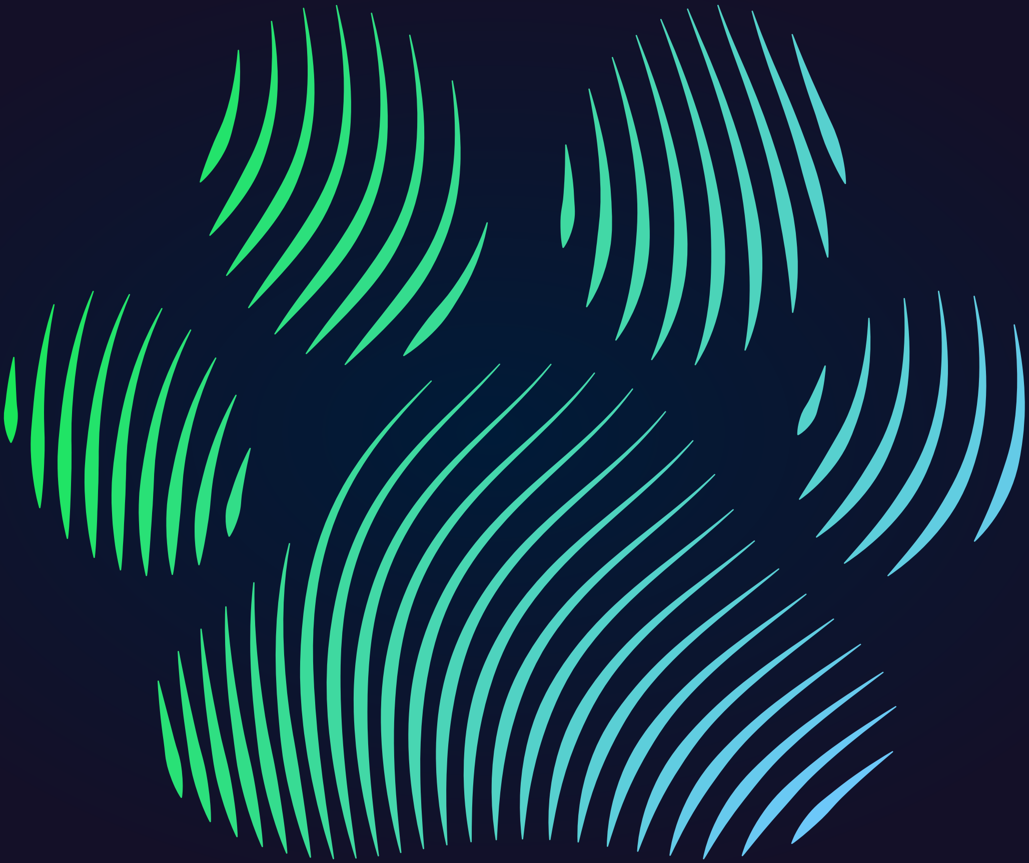
Android Studio Koala Essentials



Java Edition

Payload
publishing

Android Studio Koala Essentials

Java Edition

Android Studio Koala Essentials – Java Edition

ISBN: 978-1-951442-96-5

Rev: 1.0

Payload
publishing

https://www.payloadbooks.com

Table of Contents

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 Koala Feature Drop 2024.1.2 using the Android API 35 SDK (Vanilla Ice Cream), 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 Koala Feature Drop" 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 Koala Feature Drop 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. When prompted to select the components to install, ensure that the *Android Studio* and *Android Virtual Device* options are both 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 complete the installation process.

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:

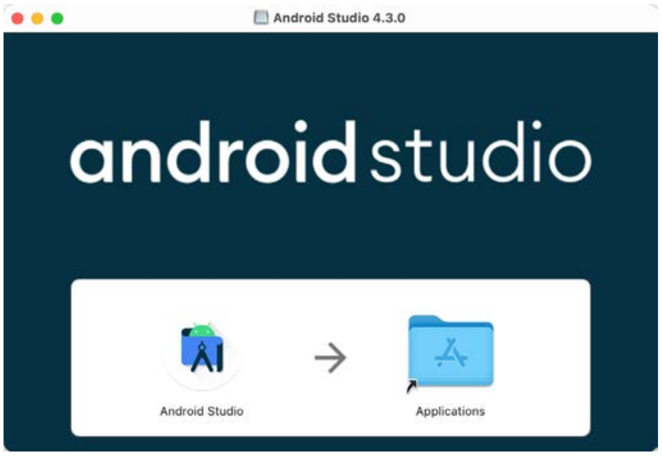


Figure 2-1

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 Installing additional Android SDK packages

When you launch Android Studio, the Welcome to Android Studio screen will appear as shown below:

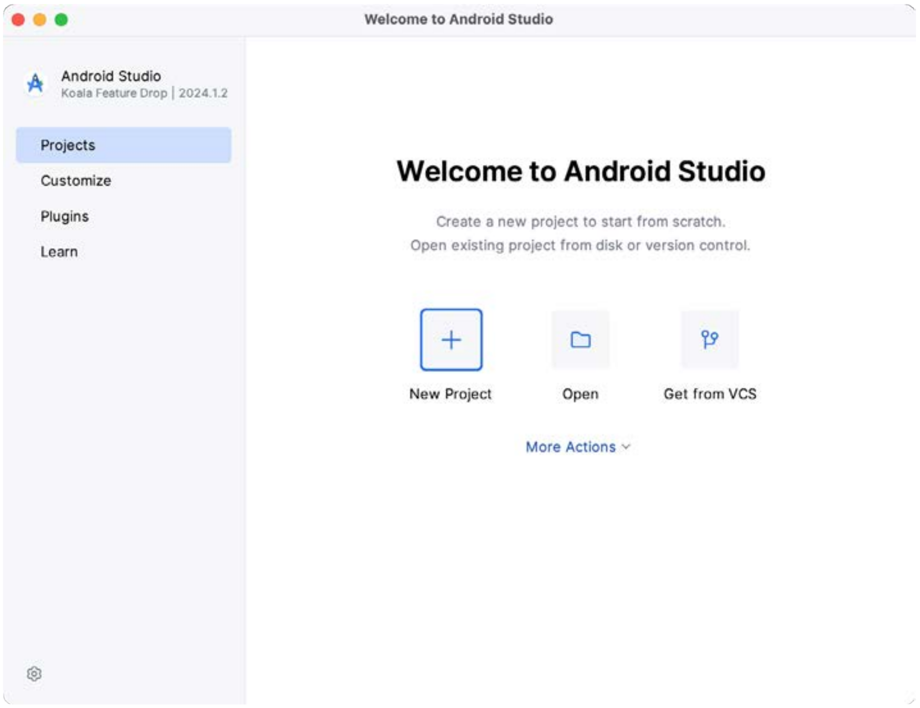
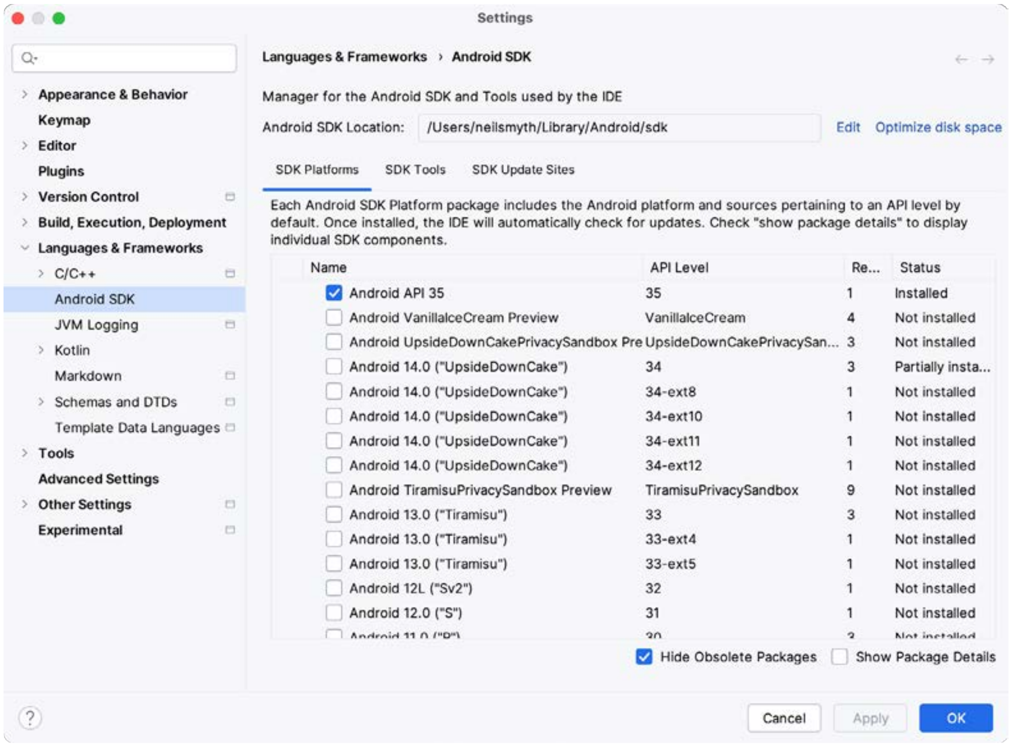

Welcome to Android Studio
Android Studio
Koala Feature Drop | 2024.1.2
Projects
Customize
Plugins
Learn
Welcome to Android Studio
Create a new project to start from scratch.
Open existing project from disk or version control.
New Project
Open
Get from VCS
More Actions

Figure 2-2

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 2-3:



Settings

Languages & Frameworks › Android SDK

Manager for the Android SDK and Tools used by the IDE

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

SDK Platforms   SDK Tools   SDK Update Sites

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

	Name	API Level	Re...	Status
✓	Android API 35	35	1	Installed
☐	Android VanillaIceCream Preview	VanillaIceCream	4	Not installed
☐	Android UpsideDownCakePrivacySandbox Pre	UpsideDownCakePrivacySan...	3	Not installed
☐	Android 14.0 ("UpsideDownCake")	34	3	Partially insta...
☐	Android 14.0 ("UpsideDownCake")	34-ext8	1	Not installed
☐	Android 14.0 ("UpsideDownCake")	34-ext10	1	Not installed
☐	Android 14.0 ("UpsideDownCake")	34-ext11	1	Not installed
☐	Android 14.0 ("UpsideDownCake")	34-ext12	1	Not installed
☐	Android TiramisuPrivacySandbox Preview	TiramisuPrivacySandbox	9	Not installed
☐	Android 13.0 ("Tiramisu")	33	3	Not installed
☐	Android 13.0 ("Tiramisu")	33-ext4	1	Not installed
☐	Android 13.0 ("Tiramisu")	33-ext5	1	Not installed
☐	Android 12L ("Sv2")	32	1	Not installed
☐	Android 12.0 ("S")	31	1	Not installed
☐	Android 11.0 ("R")	30	3	Not installed

✓ Hide Obsolete Packages   ☐ Show Package Details

Figure 2-3

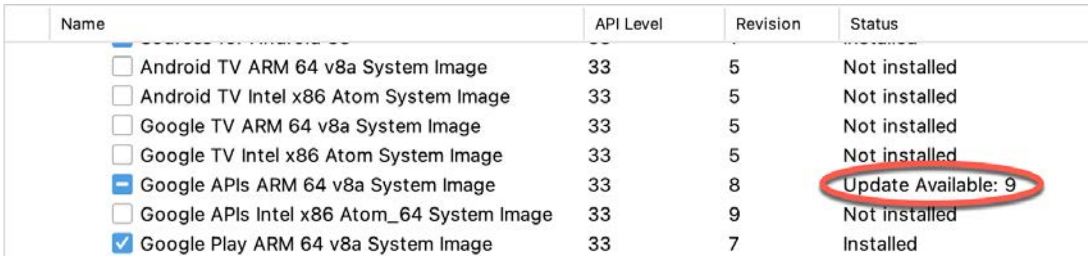
Google pairs each release of Android Studio with a maximum supported Application Programming Interface (API) level of the Android SDK. In the case of Android Studio Koala Feature Drop, this is Android Vanilla Ice Cream (API Level 35). This information can be confirmed using the following link:

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 Vanilla Ice Cream (API Level 35) 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 2-4:

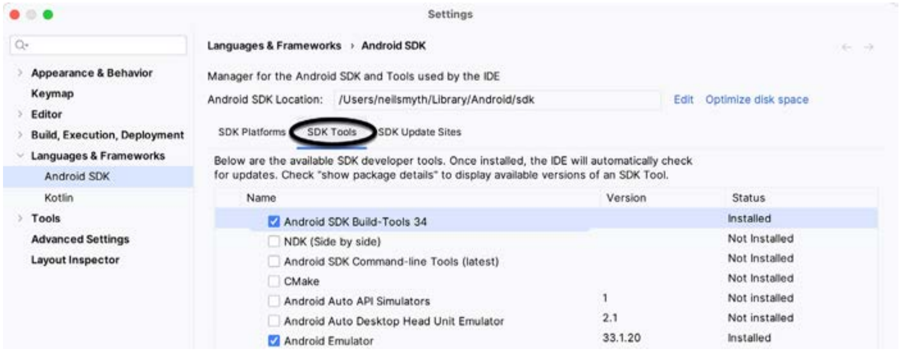


	Name	API Level	Revision	Status
☐	Android TV ARM 64 v8a System Image	33	5	Not installed
☐	Android TV Intel x86 Atom System Image	33	5	Not installed
☐	Google TV ARM 64 v8a System Image	33	5	Not installed
☐	Google TV Intel x86 Atom System Image	33	5	Not installed
⊟	Google APIs ARM 64 v8a System Image	33	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 2-4

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 2-5:



Settings

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

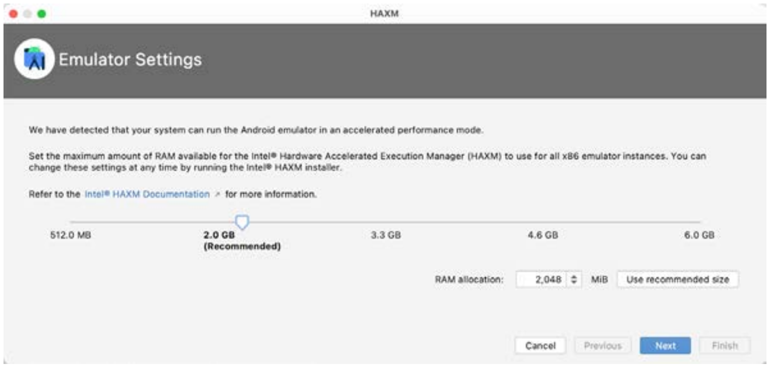
Figure 2-5

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 31-35

*Note that the Intel x86 Emulator Accelerator (HAXM installer) requires an Intel processor with VT-x support enabled. It cannot be installed on Apple silicon-based Macs or AMD-based PCs.

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:



HAXM

Emulator Settings

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

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

Refer to the Intel® HAXM Documentation ↗ for more information.

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

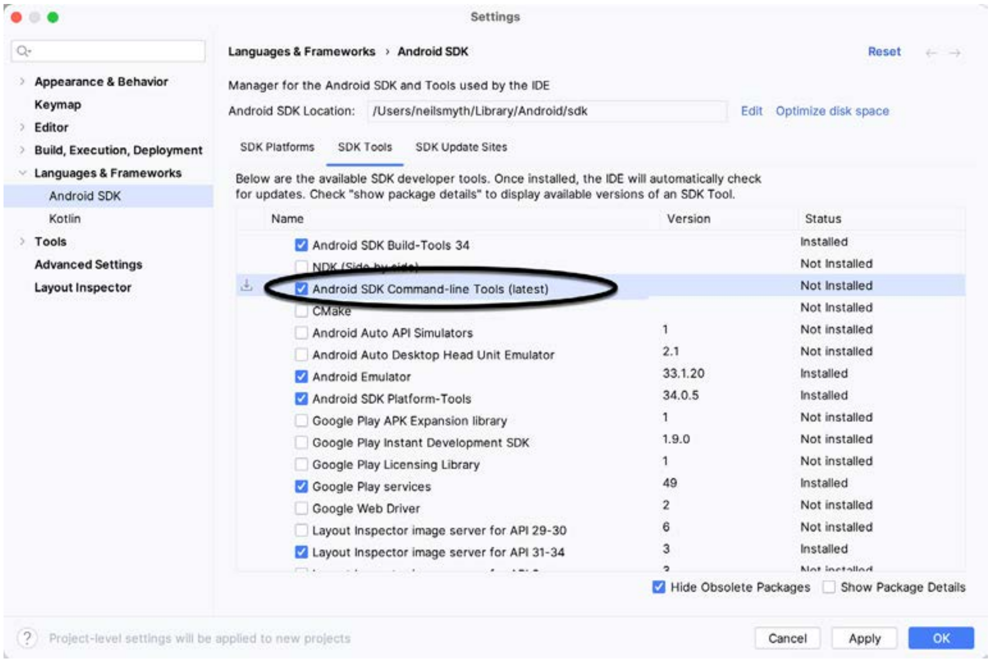
RAM allocation: 2,048 MiB

Figure 2-6

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.

2.5 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 2-7:



Settings

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

✓ Hide Obsolete Packages ☐ Show Package Details

Project-level settings will be applied to new projects

Figure 2-7

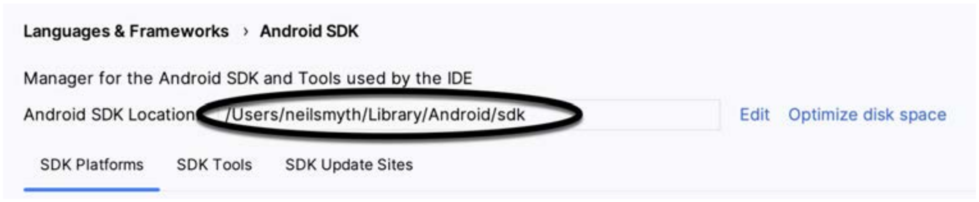
If the command-line tools package is not already installed, enable it and click Apply, followed by OK to complete the installation. When the installation completes, click Finish and close the SDK Manager dialog.

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 2-8:


Languages & Frameworks › Android SDK

Manager for the Android SDK and Tools used by the IDE

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

Figure 2-8

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

2.5.1 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.
```

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.5.5 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
```

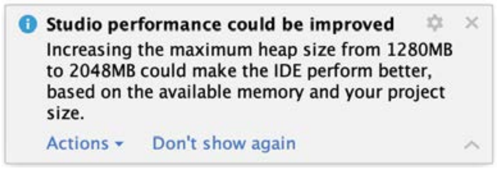
Note that since this is a system directory, it will be necessary to use the *sudo* command when creating the file. For example:

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

2.6 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.

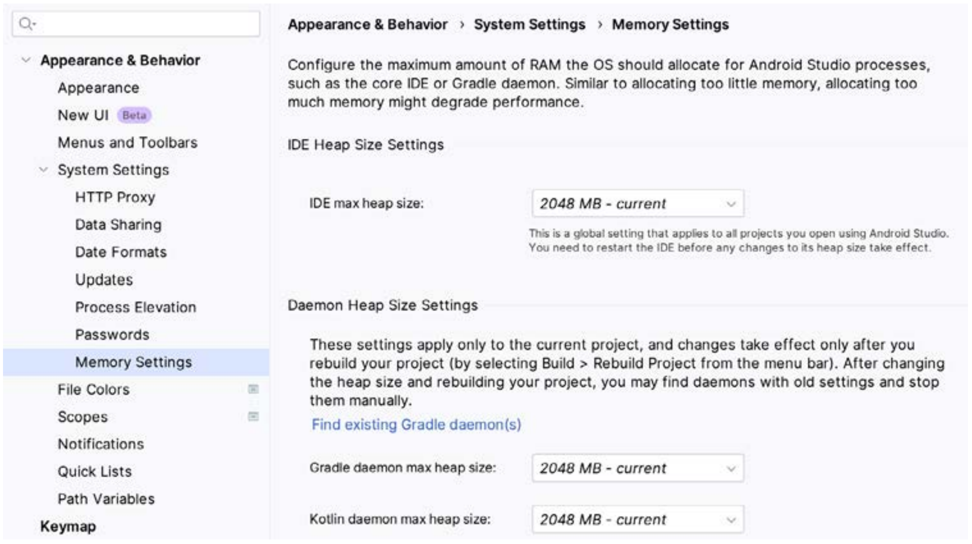
If you are running Android Studio on a system with sufficient unused RAM to increase these values (this feature is only available on 64-bit systems with 5GB or more of RAM) and find that Android Studio performance appears to be degraded, it may be worth experimenting with these memory settings. Android 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 2-9

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 2-10 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 2-10

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.

2.7 Updating Android Studio and the SDK

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

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

2.8 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:

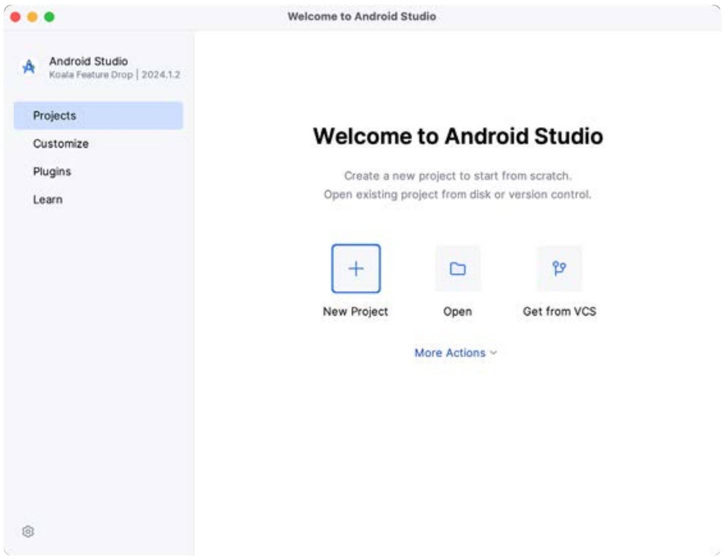

Welcome to Android Studio
Android Studio
Koala Feature Drop | 2024.1.2
Projects
Customize
Plugins
Learn
Welcome to Android Studio
Create a new project to start from scratch.
Open existing project from disk or version control.
New Project
Open
Get from VCS
More Actions

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.

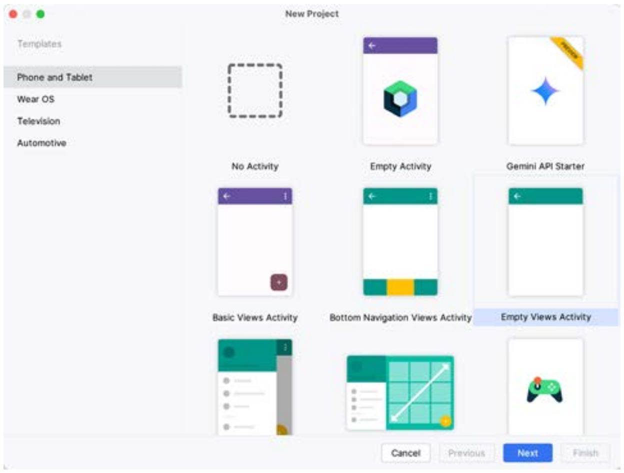


Figure 3-2

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:

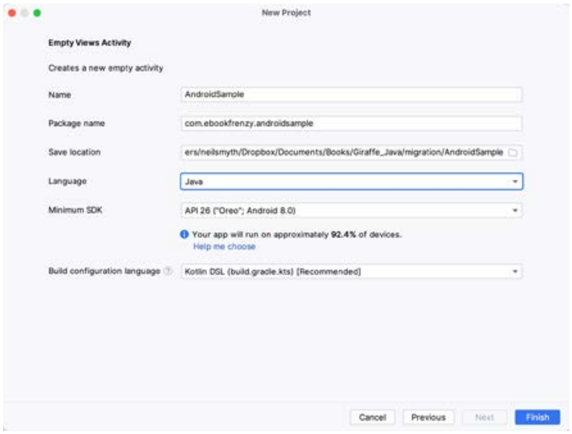
```
com.mycompany.androidsample
```

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:


New Project

Empty Views Activity

Creates a new empty activity

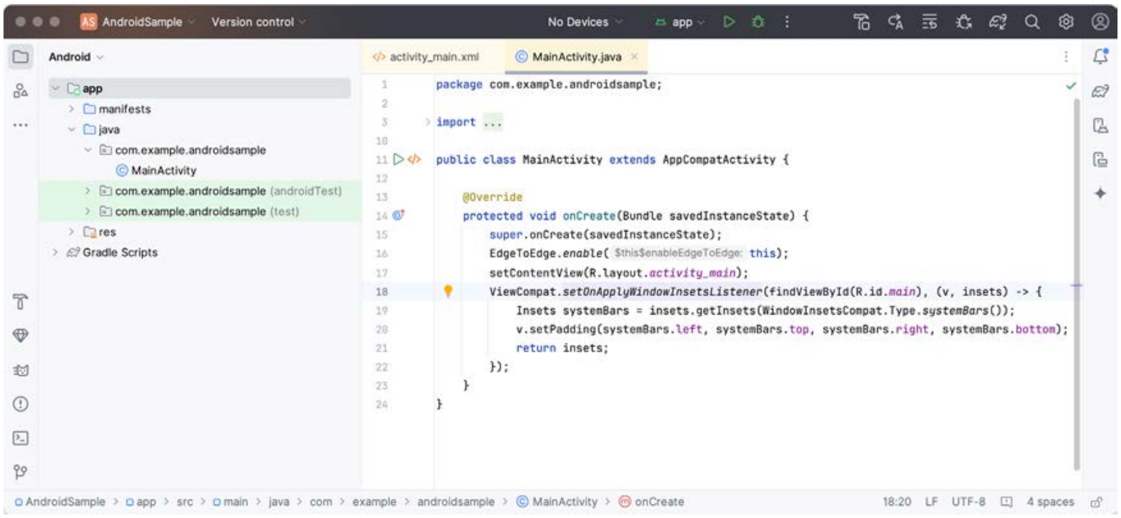
Name	AndroidSample
Package name	com.ebookfrenzy.androidsample
Save location	ers/neilsmyth/Dropbox/Documents/Books/Giraffe_Java/migration/AndroidSample
Language	Java
Minimum SDK	API 26 ("Oreo"; Android 8.0)
	Your app will run on approximately **92.4%** of devices. Help me choose
Build configuration language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

Once the project has been created, the main window will appear containing our AndroidSample project, as illustrated in Figure 3-4 below:


AndroidSample
Version control
No Devices
app
Android
app
manifests
java
com.example.androidsample
MainActivity
com.example.androidsample (androidTest)
com.example.androidsample (test)
res
Gradle Scripts
activity_main.xml
MainActivity.java
AndroidSample > app > src > main > java > com > example > androidsample > MainActivity > onCreate
18:20 LF UTF-8 4 spaces

```
package com.example.androidsample;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        EdgeToEdge.enable( $this$enableEdgeToEdge: this);
        setContentView(R.layout.activity_main);
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main), (v, insets) -> {
            Insets systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars());
            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom);
            return insets;
        });
    }
}
```

Figure 3-4

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 3-5. If the panel is not currently in Android mode, use the menu to switch mode:

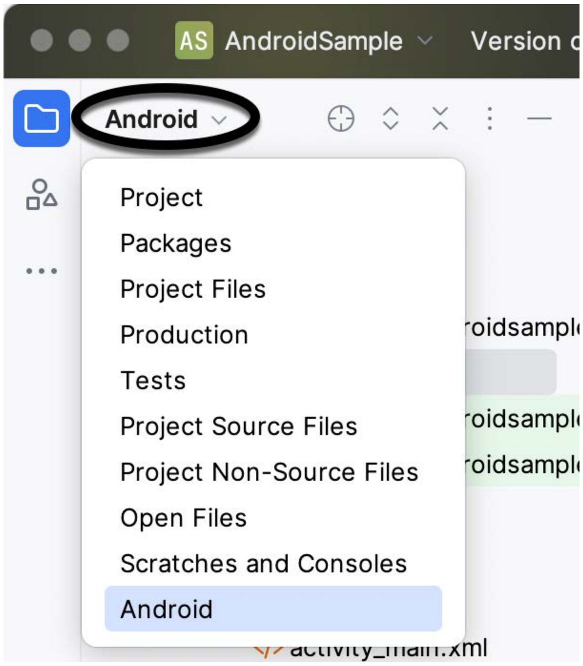


Figure 3-5

3.6 Modifying the User Interface

The user interface design for our activity is stored in a file named *activity_main.xml* which, in turn, is located under *app* -> *res* -> *layout* in the Project 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:

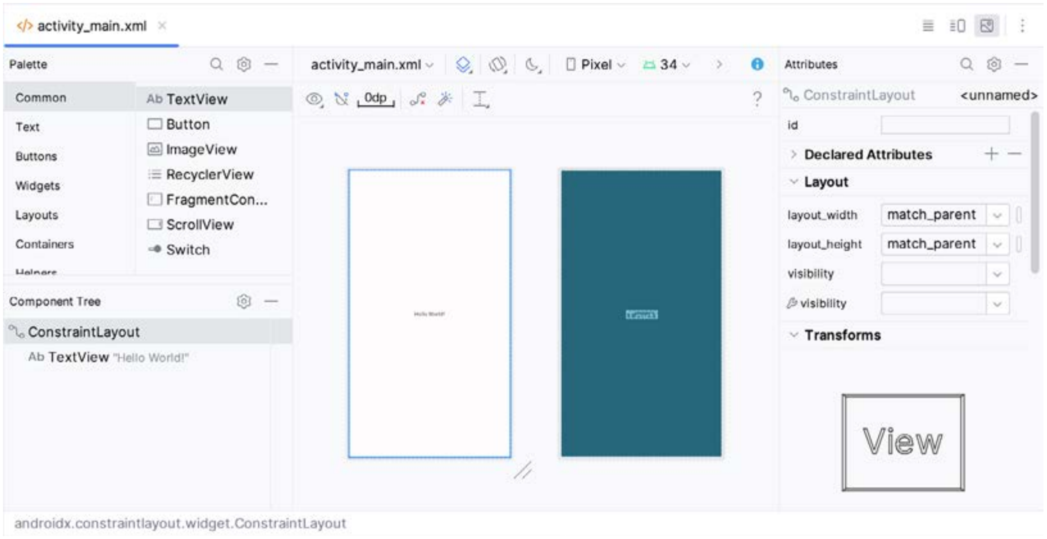


Figure 3-6

In the toolbar across the top of the Layout Editor window is a menu (currently set to *Pixel* in the above figure) which is reflected in the visual representation of the device within the Layout Editor panel. A 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 3-7:

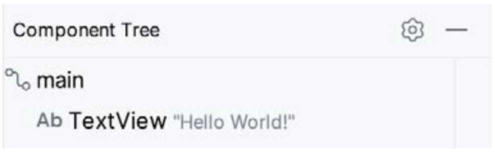


Figure 3-7

As we can see from the component tree hierarchy, the user interface layout consists of a ConstraintLayout parent called *main* 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 3-8). If necessary, re-enable Autoconnect mode by clicking on this button.

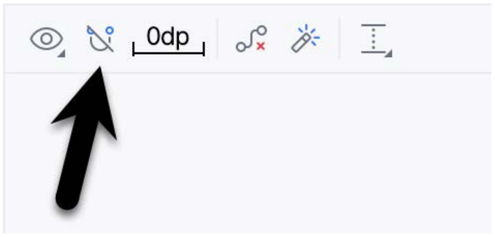


Figure 3-8

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

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

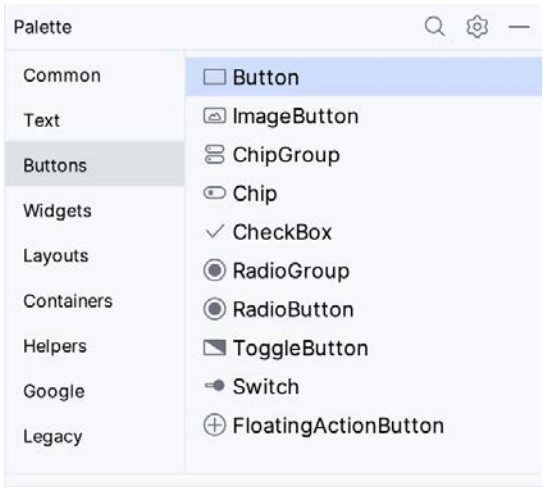


Figure 3-9

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

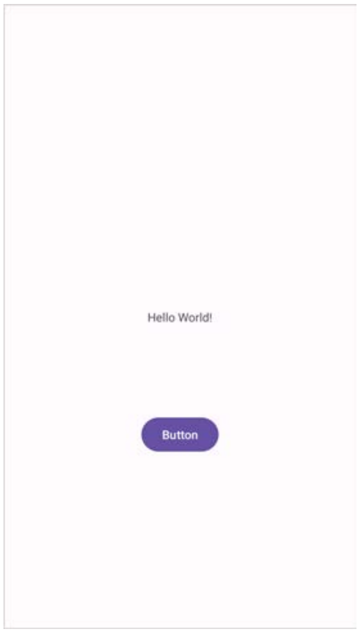
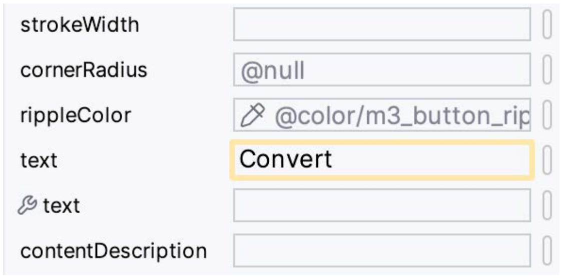


Figure 3-10

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



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

Figure 3-11

The second text property with a wrench next to it allows a text property to be set, which only appears within the Layout Editor tool but is not shown at runtime. This is useful for testing 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 3-12) to add any missing constraints to the layout:

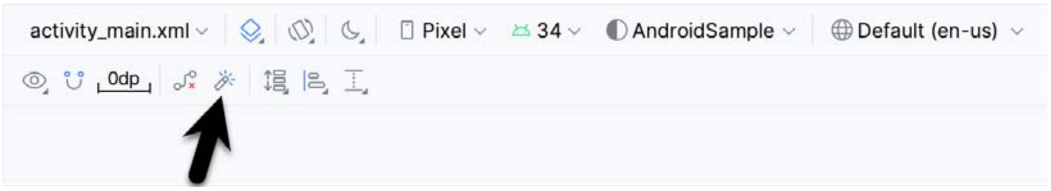


Figure 3-12

It is important to explain the warning button in the top right-hand corner of the Layout Editor tool, as indicated in Figure 3-13. This warning indicates potential problems with the layout. For details on any problems, click on the button:

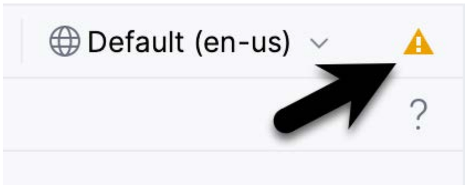
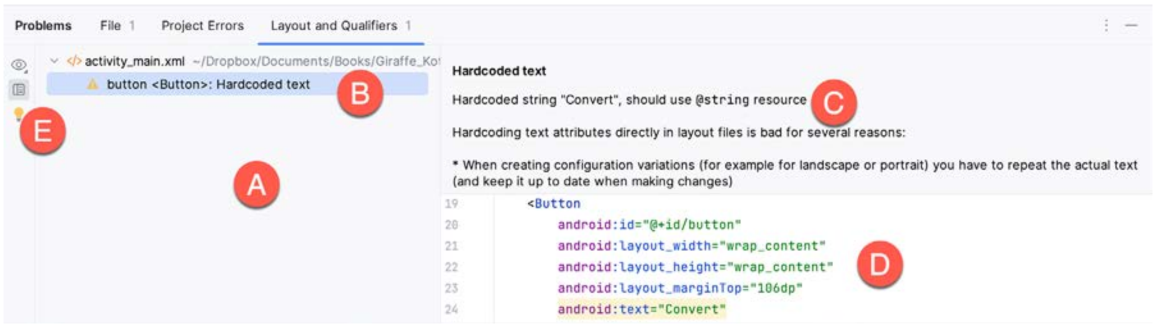


Figure 3-13

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



activity_main.xml ~/Dropbox/Documents/Books/Giraffe_Kot

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 3-14

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

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

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

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

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

This I18N message 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 3-15:

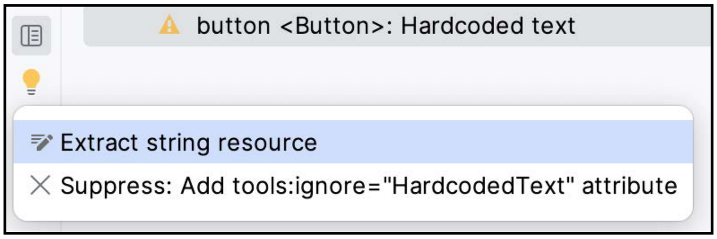
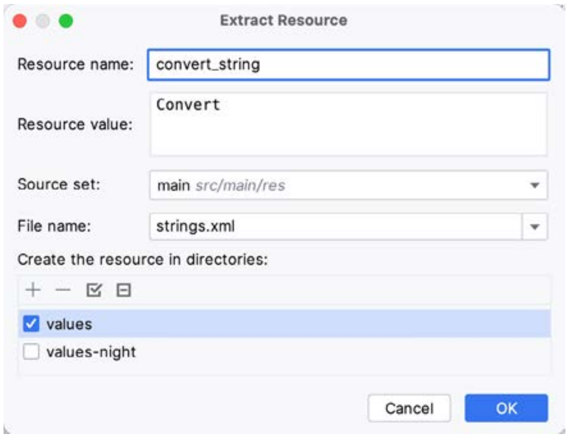

button <Button>: Hardcoded text
Extract string resource
Suppress: Add tools:ignore="HardcodedText" attribute

Figure 3-15

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



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

Figure 3-16

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

The code written later in this chapter will need to access the dollar value entered by the user into the EditText field. It will do this by referencing the id assigned to the widget in the user interface layout. The default id assigned to the widget by Android Studio can be viewed and changed from within the Attributes tool window when the widget is selected in the layout, as shown in Figure 3-17:

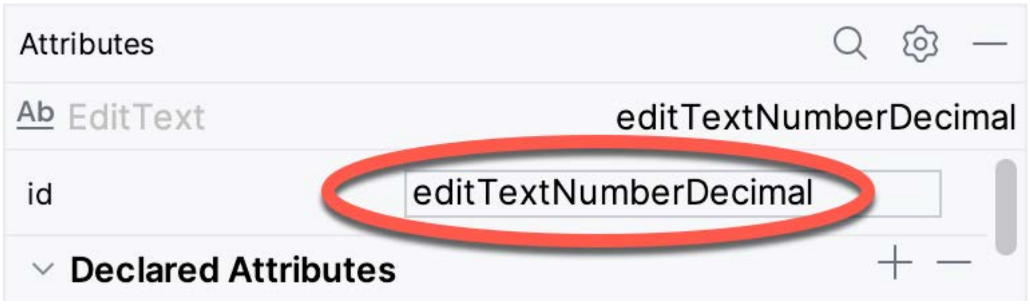


Figure 3-17

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

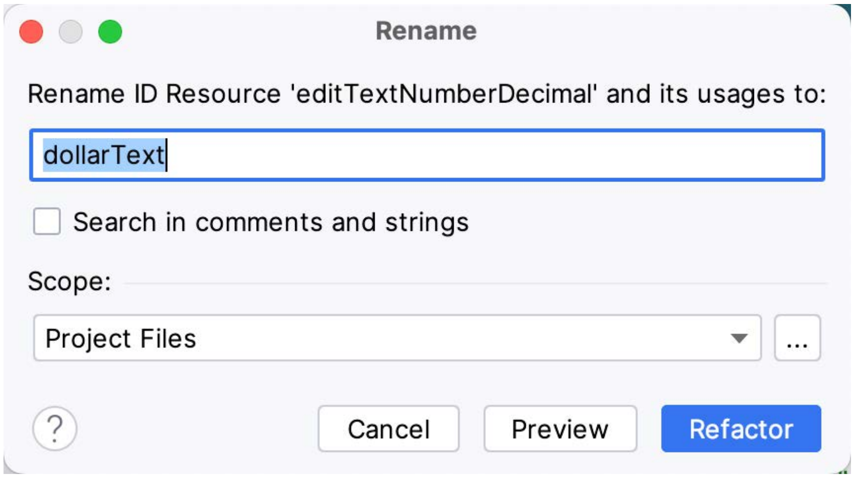

Rename
Rename ID Resource 'editTextNumberDecimal' and its usages to:
dollarText
Search in comments and strings
Scope:
Project Files
...
?
Cancel
Preview
Refactor

Figure 3-18

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 3-19:

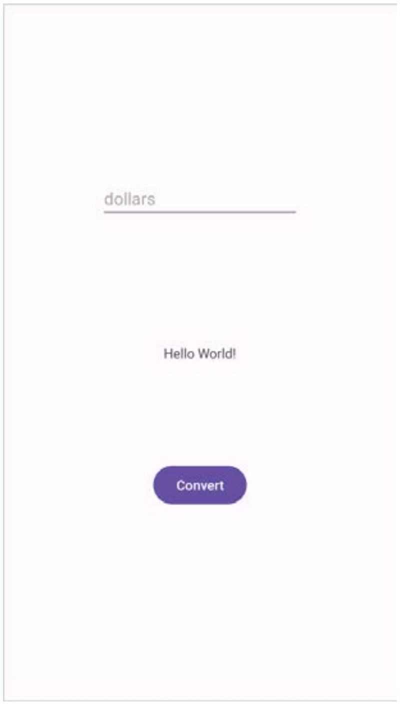


Figure 3-19

3.7 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 3-20 below:

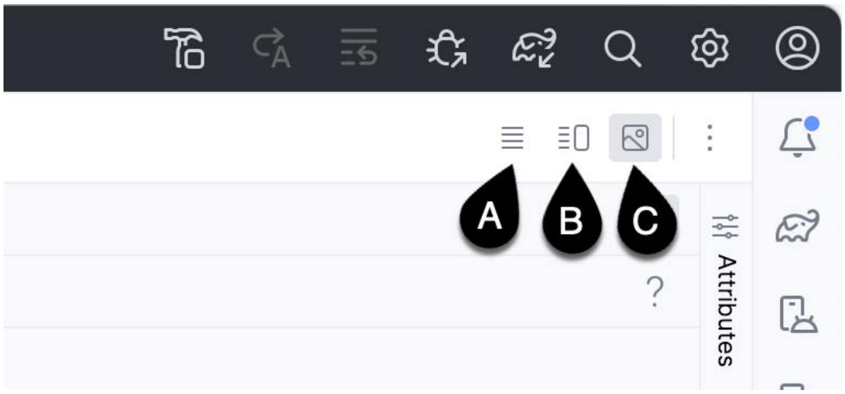
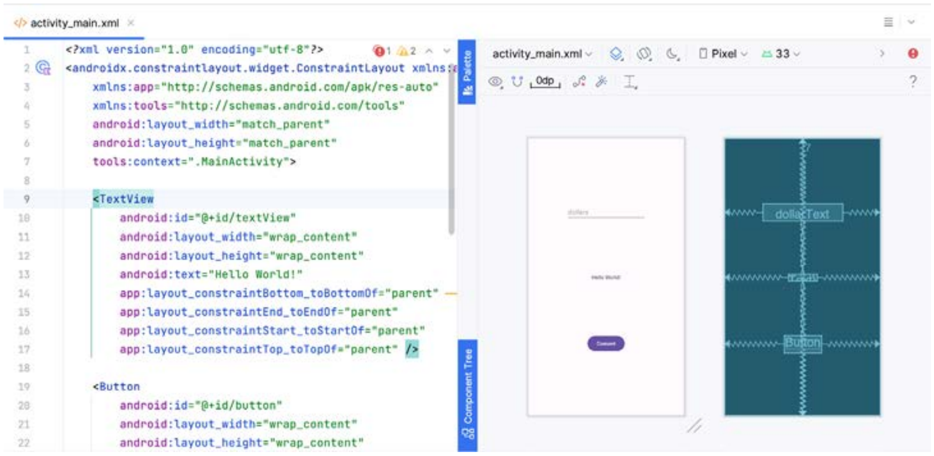


Figure 3-20

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 3-21:



```
<?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 3-21

The button to the left of the View Modes button (marked B in Figure 3-20 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:id="@+id/main"
    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:

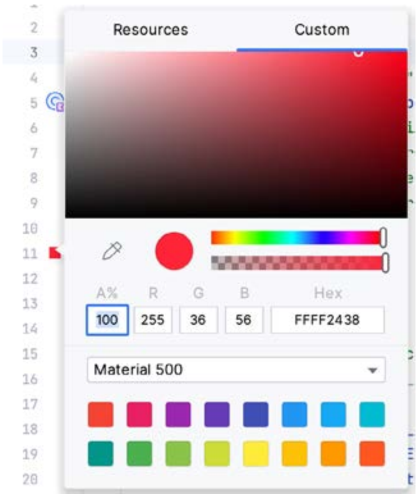


Figure 3-22

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

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

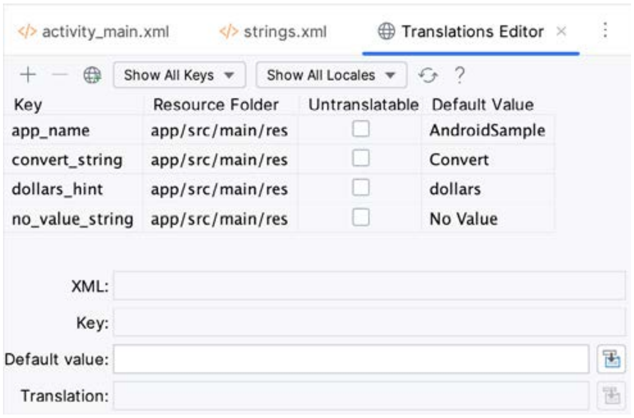
```
<resources>
    <string name="app_name">AndroidSample</string>
    <string name="convert_string">Convert</string>
    <string name="dollars_hint">dollars</string>
</resources>
```

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

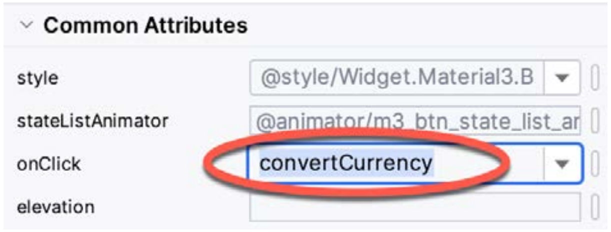
XML:
Key:
Default value:
Translation:

Figure 3-23

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

3.8 Adding 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 3-24

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.example.androidsample;

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);
.
.
    }

    public void convertCurrency(View view) {

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

        if (!dollarText.getText().toString().isEmpty()) {

            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"*.

3.9 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 10

10. An Overview of the Android Architecture

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

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

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

10.1 The Android Software Stack

Android is structured as a software stack comprising applications, an operating system, a runtime environment, middleware, services, and libraries. This architecture can best be represented visually, as Figure 10-1 outlines. Each layer of the stack, and the corresponding elements within each layer, are tightly integrated and carefully tuned to provide the optimal application development and execution environment for mobile devices. The remainder of this chapter will work through the different layers of the Android stack, starting at the bottom with the Linux Kernel.

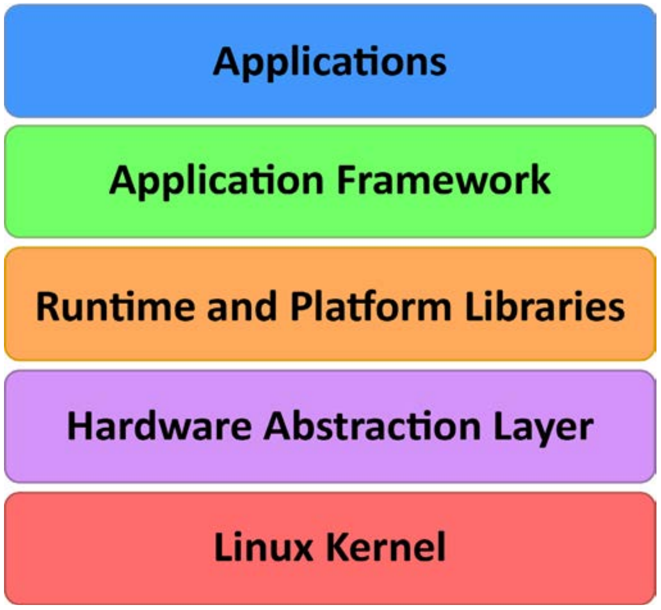


Figure 10-1

10.2 The Linux Kernel

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

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

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

10.3 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) comprises a set of library modules that interface with device components such as the camera, microphone, and accelerometer. When the Android stack needs to access a hardware component, it uses the HAL library modules. Each Android device manufacturer has an abstraction layer for its specific hardware configuration, allowing the standard Android libraries and frameworks to run on any device without being altered for specific hardware.

10.4 Android Runtime – ART

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

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

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

10.5 Android Libraries

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

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

- **android.app** – Provides access to the application model and is the cornerstone of all Android applications.
- **android.content** – Facilitates content access, publishing, and messaging between applications and application components.
- **android.database** – Used to access data published by content providers and includes SQLite database management classes.

- **android.graphics** – A low-level 2D graphics drawing API including colors, points, filters, rectangles, and canvases.
- **android.hardware** – Presents an API providing access to hardware such as the accelerometer and light sensor.
- **android.opengl** – A Java interface to the OpenGL ES 3D graphics rendering API.
- **android.os** – Provides applications with access to standard operating system services, including messages, system services, and inter-process communication.
- **android.media** – Provides classes to enable playback of audio and video.
- **android.net** – A set of APIs providing access to the network stack. Includes *android.net.wifi*, which provides access to the device's wireless stack.
- **android.print** – Includes a set of classes that enable content to be sent to configured printers from within Android applications.
- **android.provider** – A set of convenience classes that provide access to standard Android content provider databases such as those maintained by the calendar and contact applications.
- **android.text** – Used to render and manipulate text on a device display.
- **android.util** – A set of utility classes for performing tasks such as string and number conversion, XML handling and date and time manipulation.
- **android.view** – The fundamental building blocks of application user interfaces.
- **android.widget** - A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
- **android.webkit** – A set of classes intended to allow web-browsing capabilities to be built into applications.

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

10.5.1 C/C++ Libraries

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

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

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

10.6 Application Framework

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

The Android framework includes the following key services:

- **Activity Manager** – Controls all aspects of the application lifecycle and activity stack.
- **Content Providers** – Allows applications to publish and share data with other applications.
- **Resource Manager** – Provides access to non-code embedded resources such as strings, color settings, and user interface layouts.
- **Notifications Manager** – Allows applications to display alerts and notifications to the user.
- **View System** – An extensible set of views used to create application user interfaces.
- **Package Manager** – The system by which applications can find information about other applications currently installed on the device.
- **Telephony Manager** – Provides information to the application about the telephony services available on the device, such as status and subscriber information.
- **Location Manager** – Provides access to the location services allowing an application to receive updates about location changes.

10.7 Applications

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

10.8 Summary

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

Chapter 18

18. A Guide to the Android Studio Layout Editor Tool

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

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

18.1 Basic vs. Empty Views Activity Templates

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

When creating a new Android Studio project, several templates are available to be used as the starting point for the user interface of the main activity. The most basic templates are the Basic Views Activity and Empty Views Activity templates. Although these seem similar at first glance, there are considerable differences between the two options. To see these differences within the layout editor, use the View Options menu to enable Show System UI, as shown in Figure 18-1 below:

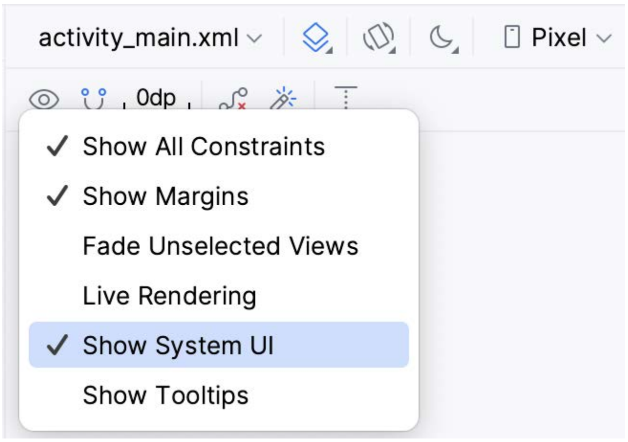


Figure 18-1

The Empty Views Activity template creates a single layout file consisting of a ConstraintLayout manager instance containing a TextView object, as shown in Figure 18-2:

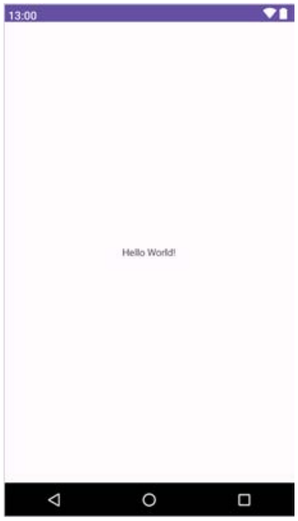
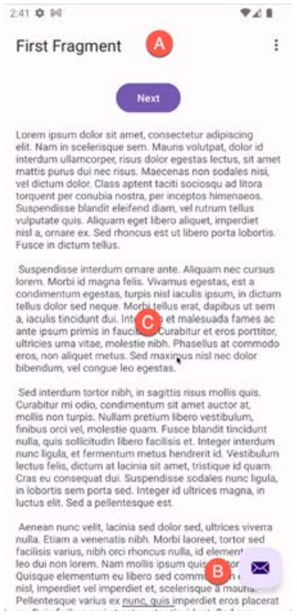


Figure 18-2

The Basic Views Activity, on the other hand, consists of multiple layout files. The top-level layout file has a CoordinatorLayout as the root view, a configurable app bar (which contains a toolbar) that appears across the top of the device screen (marked A in Figure 18-3), and a floating action button (the email button marked B). In addition to these items, the *activity_main.xml* layout file contains a reference to a second file named *content_main.xml* containing the content layout (marked C):



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

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

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

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

Figure 18-3

The Basic Views Activity contains layouts for two screens containing a button and a text view. This template aims to demonstrate how to implement navigation between multiple screens within an app. If an unmodified app using the Basic Views Activity template were to be run, the first of these two screens would appear (marked A in Figure 18-4). Pressing the Next button would navigate to the second screen (B), which, in turn, contains a button to return to the first screen:

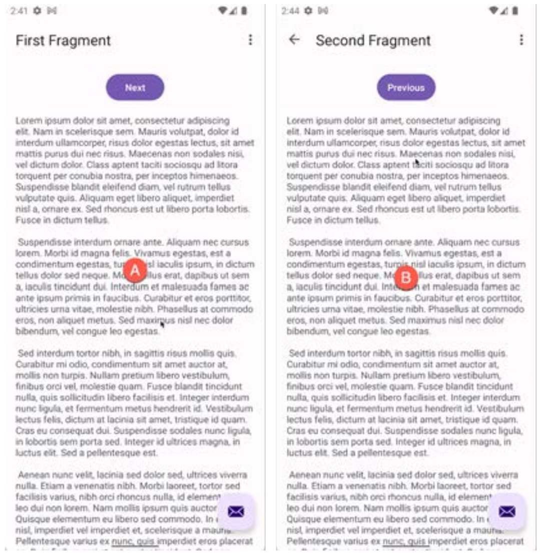


Figure 18-4

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

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

The Empty Views Activity template is helpful if you need neither a floating action button nor a menu in your activity and do not need the special app bar behavior provided by the CoordinatorLayout, such as options to make the app bar and toolbar collapse from view during certain scrolling operations (a topic covered in the chapter entitled "*Working with the AppBar and Collapsing Toolbar Layouts*"). However, the Basic Views Activity is helpful because it provides these elements by default. In fact, it is often quicker to create a new activity using the Basic Views Activity template and delete the elements you do not require than to use the Empty Views Activity template and manually implement behavior such as collapsing toolbars, a menu, or a floating action button.

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

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

1. Double-click on the main *activity_main.xml* layout file in the Project tool window under *app -> res -> layout* to load it into the Layout Editor. With the layout loaded into the Layout Editor tool, select the floating action button and tap the keyboard *Delete* key to remove the object from the layout.
2. Locate and edit the Java code for the activity (located under *app -> java -> <package name> -> <activity class name>* and remove the floating action button code from the onCreate method as follows:

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

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

    setSupportActionBar(binding.toolbar);

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

~~    binding.fab.setOnClickListener(new View.OnClickListener() {~~
~~        @Override~~
~~        public void onClick(View view) {~~
~~            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_~~
~~LONG)~~
~~                    .setAction("Action", null).show();~~
~~        }~~
~~    });~~
}
```

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

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

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

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

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

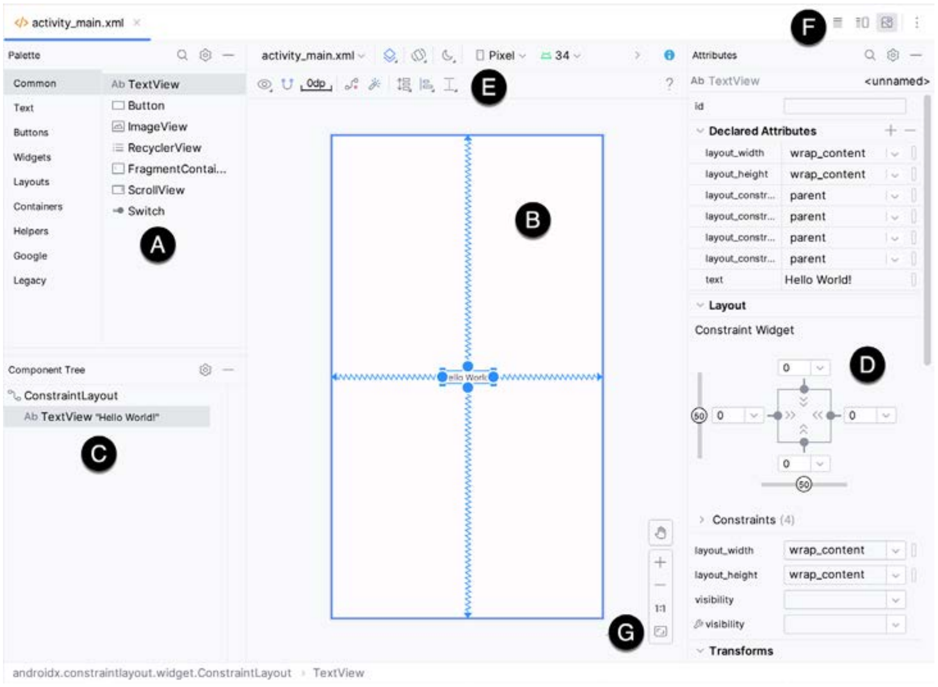
18.2 The Android Studio Layout Editor

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

Under the surface, the Layout Editor tool constructs an XML resource file containing the definition of the user interface that is being designed. As such, the Layout Editor tool operates in three distinct modes: Design, Code, and Split.

18.3 Design Mode

In design mode, the user interface can be visually manipulated by directly working with the view palette and the graphical representation of the layout. Figure 18-5 highlights the key areas of the Android Studio Layout Editor tool in design mode:



layout_width wrap_content
layout_height wrap_content
layout_constr... parent
layout_constr... parent
layout_constr... parent
layout_constr... parent
text Hello World!
layout_width wrap_content
layout_height wrap_content
visibility
visibility

Figure 18-5

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

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

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

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

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

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

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

18.4 The Palette

The Layout Editor palette is organized into two panels designed to make it easy to locate and preview view components for addition to a layout design. The category panel (marked A in Figure 18-6) lists the different categories of view components supported by the Android SDK. When a category is selected from the list, the second panel (B) updates to display a list of the components that fall into that category:

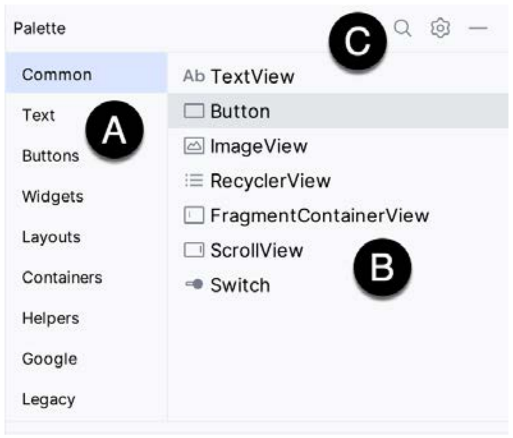


Figure 18-6

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

A search for a specific component within the selected category may be initiated by clicking the search button (marked C in Figure 18-6 above) in the palette toolbar and typing in the component name. As characters are typed, matching results will appear in the component list panel. If you are unsure of the component's category, select the All Results category before or during the search operation.

18.5 Design Mode and Layout Views

The layout editor will appear in Design mode by default, as shown in Figure 18-5 above. This mode provides a visual representation of the user interface. Design mode can be selected by clicking on the button marked C in Figure 18-7:

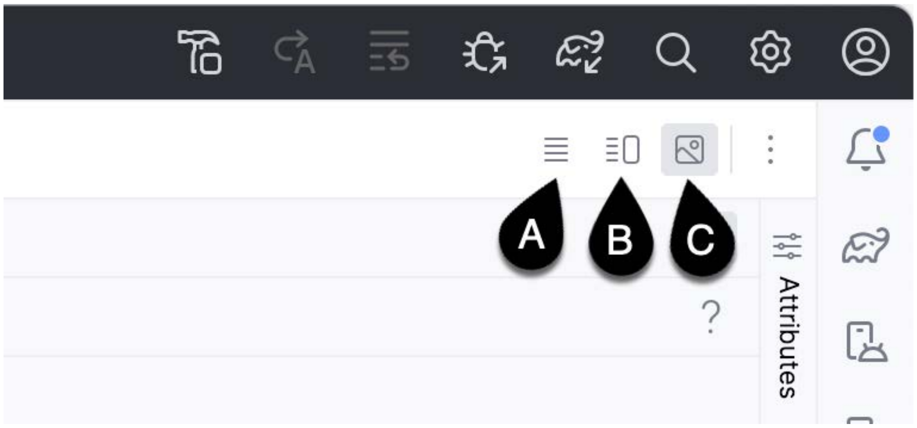


Figure 18-7

When the Layout Editor tool is in Design mode, the layout can be viewed in two ways. The view shown in Figure 18-5 above is the Design view and shows the layout and widgets as they will appear in the running app. A second mode, the Blueprint view, can be shown instead of or concurrently with the Design view. The toolbar menu in Figure 18-8 provides options to display the Design, Blueprint, or both views. Settings are also available to adjust for color blindness. A fifth option, *Force Refresh Layout*, causes the layout to rebuild and redraw. This can be useful when the layout enters an unexpected state or is not accurately reflecting the current design settings:

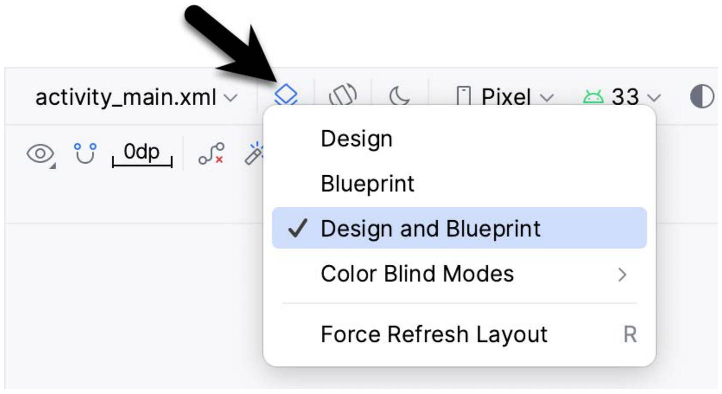


Figure 18-8

Whether to display the layout view, design view, or both is a matter of personal preference. A good approach is to begin with both displayed as shown in Figure 18-9:

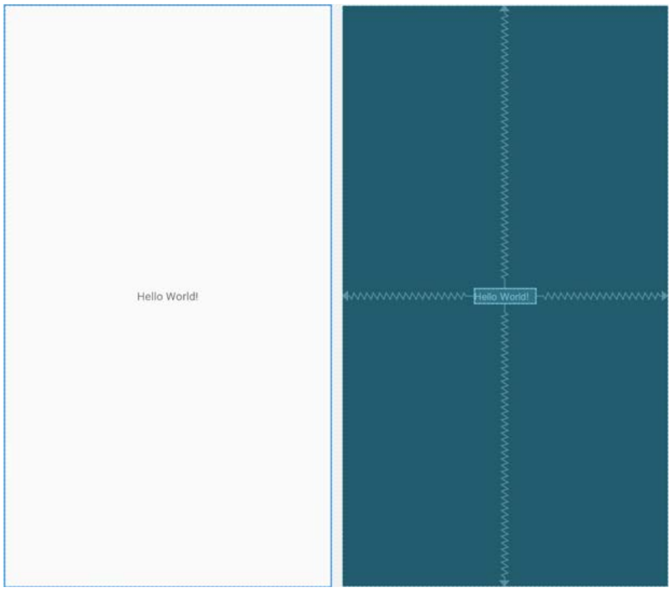


Figure 18-9

18.6 Night Mode

To view the layout in night mode during the design work, select the menu shown in Figure 18-10 below and change the setting to *Night*:

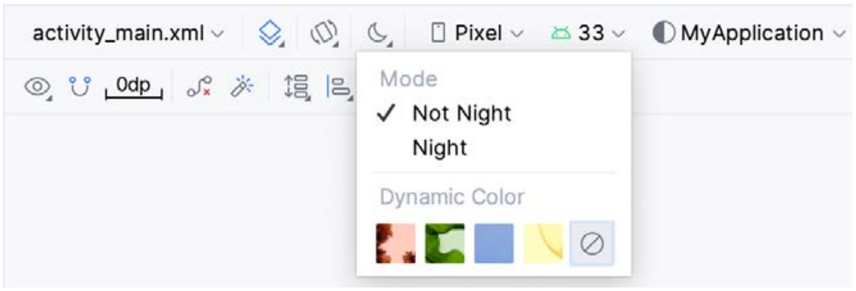


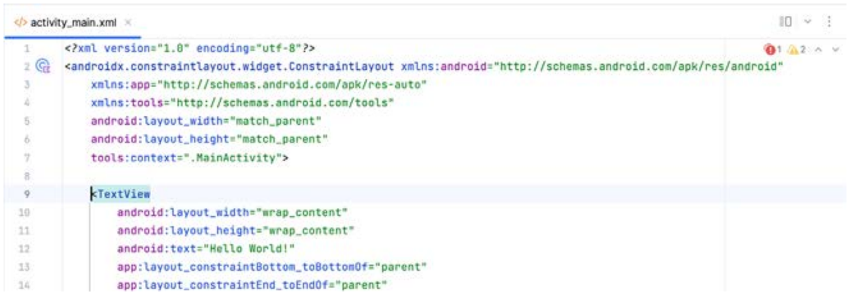
Figure 18-10

The mode menu also includes options for testing dynamic colors, a topic covered in the chapter *"A Material Design 3 Theming and Dynamic Color Tutorial"*.

18.7 Code Mode

It is important to remember when using the Android Studio Layout Editor tool that all it is doing is providing a user-friendly approach to creating XML layout resource files. The underlying XML can be viewed and directly edited during the design process by selecting the button marked A in Figure 18-7 above.

Figure 18-11 shows the Android Studio Layout Editor tool in Code mode, allowing changes to be made to the user interface declaration by modifying the XML:



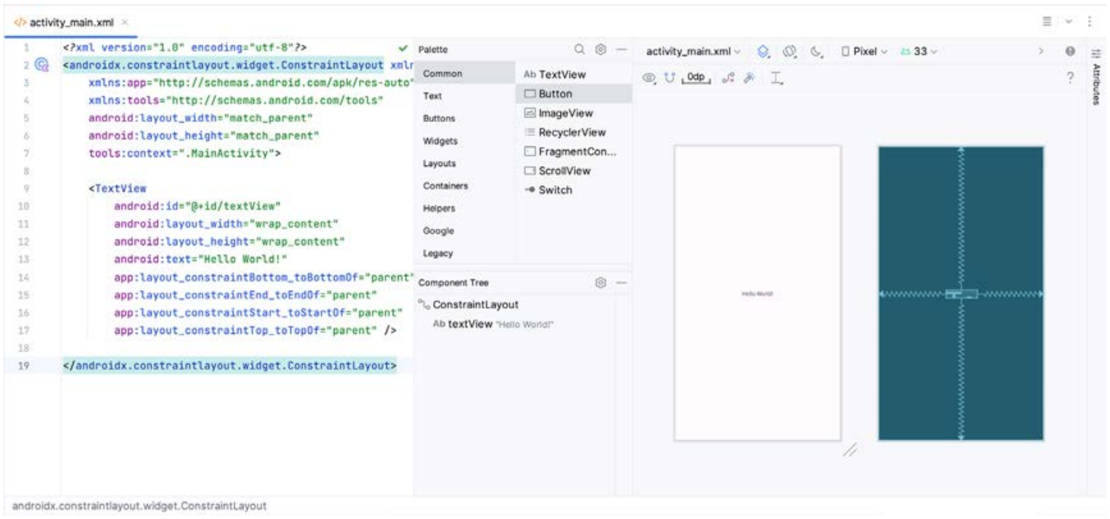
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
```

Figure 18-11

18.8 Split Mode

In Split mode, the editor shows the Design and Code views side-by-side, allowing the user interface to be modified visually using the design canvas and making changes directly to the XML declarations. Split mode is selected using the button marked B Figure 18-7 above.

Any changes to the XML are automatically reflected in the design canvas and vice versa. Figure 18-12 shows the editor in Split mode:



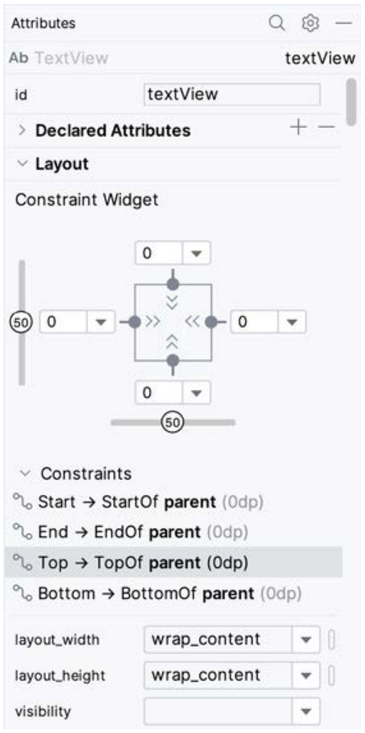
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlr
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <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" />

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

androidx.constraintlayout.widget.ConstraintLayout

Figure 18-12

18.9 Setting Attributes

The Attributes panel provides access to all available settings for the currently selected component. Figure 18-13, for example, shows some of the attributes for the TextView widget:


Attributes

Ab TextView textView

id textView

Declared Attributes

Layout

Constraint Widget

Constraints

Start → StartOf parent (0dp)

End → EndOf parent (0dp)

Top → TopOf parent (0dp)

Bottom → BottomOf parent (0dp)

layout_width wrap_content

layout_height wrap_content

visibility

Figure 18-13

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

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

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

Some attributes contain a narrow button to the right of the value field. This indicates that the Resources dialog is available to assist in selecting a suitable property value. To display the dialog, click on the button. The appearance of this button changes to reflect whether or not the corresponding property value is stored in a resource file or hard-coded. If the value is stored in a resource file, the button to the right of the text property field will be filled in to indicate that the value is not hard-coded, as highlighted in Figure 18-14 below:

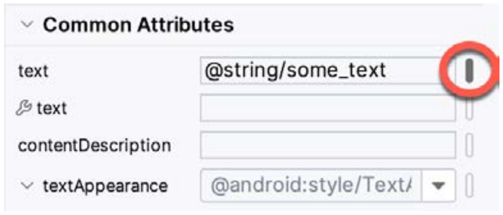


Figure 18-14

Attributes for which a finite number of valid options are available will present a drop-down menu (Figure 18-15) from which a selection may be made.

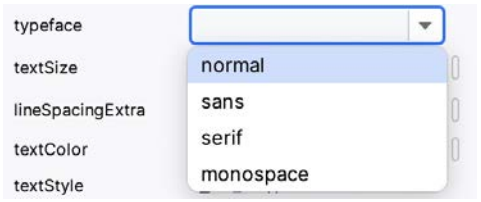


Figure 18-15

A dropper icon can be clicked to display the color selection palette. Similarly, when a flag icon appears, it can be clicked to display a list of options available for the attribute, while an image icon opens the resource manager panel allowing images and other resource types to be selected for the attribute.

18.10 Transforms

The transforms panel within the Attributes tool window (Figure 18-16) provides a set of controls and properties that control visual aspects of the currently selected object in terms of rotation, alpha (used to fade a view in and out), scale (size), and translation (offset from current position):

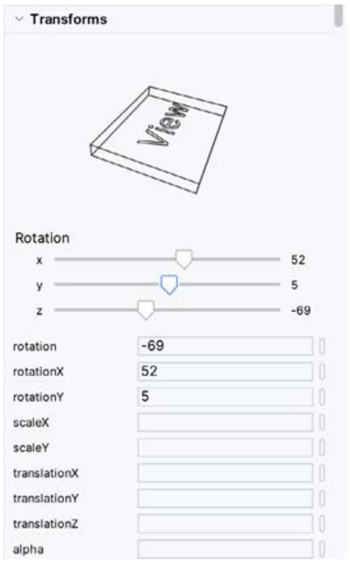


Figure 18-16

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

18.11 Tools Visibility Toggles

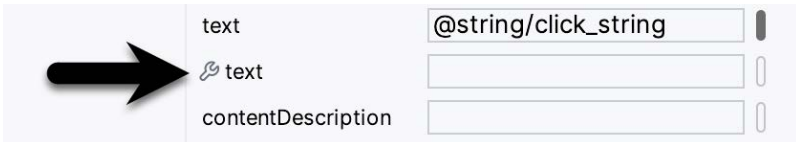
When reviewing the content of an Android Studio XML layout file in Code mode, you will notice that many attributes that define how a view appears and behaves begin with the *android:* prefix. This indicates that the attributes are set within the *android* namespace and will take effect when the app is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

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

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

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

A tool attribute of this type is set in the Attributes tool window by entering the value into the property fields marked by the wrench icon, as shown in Figure 18-17:



text	@string/click_string
text	
contentDescription	

Figure 18-17

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

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

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

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

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

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

While these visibility tools attributes are useful, having to manually edit the XML layout file is a cumbersome process. To make it easier to change these settings, Android Studio provides a set of toggles within the layout editor Component Tree panel. To access these controls, click in the margin to the right of the corresponding view in the panel. Figure 18-18, for example, shows the tools visibility toggle controls for a Button view named myButton:

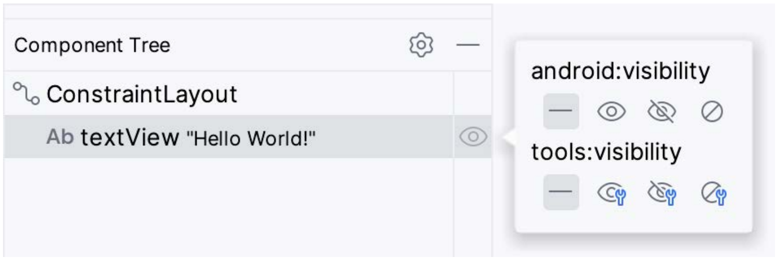


Figure 18-18

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

In addition to the visibility toggles in the Component Tree panel, the layout editor also includes the *tools visibility and position* toggle button shown highlighted in Figure 18-19 below:

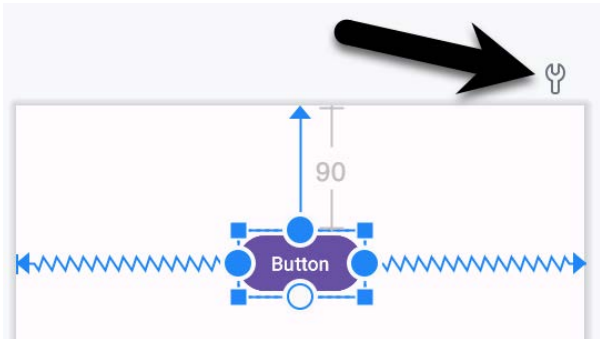


Figure 18-19

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

18.12 Converting Views

Changing a view in a layout from one type to another (such as converting a TextView to an EditText) can be performed easily within the Android Studio layout editor by right-clicking on the view either within the screen layout or Component tree window and selecting the *Convert view...* menu option (Figure 18-20):

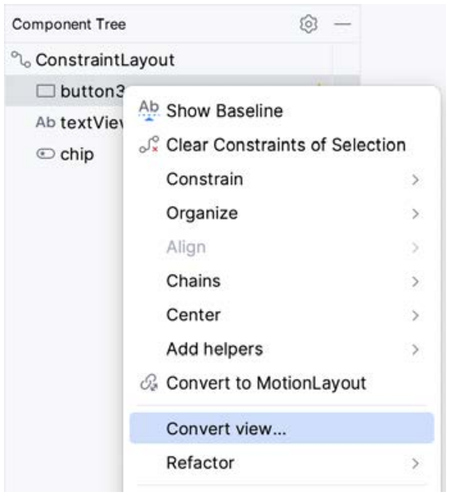


Figure 18-20

Once selected, a dialog containing a list of compatible view types to which the selected object is eligible for conversion will appear. Figure 18-21, for example, shows the types to which an existing TextView view may be converted:

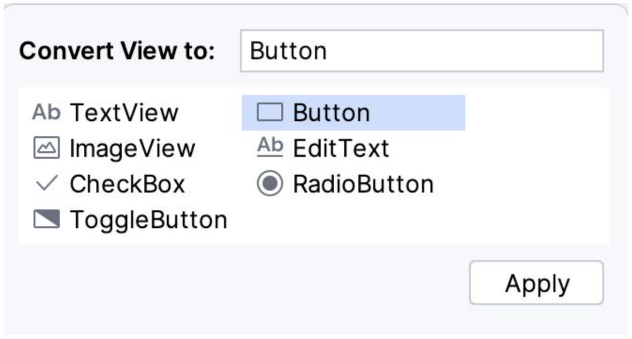
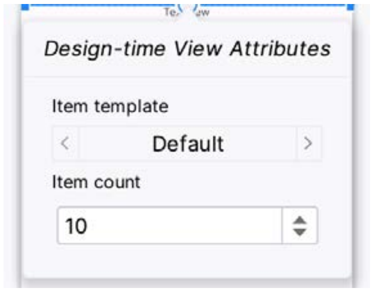


Figure 18-21

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

18.13 Displaying Sample Data

When designing layouts in Android Studio, situations will arise where the content to be displayed within the user interface will not be available until the app is completed and running. This can sometimes make it difficult to assess how the layout will appear at app runtime from within the layout editor. To address this issue, the layout editor allows sample data to be specified, which will populate views within the layout editor with sample images and data. This sample data only appears within the layout editor and is not displayed when the app runs. Sample data may be configured either by directly editing the XML for the layout or visually using the design-time helper by right-clicking on the widget in the design area and selecting the *Set Sample Data* menu option. The design-time helper panel will display a range of preconfigured options for sample data to be displayed on the selected view item, including combinations of text and images in various configurations. Figure 18-22, for example, shows the sample data options displayed when selecting sample data to appear in a RecyclerView list:



Design-time View Attributes

Item template: Default

Item count: 10

Figure 18-22

Alternatively, custom text and images may be provided for display during the layout design process. An example of using sample data within the layout editor is included in a later chapter entitled *"A Layout Editor Sample Data Tutorial"*. Since sample data is implemented as a *tools* attribute, the visibility of the data within the preview can be controlled using the toggle button highlighted in Figure 18-19 above.

18.14 Creating a Custom Device Definition

The device menu in the Layout Editor toolbar (Figure 18-23) provides a list of pre-configured device types, which, when selected, will appear as the device screen canvas. In addition to the pre-configured device types, any AVD instances previously configured within the Android Studio environment will also be listed within the menu. To add additional device configurations, display the device menu, select the *Add Device Definition* option and follow the steps outlined in the chapter entitled *"Creating an Android Virtual Device (AVD) in Android Studio"*.

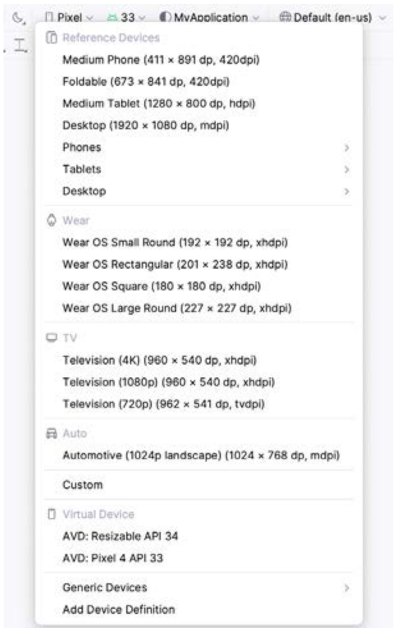

Pixel
33
MyApplication
Default (en-us)
Reference Devices
Medium Phone (411 × 891 dp, 420dpi)
Foldable (673 × 841 dp, 420dpi)
Medium Tablet (1280 × 800 dp, hdpi)
Desktop (1920 × 1080 dp, mdpi)
Phones
Tablets
Desktop
Wear
Wear OS Small Round (192 × 192 dp, xhdpi)
Wear OS Rectangular (201 × 238 dp, xhdpi)
Wear OS Square (180 × 180 dp, xhdpi)
Wear OS Large Round (227 × 227 dp, xhdpi)
TV
Television (4K) (960 × 540 dp, xhdpi)
Television (1080p) (960 × 540 dp, xhdpi)
Television (720p) (962 × 541 dp, tvdpi)
Auto
Automotive (1024p landscape) (1024 × 768 dp, mdpi)
Custom
Virtual Device
AVD: Resizable API 34
AVD: Pixel 4 API 33
Generic Devices
Add Device Definition

Figure 18-23

18.15 Changing the Current Device

As an alternative to the device selection menu, the current device format may be changed by selecting the *Custom* option from the device menu, clicking on the resize handle located next to the bottom right-hand corner of the device screen (Figure 18-24), and dragging to select an alternate device display format. As the screen resizes, markers will appear indicating the various size options and orientations available for selection:

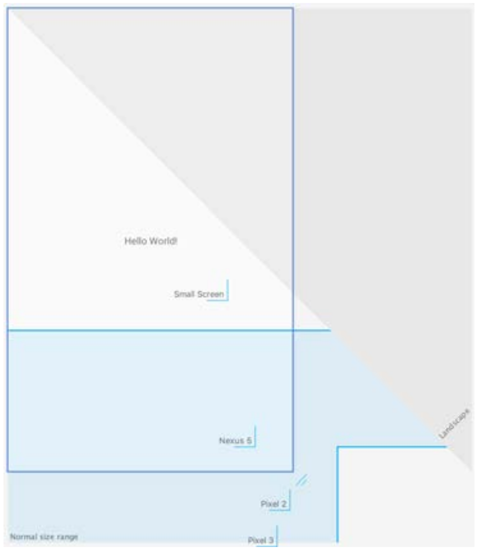


Figure 18-24

18.16 Layout Validation

The layout validation option allows the user interface layout to be previewed simultaneously on a range of Pixel-sized screens. To access the layout validation tool window, select the *View -> Tool Windows -> Layout Validation* menu option. Once loaded, the panel will appear as shown in Figure 18-25, with the layout rendered on multiple device screen configurations:

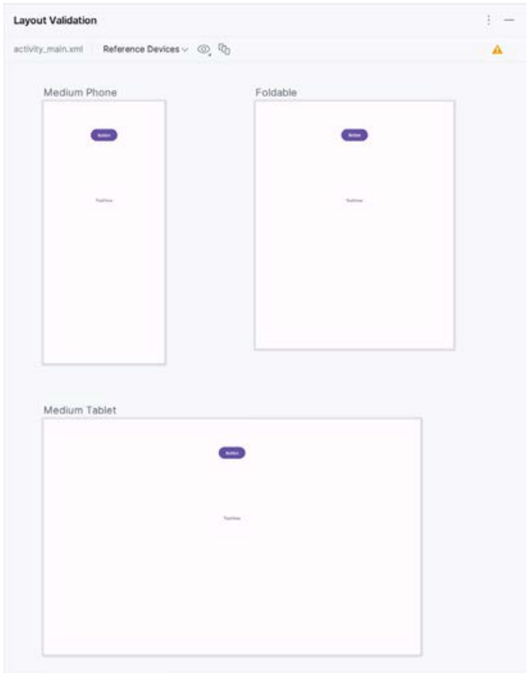


Figure 18-25

18.17 Summary

A key part of developing Android applications involves the creation of the user interface. This is performed within the Android Studio environment using the Layout Editor tool, which operates in three modes. In Design mode, view components are selected from a palette, positioned on a layout representing an Android device screen, and configured using a list of attributes. The underlying XML representing the user interface layout can be directly edited in Code mode. Split mode, on the other hand, allows the layout to be created and modified both visually and via direct XML editing. These modes combine to provide an extensive and intuitive user interface design environment.

The layout validation panel allows user interface layouts to be quickly previewed on various device screen sizes.

34. 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.

34.1 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.

34.2 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.

34.3 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.

34.4 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.

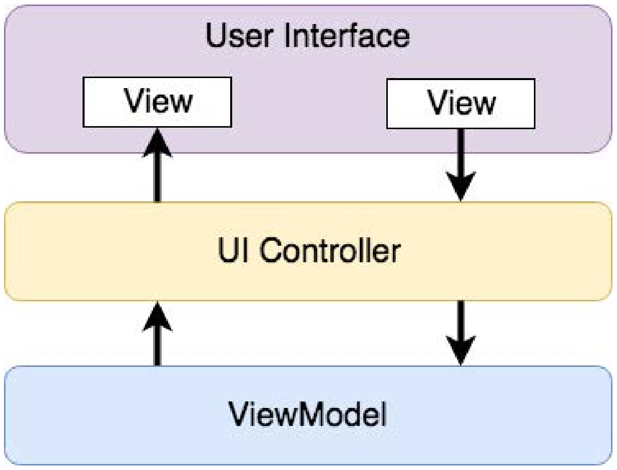


Figure 34-1

34.5 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.

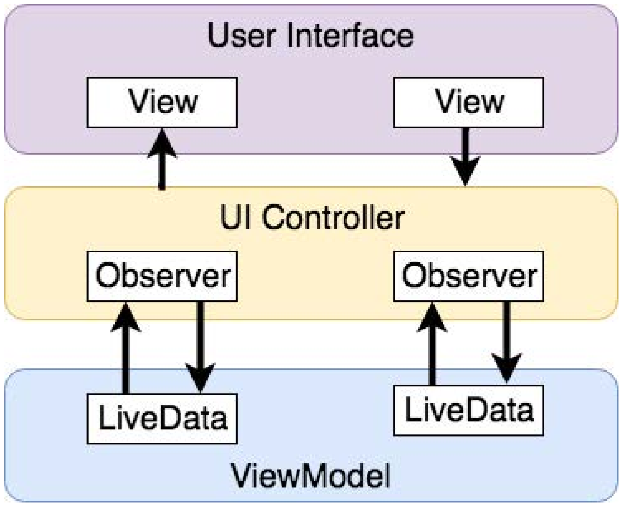


Figure 34-2

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.

34.6 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.

34.7 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.

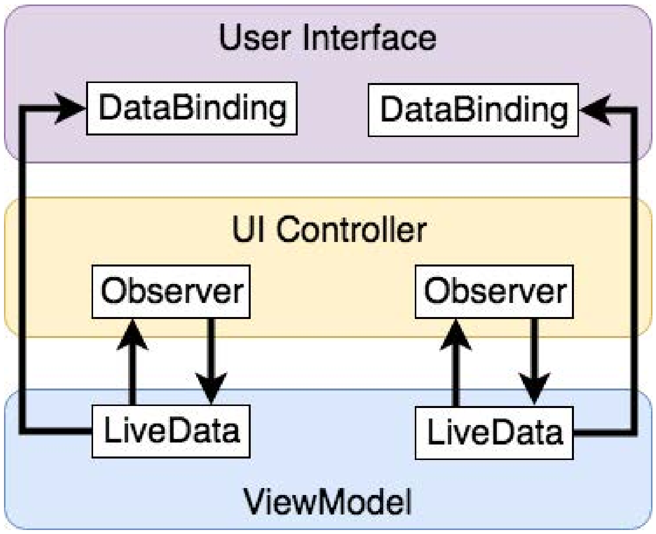


Figure 34-3

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

34.8 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.

34.9 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.

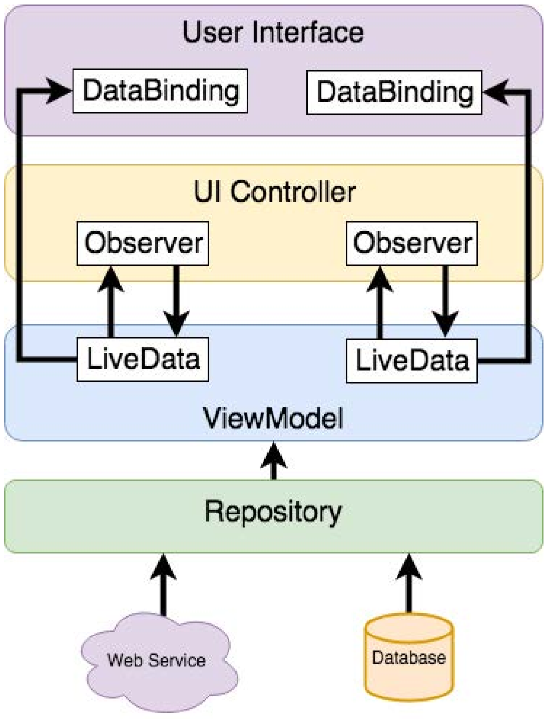


Figure 34-4

34.10 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.

Chapter 35

35. An Android ViewModel Tutorial

The previous chapter introduced the fundamental concepts of Android Jetpack and outlined the basics of modern Android app architecture. Jetpack defines a set of recommendations describing how an Android app project should be structured while providing a set of libraries and components that make it easier to conform to these guidelines to develop reliable apps with less coding and fewer errors.

To help reinforce and clarify the information provided in the previous chapter, this chapter will step through creating an example app project that uses the ViewModel component. The next chapter will further enhance this example by including LiveData and data binding support.

35.1 About the Project

In the chapter entitled *"Creating an Example Android App in Android Studio"*, a project named AndroidSample was created in which all of the code for the app was bundled into the main Activity class file. In the following chapter, an AVD emulator was created and used to run the app. While the app was running, we experienced first-hand the problems that occur when developing apps in this way when the data displayed on a TextView widget was lost during a device rotation.

This chapter will implement the same currency converter app, using the ViewModel component and following the Google app architecture guidelines to avoid Activity lifecycle complications.

35.2 Creating the ViewModel Example Project

When the AndroidSample project was created, the Empty Views Activity template was chosen as the basis for the project. However, the Basic Views Template template will be used for this project.

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

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

35.3 Removing Unwanted Project Elements

As outlined in the *"A Guide to the Android Studio Layout Editor Tool"*, the Basic Views Activity template includes features not required by all projects. Before adding the ViewModel to the project, we first need to remove the navigation features, the second content fragment, and the floating action button as follows:

1. Double-click on the *activity_main.xml* layout file in the Project tool window, select the floating action button, and tap the keyboard delete key to remove the object from the layout.

2. Edit the *MainActivity.java* file and remove the floating action button code from the onCreate method as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
.
.
```

```
    ~~binding.fab.setOnClickListener(new View.OnClickListener() {~~
        ~~@Override~~
        ~~public void onClick(View view) {~~
            ~~Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_~~
~~LONG)~~
                    ~~.setAnchorView(R.id.fab)~~
                    ~~.setAction("Action", null).show();~~
        ~~}~~
    ~~});~~
}
```

3. Within the Project tool window, navigate to and double-click on the *app -> res -> navigation -> nav_graph.xml* file to load it into the navigation editor.

4. Within the editor, select the SecondFragment entry in the graph panel and tap the keyboard delete key to remove it from the graph.

5. Locate and delete the *SecondFragment.java* and *fragment_second.xml* files.

6. The final task is to remove some code from the FirstFragment class so that the Button view no longer navigates to the now non-existent second fragment when clicked. Edit the *FirstFragment.java* file and remove the code from the *onViewCreated()* method so that it reads as follows:

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

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

35.4 Designing the Fragment Layout

The next step is to design the layout of the fragment. First, locate the *fragment_first.xml* file in the Project tool window and double-click on it to load it into the layout editor. Once the layout has loaded, select and delete the existing Button, TextView, and ConstraintLayout components. Next, right-click on the NestedScrollView instance in the Component Tree panel and select the *Convert NestedScrollView to ConstraintLayout* menu option as shown in Figure 35-1, and accept the default settings in the resulting dialog:

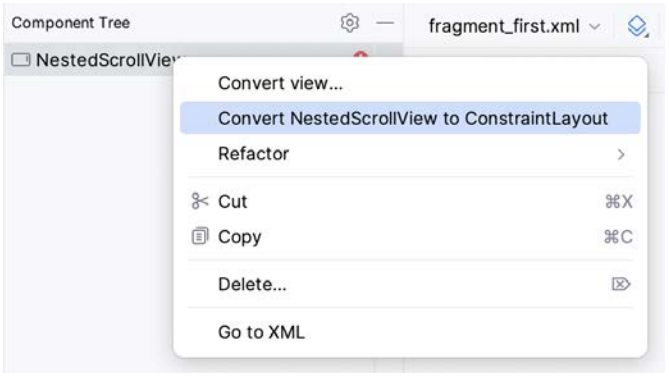


Figure 35-1

Select the converted ConstraintLayout component and use the Attributes tool window to change the id to *constraintLayout*.

Add a new TextView, position it in the center of the layout, and change the id to *resultText*. Next, drag a Number (Decimal) view from the palette and position it above the existing TextView. With the view selected in the layout, refer to the Attributes tool window and change the id to *dollarText*.

Drag a Button widget onto the layout to position it below the TextView, and change the text attribute to read "Convert". With the button still selected, change the id property to *convertButton*. At this point, the layout should resemble that illustrated in Figure 35-2 (note that the three views have been constrained using a vertical chain):

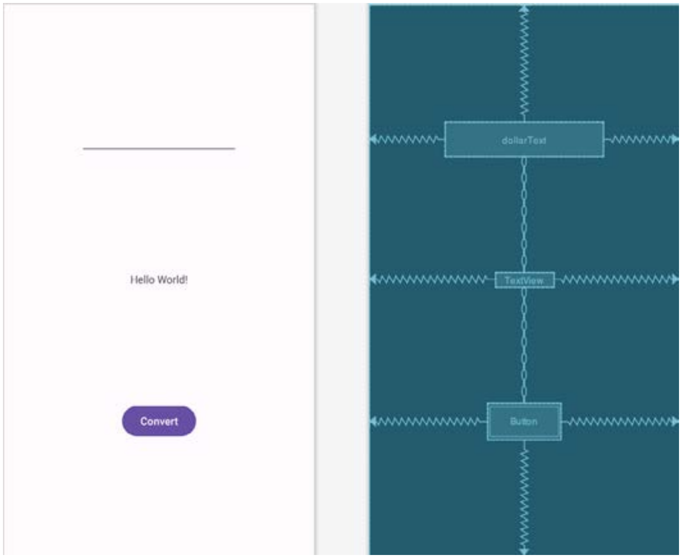


Figure 35-2

Finally, click on the warning icon in the top right-hand corner of the layout editor and convert the hard-coded strings to resources.

35.5 Implementing the View Model

With the user interface layout completed, the data model for the app needs to be created within the view model. Begin by locating the *com.ebookfrenzy.viewmodeldemo* entry in the Project tool window, right-clicking on it, and selecting the *New -> Java Class* menu option. Name the new class MainViewModel and press the keyboard enter key. Edit the new class file so that it reads as follows:

```
package com.ebookfrenzy.viewmodeldemo.ui.main;

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;
    }
}
```

The class declares variables to store the current dollar string value and the converted amount together with getter and setter methods to provide access to those data values. When called, the *setAmount()* method takes the current dollar amount as an argument and stores it in the local *dollarText* variable. The dollar string value is converted to a floating point number, multiplied by a fictitious exchange rate, and the resulting euro value is stored in the *result* variable. The *getResult()* method, on the other hand, returns the current value assigned to the *result* variable.

35.6 Associating the Fragment with the View Model

There needs to be some way for the fragment to obtain a reference to the ViewModel to access the model and observe data changes. A Fragment or Activity maintains references to the ViewModels on which it relies for data using an instance of the ViewModelProvider class.

A ViewModelProvider instance is created using the ViewModelProvider class from within the Fragment. When called, the class initializer is passed a reference to the current Fragment or Activity and returns a ViewModelProvider instance as follows:

```
ViewModelProvider viewModelProvider = new ViewModelProvider(this);
```

Once the ViewModelProvider instance has been created, an index value can be used to request a specific ViewModel class. The provider will then either create a new instance of that ViewModel class or return an existing instance, for example:

```
ViewModel viewModel = viewModelProvider.get(MainViewModel.class);
```

Edit the *FirstFragment.java* file and override the *onCreate()* method to set up the ViewModelProvider:

```
.
.
import androidx.lifecycle.ViewModelProvider;
import androidx.annotation.Nullable;
.
.
public class FirstFragment extends Fragment {

    private MainViewModel viewModel;
.
.
  @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        viewModel = new ViewModelProvider(this).get(MainViewModel.class);
    }
.
.
```

With access to the model view, code can now be added to the Fragment to begin working with the data model.

35.7 Modifying the Fragment

The fragment class needs to be updated to react to button clicks and interact with the data values stored in the ViewModel. The class will also need references to the three views in the user interface layout to react to button clicks, extract the current dollar value, and display the converted currency amount.

In the chapter entitled *"Creating an Example Android App in Android Studio"*, the onClick property of the Button widget was used to designate the method to be called when the user clicks the button. Unfortunately, this property can only call methods on an Activity and cannot be used to call a method in a Fragment. To overcome this limitation, we must add some code to the Fragment class to set up an onClick listener on the button. This can be achieved in the *onViewCreated()* lifecycle method in the *FirstFragment.java* file as outlined below:

```
.
.
public class MainFragment extends Fragment {

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

        binding.convertButton.setOnClickListener(v -> {

        });
    }
.
.
}
```

With the listener added, any code placed within the *onClick()* method will be called whenever the user clicks the button.

35.8 Accessing the ViewModel Data

When the button is clicked, the *onClick()* method needs to read the current value from the EditText view, confirm that the field is not empty, and then call the *setAmount()* method of the ViewModel instance. The method will then need to call the ViewModel's *getResult()* method and display the converted value on the TextView widget.

Since LiveData has yet to be used in the project, it will also be necessary to get the latest result value from the ViewModel each time the Fragment is created.

Remaining in the *FirstFragment.java* file, implement these requirements as follows in the *onViewCreated()* method:

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

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

        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");
            }
        });
}
```

35.9 Testing the Project

With this project development phase completed, build and run the app on the simulator or a physical device, enter a dollar value, and click the Convert button. The converted amount should appear on the TextView, indicating that the UI controller and ViewModel re-structuring is working as expected.

When the original AndroidSample app was run, rotating the device caused the value displayed on the *resultText* TextView widget to be lost. Repeat this test now with the ViewModelDemo app and note that the current euro value is retained after the rotation. This is because the ViewModel remained in memory as the Fragment was destroyed and recreated, and code was added to the *onViewCreated()* method to update the TextView with the result data value from the ViewModel each time the Fragment re-started.

While this is an improvement on the original AndroidSample app, much more can be done to simplify the project by using LiveData and data binding, both of which are the topics of the next chapters.

35.10 Summary

In this chapter, we revisited the AndroidSample project created earlier in the book and created a new version of the project structured to comply with the Android Jetpack architectural guidelines. The example project also demonstrated the use of ViewModels to separate data handling from user interface-related code. Finally, the chapter showed how the ViewModel approach avoids problems handling Fragment and Activity lifecycles.

Chapter 48

48. Creating a Tabbed Interface using the TabLayout Component

The previous chapter outlined the concept of material design in Android. It introduced two of the components provided by the design support library in the form of the floating action button and the Snackbar. This chapter will demonstrate how to use another of the design library components, the TabLayout, which can be combined with the ViewPager class to create a tab-based interface within an Android activity.

48.1 An Introduction to the ViewPager2

Although not part of the design support library, ViewPager2 is a useful companion class when used with the TabLayout component to implement a tabbed user interface. The primary role of ViewPager2 is to allow the user to flip through different pages of information where a layout fragment most typically represents each page. The fragments associated with ViewPager2 are managed by an instance of the FragmentStateAdapter class.

At a minimum, the pager adapter assigned to ViewPager2 must implement two methods. The first, named *getItemCount()*, must return the total number of page fragments to be displayed to the user. The second method, *createFragment()*, is passed a page number and must return the corresponding fragment object ready to be presented to the user.

48.2 An Overview of the TabLayout Component

As previously discussed, TabLayout is one of the components introduced in material design and is included in the design support library. The purpose of the TabLayout is to present the user with a row of tabs that can be selected to display different pages to the user. The tabs can be fixed or scrollable, whereby the user can swipe left or right to view more tabs than will currently fit on the display. The information displayed on a tab can be text-based, an image, or a combination of text and images. Figure 48-1, for example, shows the tab bar for an app consisting of four tabs displaying images:

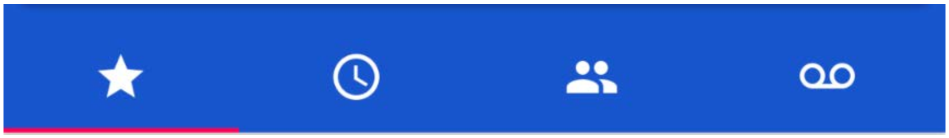


Figure 48-1

Figure 48-2, on the other hand, shows a TabLayout configuration consisting of four tabs displaying text in a scrollable configuration:

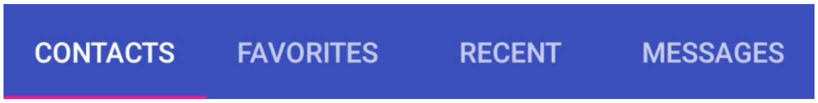


Figure 48-2

The remainder of this chapter will work through creating an example project that demonstrates the TabLayout component together with a ViewPager2 and four fragments.

48.3 Creating the TabLayoutDemo Project

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

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

Once the project has been created, load the *content_main.xml* file into the Layout Editor tool, select the NavHostFragment object, and then delete it. Since we will not be using the navigation features of the Basic Views Activity template, edit the *MainActivity.java* file and modify the *onCreate()* method to remove the navigation code:

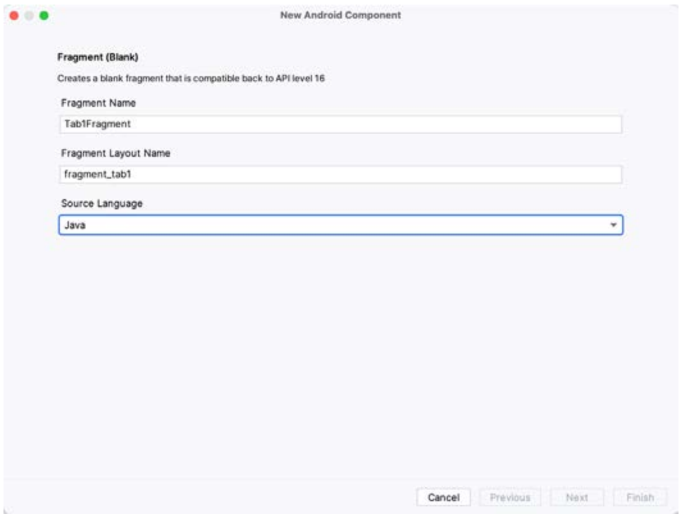
```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

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

    setSupportActionBar(binding.toolbar);

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

    binding.fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_
LONG)
                    .setAction("Action", null).show();
        }
    });
}
```

Finally, delete the *onSupportNavigateUp()* method:

```
~~@Override~~
~~public boolean onSupportNavigateUp() {~~
~~    NavController navController = Navigation.findNavController(this, R.id.nav_~~
~~host_fragment_content_main);~~
~~    return NavigationUI.navigateUp(navController, appBarConfiguration)~~
~~            || super.onSupportNavigateUp();~~
~~}~~
```

48.4 Creating the First Fragment

Each tab on the TabLayout will display a different fragment when selected. Create the first of these fragments by right-clicking on the *app -> java -> com.ebookfrenzy.tablayoutdemo* entry in the Project tool window and selecting the *New -> Fragment -> Fragment (Blank)* option. In the resulting dialog, enter *Tab1Fragment* into the *Fragment Name:* field and *fragment_tab1* into the *Fragment Layout Name:* field. Click on the *Finish* button to create the new fragment:


New Android Component

Fragment (Blank)

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

Fragment Name: Tab1Fragment

Fragment Layout Name: fragment_tab1

Source Language: Java

Figure 48-3

Edit the *Tab1Fragment.java* file, and if Android Studio has not added one automatically, add an OnFragmentInteractionListener interface declaration as follows:

```
.
.
import android.net.Uri;
.
.
    public interface OnFragmentInteractionListener {
        // TODO: Update argument type and name
        void onFragmentInteraction(Uri uri);
    }
.
.
```

Load the newly created *fragment_tab1.xml* file (located under *app -> res -> layout*) into the Layout Editor tool, right-click on the FrameLayout entry in the Component Tree panel, and select the *Convert FrameLayout to ConstraintLayout* menu option. In the resulting dialog, verify that all conversion options are selected before clicking on OK. Change the ID of the layout to *constraintLayout*.

Once the layout has been converted to a ConstraintLayout, delete the TextView from the layout. From the Palette, locate the TextView widget and drag and drop it so it is positioned in the center of the layout. Edit the object's text property to read "Tab 1 Fragment", extract the string to a resource named *tab_1_fragment*, and click the *Infer Constraints* toolbar button. At this point, the layout should match that of Figure 48-4:

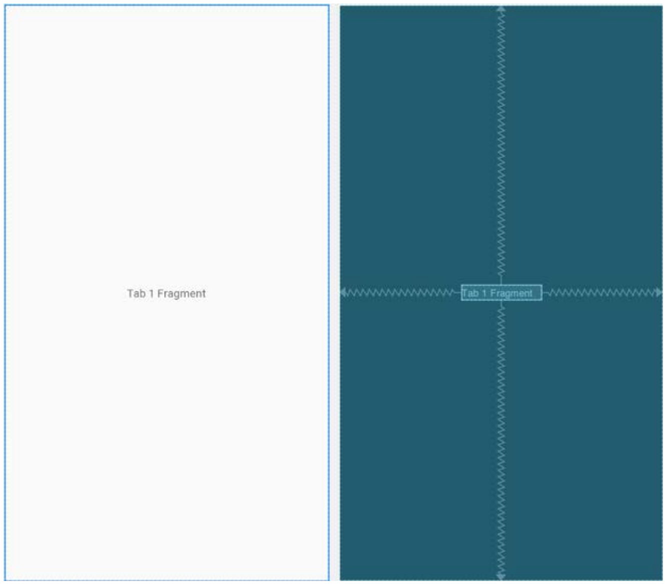


Figure 48-4

48.5 Duplicating the Fragments

So far, the project contains one of the four required fragments. It would be quicker to duplicate the first fragment instead of creating the remaining three fragments using the previous steps. Each fragment consists of a layout XML file and a Java class file, each needing to be duplicated.

Right-click on the *fragment_tab1.xml* file in the Project tool window and select the Copy option from the resulting menu. Right-click on the *layout* entry, this time selecting the Paste option. Name the new layout file *fragment_tab2.xml* in the resulting dialog before clicking the OK button. Edit the new *fragment_tab2.xml* file and change the text on the Text View to "Tab 2 Fragment", following the usual steps to extract the string to a resource named *tab_2_fragment*.

To duplicate the Tab1Fragment class file, right-click on the class listed under *app -> java -> com.ebookfrenzy.tablayoutdemo* and select Copy. Right-click on the *com.ebookfrenzy.tablayoutdemo* entry and select Paste. In the Copy Class dialog, enter Tab2Fragment into the *New name:* field and click OK.

Edit the new *Tab2Fragment.java* file and modify the *onCreateView()* method to inflate the *fragment_tab2* layout file:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.fragment_tab2, container, false);
}
```

Perform the above duplication steps twice to create the fragment layout and class files for the remaining two fragments. On completion of these steps, the project structure should match that of Figure 48-5:

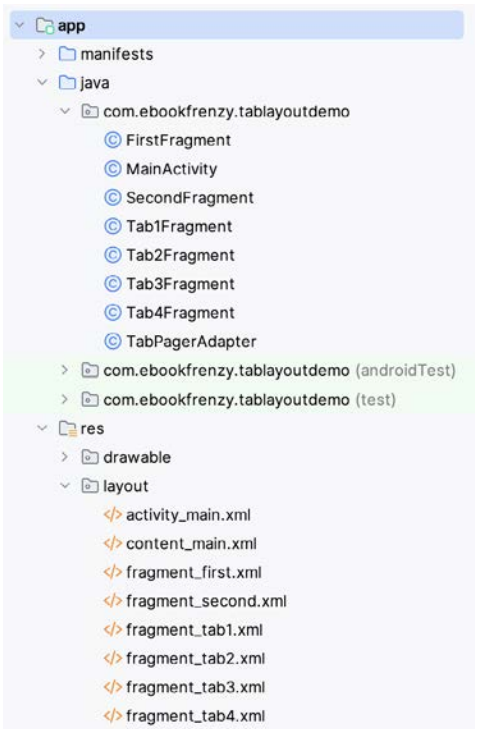


Figure 48-5

48.6 Adding the TabLayout and ViewPager2

With the fragment creation process now complete, the next step is to add the TabLayout and ViewPager2 to the main activity layout file. Edit the *activity_main.xml* file and add these elements as outlined in the following XML listing. Note that the TabLayout component is embedded into the AppBarLayout element while the ViewPager2 is placed after the AppBarLayout:

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

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/Theme.TabLayoutDemo.AppBarOverlay">

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/Theme.TabLayoutDemo.PopupOverlay" />

        <com.google.android.material.tabs.TabLayout
            android:id="@+id/tabLayout"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            app:tabMode="fixed"
            app:tabGravity="fill"/>

    </com.google.android.material.appbar.AppBarLayout>

    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/view_pager"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior" />

    <include layout="@layout/content_main" />

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_marginEnd="@dimen/fab_margin"
        android:layout_marginBottom="16dp"
        app:srcCompat="@android:drawable/ic_dialog_email" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

48.7 Creating the Pager Adapter

This example will use the ViewPager2 approach to handling the fragments assigned to the TabLayout tabs, with ViewPager2 added to the layout resource file, a new class which subclasses FragmentStateAdapter needs to be added to the project to manage the fragments that will be displayed when the user selects the tab items.

Add a new class to the project by right-clicking on the *com.ebookfrenzy.tablayoutdemo* entry in the Project tool window and selecting the *New -> Java Class* menu option. In the new class dialog, enter *TabPagerAdapter* into the *Name:* field, select the Class item in the list, and press the keyboard Return key.

Edit the *TabPagerAdapter.java* file so that it reads as follows:

```
package com.ebookfrenzy.tablayoutdemo;

import androidx.annotation.NonNull;
import androidx.fragment.app.*;
import androidx.viewpager2.adapter.FragmentStateAdapter;
```

```
public class TabPagerAdapter extends FragmentStateAdapter {

    int tabCount;

    public TabPagerAdapter(@NonNull FragmentActivity fragmentActivity, int 
numberOfTabs) {
        super(fragmentActivity);
        this.tabCount = numberOfTabs;
    }

    @NonNull
    @Override
    public Fragment createFragment(int position) {
        switch (position) {
            case 0:
                return new Tab1Fragment();
            case 1:
                return new Tab2Fragment();
            case 2:
                return new Tab3Fragment();
            case 3:
                return new Tab4Fragment();
            default:
                return null;
        }
    }

    @Override
    public int getItemCount() {
        return tabCount;
    }
}
```

The class is declared as extending the FragmentStateAdapter class, and a constructor is implemented, allowing the number of pages required to be passed to the class when an instance is created. The *createFragment()* method will be called when a specific page is required. A switch statement is used to identify the page number being requested and to return a corresponding fragment instance. Finally, the *getItemCount()* method returns the count value passed through when the object instance was created.

48.8 Performing the Initialization Tasks

The remaining tasks involve initializing the TabLayout, ViewPager2, and TabPagerAdapter instances and declaring the main activity class as implementing fragment interaction listeners for each of the four tab fragments. Edit the *MainActivity.java* file so that it reads as follows:

```
package com.ebookfrenzy.tablayoutdemo;
.
.
import android.net.Uri;
import com.google.android.material.tabs.TabLayoutMediator;
.
.
public class MainActivity extends AppCompatActivity implements
        Tab1Fragment.OnFragmentInteractionListener,
        Tab2Fragment.OnFragmentInteractionListener,
        Tab3Fragment.OnFragmentInteractionListener,
        Tab4Fragment.OnFragmentInteractionListener {

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

    protected void configureTabLayout() {

        for (int i = 0; i < 4; i++) {
            binding.tabLayout.addTab(binding.tabLayout.newTab());
        }

        final TabPagerAdapter adapter = new TabPagerAdapter
                (this, binding.tabLayout.getTabCount());
        binding.viewPager.setAdapter(adapter);

        new TabLayoutMediator(binding.tabLayout, binding.viewPager,
                (tab, position) -> tab.setText("Tab " + (position + 1) +
                                " Item")).attach();
    }

   @Override
   public void onFragmentInteraction(Uri uri) {
   }
.
.
}
```

The code begins by creating four tabs and adding them to the TabLayout instance as follows:

```
for (int i = 0; i < 4; i++) {
    binding.tabLayout.addTab(binding.tabLayout.newTab());
}
```

Next, an instance of the TabPagerAdapter class is created. Note that the code to create the TabPagerAdapter instance passes through the number of tabs assigned to the TabLayout component. The TabPagerAdapter instance is then assigned as the adapter for the ViewPager2 instance:

```
final TabPagerAdapter adapter = new TabPagerAdapter
        (this, binding.tabLayout.getTabCount());
binding.viewPager.setAdapter(adapter);
```

Finally, an instance of the TabLayoutMediator class is used to connect the TabLayout with the ViewPager2 object:

```
new TabLayoutMediator(binding.tabLayout, binding.viewPager,
        (tab, position) -> tab.setText("Tab " + (position + 1) + " Item")).
attach();
```

This class ensures that the TabLayout tabs remain synchronized with the currently selected fragment. This process involves ensuring that the correct text is displayed on each tab. In this case, the text is configured to read "Tab <*n*> Item" where <*n*> is replaced by the number of the currently selected tab.

48.9 Testing the Application

Compile and run the app on a device or emulator and make sure that selecting a tab causes the corresponding fragment to appear in the content area of the screen:

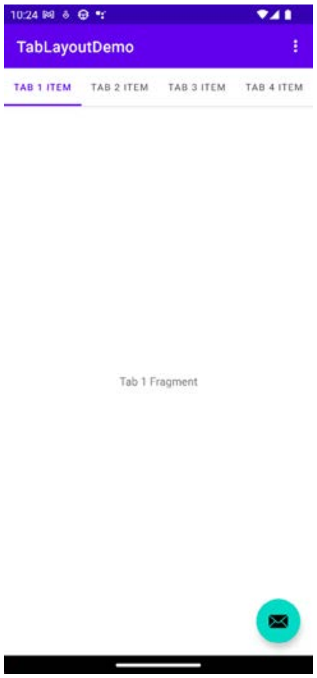


Figure 48-6

48.10 Customizing the TabLayout

The TabLayout in this example project is configured using *fixed* mode. This mode works well for a limited number of tabs with short titles. A greater number of tabs or longer titles can quickly become a problem when using fixed mode, as illustrated by Figure 48-7:

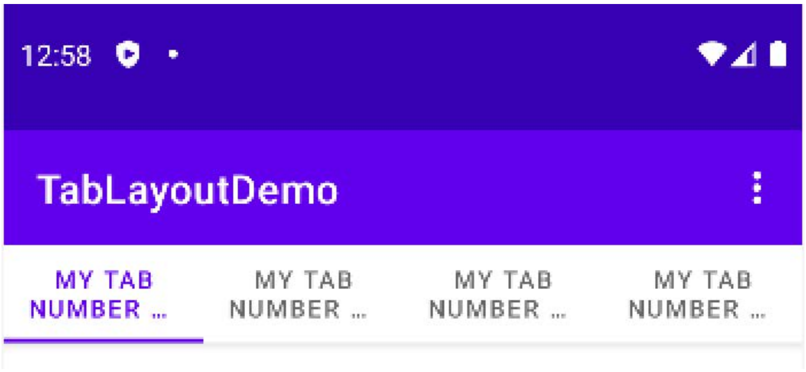


Figure 48-7

To fit the tabs into the available display width, the TabLayout has used multiple lines of text. Even so, the second line is truncated, making it impossible to see the full title. The best solution to this problem is to switch the TabLayout to scrollable mode. In this mode, the titles appear in full-length, single-line format allowing the user to swipe to scroll horizontally through the available items, as demonstrated in Figure 48-8:

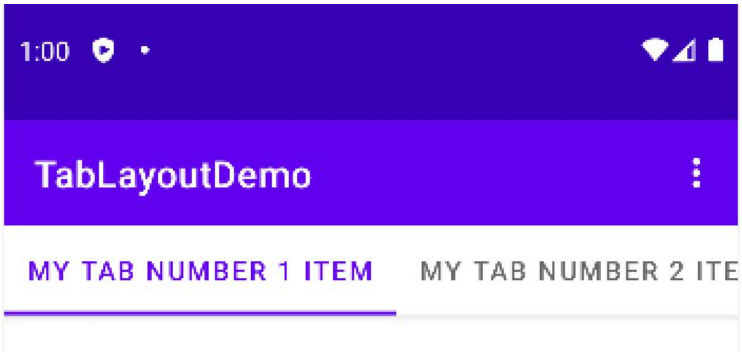


Figure 48-8

To switch a TabLayout to scrollable mode, change the *app:tabMode* property in the *activity_main.xml* layout resource file from "fixed" to "scrollable":

```
<android.support.design.widget.TabLayout
    android:id="@+id/tabLayout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:tabMode="scrollable"
    app:tabGravity="fill"/>
</android.support.design.widget.AppBarLayout>
```

When in fixed mode, the TabLayout may be configured to control how the tab items are displayed to take up the available space on the screen. This is controlled via the *app:tabGravity* property, the results of which are more noticeable on wider displays such as tablets in landscape orientation. When set to "fill", for example, the items will be distributed evenly across the width of the TabLayout, as shown in Figure 48-9:

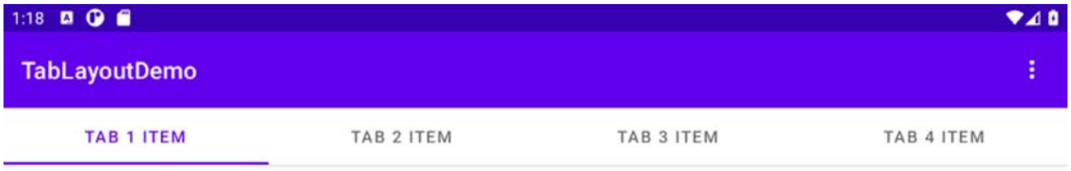


Figure 48-9

Changing the property value to “center” will cause the items to be positioned relative to the center of the tab bar:

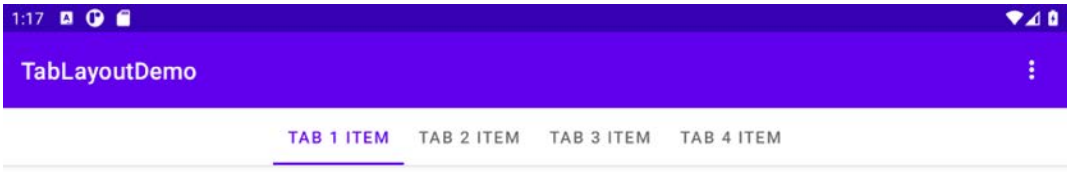


Figure 48-10

48.11 Summary

TabLayout is one of the components introduced in the Android material design implementation. The purpose of the TabLayout component is to present a series of tab items that display different content to the user when selected. The tab items can display text, images, or a combination. When combined with the ViewPager2 class and fragments, tab layouts can be created relatively easily, with each tab item selection displaying a different fragment.

Chapter 70

70. 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 chapters, this chapter will explore 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.

70.1 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 70-1 outlines the recommended architecture for a typical Android app:

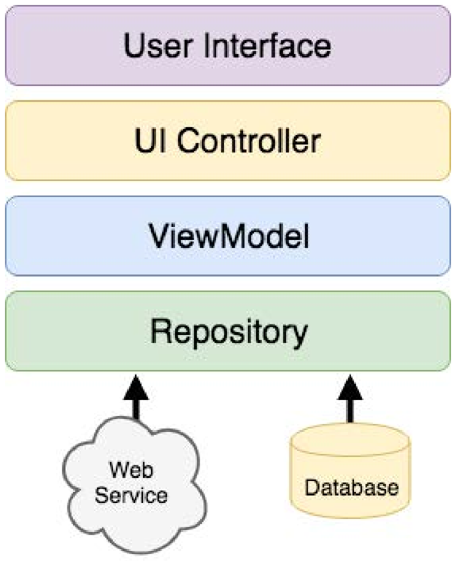


Figure 70-1

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.

70.2 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:

70.2.1 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.

70.2.2 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.

70.2.3 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.

70.2.4 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.

70.2.5 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 70-2 illustrates how these different elements interact to provide Room-based database storage within an Android app:

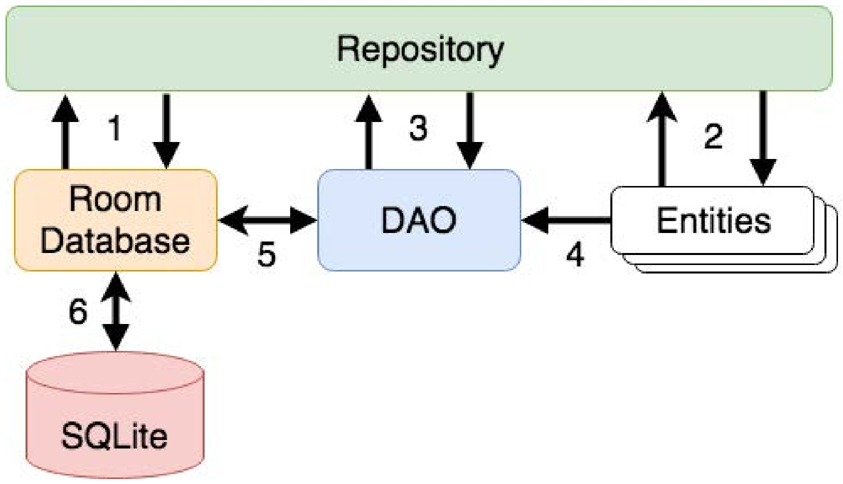


Figure 70-2

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.

70.3 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.

70.4 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);
```

70.5 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.

70.6 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*").

70.7 In-Memory Databases

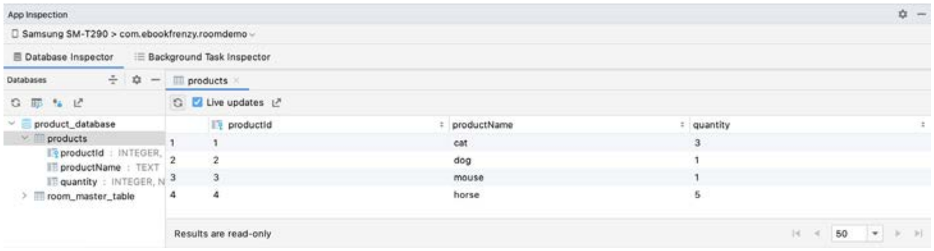
The examples outlined in this chapter use a SQLite database that exists as a database file on the persistent storage of an Android device. This ensures that the data persists even after the app process is terminated.

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();
```

70.8 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 70-3:



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

Figure 70-3

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

70.9 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.

91. Working with Material Design 3 Theming

The appearance of an Android app is intended to conform to a set of guidelines defined by Material Design. Google developed Material Design to provide a level of design consistency between different apps while also allowing app developers to include their own branding in terms of color, typography, and shape choices (a concept referred to as Material theming). In addition to design guidelines, Material Design also includes a set of UI components for use when designing user interface layouts, many of which we have used throughout this book.

This chapter will provide an overview of how theming works within an Android Studio project and explore how the default design configurations provided for newly created projects can be modified to meet your branding requirements.

91.1 Material Design 2 vs. Material Design 3

Before beginning, it is important to note that Google is transitioning from Material Design 2 to Material Design 3 and that Android Studio Koala projects default to Material Design 3. Material Design 3 provides the basis for Material You, a feature introduced in Android 12 that allows an app to automatically adjust theme elements to complement preferences configured by the user on the device. For example, dynamic color support provided by Material Design 3 allows the colors used in apps to adapt automatically to match the user's wallpaper selection.

91.2 Understanding Material Design Theming

We know that Android app user interfaces are created by assembling components such as layouts, text fields, and buttons. These components appear using default colors unless we specifically override a color attribute in the XML layout resource file or by writing code. The project's theme defines these default colors. The theme consists of a set of color slots (declared in *themes.xml* files) which are assigned color values (declared in the *colors.xml* file). Each UI component is programmed internally to use theme color slots as the default color for specific attributes (such as the foreground and background colors of the Text widget). It follows, therefore, that we can change the application-wide theme of an app by changing the colors assigned to specific theme slots. When the app runs, the new default colors will be used for all widgets when the user interface is rendered.

91.3 Material Design 3 Theming

Before exploring Material Design 3, we must consider how it is used in an Android Studio project. The theme used by an application project is declared as a property of the *application* element within the *AndroidManifest.xml* file, for example:

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

    <application
.
.
```

```
        android:supportsRtl="true"
        android:theme="@style/Theme.MyDemoApp"
        tools:targetApi="31">
        <activity
.
.
```

As previously discussed, all of the files associated with the project theme are contained within the *colors.xml* and *themes.xml* files located in the *res -> values* folder, as shown in Figure 91-1:

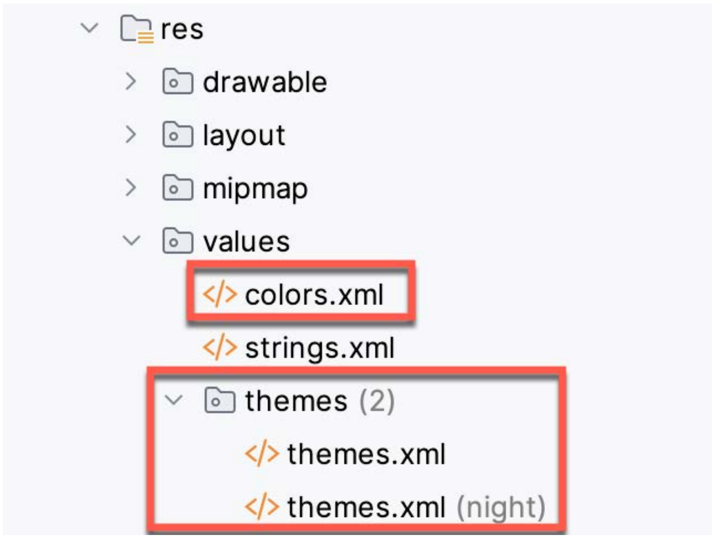


Figure 91-1

The theme itself is declared in the two *themes.xml* files located in the *themes* folder. These resource files declare different color palettes containing Material Theme color slots for use when the device is in light or dark (night) mode. Note that the style name property in each file must match that referenced in the *AndroidManifest.xml* file, for example:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Base.Theme.MyDemoApp" parent="Theme.Material3.DayNight.
NoActionBar">
        <!-- Customize your light theme here. -->
        <!-- <item name="colorPrimary">@color/my_light_primary</item> -->
    </style>

    <style name="Theme.MyDemoApp" parent="Base.Theme.MyDemoApp" />
</resources>
```

These color slots (also referred to as *color attributes*) are used by the Material components to set colors when they are rendered on the screen. For example, the *colorPrimary* color slot is used as the background color for the Material Button component.

Color slots in MD3 are grouped as Primary, Secondary, Tertiary, Error, Background, and Surface. These slots are further divided into pairs consisting of a *base color* and an *"on" base color*. This generally translates to the background and foreground colors of a Material component.

The particular group used for coloring will differ between widgets. A Material Button widget, for example, will use the *colorPrimary* base color for the background color and *colorOnPrimary* for its content (i.e., the text or icon it displays). The FloatingActionButton component, on the other hand, uses *colorPrimaryContainer* as the background color and *colorOnPrimaryContainer* for the foreground. The correct group for a specific widget type can usually be identified quickly by changing color settings in the theme files and reviewing the rendering in the layout editor.

Suppose that we need to change *colorPrimary* to red. We achieve this by adding a new entry to the *colors.xml* file for the red color and then assigning it to the *colorPrimary* slot in the *themes.xml* file. The *colorPrimary* slot in an MD3 theme night, therefore, read as follows:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Base.Theme.MyDemoApp" parent="Theme.Material3.DayNight.
NoActionBar">
        <item name="colorPrimary">@color/my_bright_primary</item>
    </style>

    <style name="Theme.MyDemoApp" parent="Base.Theme.MyDemoApp" />
</resources>
```

This color is then declared in the *colors.xml* file:

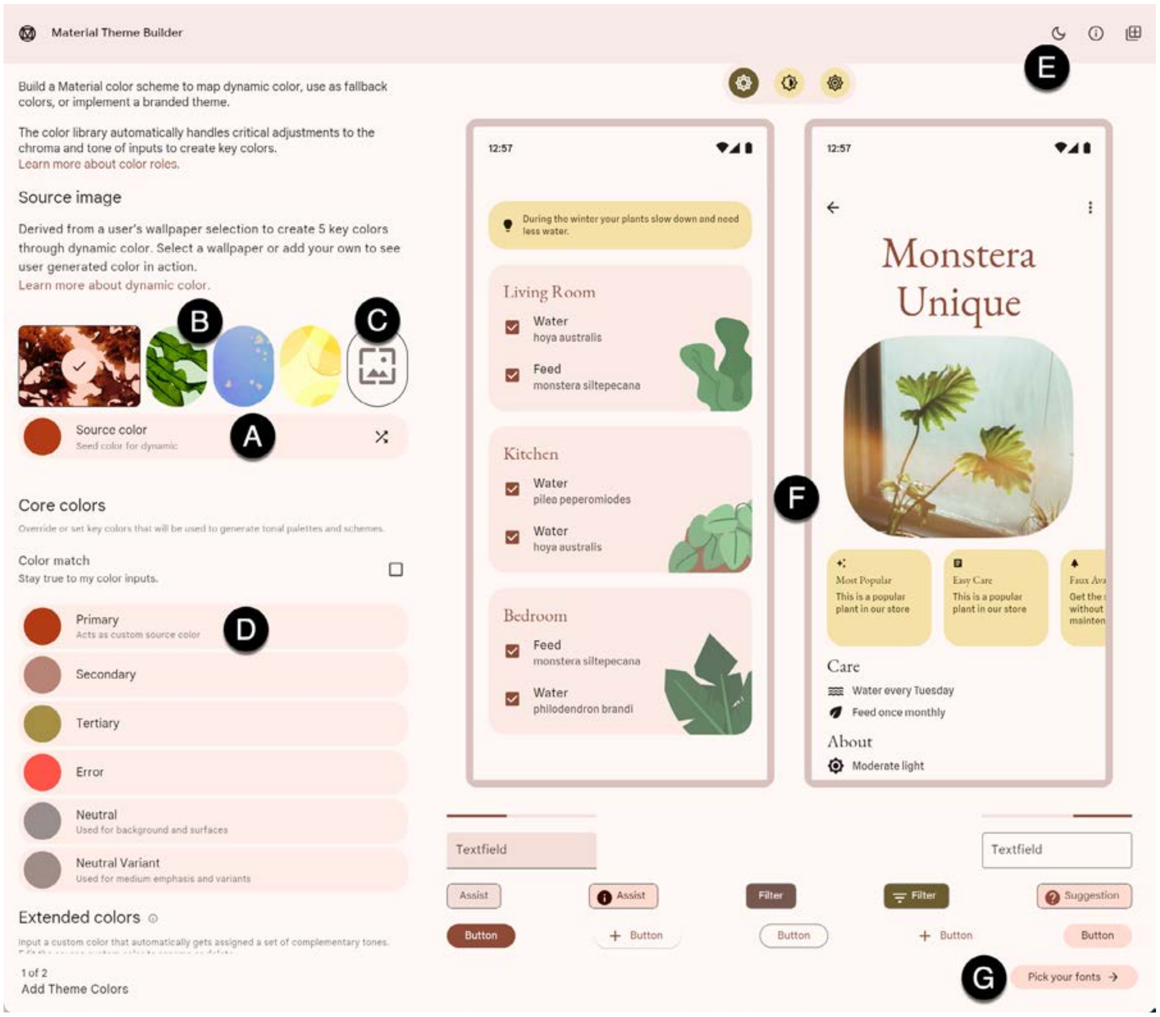
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
.
.
    <color name="my_bright_primary">#FC0505</color>
</resources>
```

91.4 Building a custom theme

As we have seen, the coding work in implementing a theme is relatively simple. The difficult part, however, is often choosing complementary colors to make up the theme. Fortunately, Google has developed a tool that makes it easy to design custom color and font themes for your apps. This tool is called the Material Theme Builder and is available at:

https://m3.material.io/theme-builder

Figure 91-2 shows a recent version of the Theme Builder. However, Google redesigns the builder every few weeks, so it is possible that the user interface you see will not match the current design:


Material Theme Builder

Build a Material color scheme to map dynamic color, use as fallback colors, or implement a branded theme.

The color library automatically handles critical adjustments to the chroma and tone of inputs to create key colors.
Learn more about color roles.

Source image

Derived from a user's wallpaper selection to create 5 key colors through dynamic color. Select a wallpaper or add your own to see user generated color in action.
Learn more about dynamic color.

Source color
Seed color for dynamic

Core colors
Override or set key colors that will be used to generate tonal palettes and schemes.

Color match
Stay true to my color inputs.

Primary
Acts as custom source color

Secondary

Tertiary

Error

Neutral
Used for background and surfaces

Neutral Variant
Used for medium emphasis and variants

Extended colors
Input a custom color that automatically gets assigned a set of complementary tones.

1 of 2
Add Theme Colors

12:57

During the winter your plants slow down and need less water.

Living Room
Water
hoya australis
Feed
monstera siltepecana

Kitchen
Water
pilea peperomiodes
Water
hoya australis

Bedroom
Feed
monstera siltepecana
Water
philodendron brandi

12:57

Monstera Unique

Most Popular
This is a popular plant in our store

Easy Care
This is a popular plant in our store

Care
Water every Tuesday
Feed once monthly

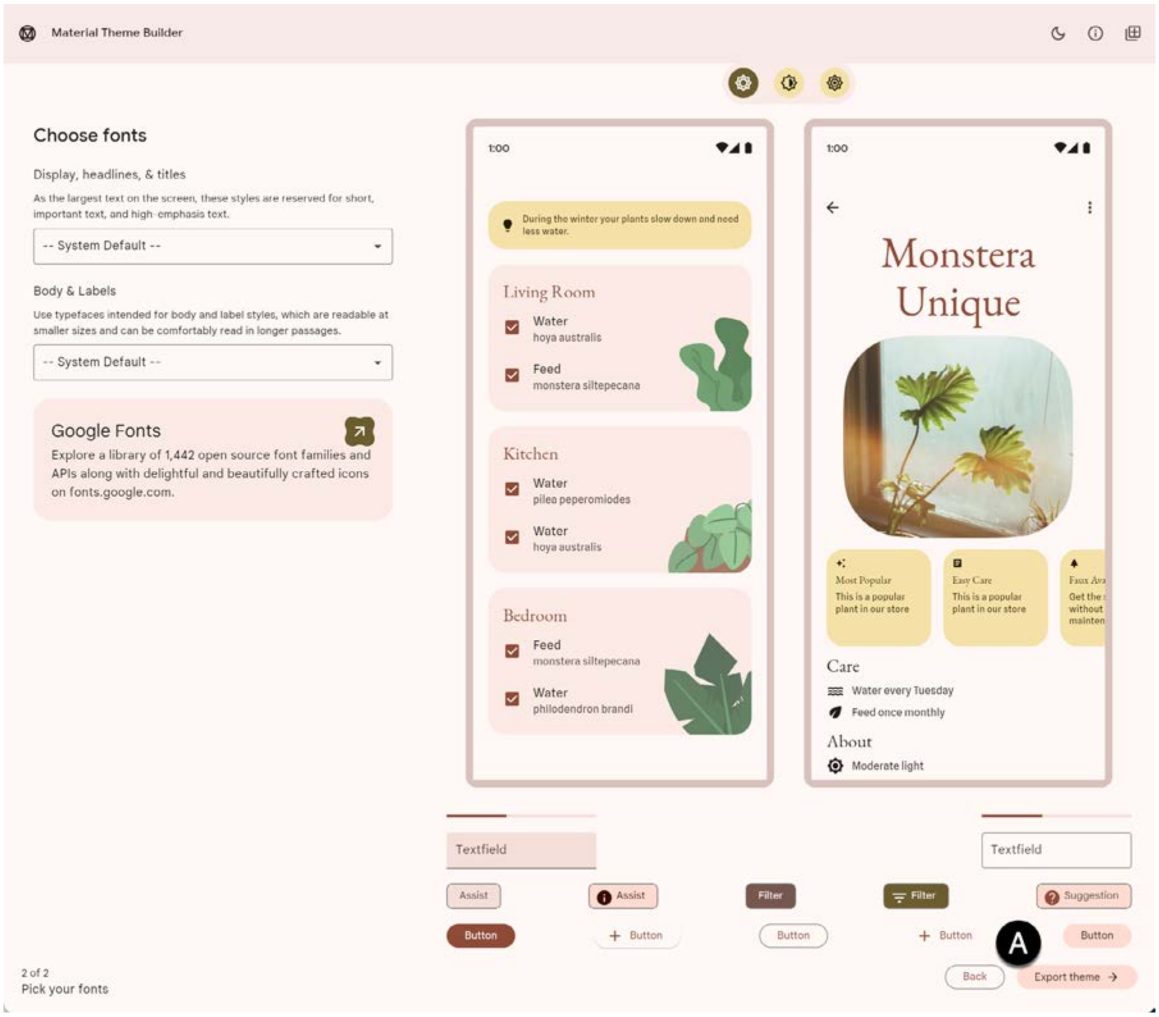
About
Moderate light

Figure 91-2

A custom Material color theme can be generated by providing a source color of your own choosing (marked A) in the figure above, selecting a standard wallpaper (B), uploading a source image (C), or making individual core color selections (D). The button marked E previews the color scheme in light and dark modes.

The area marked F displays example app interfaces, light and dark color scheme charts, and widgets that update to preview your color selections. Since the panel is longer than the typical browser window, you must scroll down to see all the information.

Once the theme colors have been defined, clicking on the Pick your fonts button (G) button will display the screen shown in Figure 91-3, where you can select the fonts to be used by your app:



As the largest text on the screen, these styles are reserved for short, important text, and high-emphasis text.

Use typefaces intended for body and label styles, which are readable at smaller sizes and can be comfortably read in longer passages.

Explore a library of 1,442 open source font families and APIs along with delightful and beautifully crafted icons on fonts.google.com.

During the winter your plants slow down and need less water.

This is a popular plant in our store

Figure 91-3

To incorporate the theme into your design, click the *Export theme* button (marked A above) and select the *Android Views (XML)* option. Once downloaded, the *themes.xml* and *colors.xml* files can replace the existing files in your project. Note that the theme name in the exported *themes.xml* files must be changed to match your project.

91.5 Summary

Material Design provides guidelines and components defining how Android apps appear. Individual branding can be applied to an app by designing themes that specify the colors, fonts, and shapes used when displaying the app. Google recently introduced Material Design 3, which replaces Material Design 2 and supports the new features of Material You, including dynamic colors. Google also provides the Material Theme Builder for designing your own themes, which eases the task of choosing complementary theme colors. Once this tool has been used to design a theme, the corresponding files can be exported and used within an Android Studio project.

Index