

73. Handling Different Android Devices and Displays

Before being made available for purchase on the Google Play App Store, an application must first be submitted to the portal for review and approval. One of the most important steps to take before submitting an application is to decide which Android device models the application is intended to support and, more importantly, that the application runs without issue on those devices.

This chapter will cover some of the areas to consider when making sure that an application runs on the widest possible range of Android devices.

73.1 Handling Different Device Displays

Android devices come in a variety of different screen sizes and resolutions. The ideal solution is to design the user interface of your application so that it appears correctly on the widest possible range of devices. The best way to achieve this is to design the user interface using layout managers that do not rely on absolute positioning (i.e. specific X and Y coordinates) such as the `ConstraintLayout` so that views are positioned relative to both the size of the display and each other.

Similarly, avoid using specific width and height properties wherever possible. When such properties are unavoidable, always use *density-independent (dp)* values as these are automatically scaled to match the device display at application runtime.

Having designed the user interface, be sure to test it on each device on which it is intended to be supported. In the absence of the physical device hardware, use the emulator templates, wherever possible, to test on the widest possible range of devices.

In the event that it is not possible to design the user interface such that a single design will work on all Android devices, another option is to provide a different layout for each display.

73.2 Creating a Layout for each Display Size

The ideal solution to the multiple display problem is to design user interface layouts that adapt to the display size of the device on which the application is running. This, for example, has the advantage of having only one layout to manage when modifying the application. Inevitably, however, there will be situations where this ideal is unachievable given the vast difference in screen size between a phone and a tablet. Another option is to provide different layouts, each tailored to a specific display category. This involves identifying the *smallest width* qualifier value of each display and creating an XML layout file for each one. The smallest width value of a display indicates the minimum width of that display measured in dp units.

Display-specific layouts are implemented by creating additional sub-directories under the *res* directory of a project. The naming convention for these folders is:

layout-*<smallest-width>*

For example, layout resource folders for a range of devices might be configured as follows:

- *res/layout* – The default layout file
- *res/layout-sw200dp*

- *res/layout-sw600dp*
- *res/layout-sw800dp*

Alternatively, more general categories can be created by targeting *small*, *normal*, *large* and *xlarge* displays:

- *res/layout* – The default layout file
- *res/layout-small*
- *res/layout-normal*
- *res/layout-large*
- *res/layout-xlarge*
- *res/layout-land*

Each folder must, in turn, contain a copy of the layout XML file adapted for the corresponding display, all of which must have matching file names. Once implemented, the Android runtime system will automatically select the correct layout file to display to the user to match the device display.

73.3 Creating Layout Variants in Android Studio

Android Studio makes it easy to add additional layout size variants using the *Layout Variants* button located in the Layout Editor toolbar as highlighted in Figure 72-1:

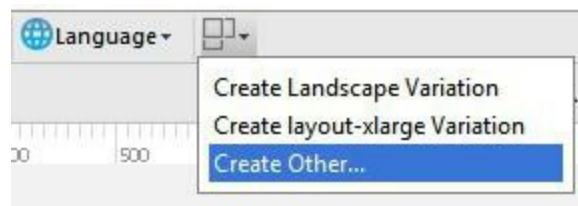


Figure 73-1

When selected, the menu provides options to create either a preconfigured landscape (*res/layout-land*) or *xlarge* (*res/layout-xlarge*) variants. Alternatively, the *Create Other...* menu option may be used to create variants for other sizes. To create a custom variant, select the *Size* qualifier in the *Select Resource Directory* dialog, click on the button displaying the '>>' character sequence and then make a selection from the *Screen size* drop-down menu:

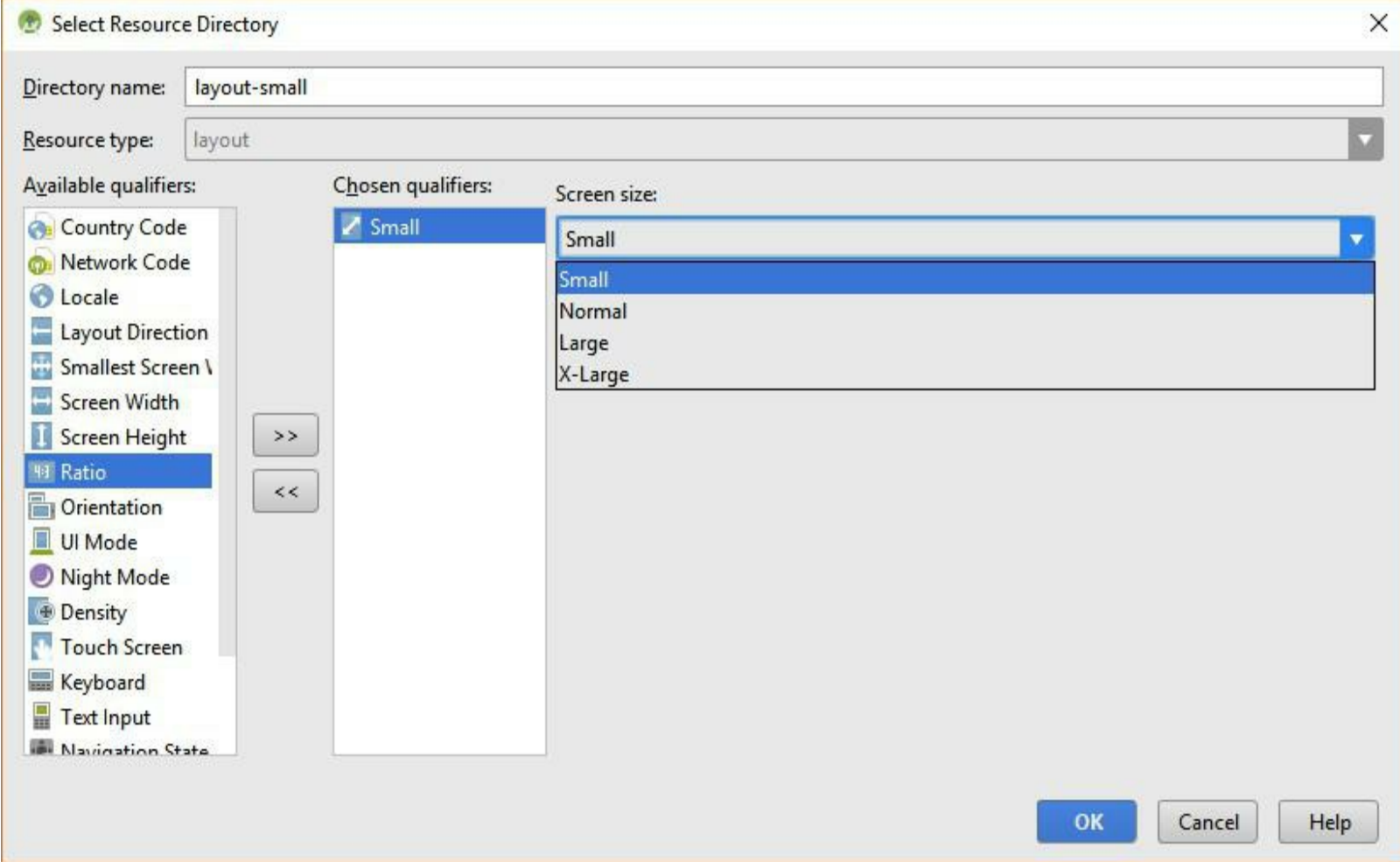


Figure 73-2

At any time during the layout design process, use the Layout Variants menu to switch to one of the different variants to see how the user interface will appear when running on a device with the corresponding screen size:

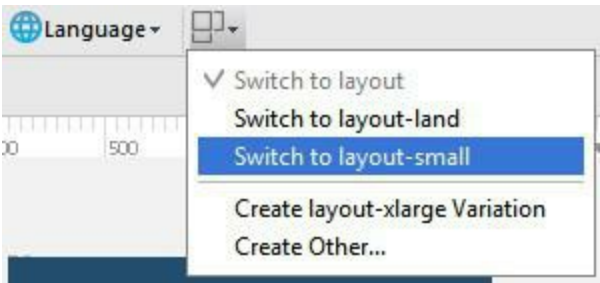


Figure 73-3

73.4 Providing Different Images

User interface layouts are not the only area of concern when adapting an application for different screen densities, dimensions and aspect ratios. Another area to pay attention to is that of images. An image that appears correctly scaled on a large tablet screen, for example, might not appear correctly scaled on a smaller phone based device. As with layouts, however, multiple sets of images can be bundled with the application, each tailored to a specific display. This can once again be achieved by referencing the smallest width value. In this case, *drawable* folders need to be created in the *res* directory. For example:

- *res/drawable* – The default image folder
- *res/drawable-sw200dp*

- *res/drawable-sw600dp*
- *res/drawable-sw800dp*

Having created the folders, simply place the display specific versions of the images into the corresponding folder, using the same name for each of the images.

Alternatively, the images may be categorized into broader display densities using the following directories based on the pixel density of the display:

- *res/drawable-ldpi* - Images for low density screens (approx. 120 dpi)
- *res/drawable-mdpi* – Images for medium-density screens (approx. 160 dpi)
- *res/drawable-hdpi* – Images for high-density screens (approx. 240 dpi)
- *res/drawable-xhdpi* – Images for extra high-density screens (approx. 320 dpi)
- *res/drawable-tvdpi* – Images for displays between medium and high density (approx. 213 dpi)
- *res/drawable-nodpi* – Images that must not be scaled by the system

73.5 Checking for Hardware Support

By now, it should be apparent that not all Android devices were created equal. An application that makes use of specific hardware features (such as a microphone or camera) should include code to gracefully handle the absence of that hardware. This typically involves performing a check to find out if the hardware feature is missing, and subsequently reporting to the user that the corresponding application functionality will not be available.

The following method can be used to check for the presence of a microphone:

```
protected boolean hasMicrophone() {
    PackageManager pmanager = this.getPackageManager();
    return pmanager.hasSystemFeature(
        PackageManager.FEATURE_MICROPHONE);
}
```

Similarly, the following method is useful for checking for the presence of a front facing camera:

```
private boolean hasCamera() {
    if (getPackageManager().hasSystemFeature(
        PackageManager.FEATURE_CAMERA_FRONT)) {
        return true;
    } else {
        return false;
    }
}
```

73.6 Providing Device Specific Application Binaries

Even with the best of intentions, there will inevitably be situations where it is not possible to target all Android devices within a single application (though Google certainly encourages developers to target as many devices as possible within a single application binary package). In this situation, the application submission process allows multiple application binaries to be uploaded for a single application. Each binary is then configured to indicate to Google the devices with which the binary is configured to work. When a user subsequently purchases the application, Google ensures that the

correct binary is downloaded for the user's device.

It is also important to be aware that it may not always make sense to try to provide support for every Android device model. There is little point, for example, in making an application that relies heavily on a specific hardware feature available on devices that lack that specific hardware. These requirements can be defined using Google Play Filters as outlined at:

<http://developer.android.com/google/play/filters.html>

73.7 Summary

There is more to completing an Android application than making sure it works on a single device model. Before an application is submitted to the Google Play Developer Console, it should first be tested on as wide a range of display sizes as possible. This includes making sure that the user interface layouts and images scale correctly for each display variation and taking steps to ensure that the application gracefully handles the absence of certain hardware features. It is also possible to submit to the developer console a different application binary for specific Android models, or to state that a particular application simply does not support certain Android devices.

74. Signing and Preparing an Android Application for Release

Once the development work on an Android application is complete and it has been tested on a wide range of Android devices, the next step is to prepare the application for submission to the Google Play App Store. Before submission can take place, however, the application must be packaged for release and signed with a private key. This chapter will work through the steps involved in obtaining a private key and preparing the application package for release.

74.1 The Release Preparation Process

Up until this point in the book, we have been building application projects in a mode suitable for testing and debugging. Building an application package for release to customers via the Google Play store, on the other hand, requires that some additional steps be taken. The first requirement is that the application be compiled in *release mode* instead of *debug mode*. Secondly, the application must be signed with a private key that uniquely identifies you as the application's developer. Finally, the application package must be *aligned*. This is simply a process by which some data files in the application package are formatted with a certain byte alignment to improve performance.

While each of these tasks can be performed outside of the Android Studio environment, the procedures can more easily be performed using the Android Studio build mechanism as outlined in the remainder of this chapter.

74.2 Changing the Build Variant

The first step in the process of generating a signed application APK file involves changing the build variant for the project from debug to release. This is achieved using the *Build Variants* tool window which can be accessed from the tool window quick access menu (located in the bottom left-hand corner of the Android Studio main window as shown in Figure 74-1).

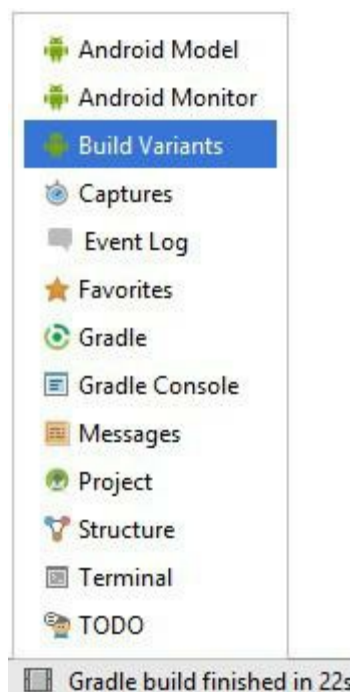


Figure 74-1
WOW! eBook
www.wowebook.org

Once the Build Variants tool window is displayed, change the Build Variant settings for all the modules listed from *debug* to *release*:

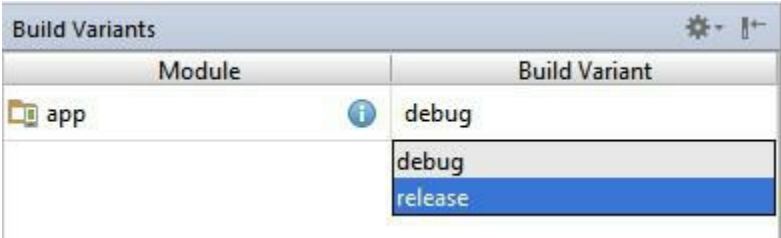


Figure 74-2

The project is now configured to build in release mode. The next step is to configure signing key information for use when generating the signed application package.

74.3 Enabling ProGuard

When generating an application package, the option is available to use ProGuard during the package creation process. ProGuard performs a series of optimization and verification tasks that result in smaller and more efficient byte code. In order to use ProGuard, it is necessary to enable this feature within the Project Structure settings prior to generating the APK file.

The steps to enable ProGuard are as follows:

1. Display the Project Structure dialog (*File -> Project Structure*).
2. Select the "app" module in the far left panel.
3. Select the "Build Types" tab in the main panel and the "release" entry from the middle panel.
4. Change the "Minify Enabled" option from "false" to "true" and click on *OK*.
5. Follow the steps to create a keystore file and build the release APK file.

74.4 Creating a Keystore File

To create a keystore file, select the *Build -> Generate Signed APK...* menu option to display the Generate Signed APK Wizard dialog as shown in Figure 74-3:



Figure 74-3

In the event that you have an existing release key store file, click on the *Choose existing...* button and

navigate to and select the file. In the event that you have yet to create a keystore file, click on the *Create new...* button to display the *New Key Store* dialog (Figure 74-4). Click on the button to the right of the Key store path field and navigate to a suitable location on your file system, enter a name for the keystore file (for example, *release.keystore.jks*) and click on the OK button.

The New Key Store dialog is divided into two sections. The top section relates to the keystore file. In this section, enter a strong password with which to protect the keystore file into both the *Password* and *Confirm* fields. The lower section of the dialog relates to the release key that will be stored in the key store file.

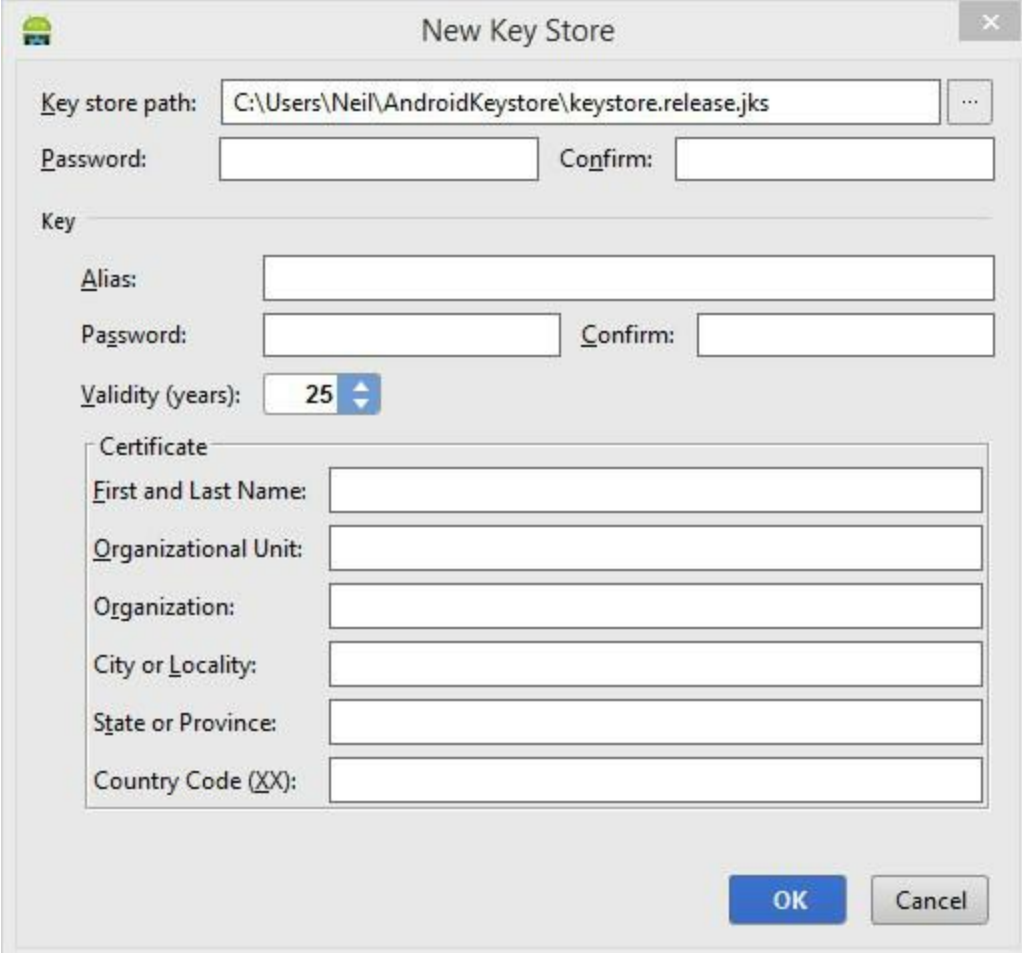


Figure 74-4

74.5 Generating a Private Key

The next step is to generate a new private key which will be used to sign the application package. Within the *Key* section of the New Key Store dialog, enter the following details:

- An alias by which the key will be referenced. This can be any sequence of characters, though only the first 8 are used by the system.
- A suitably strong password to protect the key.
- The number of years for which the key is to be valid (Google recommends a duration in excess of 25 years).

In addition, information must be provided for at least one of the remaining fields (for example, your first and last name, or organization name).

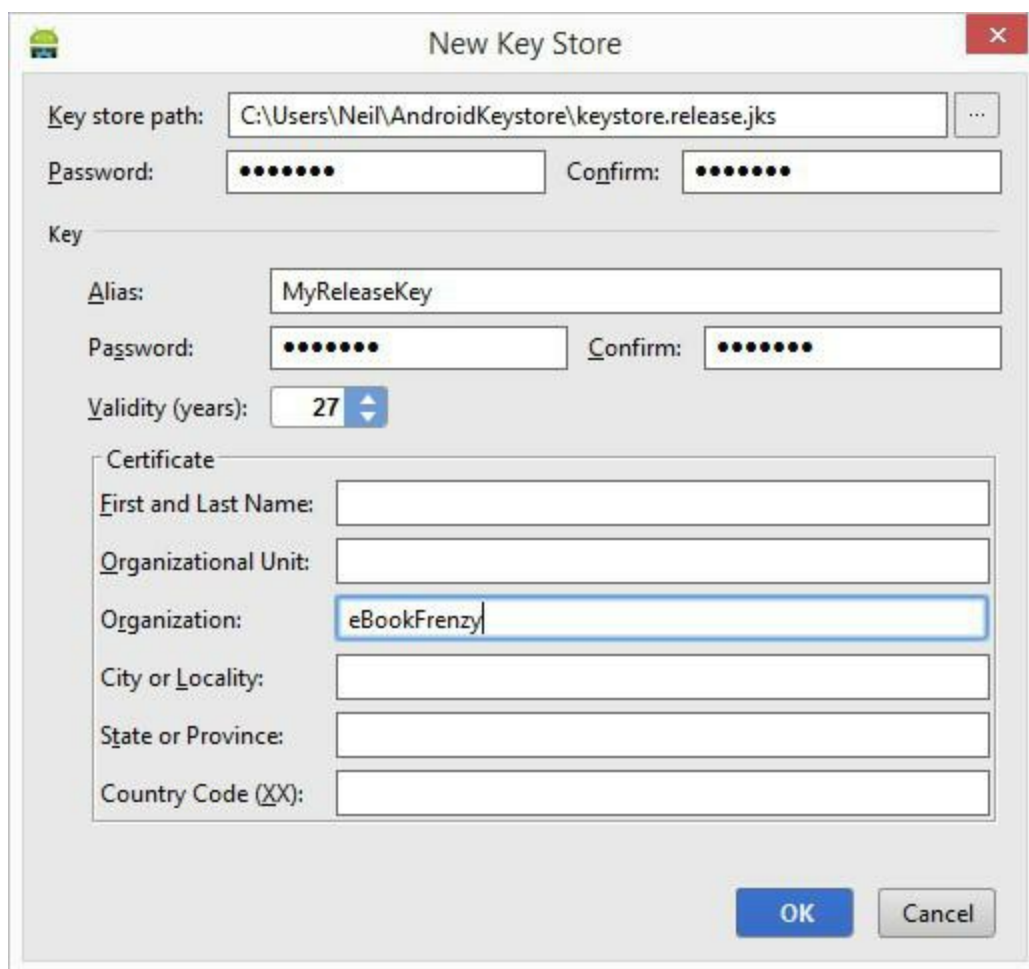


Figure 74-5

Once the information has been entered, click on the *OK* button to proceed with the package creation.

74.6 Creating the Application APK File

The next task to be performed is to instruct Android Studio to build the application APK package file in release mode and then sign it with the newly created private key. At this point the *Generate Signed APK Wizard* dialog should still be displayed with the keystore path, passwords and key alias fields populated with information:



Figure 74-6

Assuming that the settings are correct, click on the *Next* button to proceed to the APK generation screen (Figure 74-7). Within this screen, review the *Destination APK path*: setting to verify that the location into which the APK file will be generated is acceptable. In the event that another location is preferred, click on the button to the right of the text field and navigate to the desired file system location.

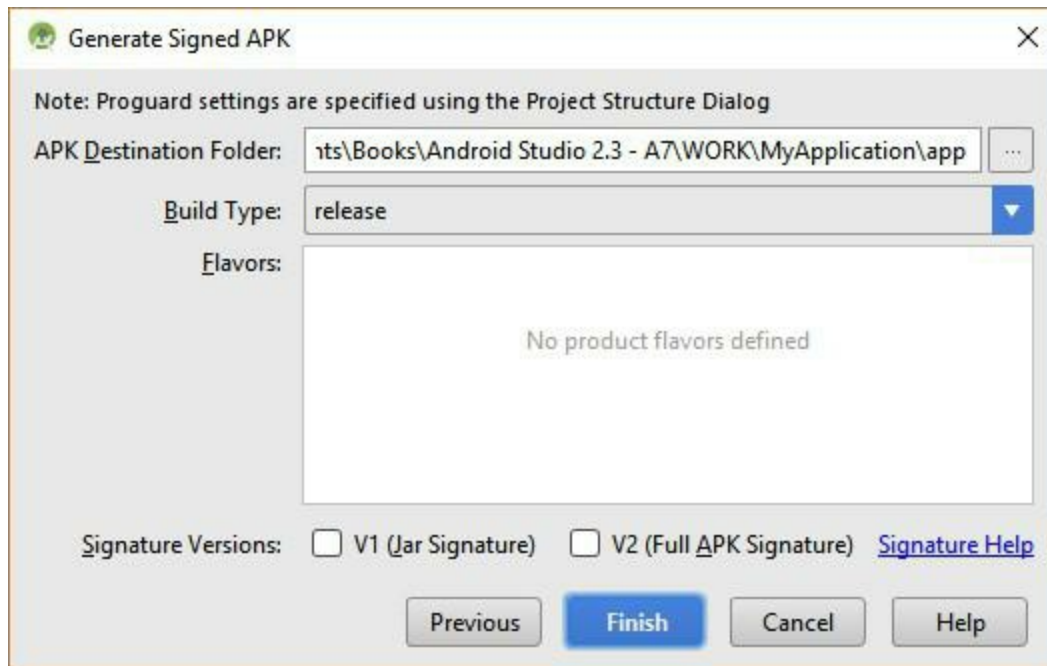


Figure 74-7

Two signature options are provided for selection within the APK generation dialog. The recommended option is V2 (Full APK Signature). This provides additional security to protect the APK from malicious alteration together with faster app installation times. If problems occur when using the V2 option, repeat the generation process using the V1 option.

The Gradle system will now compile the application in release mode. Once the build is complete, a dialog will appear providing the option to open the folder containing the APK file in an explorer window:



Figure 74-8

At this point the application is ready to be submitted to the Google Play store.

The private key generated as part of this process should be used when signing and releasing future applications and, as such, should be kept in a safe place and securely backed up.

The final step in the process of bringing an Android application to market involves submitting it to the Google Play Developer Console. Once submitted, the application will be available for download from the Google Play App Store.

74.7 Register for a Google Play Developer Console Account

The first step in the application submission process is to create a Google Play Developer Console account. For more information, visit www.wowebook.org.

account. To do so, navigate to <https://play.google.com/apps/publish/signup/> and follow the instructions to complete the registration process. Note that there is a one-time \$25 fee to register. Once an application goes on sale, Google will keep 30% of all revenues associated with the application.

Once the account has been created, the next step is to gather together information about the application. In order to bring your application to market, the following information will be required:

- **Title** – The title of the application.
- **Short Description** - Up to 80 words describing the application.
- **Description** – Up to 4000 words describing the application.
- **Screenshots** – Up to 8 screenshots of your application running (a minimum of two is required). Google recommends submitted screenshots of the application running on a 7” or 10” tablet.
- **Language** – The language of the application (the default is US English).
- **Promotional Text** – The text that will be used when your application appears in special promotional features within the Google Play environment.
- **Application Type** – Whether your application is considered to be a *game* or an *application*.
- **Category** – The category that best describes your application (for example finance, health and fitness, education, sports, etc.).
- **Locations** – The geographical locations into which you wish your application to be made available for purchase.
- **Contact Details** – Methods by which users may contact you for support relating to the application. Options include web, email and phone.
- **Pricing & Distribution** – Information about the price of the application and the geographical locations where it is to be marketed and sold.

Having collected the above information and prepared the application package file for release, simply follow the steps in the Google Play Developer Console to submit the application for testing and sale.

74.8 Uploading New APK Versions to the Google Play Developer Console

The first APK file uploaded for your application will invariably have a version code of 1. If an attempt is made to upload another APK file with the same version code number, the console will reject the file with the following error:

```
You need to use a different version code for your APK because you already  
have one with version code 1.
```

To resolve this problem, the version code embedded into the APK needs to be increased. This is performed in the *module* level build.gradle file of the project, shown highlighted in Figure 74-9. It is important to note that this is not the *top* level build.gradle file positioned lower in the project hierarchy listing:

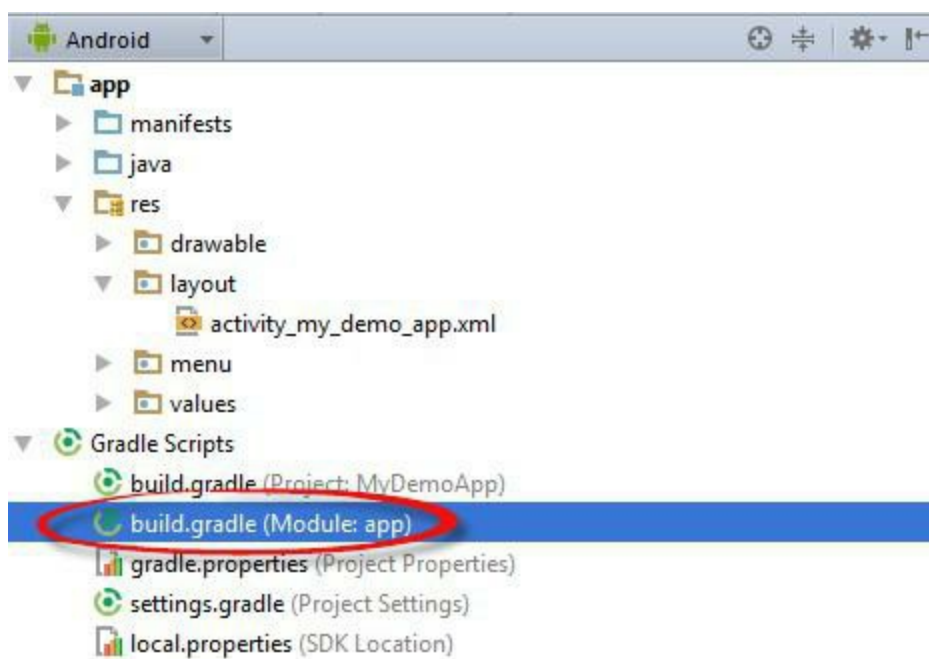


Figure 74-9

By default, this file will typically read as follows:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.ebookfrenzy.myapplication"
        minSdkVersion 14
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner
        "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
            'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    androidTestCompile('com.android.support.test.espresso:espresso-
core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.1.0'
    compile 'com.android.support.constraint:constraint-layout:1.0.0-beta4'
    testCompile 'junit:junit:4.12'
}
```

To change the version code, simply change the number declared next to *versionCode*. To also change

the version number displayed to users of your application, change the *versionName* string. For example:

```
versionCode 2
versionName "2.0"
```

Having made these changes, rebuild the APK file and perform the upload again.

74.9 Analyzing the APK File

Android Studio provides the ability to analyze the content of an APK file. This can be useful, for example, when attempting to find out why the APK file is larger than expected or to review the class structure of the application's dex file.

To analyze an APK file, select the Android Studio *Build -> Analyze APK...* menu option and navigate to and choose the APK file to be reviewed. Once loaded into the tool, information will be displayed about the raw and download size of the package together with a listing the of file structure of the package as illustrated in Figure 74-10:

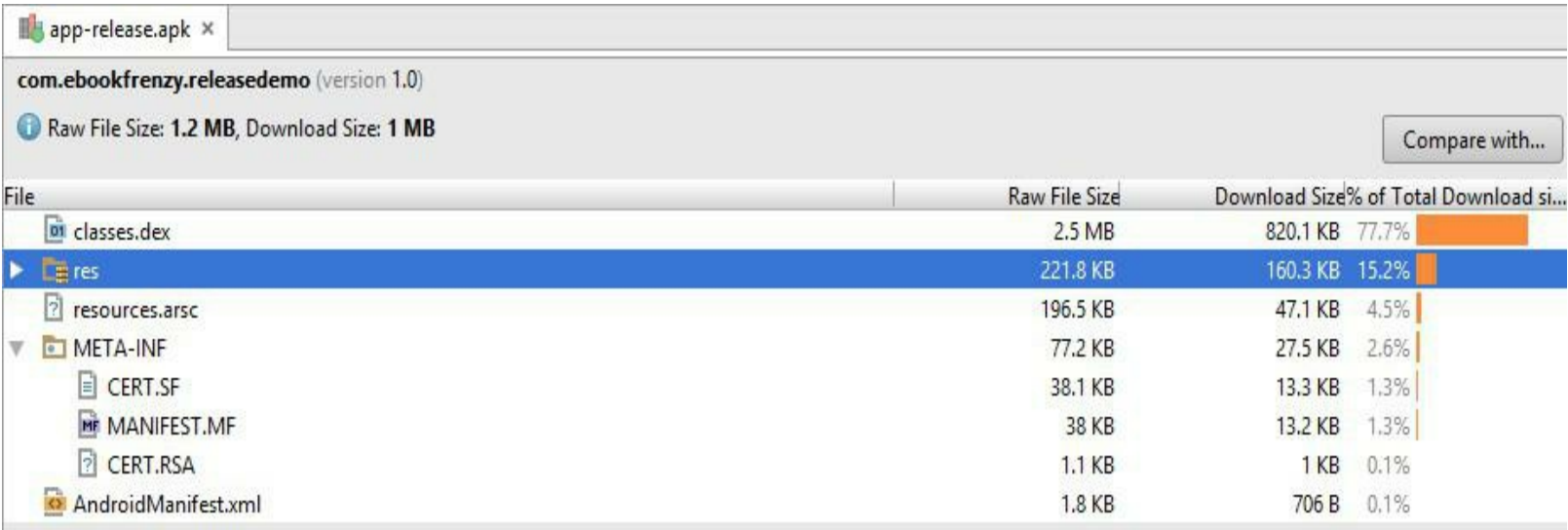


Figure 74-10

Selecting the *classes.dex* file will display the class structure of the file in the lower panel. Within this panel, details of the individual classes may be explored down to the level of the methods within a class:

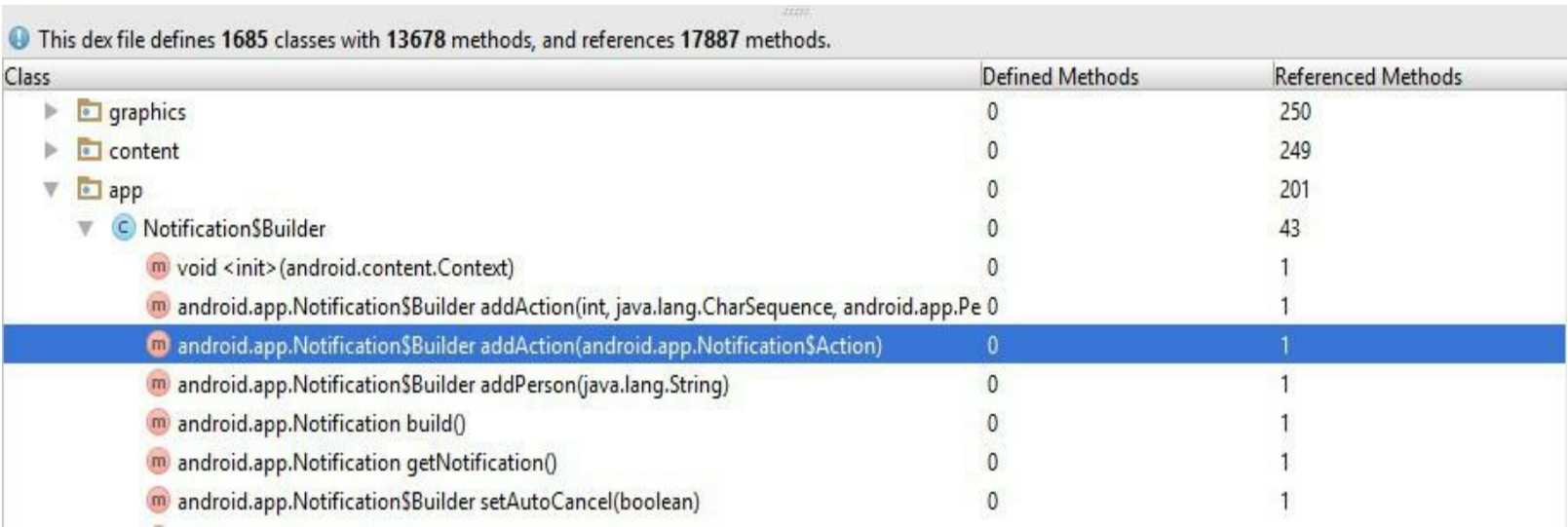


Figure 74-11

Similarly, selecting a resource or image file within the file list will display the file content within the lower panel. In Figure 74-12, for example, an image file has been selected from a drawable folder within the package file list:

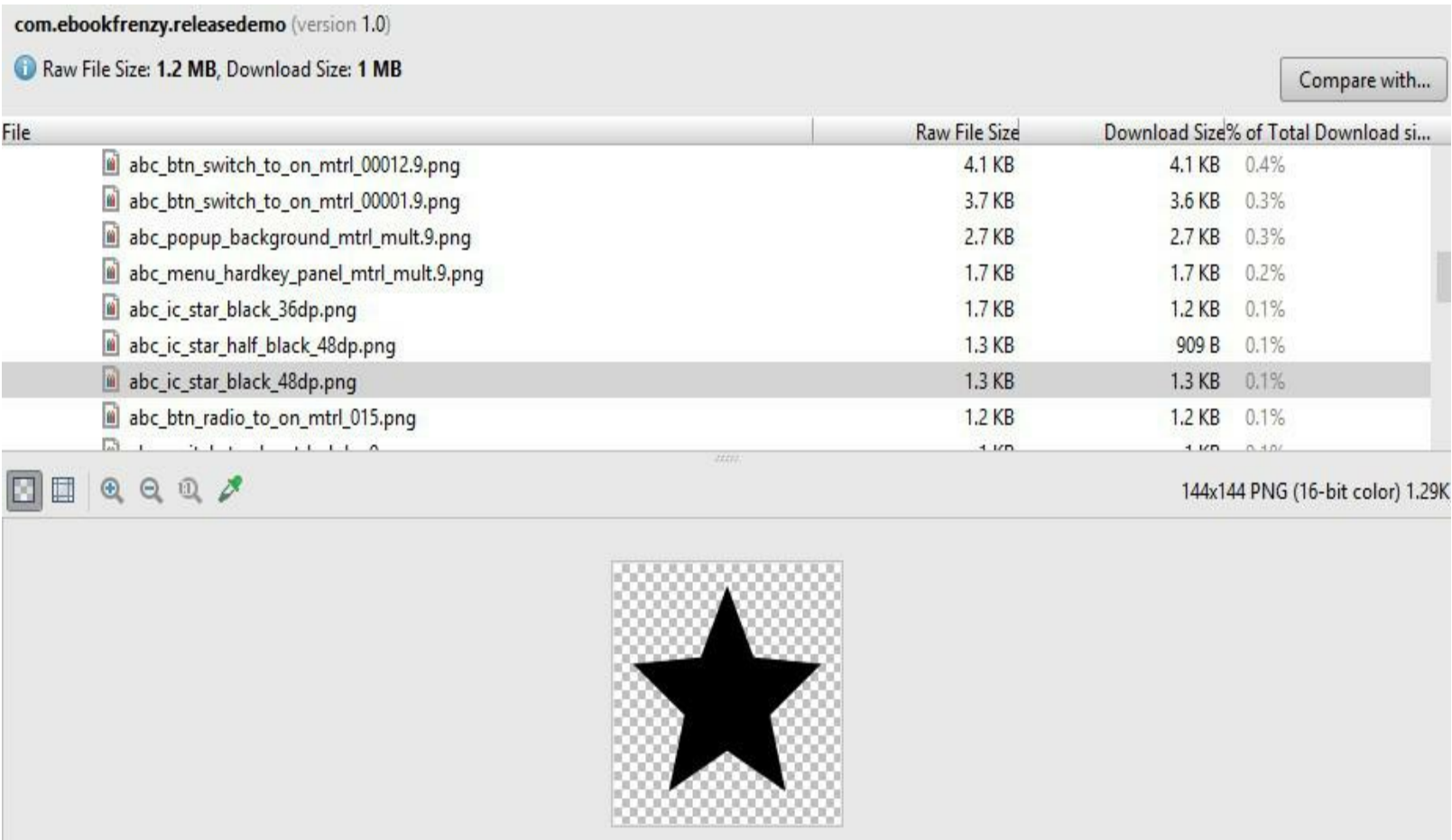


Figure 74-12

The size differences between two APK files may be reviewed by clicking on the *Compare with...* button and selecting a second APK file.

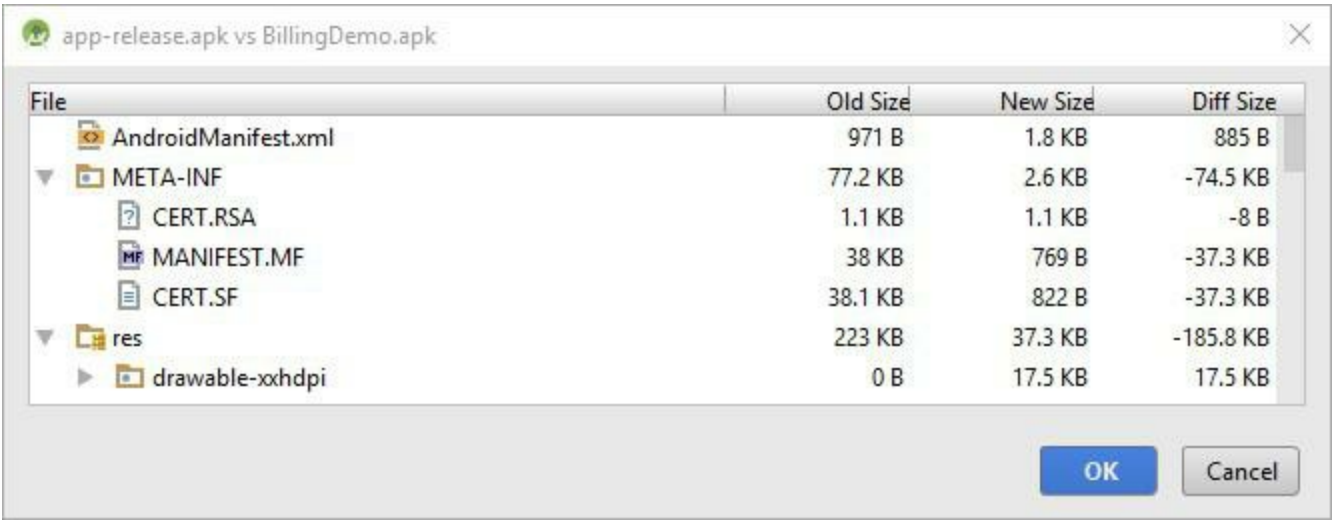


Figure 74-13

74.10 Summary

Before an application can be submitted to the Google Play store, it must first be built in release mode, signed with a private certificate and the resulting APK package file subjected to a process referred to as *alignment*. As outlined in this chapter, all of these steps can be performed with relative ease

through the use of the Android Studio build system.