfrenzy › androidsample › MainActivity › onCreate
12:30 LF UTF-8 4 spaces

```
package com.ebookfrenzy.androidsample;

import ...

2 usages
public class MainActivity extends AppCompatActivity {

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

Figure 3-6

The newly created project and references to associated files are listed in the *Project* tool window on the left side of the main project window. The Project tool window has several modes in which information can be displayed. By default, this panel should be in *Android* mode. This setting is controlled by the menu at the top of the panel as highlighted in Figure 3-7. If the panel is not currently in Android mode, use the menu to switch mode:

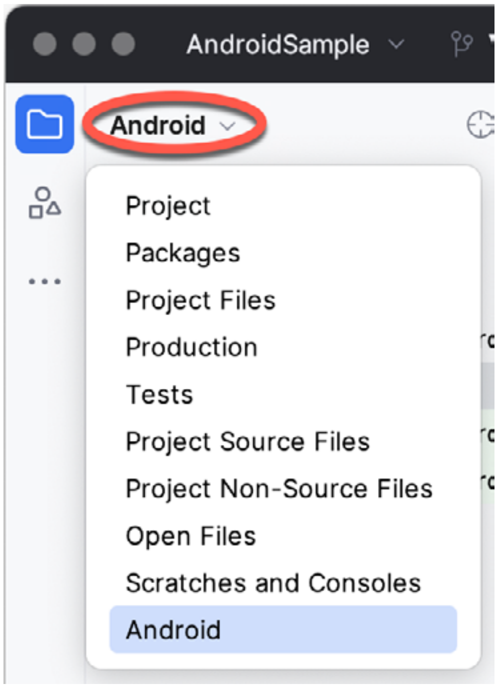


Figure 3-7

3.7 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 tool window 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:

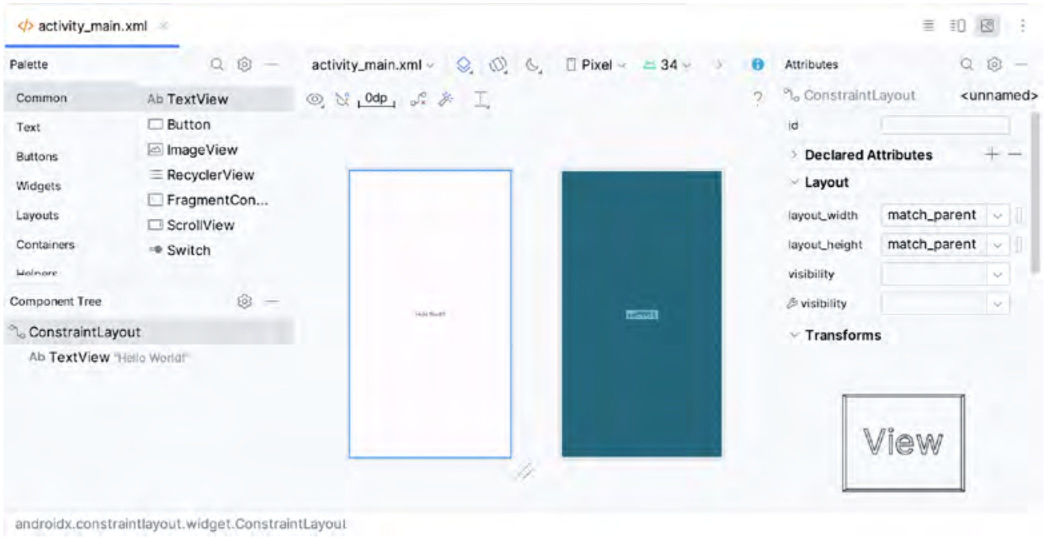


Figure 3-8

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 range of other device options are available by clicking on this menu.

Use the System UI Mode button ( ) to turn Night mode on and off for the device screen layout. To change the orientation of the device representation between landscape and portrait, use the drop-down menu showing the icon.

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. However, it should be noted that not all user interface components are 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 3-9:

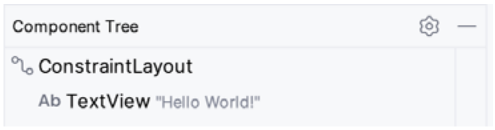


Figure 3-9

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, 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 ensure 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 U-shaped icon. When disabled, the icon appears with a diagonal line through it (Figure 3-10). If necessary, re-enable Autoconnect mode by clicking on this button.

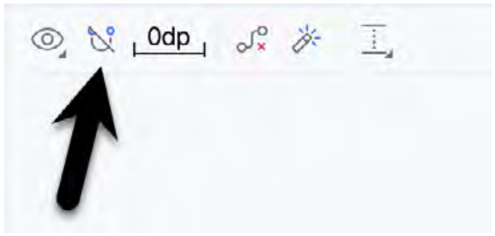


Figure 3-10

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 3-11, for example, the Button view is currently selected within the Buttons category:

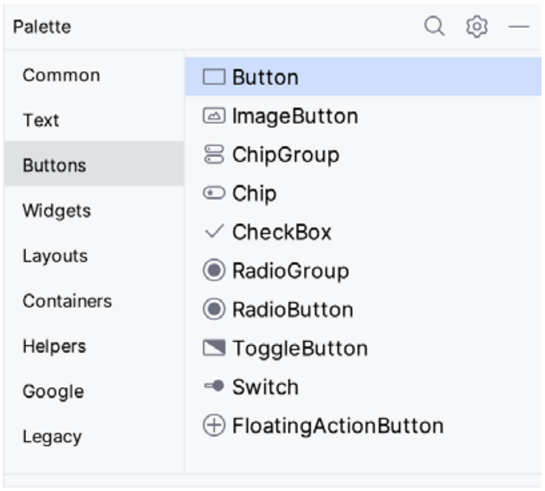


Figure 3-11

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:

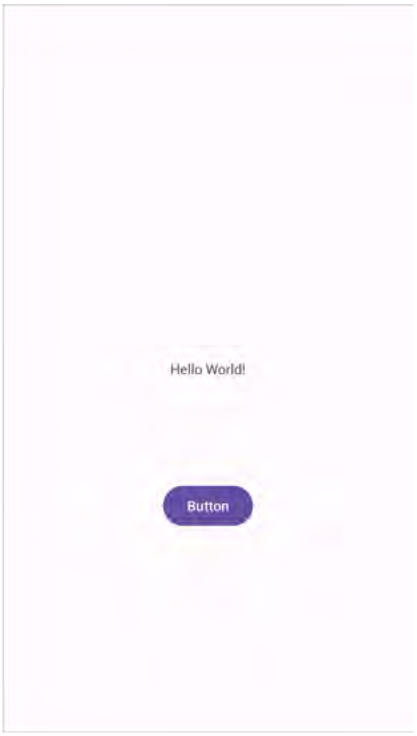
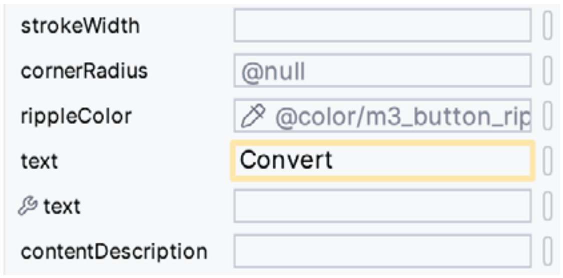


Figure 3-12

The next step is to change the text 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 3-13:



strokeWidth	
cornerRadius	@null
rippleColor	@color/m3_button_rip
text	Convert
text	
contentDescription	

Figure 3-13

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 how a visual component and the layout will behave with different settings without running the app repeatedly.

Just in case the Autoconnect system failed to set all of the layout connections, click on the Infer Constraints button (Figure 3-14) to add any missing constraints to the layout:

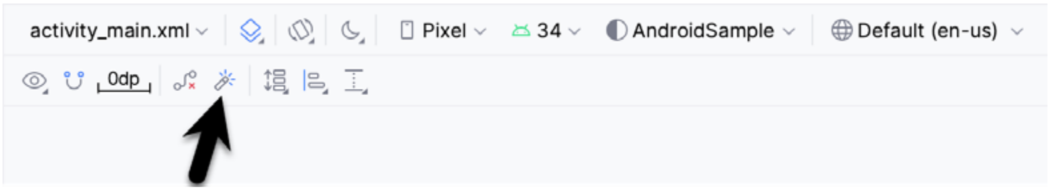


Figure 3-14

It is important to explain the warning button in the top right-hand corner of the Layout Editor tool, as indicated in Figure 3-15. This warning indicates potential problems with the layout. For details on any problems, click on the button:

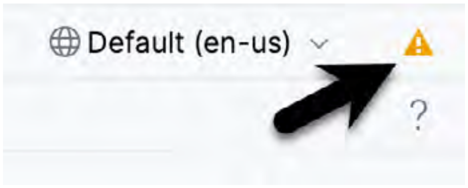
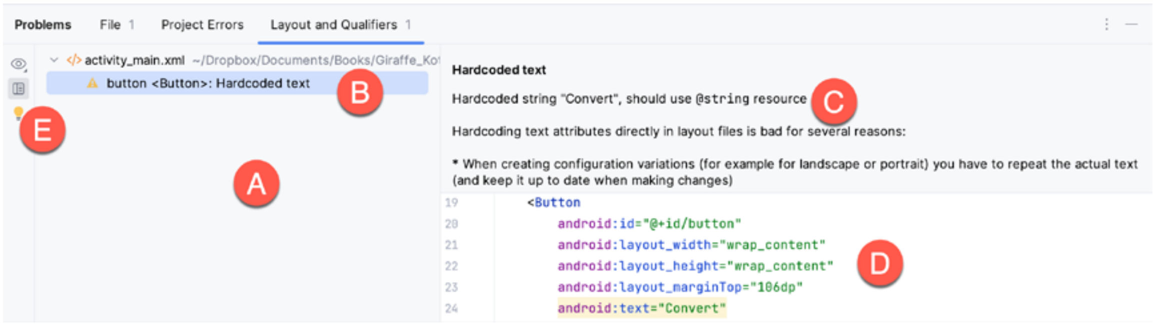


Figure 3-15

When clicked, the Problems tool window (Figure 3-16) will appear, describing the nature of the problems:



button <Button>: Hardcoded text

Hardcoded text

Hardcoded string "Convert", should use @string resource

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

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

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="106dp"
    android:text="Convert"
```

Figure 3-16

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 informs us that a potential issue exists concerning 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 reminds us that attributes and values such as text strings should be stored as *resources* wherever possible when developing Android applications. 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 3-17:

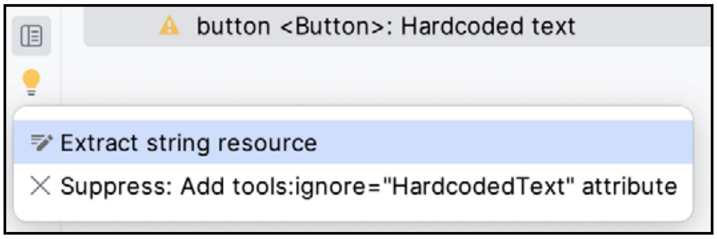

button <Button>: Hardcoded text
Extract string resource
Suppress: Add tools:ignore="HardcodedText" attribute

Figure 3-17

After selecting this option, the *Extract Resource* panel (Figure 3-18) 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:

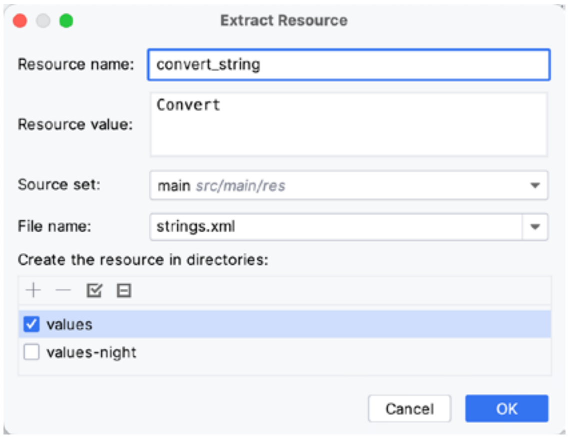

Extract Resource
Resource name: convert_string
Resource value: Convert
Source set: main src/main/res
File name: strings.xml
Create the resource in directories:
values
values-night
Cancel
OK

Figure 3-18

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 3-19:

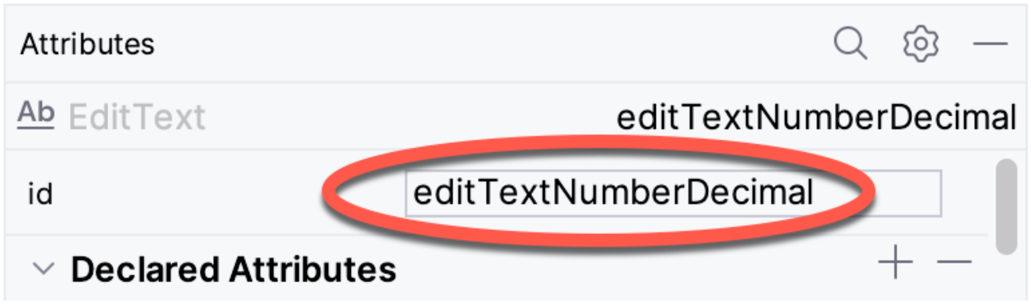


Figure 3-19

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

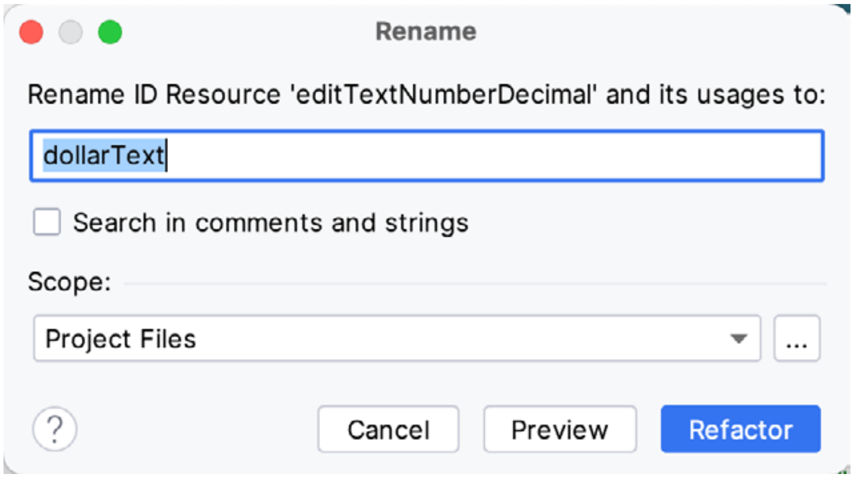

Rename
Rename ID Resource 'editTextNumberDecimal' and its usages to:
dollarText
Search in comments and strings
Scope:
Project Files
...
?
Cancel
Preview
Refactor

Figure 3-20

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

Add any missing layout constraints by clicking on the *Infer Constraints* button. At this point, the layout should resemble that shown in Figure 3-21:

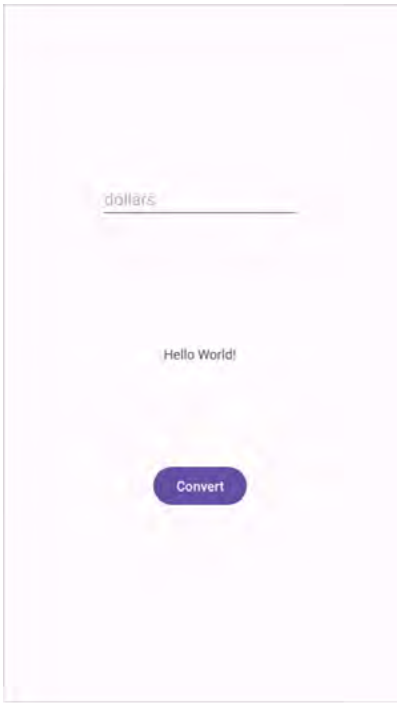


Figure 3-21

3.8 Reviewing the Layout and Resource Files

Before moving on to the next step, we will look at some internal aspects of user interface design and resource handling. In the previous section, we changed 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 the View Modes buttons marked A through C in Figure 3-22 below:

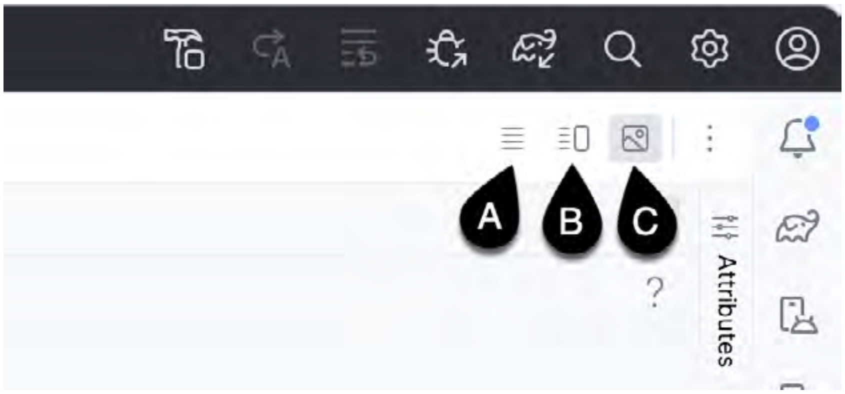
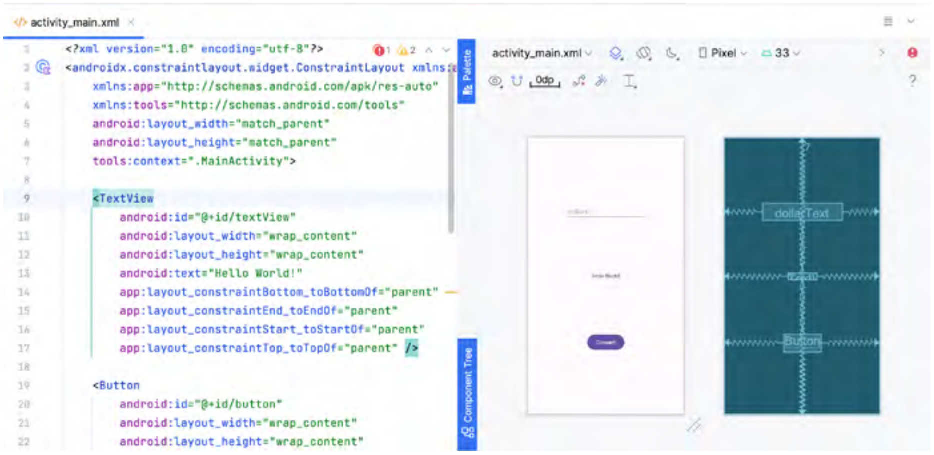


Figure 3-22

By default, the editor will be in *Design* mode (button C), whereby only the visual representation of the layout is displayed. In *Code* mode (A), the editor will display the XML for the layout, while in *Split* mode (B), both the layout and XML are displayed, as shown in Figure 3-23:



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

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

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

Figure 3-23

The button to the left of the View Modes button (marked B in Figure 3-22 above) is used to toggle between Code and Split modes quickly.

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 complexity and content vary, 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 layout color changes in real-time to match the new setting in the XML file. Note also that a small red square appears in the XML editor's left margin (also called the *gutter*) 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:

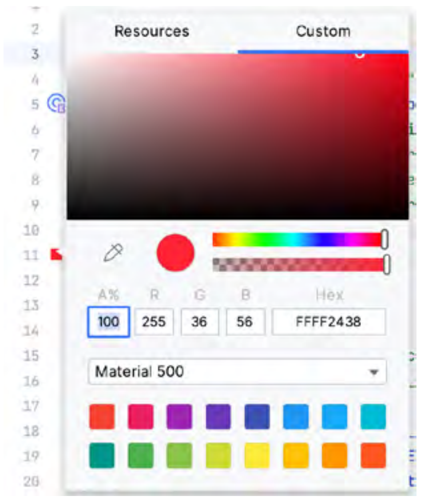


Figure 3-24

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:

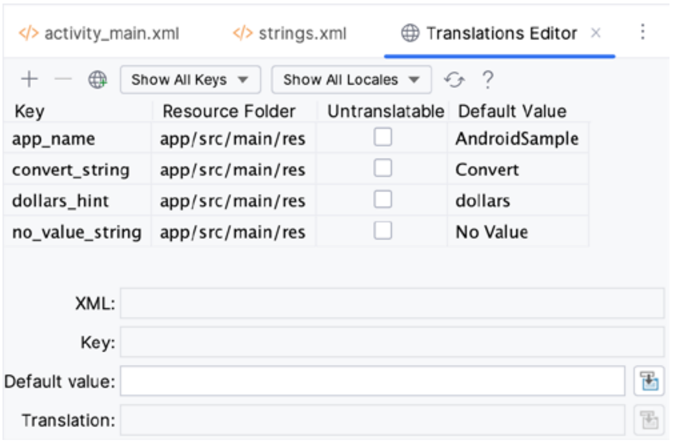
```
<resources>
    <string name="app_name">AndroidSample</string>
    <string name="convert_string">Convert</string>
    <string name="dollars_hint">dollars</string>
</resources>
```

To demonstrate 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 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="app_name">AndroidSample</string>
    <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 right-hand corner of the editor window. This will display the Translation Editor in the main panel of the Android Studio window:



Key	Resource Folder	Untranslatable	Default Value
app_name	app/src/main/res	☐	AndroidSample
convert_string	app/src/main/res	☐	Convert
dollars_hint	app/src/main/res	☐	dollars
no_value_string	app/src/main/res	☐	No Value

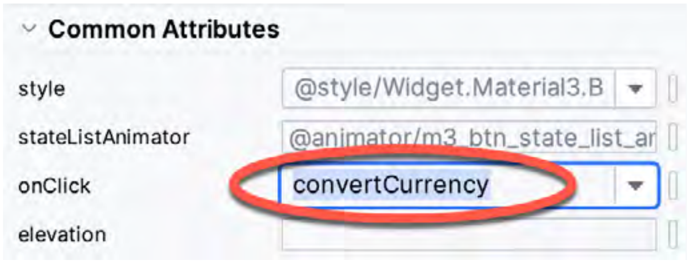
XML:
Key:
Default value:
Translation:

Figure 3-25

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

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 several ways and is covered in a later chapter entitled *"An Overview and Example of Android Event Handling"*. Return the layout editor to Design mode, select the Button widget in the layout editor, refer to the Attributes tool window, and specify a method named *convertCurrency* as shown below:



style: @style/Widget.Material3.B
stateListAnimator: @animator/m3_btn_state_list_ar
onClick: convertCurrency
elevation:

Figure 3-26

Next, double-click on the *MainActivity.java* file in the Project tool window (*app -> java -> <package name> -> MainActivity*) to load it into the code editor and add the code for the *convertCurrency* method to the class file so that it reads as follows, noting that it is also necessary to import some additional Android packages:

```
package com.ebookfrenzy.androidsample;

import androidx.appcompat.app.AppCompatActivity;

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

```
.
.
import java.util.Locale;

public class MainActivity extends AppCompatActivity {

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

    public void convertCurrency(View view) {

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

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

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

The method begins by obtaining references to the EditText and TextView objects by making a call to a method named findViewById, passing through the id assigned within the layout file. A check is then made to ensure that the user has entered a dollar value, and if so, that value is extracted, converted from a String to a floating point value, and converted to euros. Finally, the result is displayed on the TextView widget.

If any of this is unclear, rest assured that these concepts will be covered in greater detail in later chapters. In particular, the topic of accessing widgets from within code using findByViewId and an introduction to an alternative technique referred to as *view binding* will be covered in the chapter entitled *"An Overview of Android View Binding"*.

3.10 Summary

While not excessively complex, several steps are involved in setting up an Android development environment. Having performed those steps, it is worth working through an example to ensure 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 string values, and briefly touched on layouts. Next, we looked at the underlying XML used to store Android application user interface designs.

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

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

Chapter 20

20. Working with ConstraintLayout Chains and Ratios in Android Studio

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

20.1 Creating a Chain

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

Consider a layout consisting of three Button widgets constrained to be positioned in the top-left, top-center, and top-right of the ConstraintLayout parent, as illustrated in Figure 20-1:

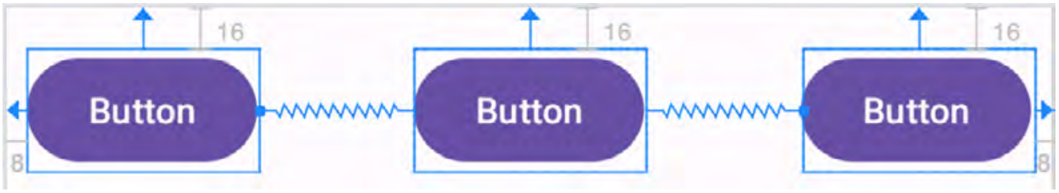


Figure 20-1

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

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toStartOf="@+id/button3"
    app:layout_constraintStart_toEndOf="@+id/button1"
    app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

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

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/button2" />

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toStartOf="@+id/button3"
    app:layout_constraintStart_toEndOf="@+id/button1"
    app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toEndOf="@+id/button2" />
```

With these changes, the widgets now have bi-directional horizontal constraints configured. This constitutes a ConstraintLayout chain represented visually within the Layout Editor by chain connections, as shown in Figure 20-2 below. Note that the chain has defaulted to the *spread* chain style in this configuration.

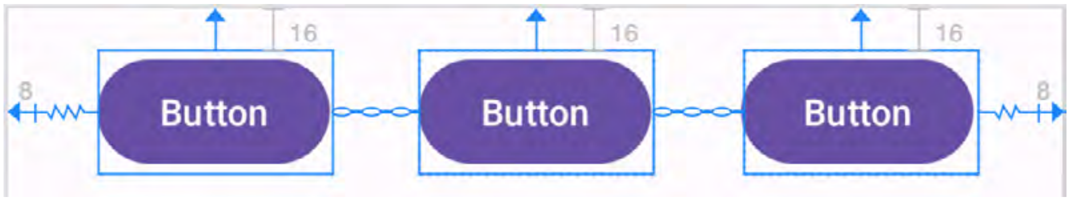


Figure 20-2

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

20.2 Changing the Chain Style

If no chain style is configured, the ConstraintLayout will default to the spread chain style. The chain style can be altered by right-clicking any of the widgets in the chain and selecting the *Cycle Chain Mode* menu option. Each time the menu option is clicked, the style will switch to another setting in the order of spread, spread inside, and packed.

Alternatively, the style may be specified in the Attributes tool window unfolding the *layout_constraints* property and changing either the *horizontal_chainStyle* or *vertical_chainStyle* property depending on the orientation of the chain:

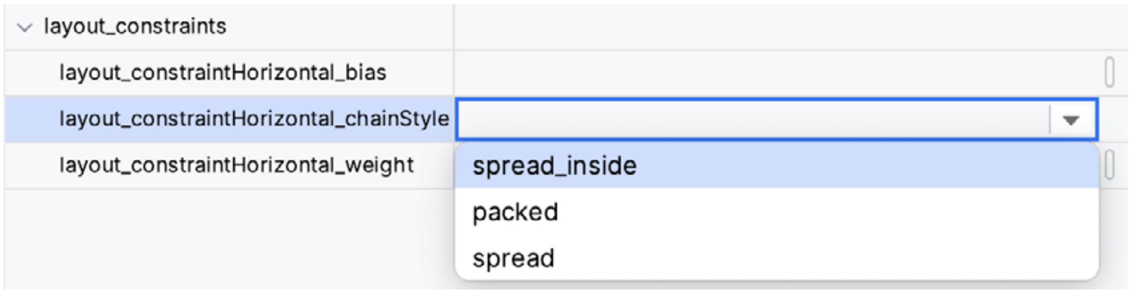

layout_constraints
layout_constraintHorizontal_bias
layout_constraintHorizontal_chainStyle
layout_constraintHorizontal_weight
spread_inside
packed
spread

Figure 20-3

20.3 Spread Inside Chain Style

Figure 20-4 illustrates the effect of changing the chain style to the *spread inside* chain style using the above techniques:

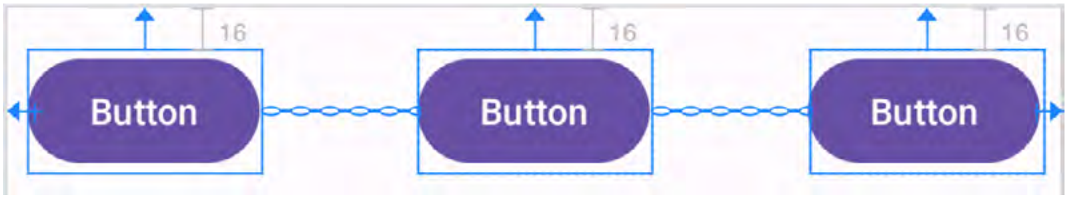


Figure 20-4

20.4 Packed Chain Style

Using the same technique, changing the chain style property to *packed* causes the layout to change, as shown in Figure 20-5:

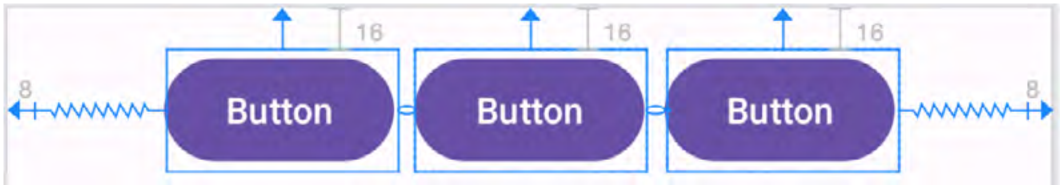


Figure 20-5

20.5 Packed Chain Style with Bias

The positioning of the packed chain may be influenced by applying a bias value. The bias can be between 0.0 and 1.0, with 0.5 representing the parent's center. Bias is controlled by selecting the chain head widget and assigning a value to the *layout_constraintHorizontal_bias* or *layout_constraintVertical_bias* attribute in the Attributes panel. Figure 20-6 shows a packed chain with a horizontal bias setting of 0.2:

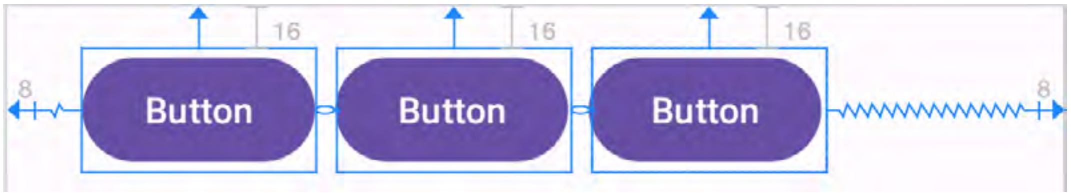
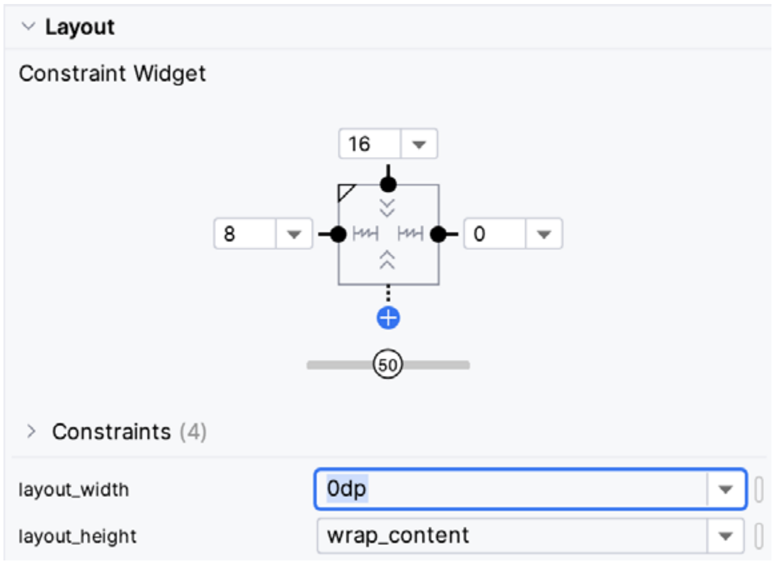


Figure 20-6

20.6 Weighted Chain

The final area of chains to explore involves weighting the individual widgets to control how much space each widget in the chain occupies within the available space. A weighted chain may only be implemented using the spread chain style, and any widget within the chain that responds to the weight property must have the corresponding dimension property (height for a vertical chain and width for a horizontal chain) configured for *match constraint* mode. Match constraint mode for a widget dimension may be configured by selecting the widget, displaying the Attributes panel, and changing the dimension to *match_constraint* (equivalent to 0dp). In Figure 20-7, for example, the *layout_width* constraint for a button has been set to *match_constraint (0dp)* to indicate that the width of the widget is to be determined based on the prevailing constraint settings:



layout_width: 0dp
layout_height: wrap_content

Figure 20-7

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

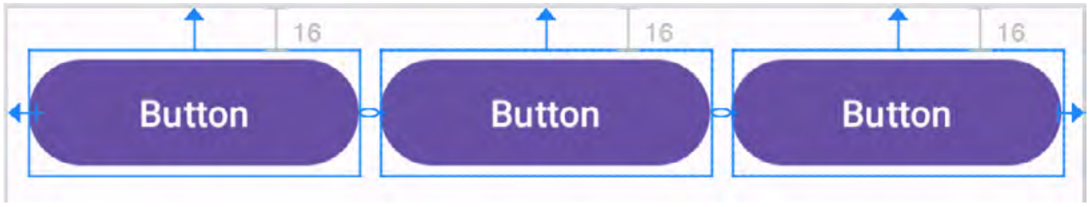


Figure 20-8

The amount of space occupied by each widget relative to the other widgets in the chain can be controlled by adding weight properties to the widgets. Figure 20-9 shows the effect of setting the *layout_constraintHorizontal_weight* property to 4 on button1, and to 2 on both button2 and button3:

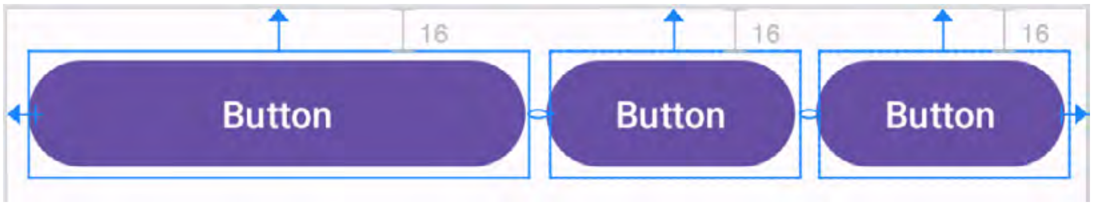


Figure 20-9

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

20.7 Working with Ratios

ConstraintLayout ratios allow one widget dimension to be sized relative to the widget's other dimension (also referred to as aspect ratio). For example, an aspect ratio setting could be applied to an ImageView to ensure that its width is always twice its height.

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

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

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

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

```
<ImageView
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:id="@+id/imageView"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintDimensionRatio="W,1:3" />
```

In the above example, the height will be defined subject to the constraints applied to it. In this case, constraints have been configured such that it is attached to the top and bottom of the parent view, essentially stretching the widget to fill the entire height of the parent. On the other hand, the width dimension has been constrained to be one-third of the ImageView's height dimension. Consequently, whatever size screen or orientation the layout appears on, the ImageView will always be the same height as the parent and the width one-third of that height.

The same results may also be achieved without manually editing the XML resource file. Whenever a widget dimension is set to match constraint mode, a ratio control toggle appears in the Inspector area of the property panel. Figure 20-10, for example, shows the layout width and height attributes of a button widget set to match constraint mode and 100dp respectively, and highlights the ratio control toggle in the widget sizing preview:

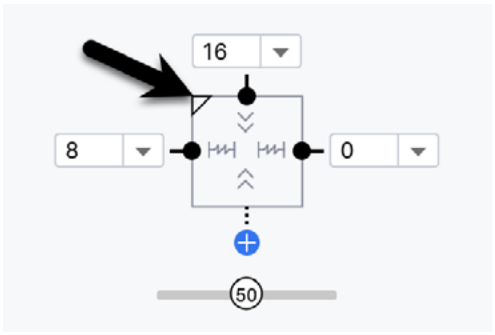


Figure 20-10

By default, the ratio sizing control is toggled off. Clicking on the control enables the ratio constraint and displays an additional field where the ratio may be changed:

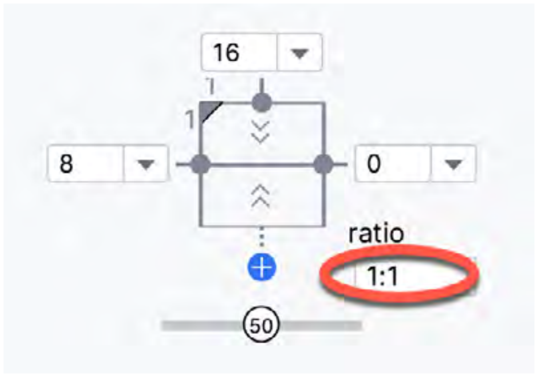


Figure 20-11

20.8 Summary

Both chains and ratios are powerful features of the ConstraintLayout class intended to provide additional options for designing flexible and responsive user interface layouts within Android applications. As outlined in this chapter, the Android Studio Layout Editor has been enhanced to make it easier to use these features during the user interface design process.

Chapter 25

25. A Guide to Using Apply Changes in Android Studio

Now that some of the basic concepts of Android development using Android Studio have been covered, this is a good time to introduce the Android Studio Apply Changes feature. As all experienced developers know, every second spent waiting for an app to compile and run is better spent writing and refining code.

25.1 Introducing Apply Changes

In early versions of Android Studio, each time a change to a project needed to be tested, Android Studio would recompile the code, convert it to Dex format, generate the APK package file, and install it on the device or emulator. Having performed these steps, the app would finally be launched and ready for testing. Even on a fast development system, this process takes considerable time to complete. It is not uncommon for it to take a minute or more for this process to complete for a large application.

Apply Changes, in contrast, allows many code and resource changes within a project to be reflected nearly instantaneously within the app while it is already running on a device or emulator session.

Consider, for example, an app being developed in Android Studio which has already been launched on a device or emulator. If changes are made to resource settings or the code within a method, Apply Changes will push the updated code and resources to the running app and dynamically "swap" the changes. The changes are then reflected in the running app without the need to build, deploy and relaunch the entire app. This often allows changes to be tested in a fraction of the time without Apply Changes.

25.2 Understanding Apply Changes Options

Android Studio provides three options for applying changes to a running app in the form of *Run App*, *Apply Changes and Restart Activity* and *Apply Code Changes*. These options can be summarized as follows:

- **Run App** - Stops the currently running app and restarts it. If no changes have been made to the project since it was last launched, this option will restart the app. If, on the other hand, changes have been made to the project, Android Studio will rebuild and re-install the app onto the device or emulator before launching it.

- **Apply Code Changes** - This option can be used when the only changes made to a project involve modifications to the body of existing methods or when a new class or method has been added. When selected, the changes will be applied to the running app without needing to restart the app or the currently running activity. This mode cannot, however, be used when changes have been made to any project resources, such as a layout file. Other restrictions include removing methods, changing a method signature, renaming classes, and other structural code changes. It is also impossible to use this option when changes have been made to the project manifest.

- **Apply Changes and Restart Activity** - When selected, this mode will dynamically apply any code or resource changes made within the project and restart the activity without re-installing or restarting the app. Unlike the Apply Code changes option, this can be used when changes have been made to the code and resources of the project. However, the same restrictions involving some structural code changes and manifest modifications apply.

25.3 Using Apply Changes

When a project has been loaded into Android Studio but is not yet running on a device or emulator, it can be launched as usual using either the run (marked A in Figure 25-1) or debug (B) button located in the toolbar:

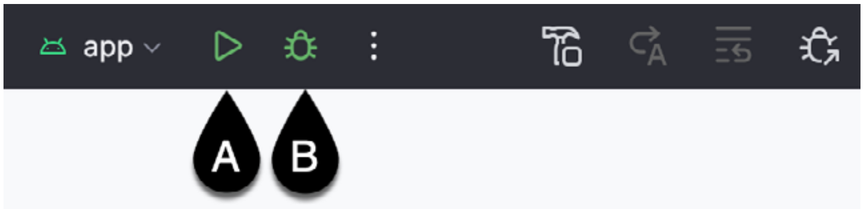


Figure 25-1

After the app has launched and is running, a stop button (marked A in Figure 25-2) will appear, and the *Apply Changes and Restart Activity* (B) and *Apply Code Changes* (C) buttons will be enabled:

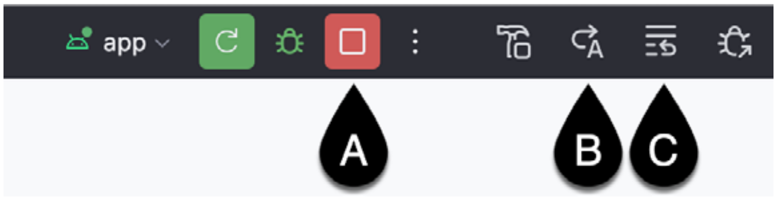
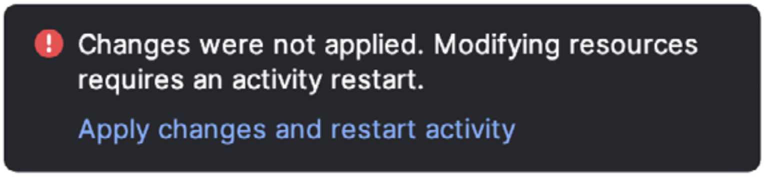


Figure 25-2

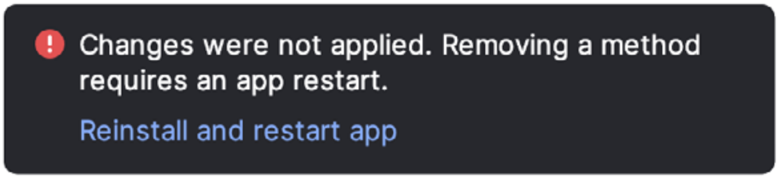
If the changes cannot be applied when one of the Apply Changes buttons is selected, Android Studio will display a message indicating the failure and an explanation. Figure 25-3, for example, shows the message displayed by Android Studio when the *Apply Code Changes* option is selected after a change has been made to a resource file:


Changes were not applied. Modifying resources requires an activity restart.

Apply changes and restart activity

Figure 25-3

In this situation, the solution is to use the *Apply Changes and Restart Activity* option (for which a link is provided). Similarly, the following message will appear when an attempt to apply changes that involve the removal of a method is made:


Changes were not applied. Removing a method requires an app restart.

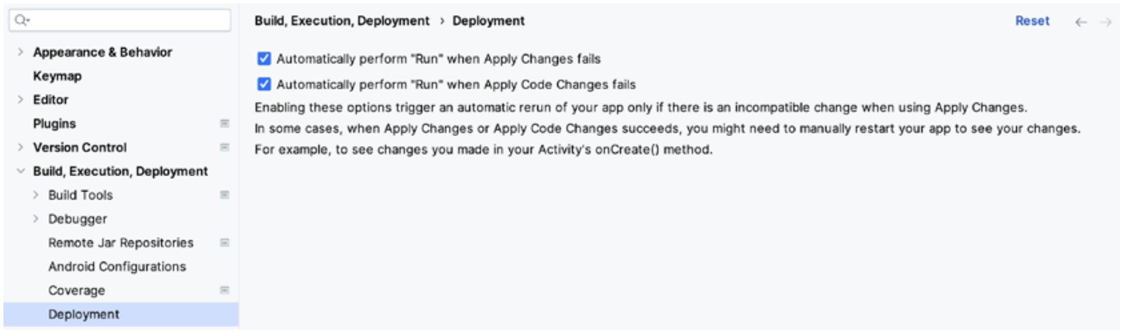
Reinstall and restart app

Figure 25-4

In this case, the only option is to click on the *Run App* button to re-install and restart the app. As an alternative to manually selecting the correct option, Android Studio may be configured to automatically fall back to performing a Run App operation.

25.4 Configuring Apply Changes Fallback Settings

The Apply Changes fallback settings are located in the Android Studio Settings dialog. Within the Settings dialog, select the *Build, Execution, Deployment* entry in the left-hand panel, followed by *Deployment*, as shown in Figure 25-5:



Build, Execution, Deployment › Deployment

☑ Automatically perform "Run" when Apply Changes fails
☑ Automatically perform "Run" when Apply Code Changes fails

Enabling these options trigger an automatic rerun of your app only if there is an incompatible change when using Apply Changes.
In some cases, when Apply Changes or Apply Code Changes succeeds, you might need to manually restart your app to see your changes.
For example, to see changes you made in your Activity's onCreate() method.

Figure 25-5

Once the required options have been enabled, click on Apply, followed by the OK button to commit the changes and dismiss the dialog. After these defaults have been enabled, Android Studio will automatically re-install and restart the app when necessary.

25.5 An Apply Changes Tutorial

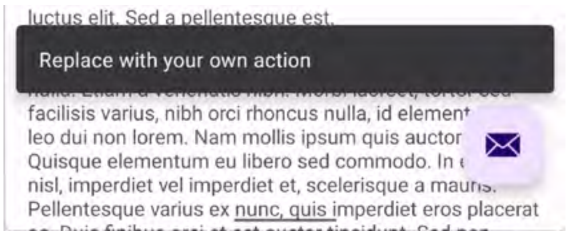
Launch Android Studio, select the New Project option from the welcome screen, and choose the Basic Views Activity template within the resulting new project dialog before clicking the Next button.

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

25.6 Using Apply Code Changes

Begin by clicking the run button and selecting an emulator or physical device as the run target. After clicking the run button, track the time before the example app appears on the device or emulator.

Once running, click on the action button (the button displaying an envelope icon in the screen's lower right-hand corner). Note that a Snackbar instance appears, displaying text which reads "Replace with your own action", as shown in Figure 25-6:



Replace with your own action

Figure 25-6

Once the app is running, the Apply Changes buttons should have been enabled, indicating that certain project changes can be applied without reinstalling and restarting the app. To see this in action, edit the *MainActivity.java* file, locate the *onCreate* method, and modify the action code so that a different message is displayed when the action button is selected:

```
binding.fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Snackbar.make(view, "Apply Changes is Amazing!", Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
    }
});
```

With the code change implemented, click the *Apply Code Changes* button and note that a message appears within a few seconds indicating the app has been updated. Tap the action button and note that the new message is now displayed in the Snackbar.

25.7 Using Apply Changes and Restart Activity

Any resource change will require the use of the *Apply Changes and Restart Activity* option. Within Android Studio, select the *app -> res -> layout -> fragment_first.xml* layout file. With the Layout Editor tool in Design mode, select the default TextView component and change the text property in the attributes tool window to "Hello Android".

Ensure that the fallback options outlined in *"Configuring Apply Changes Fallback Settings"* above are turned off before clicking on the *Apply Code Changes* button. Note that the request fails because this change involves project resources. Click on the *Apply Changes and Restart Activity* button and verify that the activity restarts and displays the new text on the TextView widget.

25.8 Using Run App

As previously described, removing a method requires the complete re-installation and restart of the running app. To experience this, edit the *MainActivity.java* file and add a new method after the *onCreate* method as follows:

```
public void demoMethod() {
}
```

Use the *Apply Code Changes* button and confirm that the changes are applied without re-installing the app.

Next, delete the new method and verify that clicking on either of the two Apply Changes buttons will result in the request failing. The only way to run the app after such a change is to click the Run App button.

25.9 Summary

Apply Changes is a feature of Android Studio designed to significantly accelerate the code, build and run cycle performed when developing an app. The Apply Changes feature can push updates to the running application, in many cases, without reinstalling or restarting the app. Apply Changes provides several different levels of support depending on the nature of the modification being applied to the project.

Chapter 32

32. Modern Android App Architecture with Jetpack

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

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

32.1 What is Android Jetpack?

Android Jetpack consists of Android Studio, the Android Architecture Components, the Android Support Library, and a set of guidelines recommending how an Android App should be structured. The Android Architecture Components are designed to make it quicker and easier to perform common tasks when developing Android apps while also conforming to the key principle of the architectural guidelines.

While all Android Architecture Components will be covered in this book, this chapter will focus on the key architectural guidelines and the ViewModel, LiveData, and Lifecycle components while introducing Data Binding and Repositories.

Before moving on, it is important to understand that the Jetpack approach to app development is optional. While highlighting some of the shortcomings of other techniques that have gained popularity over the years, Google stopped short of completely condemning those approaches to app development. Google is taking the position that while there is no right or wrong way to develop an app, there is a recommended way.

32.2 The "Old" Architecture

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

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

32.3 Modern Android Architecture

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

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

32.4 The ViewModel Component

The purpose of ViewModel is to separate the user interface-related data model and logic of an app from the code responsible for displaying and managing the user interface and interacting with the operating system. When designed this way, an app will consist of one or more UI Controllers, such as an activity, together with ViewModel instances responsible for handling the data those controllers need.

The ViewModel only knows about the data model and corresponding logic. It knows nothing about the user interface and does not attempt to directly access or respond to events relating to views within the user interface. When a UI controller needs data to display, it asks the ViewModel to provide it. Similarly, when the user enters data into a view within the user interface, the UI controller passes it to the ViewModel for handling.

This separation of responsibility addresses the issues relating to the lifecycle of UI controllers. Regardless of how often the UI controller is recreated during the lifecycle of an app, the ViewModel instances remain in memory, thereby maintaining data consistency. For example, a ViewModel used by an activity will remain in memory until the activity finishes, which, in the single activity app, is not until the app exits.

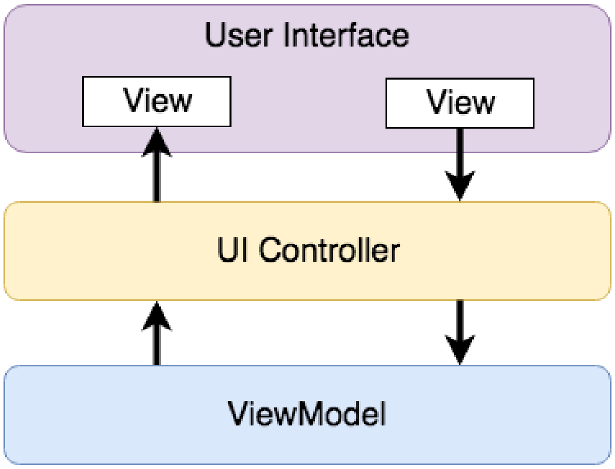


Figure 32-1

32.5 The LiveData Component

Consider an app that displays real-time data, such as the current price of a financial stock. The app could use a stock price web service to continuously update the data model within the ViewModel with the latest information. This real-time data is of use only if it is displayed to the user promptly. There are only two ways that the UI controller can ensure that the latest data is displayed in the user interface. One option is for the controller to continuously check with the ViewModel to determine if the data has changed since it was last displayed. However, the problem with this approach is that it could be more efficient. To maintain the real-time nature of the data feed, the UI controller would have to run on a loop, continuously checking for the data to change.

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

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

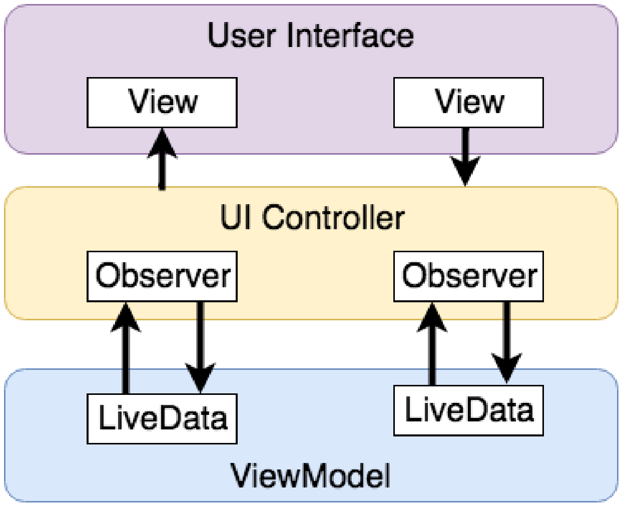


Figure 32-2

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

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

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

32.6 ViewModel Saved State

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

32.7 LiveData and Data Binding

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

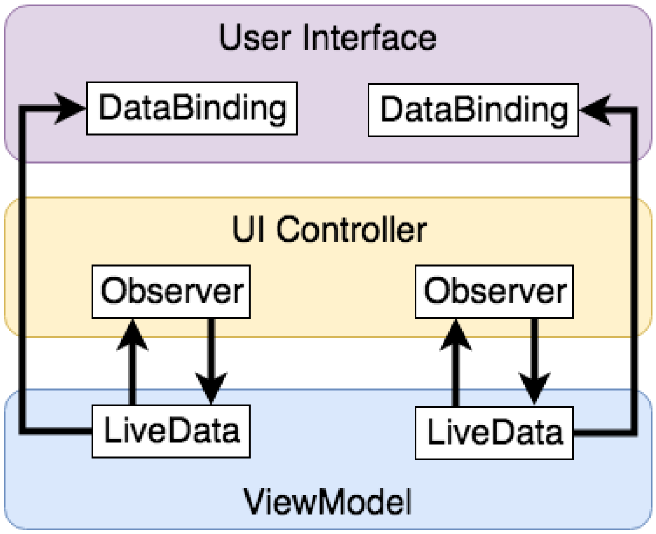


Figure 32-3

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

32.8 Android Lifecycles

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

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

Objects that can detect and react to lifecycle state changes in other objects are said to be *lifecycle-aware*. In contrast, objects that provide access to their lifecycle state are called *lifecycle owners*. The chapter entitled *"Working with Android Lifecycle-Aware Components"* will cover Lifecycles in greater detail.

32.9 Repository Modules

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

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

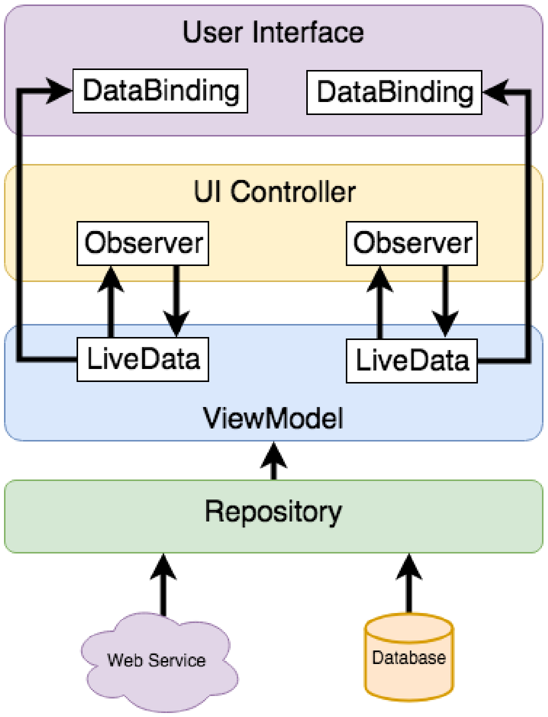


Figure 32-4

32.10 Summary

Until recently, Google has tended not to recommend any particular approach to structuring an Android app. That has now changed with the introduction of Android Jetpack, consisting of tools, components, libraries, and architecture guidelines. Google now recommends that an app project be divided into separate modules, each responsible for a particular area of functionality, otherwise known as "separation of concerns".

In particular, the guidelines recommend separating the view data model of an app from the code responsible for handling the user interface. In addition, the code responsible for gathering data from data sources such as web services or databases should be built into a separate repository module instead of being bundled with the view model.

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

34. An Android Jetpack LiveData Tutorial

The previous chapter began building an app to conform to the recommended Jetpack architecture guidelines. These initial steps involved implementing the data model for the app user interface within a ViewModel instance.

This chapter will further enhance the app design using the LiveData architecture component. Once LiveData support has been added to the project in this chapter, the next chapters (starting with *"An Overview of Android Jetpack Data Binding"*) will use the Jetpack Data Binding library to eliminate even more code from the project.

34.1 LiveData - A Recap

LiveData was previously introduced in the *"Modern Android App Architecture with Jetpack"* chapter. As described earlier, the LiveData component can be used as a wrapper around data values within a view model. Once contained in a LiveData instance, those variables become observable to other objects within the app, typically UI controllers such as Activities and Fragments. This allows the UI controller to receive a notification whenever the underlying LiveData value changes. An observer is set up by creating an instance of the Observer class and defining an *onChange()* method to be called when the LiveData value changes. Once the Observer instance has been created, it is attached to the LiveData object via a call to the LiveData object's *observe()* method.

LiveData instances can be declared mutable using the MutableLiveData class, allowing both the ViewModel and UI controller to change the underlying data value.

34.2 Adding LiveData to the ViewModel

Launch Android Studio, open the ViewModelDemo project created in the previous chapter, and open the *MainViewModel.java* file, which should currently read as follows:

```
package com.ebookfrenzy.viewmodeldemo;

import androidx.lifecycle.ViewModel;

public class MainViewModel extends ViewModel {

    private static final Float rate = 0.74F;
    private String dollarText = "";
    private Float result = 0F;

    public void setAmount(String value) {
        this.dollarText = value;
        result = Float.parseFloat(dollarText)*rate;
    }

    public Float getResult()
    {
        return result;
    }
}
```

This stage in the chapter aims to wrap the *result* variable in a MutableLiveData instance (the object will need to be mutable so that the value can be changed each time the user requests a currency conversion). Begin by modifying the class so that it now reads as follows, noting that an additional package needs to be imported when making use of LiveData:

```
package com.ebookfrenzy.viewmodeldemo;

import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class  MainViewModel extends ViewModel {

    private static final Float rate = 0.74F;
    private String dollarText = "";
    ~~private Float result = 0F;~~
    final private MutableLiveData<Float> result = new MutableLiveData<>();

    public void setAmount(String value) {
        this.dollarText = value;
        result = Float.parseFloat(dollarText)*rate;
    }

    public Float getResult()
    {
        return result;
    }
}
```

Now that the result variable is contained in a mutable LiveData instance, both the *setAmount()* and *getResult()* methods must be modified. In the case of the *setAmount()* method, a value can no longer be assigned to the result variable using the assignment (=) operator. Instead, the LiveData *setValue()* method must be called, passing through the new value as an argument. As currently implemented, the *getResult()* method is declared to return a Float value and must be changed to return a MutableLiveData object. Making these remaining changes results in the following class file:

```
package com.ebookfrenzy.viewmodeldemo;

import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class  MainViewModel extends ViewModel {

    private static final Float rate = 0.74F;
    private String dollarText = "";
    final private MutableLiveData<Float> result = new MutableLiveData<>();
    public void setAmount(String value) {
        this.dollarText = value;
        ~~result = Float.parseFloat(dollarText)*rate;~~
        result.setValue(Float.parseFloat(dollarText)*rate);
    }

    ~~public Float getResult()~~
    public MutableLiveData<Float> getResult()
    {
        return result;
    }
}
```

34.3 Implementing the Observer

Now that the conversion result is contained within a LiveData instance, the next step is configuring an observer within the UI controller, which, in this example, is the FirstFragment class. Locate the *FirstFragment.java* class (*app -> java -> <package name> -> FirstFragment*), double-click on it to load it into the editor, and modify the *onViewCreated()* method to create a new Observer instance named *resultObserver*:

```
package com.ebookfrenzy.viewmodeldemo;

import androidx.lifecycle.Observer;
.
.
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle 
savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    binding.resultText.setText(String.format(Locale.ENGLISH,"%.2f",
            viewModel.getResult()));

    final Observer<Float> resultObserver = new Observer<Float>() {
        @Override
        public void onChanged(@Nullable final Float result) {
            binding.resultText.setText(String.format(Locale.ENGLISH,
                    "%.2f", result));
        }
    };
.
.
}
```

The *resultObserver* instance declares the *onChanged()* method which, when called, is passed the current result value, which it then converts to a string and displays on the resultText TextView object. The next step is to add the observer to the result LiveData object, a reference that can be obtained via a call to the *getResult()* method of the ViewModel object. Since updating the result TextView is now the responsibility of the *onChanged()* callback method, the existing lines of code to perform this task can now be deleted:

```
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle 
savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    ~~binding.resultText.setText(String.format(Locale.ENGLISH,"%.2f",
            viewModel.getResult()));~~

    final Observer<Float> resultObserver = new Observer<Float>() {
        @Override
        public void onChanged(@Nullable final Float result) {
            binding.resultText.setText(String.format(Locale.ENGLISH,
                    "%.2f", result));
        }
    };

    viewModel.getResult().observe(getViewLifecycleOwner(), resultObserver);

    binding.convertButton.setOnClickListener(v -> {
        if (!binding.dollarText.getText().toString().equals("")) {
            viewModel.setAmount(String.format(Locale.ENGLISH,"%s",
                    binding.dollarText.getText()));
            ~~binding.resultText.setText(String.format(Locale.ENGLISH,"%.2f",
                    viewModel.getResult()));~~
        } else {
            binding.resultText.setText("No Value");
        }
    });
}
```

Compile and run the app, enter a value into the dollar field, click on the Convert button, and verify that the converted euro amount appears on the TextView. This confirms that the observer received notification that the result value had changed and called the *onChanged()* method to display the latest data.

Note in the above implementation of the *onViewCreated()* method that the line of code responsible for displaying the current result value each time the method was called was removed. This was originally put in place to ensure that the displayed value was recovered if the Fragment was recreated for any reason. Because LiveData monitors the lifecycle status of its observers, this step is no longer necessary. When LiveData detects that the UI controller was recreated, it automatically triggers any associated observers and provides the latest data. Verify this by rotating the device while a euro value is displayed on the TextView object and confirming that the value is not lost.

Before moving on to the next chapter, close the project, copy the ViewModelDemo project folder, and save it as ViewModelDemo_LiveData to be used later when saving the ViewModel state.

34.4 Summary

This chapter demonstrated the use of the Android LiveData component to ensure that the data displayed to the user always matches that stored in the ViewModel. This relatively simple process consisted of wrapping a ViewModel data value within a LiveData object and setting up an observer within the UI controller subscribed to the LiveData value. Each time the LiveData value changes, the observer is notified, and the *onChanged()* method is called and passed the updated value.

Adding LiveData support to the project has gone some way towards simplifying the design of the project. Additional and significant improvements are also possible using the Data Binding Library, details of which will be covered in the next chapter.

Chapter 42

42. An Introduction to MotionLayout

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

42.1 An Overview of MotionLayout

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

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

MotionLayout animations are declared entirely in XML and do not typically require writing code. These XML declarations may be implemented manually in the Android Studio code editor, visually using the MotionLayout editor, or combining both approaches.

42.2 MotionLayout

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

Unsurprisingly since it is a subclass of ConstraintLayout, MotionLayout supports all of the layout features of the ConstraintLayout. Therefore, a user interface layout can be similarly designed when using MotionLayout for views that do not require animation.

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

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

42.3 MotionScene

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

The following listing shows a template for a MotionScene file:

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

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

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

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

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

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

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

42.4 Configuring ConstraintSets

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

- alpha
- visibility
- elevation
- rotation

- rotationX
- rotationY
- translationX
- translationY
- translationZ
- scaleX
- scaleY

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

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

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

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

42.5 Custom Attributes

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

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

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

- **motion:customBoolean** - Boolean attribute values.

- **motion:customColorValue** - Color attribute values.
- **motion:customDimension** - Dimension attribute values.
- **motion:customFloatValue** - Floating point attribute values.
- **motion:customIntegerValue** - Integer attribute values.
- **motion:customStringValue** - String attribute values

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

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

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

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

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

42.6 Triggering an Animation

Without some event to tell MotionLayout to start the animation, none of the settings in the MotionScene file will affect the layout (except that the view will be positioned based on the setting in the start ConstraintSet).

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

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

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

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

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

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

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

42.7 Arc Motion

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

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

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

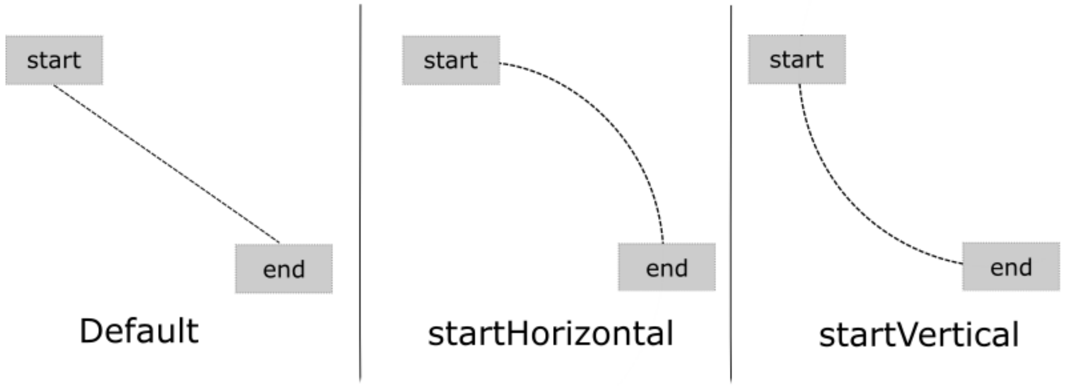


Figure 42-1

42.8 Keyframes

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

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

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

42.8.1 Attribute Keyframes

Attribute Keyframes (declared using KeyAttribute) allow view attributes to be changed at intermediate points in the animation timeline. KeyAttribute supports the attributes listed above for ConstraintSets combined with the ability to specify where the change will take effect in the animation timeline. For example, the following

Keyframe declaration will gradually cause the button view to double in size horizontally (scaleX) and vertically (scaleY), reaching full size at 50% through the timeline. For the remainder of the timeline, the view will decrease in size to its original dimensions:

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

42.8.2 Position Keyframes

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

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

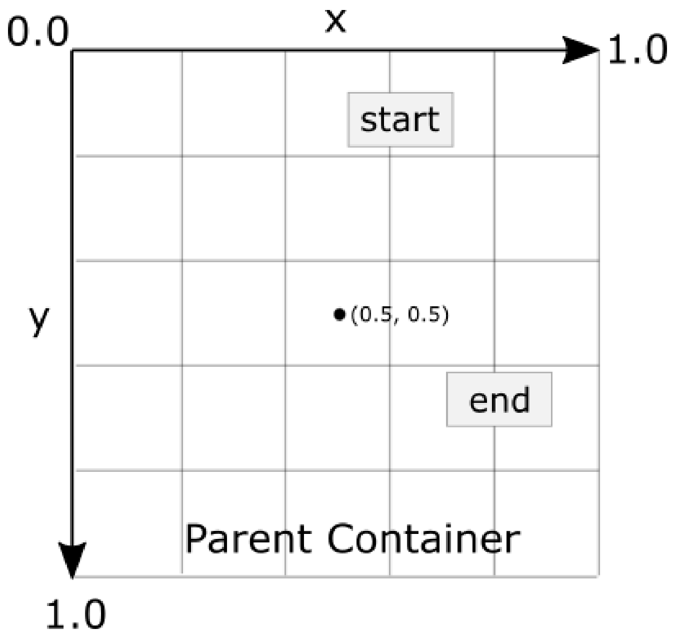


Figure 42-2

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

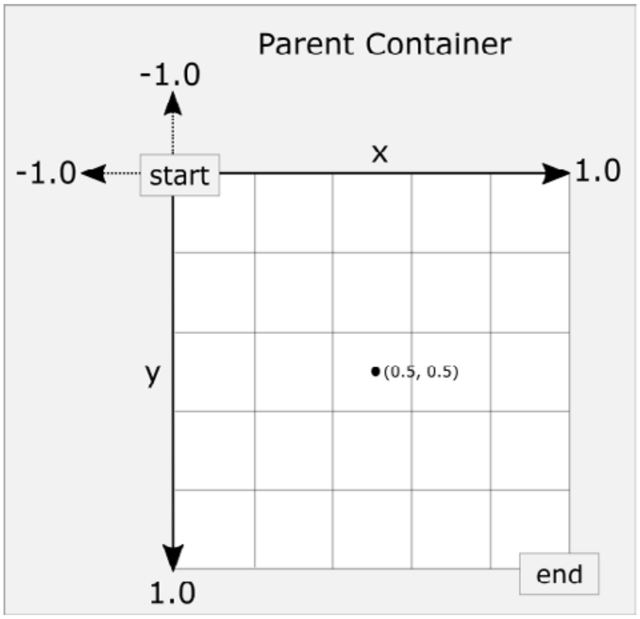


Figure 42-3

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

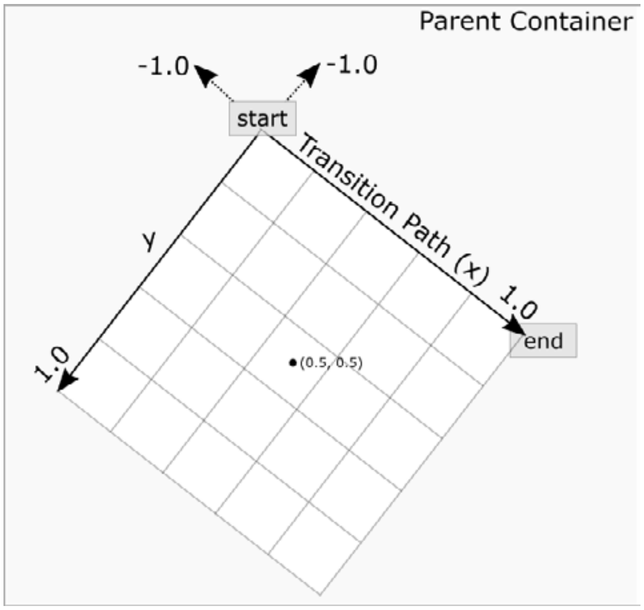


Figure 42-4

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

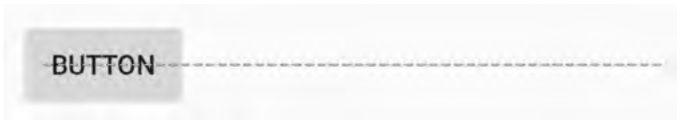


Figure 42-5

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

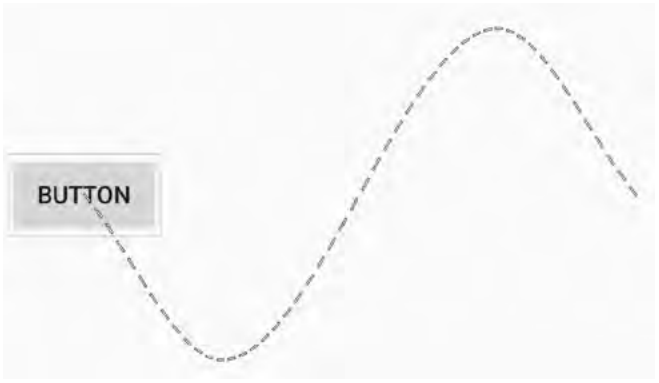


Figure 42-6

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

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

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

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

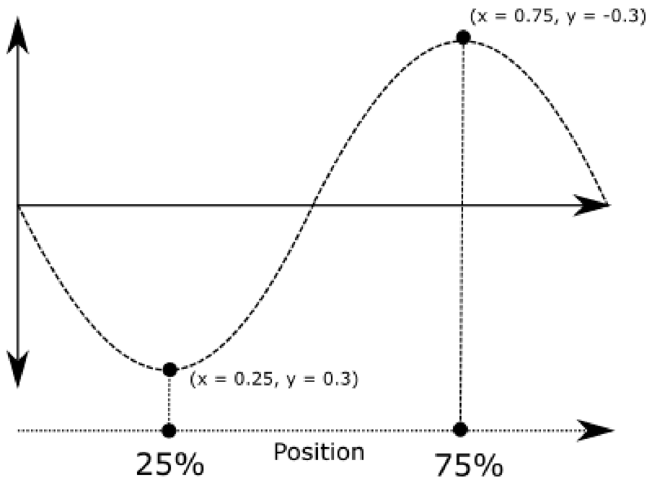


Figure 42-7

42.9 Time Linearity

Without additional settings, the animations outlined above will be performed at a constant speed. To vary the animation speed (for example, so that it accelerates and then decelerates), the transition easing attribute (transitionEasing) can be used within a ConstraintSet or Keyframe.

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

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

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

https://cubic-bezier.com/

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

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

42.10 KeyTrigger

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

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

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

```
<KeyTrigger
     motion:framePosition="10"
     motion:onNegativeCross="show"
     motion:motionTarget="@id/button2"/>
```

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

```
<KeyTrigger
     motion:framePosition="10"
     motion:onCross="show"
     motion:motionTarget="@id/button2"/>
```

42.11 Cycle and Time Cycle Keyframes

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

42.12 Starting an Animation from Code

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

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

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

```
motionLayout.setTransition(R.id.myStart, R.id.myEnd);
motionLayout.transitionToEnd();
```

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

```
motionLayout.setTransitionListener(transitionListener);

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

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

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

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

42.13 Summary

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

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

Chapter 64

64. An Overview of Android SQLite Databases

Mobile applications that do not need to store at least some persistent data are few and far between. The use of databases is an essential aspect of most applications, ranging from almost entirely data-driven applications to those that need to store small amounts of data, such as the prevailing game score.

The importance of persistent data storage becomes even more evident when considering the transient lifecycle of the typical Android application. With the ever-present risk that the Android runtime system will terminate an application component to free up resources, a comprehensive data storage strategy to avoid data loss is a key factor in designing and implementing any application development strategy.

This chapter will cover the SQLite database management system bundled with the Android operating system and outline the Android SDK classes that facilitate persistent SQLite-based database storage within an Android application. Before delving into the specifics of SQLite in the context of Android development, however, a brief overview of databases and SQL will be covered.

64.1 Understanding Database Tables

Database *Tables* provide the most basic level of data structure in a database. Each database can contain multiple tables, each designed to hold information of a specific type. For example, a database may contain a *customer* table that contains the name, address, and telephone number of each of the customers of a particular business. The same database may also include a *products* table used to store the product descriptions with associated product codes for the items sold by the business.

Each table in a database is assigned a name that must be unique within that particular database. A table name, once assigned to a table in one database, may not be used for another table except within the context of another database.

64.2 Introducing Database Schema

Database Schemas define the characteristics of the data stored in a database table. For example, the table schema for a customer database table might define the customer name as a string of no more than 20 characters long and the customer phone number is a numerical data field of a certain format.

Schemas are also used to define the structure of entire databases and the relationship between the various tables in each database.

64.3 Columns and Data Types

It is helpful at this stage to begin viewing a database table as similar to a spreadsheet where data is stored in rows and columns.

Each column represents a data field in the corresponding table. For example, a table's name, address, and telephone data fields are all *columns*.

Each column, in turn, is defined to contain a certain type of data. Therefore, a column designed to store numbers would be defined as containing numerical data.

64.4 Database Rows

Each new record saved to a table is stored in a row. Each row, in turn, consists of the columns of data associated with the saved record.

Once again, consider the spreadsheet analogy described earlier in this chapter. Each entry in a customer table is equivalent to a row in a spreadsheet, and each column contains the data for each customer (name, address, telephone, etc.). When a new customer is added to the table, a new row is created, and the data for that customer is stored in the corresponding columns of the new row.

Rows are also sometimes referred to as *records* or *entries,* and these terms can generally be used interchangeably.

64.5 Introducing Primary Keys

Each database table should contain one or more columns that can be used to identify each row in the table uniquely. This is known in database terminology as the *Primary Key*. For example, a table may use a bank account number column as the primary key. Alternatively, a customer table may use the customer's social security number as the primary key.

Primary keys allow the database management system to uniquely identify a specific row in a table. Without a primary key, retrieving or deleting a specific row in a table would not be possible because there can be no certainty that the correct row has been selected. For example, suppose a table existed where the customer's last name had been defined as the primary key. Imagine the problem if more than one customer named "Smith" were recorded in the database. Without some guaranteed way to identify a specific row uniquely, ensuring the correct data was being accessed at any given time would be impossible.

Primary keys can comprise a single column or multiple columns in a table. To qualify as a single column primary key, no two rows can contain matching primary key values. When using multiple columns to construct a primary key, individual column values do not need to be unique, but all the columns' values combined must be unique.

64.6 What is SQLite?

SQLite is an embedded, relational database management system (RDBMS). Most relational databases (Oracle, SQL Server, and MySQL being prime examples) are standalone server processes that run independently and cooperate with applications requiring database access. SQLite is referred to as *embedded* because it is provided in the form of a library that is linked into applications. As such, there is no standalone database server running in the background. All database operations are handled internally within the application through calls to functions in the SQLite library.

The developers of SQLite have placed the technology into the public domain with the result that it is now a widely deployed database solution.

The developers of SQLite have placed the technology into the public domain with the result that it is now a widely deployed database solution.

SQLite is written in the C programming language, so the Android SDK provides a Java-based "wrapper" around the underlying database interface. This consists of classes that may be utilized within an application's Java or Kotlin code to create and manage SQLite-based databases.

For additional information about SQLite, refer to *https://www.sqlite.org*.

64.7 Structured Query Language (SQL)

Data is accessed in SQLite databases using a high-level language known as Structured Query Language. This is usually abbreviated to SQL and pronounced *sequel*. SQL is a standard language used by most relational database management systems. SQLite conforms mostly to the SQL-92 standard.

SQL is a straightforward and easy-to-use language designed specifically to enable the reading and writing of database data. Because SQL contains a small set of keywords, it can be learned quickly. In addition, SQL syntax is more or less identical between most DBMS implementations, so having learned SQL for one system, your skills will likely transfer to other database management systems.

While some basic SQL statements will be used within this chapter, a detailed overview of SQL is beyond the scope of this book. However, many other resources provide a far better overview of SQL than we could ever hope to provide in a single chapter here.

64.8 Trying SQLite on an Android Virtual Device (AVD)

For readers unfamiliar with databases and SQLite, diving right into creating an Android application that uses SQLite may seem intimidating. Fortunately, Android is shipped with SQLite pre-installed, including an interactive environment for issuing SQL commands from within an adb shell session connected to a running Android AVD emulator instance. This is a useful way to learn about SQLite and SQL and an invaluable tool for identifying problems with databases created by applications running in an emulator.

To launch an interactive SQLite session, begin by running an AVD session. This can be achieved within Android Studio by launching the Android Virtual Device Manager (*Tools -> AVD Manager*), selecting a previously configured AVD, and clicking on the start button.

Once the AVD is up and running, open a Terminal or Command-Prompt window and connect to the emulator using the *adb* command-line tool as follows (note that the –e flag directs the tool to look for an emulator with which to connect, rather than a physical device):

```
adb –e shell
```

Once connected, the shell environment will provide a command prompt at which commands may be entered. Begin by obtaining superuser privileges using the *su* command:

```
Generic_x86:/ su
root@android:/ #
```

If a message indicates that superuser privileges are not allowed, the AVD instance likely includes Google Play support. To resolve this, create a new AVD and, on the “Choose a device definition” screen, select a device that does not have a marker in the “Play Store” column.

The data in SQLite databases are stored in database files on the file system of the Android device on which the application is running. By default, the file system path for these database files is as follows:

```
/data/data/<package name>/databases/<database filename>.db
```

For example, if an application with the package name *com.example.MyDBApp* creates a database named *mydatabase.db*, the path to the file on the device would read as follows:

```
/data/data/com.example.MyDBApp/databases/mydatabase.db
```

For this exercise, therefore, change directory to /data/data within the adb shell and create a sub-directory hierarchy suitable for some SQLite experimentation:

```
cd /data/data
mkdir com.example.dbexample
cd com.example.dbexample
mkdir databases
cd databases
```

With a suitable location created for the database file, launch the interactive SQLite tool as follows:

```
root@android:/data/data/databases # sqlite3 ./mydatabase.db
```

```
sqlite3 ./mydatabase.db
SQLite version 3.8.10.2 2015-05-20 18:17:19
Enter ".help" for usage hints.
sqlite>
```

At the *sqlite>* prompt, commands may be entered to perform tasks such as creating tables and inserting and retrieving data. For example, to create a new table in our database with fields to hold ID, name, address, and phone number fields, the following statement is required:

```
create table contacts (_id integer primary key autoincrement, name text, address
text, phone text);
```

Note that each row in a table should have a *primary key* that is unique to that row. In the above example, we have designated the ID field as the primary key, declared it as being of type *integer*, and asked SQLite to increment the number automatically each time a row is added. This is a common way to ensure that each row has a unique primary key. On most other platforms, the primary key's name choice is arbitrary. In the case of Android, however, the key must be named *_id* for the database to be fully accessible using all Android database-related classes. The remaining fields are each declared as being of type *text*.

To list the tables in the currently selected database, use the *.tables* statement:

```
sqlite> .tables
contacts
```

To insert records into the table:

```
sqlite> insert into contacts (name, address, phone) values ("Bill Smith", "123
Main Street, California", "123-555-2323");
sqlite> insert into contacts (name, address, phone) values ("Mike Parks", "10
Upping Street, Idaho", "444-444-1212");
```

To retrieve all rows from a table:

```
sqlite> select * from contacts;
1|Bill Smith|123 Main Street, California|123-555-2323
2|Mike Parks|10 Upping Street, Idaho|444-444-1212
```

To extract a row that meets specific criteria:

```
sqlite> select * from contacts where name="Mike Parks";
2|Mike Parks|10 Upping Street, Idaho|444-444-1212
```

To exit from the sqlite3 interactive environment:

```
sqlite> .exit
```

When running an Android application in the emulator environment, any database files will be created on the emulator's file system using the previously discussed path convention. This has the advantage that you can connect with adb, navigate to the location of the database file, load it into the sqlite3 interactive tool, and perform tasks on the data to identify possible problems occurring in the application code.

It is also important to note that while connecting with an adb shell to a physical Android device is possible, the shell is not granted sufficient privileges by default to create and manage SQLite databases. Therefore, database problem debugging is best performed using an AVD session.

64.9 The Android Room Persistence Library

As previously mentioned, SQLite is written in the C programming language, while Android applications are primarily developed using Java or Kotlin. To bridge this "language gap" in the past, the Android SDK included a set of classes that provide a layer on top of the SQLite database management system. Although available in the SDK, use of these classes still involved writing a considerable amount of code and did not take advantage of the new architecture guidelines and features such as LiveData and lifecycle management. The Android Jetpack Architecture Components include the Room persistent library to address these shortcomings. This library provides a high-level interface on top of the SQLite database system, making it easy to store data locally on Android devices with minimal coding while also conforming to the recommendations for modern application architecture.

The next few chapters will provide an overview and tutorial on SQLite database management using the Room persistence library.

64.10 Summary

SQLite is a lightweight, embedded relational database management system included in the Android framework and provides a mechanism for implementing organized persistent data storage for Android applications. When combined with the Room persistence library, Android provides a modern way to implement data storage from within an Android app.

This chapter provided an overview of databases in general and SQLite in particular within the context of Android application development. The next chapters will provide an overview of the Room persistence library, after which we will work through the creation of an example application.

Chapter 65

65. The Android Room Persistence Library

Included with the Android Architecture Components, the Room persistence library is designed to make it easier to add database storage support to Android apps in a way consistent with the Android architecture guidelines. With the basics of SQLite databases covered in the previous chapter, this chapter will explore the basic concepts behind Room-based database management, the key elements that work together to implement Room support within an Android app, and how these are implemented in terms of architecture and coding. Having covered these topics, the next two chapters will put this theory into practice with an example Room database project.

65.1 Revisiting Modern App Architecture

The chapter entitled *"Modern Android App Architecture with Jetpack"* introduced the concept of modern app architecture and stressed the importance of separating different areas of responsibility within an app. The diagram illustrated in Figure 65-1 outlines the recommended architecture for a typical Android app:

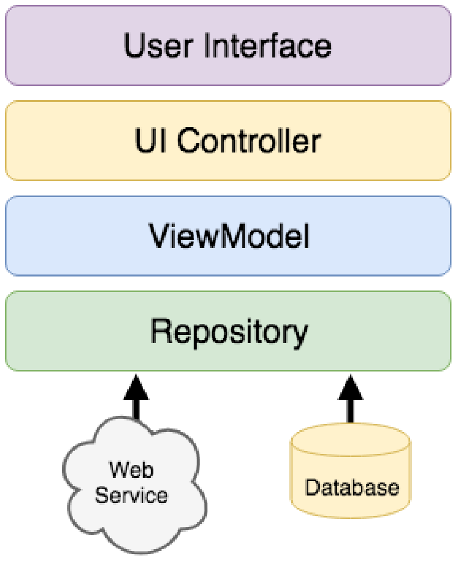


Figure 65-1

With the top three levels of this architecture covered in some detail in earlier chapters of this book, it is time to explore the repository and database architecture levels in the context of the Room persistence library.

65.2 Key Elements of Room Database Persistence

Before going into greater detail later in the chapter, it is first worth summarizing the key elements involved in working with SQLite databases using the Room persistence library:

65.2.1 Repository

As previously discussed, the repository module contains all of the code necessary for directly handling all data sources used by the app. This avoids the need for the UI controller and ViewModel to contain code directly accessing sources such as databases or web services.

65.2.2 Room Database

The room database object provides the interface to the underlying SQLite database. It also provides the repository with access to the Data Access Object (DAO). An app should only have one room database instance, which may be used to access multiple database tables.

65.2.3 Data Access Object (DAO)

The DAO contains the SQL statements required by the repository to insert, retrieve and delete data within the SQLite database. These SQL statements are mapped to methods which are then called from within the repository to execute the corresponding query.

65.2.4 Entities

An entity is a class that defines the schema for a table within the database, defines the table name, column names, and data types, and identifies which column is to be the primary key. In addition to declaring the table schema, entity classes contain getter and setter methods that provide access to these data fields. The data returned to the repository by the DAO in response to the SQL query method calls will take the form of instances of these entity classes. The getter methods will then be called to extract the data from the entity object. Similarly, when the repository needs to write new records to the database, it will create an entity instance, configure values on the object via setter calls, then call insert methods declared in the DAO, passing through entity instances to be saved.

65.2.5 SQLite Database

The SQLite database is responsible for storing and providing access to the data. The app code, including the repository, should never directly access this underlying database. All database operations are performed using a combination of the room database, DAOs, and entities.

The architecture diagram in Figure 65-2 illustrates how these different elements interact to provide Room-based database storage within an Android app:

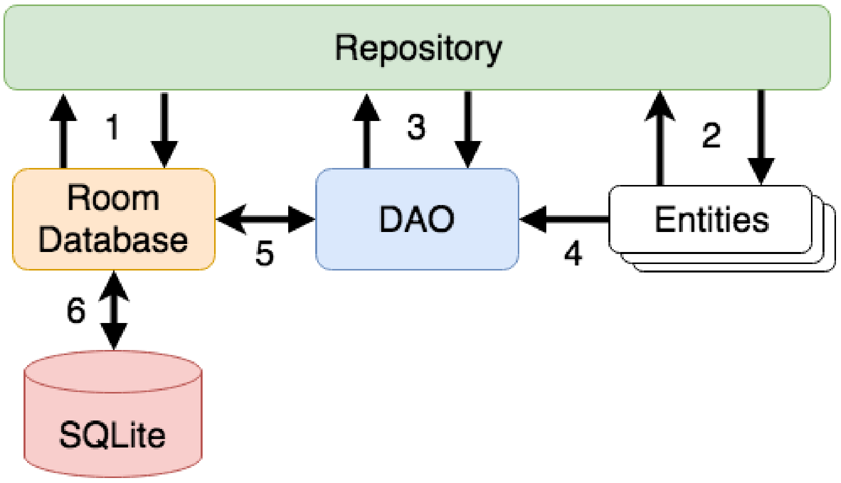


Figure 65-2

The numbered connections in the above architecture diagram can be summarized as follows:

1. The repository interacts with the Room Database to get a database instance which, in turn, is used to obtain references to DAO instances.
2. The repository creates entity instances and configures them with data before passing them to the DAO for use in search and insertion operations.
3. The repository calls methods on the DAO passing through entities to be inserted into the database and receives entity instances back in response to search queries.
4. When a DAO has results to return to the repository, it packages them into entity objects.
5. The DAO interacts with the Room Database to initiate database operations and handle results.
6. The Room Database handles all low-level interactions with the underlying SQLite database, submitting queries and receiving results.

With a basic outline of the key elements of database access using the Room persistent library covered, it is time to explore entities, DAOs, room databases, and repositories in more detail.

65.3 Understanding Entities

Each database table will have associated with it an entity class. This class defines the schema for the table and takes the form of a standard Java class interspersed with some special Room annotations. An example Java class declaring the data to be stored within a database table might read as follows:

```
public class Customer {

    private int id;
    private String name;
    private string address;

    public Customer(String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public int getId() {
        return this.id;
    }
    public String getName() {
        return this.name;
    }

    public int getAddress() {
        return this.address;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAddress(int quantity) {
        this.address = address;
    }
}
```

As currently implemented, the above code declares a basic Java class containing several variables representing database table fields and a collection of getter and setter methods. This class, however, is not yet an entity. To make this class into an entity and to make it accessible within SQL statements, some Room annotations need to be added as follows:

```
@Entity(tableName = "customers")
public class Customer {

    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "customerId")
    private int id;

    @ColumnInfo(name = "customerName")
    private String name;

    private String address;

    public Customer(String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public int getId() {
        return this.id;
    }

    public String getName() {
        return this.name;
    }

    public String getAddress() {
        return this.address;
    }
    public void setId(@NonNull int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAddress(int quantity) {
        this.address = address;
    }
}
```

The above annotations begin by declaring that the class represents an entity and assigns a table name of "customers". This is the name by which the table will be referenced in the DAO SQL statements:

```
@Entity(tableName = "customers")
```

Every database table needs a column to act as the primary key. In this case, the customer id is declared as the primary key. Annotations have also been added to assign a column name to be referenced in SQL queries and to indicate that the field cannot be used to store null values. Finally, the id value is configured to be auto-generated. This means the system automatically generates the id assigned to new records to avoid duplicate keys:

```
@PrimaryKey(autoGenerate = true)
@NonNull
@ColumnInfo(name = "customerId")
private int id;
```

A column name is also assigned to the customer name field. Note, however, that no column name was assigned to the address field. This means that the address data will still be stored within the database but is not required to be referenced in SQL statements. If a field within an entity is not required to be stored within a database, use the @Ignore annotation:

```
@Ignore
private String myString;
```

Finally, the setter method for the id variable is modified to prevent attempts to assign a null value:

```
public void setId(@NonNull int id) {
    this.id = id;
}
```

Annotations may also be included within an entity class to establish relationships with other entities using a relational database concept referred to as *foreign keys*. Foreign keys allow a table to reference the primary key in another table. For example, a relationship could be established between an entity named Purchase and our existing Customer entity as follows:

```
@Entity(foreignKeys = {@ForeignKey(entity = Customer.class,
        parentColumns = "customerId",
        childColumns = "buyerId",
        onDelete = ForeignKey.CASCADE,
        onUpdate = ForeignKey.RESTRICT})
public class Purchase {
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "purchaseId")
    private int purchaseId;

    @ColumnInfo(name = "buyerId")
    private int buyerId;

}
```

Note that the foreign key declaration also specifies the action to be taken when a parent record is deleted or updated. Available options are CASCADE, NO_ACTION, RESTRICT, SET_DEFAULT, and SET_NULL.

65.4 Data Access Objects

A Data Access Object allows access to the data stored within a SQLite database. A DAO is declared as a standard Java interface with additional annotations that map specific SQL statements to methods that the repository may then call.

The first step is to create the interface and declare it as a DAO using the @Dao annotation:

```
@Dao
public interface CustomerDao {
}
```

Next, entries are added consisting of SQL statements and corresponding method names. The following declaration, for example, allows all of the rows in the customers table to be retrieved via a call to a method named *getAllCustomers()*:

```
@Dao
public interface CustomerDao {
    @Query("SELECT * FROM customers")
    LiveData<List<Customer>> getAllCustomers();
}
```

The *getAllCustomers()* method returns a List object containing a Customer entity object for each record retrieved from the database table. The DAO is also using LiveData so that the repository can observe changes to the database.

Arguments may also be passed into the methods and referenced within the corresponding SQL statements. Consider the following DAO declaration, which searches for database records matching a customer's name (note that the column name referenced in the WHERE condition is the name assigned to the column in the entity class):

```
@Query("SELECT * FROM customers WHERE name = :customerName")
List<Customer> findCustomer(String customerName);
```

In this example, the method is passed a string value which is, in turn, included within an SQL statement by prefixing the variable name with a colon (:).

A basic insertion operation can be declared as follows using the @Insert *convenience annotation*:

```
@Insert
void addCustomer(Customer customer);
```

This is referred to as a convenience annotation because the Room persistence library can infer that the Customer entity passed to the *addCustomer()* method is to be inserted into the database without the need for the SQL insert statement to be provided. Multiple database records may also be inserted in a single transaction as follows:

```
@Insert
public void insertCustomers(Customer... customers);
```

The following DAO declaration deletes all records matching the provided customer name:

```
@Query("DELETE FROM customers WHERE name = :name")
void deleteCustomer(String name);
```

As an alternative to using the @Query annotation to perform deletions, the @Delete convenience annotation may also be used. In the following example, all of the Customer records that match the set of entities passed to the *deleteCustomers()* method will be deleted from the database:

```
@Delete
public void deleteCustomers(Customer... customers);
```

The @Update convenience annotation provides similar behavior when updating records:

```
@Update
public void updateCustomers(Customer... customers);
```

The DAO methods for these types of database operations may also be declared to return an int value indicating the number of rows affected by the transaction, for example:

```
@Delete
public int deleteCustomers(Customer... customers);
```

65.5 The Room Database

The Room database class is created by extending the RoomDatabase class and acts as a layer on top of the actual SQLite database embedded into the Android operating system. The class is responsible for creating and returning a new room database instance and providing access to the database's associated DAO instances.

The Room persistence library provides a database builder for creating database instances. Each Android app should only have one room database instance, so it is best to implement defensive code within the class to prevent more than one instance from being created.

An example Room Database implementation for use with the example customer table is outlined in the following code listing:

```
import android.content.Context;
import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;

@Database(entities = {Customer.class}, version = 1)
public class CustomerRoomDatabase extends RoomDatabase {

    public abstract CustomerDao customerDao();

    private static CustomerRoomDatabase INSTANCE;

    static CustomerRoomDatabase getDatabase(final Context context) {
        if (INSTANCE == null) {
            synchronized (CustomerRoomDatabase.class) {
                if (INSTANCE == null) {
                    INSTANCE = Room.databaseBuilder(
                          context.getApplicationContext(),
                            CustomerRoomDatabase.class, "customer_database")
                            .build();
                }
            }
        }
        return INSTANCE;
    }
}
```

Important areas to note in the above example are the annotation above the class declaration declaring the entities with which the database is to work, the code to check that an instance of the class has not already been created and the assignment of the name "customer_database" to the instance.

65.6 The Repository

The repository is responsible for getting a Room Database instance, using that instance to access associated DAOs, and then making calls to DAO methods to perform database operations. A typical constructor for a repository designed to work with a Room Database might read as follows:

```
public class CustomerRepository {

    private CustomerDao customerDao;
    private CustomerRoomDatabase db;

    public CustomerRepository(Application application) {
        db = CustomerRoomDatabase.getDatabase(application);
        customerDao = db.customerDao();
    }
.
.
}
```

Once the repository can access the DAO, it can call the data access methods. The following code, for example, calls the *getAllCustomers()* DAO method:

```
private LiveData<List<Customer>> allCustomers;
allCustomers = customerDao.getAllCustomers();
```

When calling DAO methods, it is important to note that unless the method returns a LiveData instance (which automatically runs queries on a separate thread), the operation cannot be performed on the app's main thread. Attempting to do so will cause the app to crash with the following diagnostic output:

```
Cannot access database on the main thread since it may potentially lock the UI
for a long period of time
```

Since some database transactions may take a longer time to complete, running the operations on a separate thread avoids the app appearing to lock up. As will be demonstrated in the chapter entitled *"An Android Room Database and Repository Tutorial"*, this problem can be easily resolved by making use of Java threads (for more information or a reminder of how to use threads, refer back to the chapter entitled *"An Overview of Java Threads,* *Handlers and Executors*").

65.7 In-Memory Databases

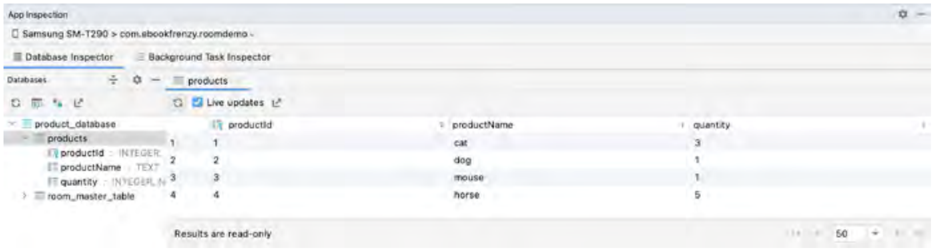
The examples outlined in this chapter use a SQLite database that exists as a database file on the persistent storage of an Android device. This ensures that the data persists even after the app process is terminated.

The Room database persistence library also supports *in-memory* databases. These databases reside entirely in memory and are lost when the app terminates. The only change necessary to work with an in-memory database is to call the *Room.inMemoryDatabaseBuilder()* method of the Room Database class instead of *Room.databaseBuilder()*. The following code shows the difference between the method calls (note that the in-memory database does not require a database name):

```
// Create a file storage based database
INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                            CustomerRoomDatabase.class, "customer_database")
                            .build();

// Create an in-memory database
INSTANCE = Room.inMemoryDatabaseBuilder(context.getApplicationContext(),
                            CustomerRoomDatabase.class)
                            .build();
```

65.8 Database Inspector

Android Studio includes a Database Inspector tool window which allows the Room databases associated with running apps to be viewed, searched, and modified, as shown in Figure 65-3:



	productId	productName	quantity
1	1	cat	3
2	2	dog	1
3	3	mouse	1
4	4	horse	5

Figure 65-3

The Database Inspector will be covered in the chapter *"An Android Room Database and Repository Tutorial"*.

65.9 Summary

The Android Room persistence library is bundled with the Android Architecture Components and acts as an abstract layer above the lower-level SQLite database. The library is designed to make it easier to work with databases while conforming to the Android architecture guidelines. This chapter has introduced the elements that interact to build Room-based database storage into Android app projects, including entities, repositories, data access objects, annotations, and Room Database instances.

With the basics of SQLite and the Room architecture component covered, the next step is to create an example app that puts this theory into practice. Since the user interface for the example application will require a forms-based layout, the next chapter, entitled *"An Android TableLayout and TableRow Tutorial"*, will detour slightly from the core topic by introducing the basics of the TableLayout and TableRow views.

Chapter 68

68. Accessing Cloud Storage using the Android Storage Access Framework

Recent years have seen the wide adoption of remote storage services (otherwise known as "cloud storage") to store user files and data. Driving this growth are two key factors. One is that most mobile devices now provide continuous, high speed internet connectivity, thereby making the transfer of data fast and affordable. The second factor is that, relative to traditional computer systems (such as desktops and laptops) these mobile devices are constrained in terms of internal storage resources. A high specification Android tablet today, for example, typically comes with 128Gb of storage capacity. When compared with a mid-range laptop system with a 750Gb disk drive, the need for the seamless remote storage of files is a key requirement for many mobile applications today.

In recognition of this fact, Google introduced the Storage Access Framework as part of the Android 4.4 SDK. This chapter will provide a high level overview of the storage access framework in preparation for the more detail oriented tutorial contained in the next chapter, entitled *"An Android Storage Access Framework Example"*.

68.1 The Storage Access Framework

From the perspective of the user, the Storage Access Framework provides an intuitive user interface that allows the user to browse, select, delete and create files hosted by storage services (also referred to as *document providers*) from within Android applications. Using this browsing interface (also referred to as the *picker*), users can, for example, browse through the files (such as documents, audio, images and videos) hosted by their chosen document providers. Figure 68-1, for example, shows the picker user interface displaying a collection of files hosted by a document provider service:

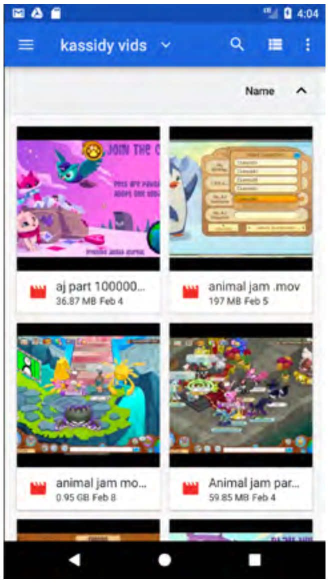


Figure 68-1

Document providers can range from cloud-based services to local document providers running on the same device as the client application. At the time of writing, the most prominent document providers compatible with the Storage Access Framework are Box and, unsurprisingly, Google Drive. It is highly likely that other cloud storage providers and application developers will soon also provide services that conform to the Android Storage Access Framework.

In addition to cloud based document providers the picker also provides access to internal storage on the device, providing a range of file storage options to the application user.

Through a set of Intents, Android application developers can incorporate these storage capabilities into applications with just a few lines of code. A particularly compelling aspect of the Storage Access Framework from the point of view of the developer is that the underlying document provider selected by the user is completely transparent to the application. Once the storage functionality has been implemented using the framework within an application, it will work with all document providers without any code modifications.

68.2 Working with the Storage Access Framework

Android includes a set of Intents designed to integrate the features of the Storage Access Framework into Android applications. These intents display the Storage Access Framework picker user interface to the user and return the results of the interaction to the application via a call to the *onActivityResult()* method of the activity that launched the intent. When the *onActivityResult()* method is called, it is passed the Uri of the selected file together with a value indicating the success or otherwise of the operation.

The Storage Access Framework intents can be summarized as follows:

- **ACTION_OPEN_DOCUMENT** – Provides the user with access to the picker user interface so that files may be selected from the document providers configured on the device. Selected files are passed back to the application in the form of Uri objects.

- **ACTION_CREATE_DOCUMENT** – Allows the user to select a document provider, a location on that provider's storage and a file name for a new file. Once selected, the file is created by the Storage Access Framework and the Uri of that file returned to the application for further processing.

68.3 Filtering Picker File Listings

The files listed within the picker user interface when an intent is started may be filtered using a variety of options. Consider, for example, the following code to configure an ACTION_OPEN_DOCUMENT intent:

```
Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
```

When launched, the above intent will cause the picker user interface to be displayed, allowing the user to browse and select any files hosted by available document providers. Once a file has been selected by the user, a reference to that file will be provided to the application in the form of a Uri object. The application can then open the file using the *openFileDescriptor(Uri, String)* method. There is some risk, however, that not all files listed by a document provider can be opened in this way. The exclusion of such files within the picker can be achieved by modifying the intent using the *CATEGORY_OPENABLE* option. For example:

```
Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
intent.addCategory(Intent.CATEGORY_OPENABLE);
```

When the intent is now displayed, files which cannot be opened using the *openFileDescriptor()* method will be listed but not selectable by the user.

Another useful approach to filtering allows the files available for selection to be restricted by file type. This involves specifying the types of the files the application is able to handle. An image editing application might, for example, only want to provide the user with the option of selecting image files from the document providers.

This is achieved by configuring the intent object with the MIME types of the files that are to be selectable by the user. The following code, for example, specifies that only image files are suitable for selection in the picker:

```
intent.addCategory(Intent.CATEGORY_OPENABLE);
intent.setType("image/*");
```

This could be further refined to limit selection to JPEG images:

```
intent.setType("image/jpeg");
```

Alternatively, an audio player app might only be able to handle audio files:

```
intent.setType("audio/*");
```

The audio app might be limited even further in only supporting the playback of MP4 based audio files:

```
intent.setType("audio/mp4");
```

A wide range of MIME type settings are available for use when working with the Storage Access Framework, the more common of which can be found listed online at:

https://en.wikipedia.org/wiki/Internet_media_type#List_of_common_media_types

68.4 Handling Intent Results

Since we are launching intents which will return a result, the *registerForActivityResult()* method (covered in the chapter entitled *"Android Explicit Intents – A Worked Example"*) needs to be used to create a launcher and declare a lambda to handle the returned data, for example:

```
ActivityResultLauncher<Intent> startOpenForResult = registerForActivityResult(
        new ActivityResultContracts.StartActivityForResult(),
        new ActivityResultCallback<ActivityResult>() {
            @Override
            public void onActivityResult(ActivityResult result) {
                if (result.getResultCode() == Activity.RESULT_OK) {
                    Intent data = result.getData();
                    // Read file content from Uri here
                }
            }
        });
```

Once declared, the intent can be configured and launched as follows:

```
Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
intent.addCategory(Intent.CATEGORY_OPENABLE);
intent.setType("text/plain");
startOpenForResult.launch(intent);
```

68.5 Reading the Content of a File

The exact steps required to read the content of a file hosted by a document provider will depend to a large extent on the type of the file. The steps to read lines from a text file, for example, differ from those for image or audio files.

An image file can be assigned to a Bitmap object by extracting the file descriptor from the Uri object and then decoding the image into a BitmapFactory instance. For example:

```
ParcelFileDescriptor pFileDescriptor =
            getContentResolver().openFileDescriptor(uri, "r");
```

```
FileDescriptor fileDescriptor =
            pFileDescriptor.getFileDescriptor();

Bitmap image = BitmapFactory.decodeFileDescriptor(fileDescriptor);

pFileDescriptor.close();

myImageView.setImageBitmap(image);
```

Note that the file descriptor is opened in "r" mode. This indicates that the file is to be opened for reading. Other options are "w" for write access and "rwt" for read and write access, where existing content in the file is truncated by the new content.

Reading the content of a text file requires slightly more work and the use of an InputStream object. The following code, for example, reads the lines from a text file:

```
InputStream inputStream = getContentResolver().openInputStream(uri);

BufferedReader reader = new BufferedReader(new InputStreamReader(
                          inputStream));
String readline;

while ((readline = reader.readLine()) != null) {
       // Do something with each line in the file
}
inputStream.close();
```

68.6 Writing Content to a File

Writing to an open file hosted by a document provider is similar to reading with the exception that an output stream is used instead of an input stream. The following code, for example, writes text to the output stream of the storage based file referenced by the specified Uri:

```
try{
       ParcelFileDescriptor pFileDescriptor = this.getContentResolver().
                openFileDescriptor(uri, "w");

       FileOutputStream fileOutputStream =
             new FileOutputStream(pFileDescriptor.getFileDescriptor());

       String textContent = "Some sample text";
       fileOutputStream.write(textContent.getBytes());
       fileOutputStream.close();
       pFileDescriptor.close();
} catch (FileNotFoundException e) {
       e.printStackTrace();
} catch (IOException e) {
       e.printStackTrace();
}
```

First, the file descriptor is extracted from the Uri, this time requesting write permission to the target file. The file descriptor is subsequently used to obtain a reference to the file's output stream. The content (in this example, some text) is then written to the output stream before the file descriptor and output stream are closed.

68.7 Deleting a File

Whether a file can be deleted from storage depends on whether or not the file's document provider supports deletion of the file. Assuming deletion is permitted, it may be performed on a designated Uri as follows:

```
if (DocumentsContract.deleteDocument(getContentResolver(), uri))
       // Deletion was successful
else
       // Deletion failed
```

68.8 Gaining Persistent Access to a File

When an application gains access to a file via the Storage Access Framework, the access will remain valid until the Android device on which the application is running is restarted. Persistent access to a specific file can be obtained by "taking" the necessary permissions for the Uri. The following code, for example, persists read and write permissions for the file referenced by the *fileUri* Uri instance:

```
final int takeFlags = intent.getFlags()
            & (Intent.FLAG_GRANT_READ_URI_PERMISSION
            | Intent.FLAG_GRANT_WRITE_URI_PERMISSION);

getContentResolver().takePersistableUriPermission(fileUri, takeFlags);
```

Once the permissions for the file have been taken by the application, and assuming the Uri has been saved by the application, the user should be able to continue accessing the file after a device restart without the user having to reselect the file from the picker interface.

If, at any time, the persistent permissions are no longer required, they can be released via a call to the *releasePersistableUriPermission()* method of the content resolver:

```
final int releaseFlags = intent.getFlags()
            & (Intent.FLAG_GRANT_READ_URI_PERMISSION
            | Intent.FLAG_GRANT_WRITE_URI_PERMISSION);

getContentResolver().releasePersistableUriPermission(fileUri,
                            releaseFlags);
```

68.9 Summary

It is interesting to consider how perceptions of storage have changed in recent years. Once synonymous with high capacity internal hard disk drives, the term "storage" is now just as likely to refer to storage space hosted remotely in the cloud and accessed over an internet connection. This is increasingly the case with the wide adoption of resource constrained, "always-connected" mobile devices with minimal internal storage capacity.

The Android Storage Access Framework provides a simple mechanism for both users and application developers to seamlessly gain access to files stored in the cloud. Through the use of a set of intents and a built-in user interface for selecting document providers and files, comprehensive cloud based storage can now be integrated into Android applications with a minimal amount of coding.

Index