76. An Overview of Gradle in Android Studio

Up until this point it has, for the most part, been taken for granted that Android Studio will take the necessary steps to compile and run the application projects that have been created. Android Studio has been achieving this in the background using a system known as *Gradle*.

It is now time to look at how Gradle is used to compile and package together the various elements of an application project and to begin exploring how to configure this system when more advanced requirements are needed in terms of building projects in Android Studio.

76.1 An Overview of Gradle

Gradle is an automated build toolkit that allows the way in which projects are built to be configured and managed through a set of build configuration files. This includes defining how a project is to be built, what dependencies need to be fulfilled for the project to build successfully and what the end result (or results) of the build process should be.

The strength of Gradle lies in the flexibility that it provides to the developer. The Gradle system is a self-contained, command-line based environment that can be integrated into other environments through the use of plug-ins. In the case of Android Studio, Gradle integration is provided through the appropriately named Android Studio Plug-in.

Although the Android Studio Plug-in allows Gradle tasks to be initiated and managed from within Android Studio, the Gradle command-line wrapper can still be used to build Android Studio based projects, including on systems on which Android Studio is not installed.

The configuration rules to build a project are declared in Gradle build files and scripts based on the Groovy programming language.

76.2 Gradle and Android Studio

Gradle brings a number of powerful features to building Android application projects. Some of the key features are as follows:

76.2.1 Sensible Defaults

Gradle implements a concept referred to as *convention over configuration*. This simply means that Gradle has a pre-defined set of sensible default configuration settings that will be used unless they are overridden by settings in the build files. This means that builds can be performed with the minimum of configuration required by the developer. Changes to the build files are only needed when the default configuration does not meet your build needs.

76.2.2 Dependencies

Another key area of Gradle functionality is that of dependencies. Consider, for example, a module within an Android Studio project which triggers an intent to load another module in the project. The first module has, in effect, a dependency on the second module since the application will fail to build if the second module cannot be located and launched at runtime. This dependency can be declared in the Gradle build file for the first module so that the second module is included in the application build, or an error flagged in the event the second module cannot be found or built. Other examples of dependencies are libraries and JAR files on which the project depends in order to compile and run.

Gradle dependencies can be categorized as *local* or *remote.* A local dependency references an item that is present on the local file system of the computer system on which the build is being performed. A remote dependency refers to an item that is present on a remote server (typically referred to as a *repository*).

Remote dependencies are handled for Android Studio projects using another project management tool named *Maven*. If a remote dependency is declared in a Gradle build file using Maven syntax then the dependency will be downloaded automatically from the designated repository and included in the build process. The following dependency declaration, for example, causes the ConstraintLayout library to be added to the project from the Google repository:

```
compile 'com.android.support.constraint:constraint-layout:1.0.0'
```

76.2.3 Build Variants

In addition to dependencies, Gradle also provides *build variant* support for Android Studio projects. This allows multiple variations of an application to be built from a single project. Android runs on many different devices encompassing a range of processor types and screen sizes. In order to target as wide a range of device types and sizes as possible it will often be necessary to build a number of different variants of an application (for example, one with a user interface for phones and another for tablet sized screens). Through the use of Gradle, this is now possible in Android Studio.

76.2.4 Manifest Entries

Each Android Studio project has associated with it an AndroidManifest.xml file containing configuration details about the application. A number of manifest entries can be specified in Gradle build files which are then auto-generated into the manifest file when the project is built. This capability is complementary to the build variants feature, allowing elements such as the application version number, application ID and SDK version information to be configured differently for each build variant.

76.2.5 APK Signing

The chapter entitled *Signing and Preparing an Android Application for Release* covered the creation of a signed release APK file using the Android Studio environment. It is also possible to include the signing information entered through the Android Studio user interface within a Gradle build file so that signed APK files can be generated from the command-line.

76.2.6 ProGuard Support

ProGuard is a tool included with Android Studio that optimizes, shrinks and obfuscates Java byte code to make it more efficient and harder to reverse engineer (the method by which the logic of an application can be identified by others through analysis of the compiled Java byte code). The Gradle build files provide the ability to control whether or not ProGuard is run on your application when it is built.

76.3 The Top-level Gradle Build File

A completed Android Studio project contains everything needed to build an Android application and consists of modules, libraries, manifest files and Gradle build files.

Each project contains one top-level Gradle build file. This file is listed as *build.gradle (Project:*

<project name>) and can be found in the project tool window as highlighted in Figure 76-1:

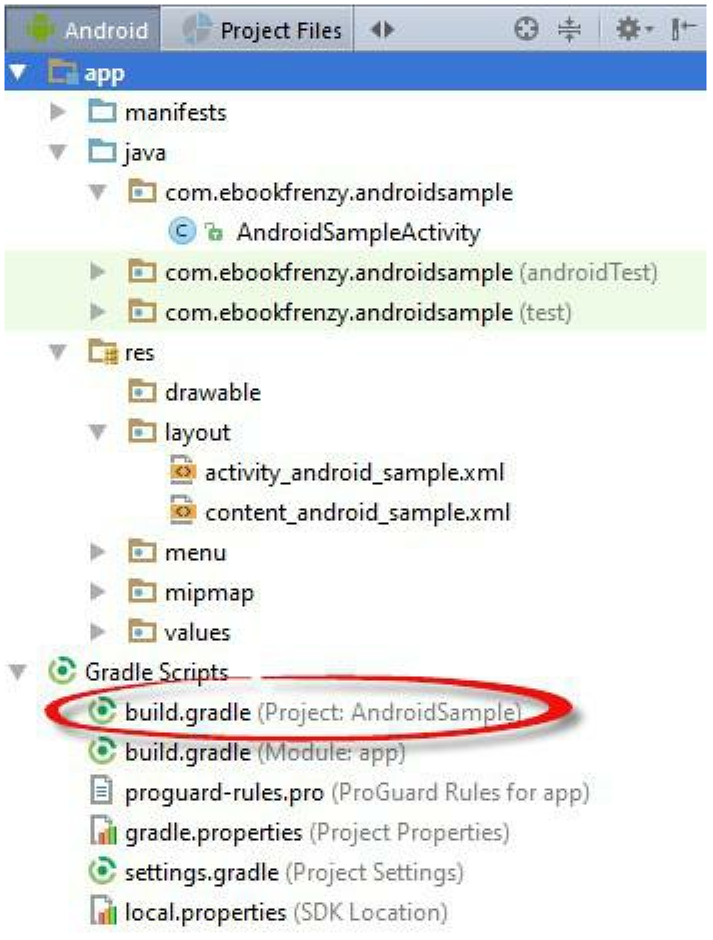
Android
Project Files
app
manifests
java
com.ebookfrenzy.androidsample
AndroidSampleActivity
com.ebookfrenzy.androidsample (androidTest)
com.ebookfrenzy.androidsample (test)
res
drawable
layout
activity_android_sample.xml
content_android_sample.xml
menu
mipmap
values
Gradle Scripts
build.gradle (Project: AndroidSample)
build.gradle (Module: app)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)

Figure 76-1

By default, the contents of the top level Gradle build file read as follows:

```
// Top-level build file where you can add configuration options common to
all sub-projects/modules.

buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.0'

        // NOTE: Do not place your application dependencies here; they
belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

As it stands all the file does is declare that remote libraries are to be obtained using the jcenter repository and that builds are dependent on the Android plugin for Gradle. In most situations it is not necessary to make any changes to this build file.

76.4 **Module Level Gradle Build Files**

An Android Studio application project is made up of one or more modules. Take, for example, a hypothetical application project named GradleDemo which contains two modules named Module1 and Module2 respectively. In this scenario, each of the modules will require its own Gradle build file. In terms of the project structure, these would be located as follows:

- Module1/build.gradle
- Module2/build.gradle

By default, the Module1 build.gradle file would resemble that of the following listing:

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

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"

    defaultConfig {
        applicationId "com.ebookfrenzy.module1"
        minSdkVersion 8
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:appcompat-v7:25.2.0'
    compile 'com.android.support:design:25.2.0'
}
```

As is evident from the file content, the build file begins by declaring the use of the Gradle Android plug-in:

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

The *android* section of the file then states the version of both the SDK and the Android Build Tools that are to be used when building Module1.

```
android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
```

The items declared in the defaultConfig section define elements that are to be generated into the module's *AndroidManifest.xml* file during the build. These settings, which may be modified in the build file, are taken from the settings entered within Android Studio when the module was first created:

```
defaultConfig {
    applicationId "com.ebookfrenzy.gradledemo.module1"
    minSdkVersion 8
    targetSdkVersion 25
    versionCode 1
    versionName "1.0"
}
```

The buildTypes section contains instructions on whether and how to run ProGuard on the APK file when a release version of the application is built:

```
buildTypes {
    release {
        runProguard false
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
                      'proguard-rules.pro'
    }
}
```

As currently configured, ProGuard will not be run when Module1 is built. To enable ProGuard, the *runProguard* entry needs to be changed from *false* to *true*. The *proguard-rules.pro* file can be found in the module directory of the project. Changes made to this file override the default settings in the *proguard-android.txt* file which is located on the Android SDK installation directory under *sdk/tools/proguard*.

Since no debug buildType is declared in this file, the defaults will be used (built without ProGuard, signed with a debug key and with debug symbols enabled).

An additional section, entitled *productFlavors* may also be included in the module build file to enable multiple build variants to be created. This topic will be covered in the next chapter entitled *An Android Studio Gradle Build Variants Example*.

Finally, the dependencies section lists any local and remote dependencies on which the module is dependent. The first dependency reads as follows:

```
compile fileTree(dir: 'libs', include: ['*.jar'])
```

This is a standard line that tells the Gradle system that any JAR file located in the module's lib sub-directory is to be included in the project build. If, for example, a JAR file named myclasses.jar was present in the GradleDemo/Module1/lib folder of the project, that JAR file would be treated as a module dependency and included in the build process.

A dependency on other modules within the same application project may also be declared within the build file. If, for example, Module1 has a dependency on Module2, the following line would need to be added to the dependencies section of the Module1 *build.gradle* file:

```
compile project(":Module2")
```

The last dependency lines in the above example file use Maven syntax to designate that the Android Support and Design libraries need to be included from the Android Repository:

```
compile 'com.android.support:appcompat-v7:25.2.0'
```

```
    compile 'com.android.support:design:25.2.0'
```

Another common repository requirement is the Google Play Services library, a dependency which can be declared as follows:

```
    compile 'com.google.android.gms:play-services:+'
```

Note that the dependency declaration can include an optional version number to indicate which version of the library should be included. No particular version is specified above for the play services (resulting in the most recent version being obtained from the repository).

76.5 **Configuring Signing Settings in the Build File**

The *Signing and Preparing an Android Application for Release* chapter of this book covered the steps involved in setting up keys and generating a signed release APK file using the Android Studio user interface. These settings may also be declared within a *signingSettings* section of the build.gradle file. For example:

```
    apply plugin: 'android'

    android {
        compileSdkVersion 25
        buildToolsVersion "25.0.2"

        defaultConfig {
            applicationId "com.ebookfrenzy.gradledemo.module1"
            minSdkVersion 8
            targetSdkVersion 25
            versionCode 1
            versionName "1.0"
        }

        signingConfigs {
            release {
                storeFile file("keystore.release")
                storePassword "your keystore password here"
                keyAlias "your key alias here"
                keyPassword "your key password here"
            }
        }
        buildTypes {
    .
    .
    .
    }
```

The above example embeds the key password information directly into the build file. Alternatives to this approach are to extract these values from system environment variables:

```
    signingConfigs {
        release {
            storeFile file("keystore.release")
            storePassword System.getenv("KEYSTOREPASSWD")
            keyAlias "your key alias here"
            keyPassword System.getenv("KEYPASSWD")
        }
    }
```

Yet another approach is to configure the build file so that Gradle prompts for the passwords to be entered during the build process:

```
signingConfigs {
    release {
        storeFile file("keystore.release")
        storePassword System.console().readLine
                 ("\nEnter Keystore password: ")
        keyAlias "your key alias here"
        keyPassword System.console().readLIne("\nEnter Key password: ")
    }
}
```

76.6 Running Gradle Tasks from the Command-line

Each Android Studio project contains a Gradle wrapper tool for the purpose of allowing Gradle tasks to be invoked from the command line. This tool is located in the root directory of each project folder. While this wrapper is executable on Windows systems, it needs to have execute permission enabled on Linux and Mac OS X before it can be used. To enable execute permission, open a terminal window, change directory to the project folder for which the wrapper is needed and execute the following command:

```
chmod +x gradlew
```

Once the file has execute permissions, the location of the file will either need to be added to your $PATH environment variable, or the name prefixed by ./ in order to run. For example:

```
./gradlew tasks
```

Gradle views project building in terms of number of different tasks. A full listing of tasks that are available for the current project can be obtained by running the following command from within the project directory (remembering to prefix the command with a ./ if running in Mac OS X or Linux):

```
gradlew tasks
```

To build a debug release of the project suitable for device or emulator testing, use the assembleDebug option:

```
gradlew assembleDebug
```

Alternatively, to build a release version of the application:

```
gradlew assembleRelease
```

76.7 Summary

For the most part, Android Studio performs application builds in the background without any intervention from the developer. This build process is handled using the Gradle system, an automated build toolkit designed to allow the ways in which projects are built to be configured and managed through a set of build configuration files. While the default behavior of Gradle is adequate for many basic project build requirements, the need to configure the build process is inevitable with more complex projects. This chapter has provided an overview of the Gradle build system and configuration files within the context of an Android Studio project. The next chapter, entitled *An Android Studio Gradle Build Variants Example* will take this a step further in the form of using Gradle to build different versions of the same application project.

77. An Android Studio Gradle Build Variants Example

The goal of this chapter is to use the build variants feature of Android Studio to create a project which can be built in two flavors designed to target phone and tablet devices respectively. The build environment will be configured such that each flavor can be built using either a release or debug build type. The end result, therefore, will be four build variant options available for selection within Android Studio:

· phoneDebug
· phoneRelease
· tabletDebug
· tabletRelease

This raises the question as to the difference between a build type and a build flavor. In general, a build type defines *how* a module is built (for example whether or not ProGuard is run, how the resulting application package is signed and whether debug symbols are to be included).

The build flavor, on the other hand, typically defines *what* is built (such as which resource and source code files are to be included in the build) for each variant of the module.

Initially the two flavors will be configured such that they differ only visually in terms of the resources that are used for each target such as layouts and string values. The project will then be further extended to provide an example of how each flavor might make use of different source code bases in order to provide differing application behavior.

77.1 Creating the Build Variant Example Project

Create a new project in Android Studio, entering *BuildExample* into the Application name field and ebookfrenzy.com as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 19: Android 4.4 (KitKat). Continue to proceed through the screens, requesting the creation of an Empty Activity named *BuildExampleActivity* with the remaining fields set to the default values.

Select the *activity_build_example.xml* layout file and load it into the Layout Editor tool. Select the ConstraintLayout entry within the Component Tree tool window and verify within the Properties tool window that the ID has been set to *activity_build_example*, changing it to this value if necessary.

Next, change the text on the "Hello World!" TextView widget to "Build Example" and extract the string to a resource named *variant_text*.

77.2 Adding the Build Flavors to the Module Build File

With the initial project created, the next step is to configure the module level *build.gradle* file to add the two build flavor configurations. Within the Android Studio Project tool window, navigate to the *build.gradle* file listed as *app -> Gradle Scripts -> build.gradle (Module: app)* (Figure 77-1) and double-click on it to load it into the editor:

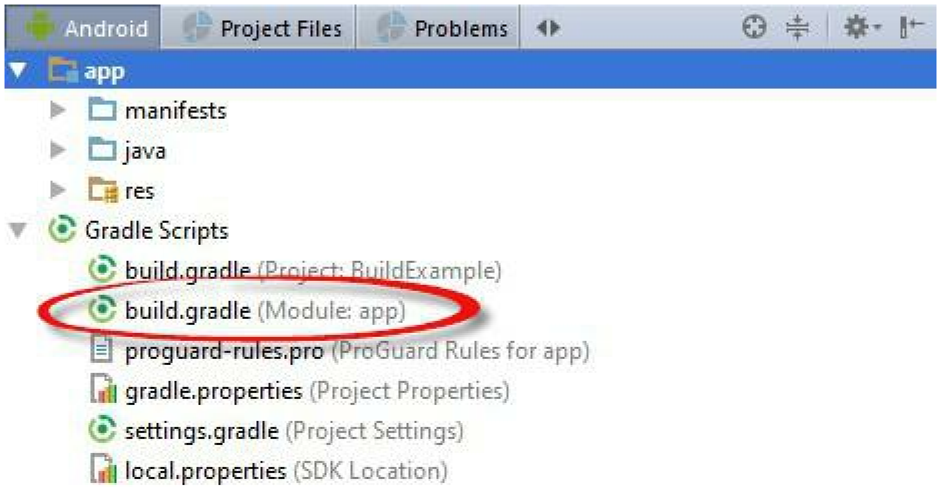
Android
Project Files
Problems
app
manifests
java
res
Gradle Scripts
build.gradle (Project: BuildExample)
build.gradle (Module: app)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)

Figure 77-1

Once loaded into the editor, the build file should resemble the following listing (allowing for differences in some version numbers):

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

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.ebookfrenzy.buildexample"
        minSdkVersion 19
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-
core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.2.0'
    compile 'com.android.support.constraint:constraint-layout:1.0.0'
    testCompile 'junit:junit:4.12'
}
```

As previously outlined, the project is going to consist of two build types (release and debug) together with two flavors (phone and tablet). As is evident from the build file, the release build type is already declared in the file so this does not need to be added. In practice, Gradle is also using sensible default settings for the debug build so this type also does not need to be added to the file. All this leaves is the requirement to declare the two build flavors as follows:

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

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.ebookfrenzy.buildexample"
        minSdkVersion 19
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
        }
    }
    productFlavors {
        phone {
            applicationId
            "com.ebookfrenzy.buildexample.app.phone"
            versionName "1.0-phone"
        }
        tablet {
            applicationId
            "com.ebookfrenzy.buildexample.app.tablet"
            versionName "1.0-tablet"
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-
core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.2.0'
    compile 'com.android.support.constraint:constraint-layout:1.0.0'
    testCompile 'junit:junit:4.12'
}
```

Once the changes have been made, a yellow warning bar will appear across the top of the editor indicating that the changes to the Gradle build file need to be synchronized with the rest of the project. Click on the *Sync Now* link located in the warning panel to perform the synchronization.

Before proceeding to the next step, open the Build Variants tool window either using the quick access menu located in the status bar in the bottom left-hand corner of the Android Studio main window or using the *Build Variant* tool window bar. Once loaded, clicking in the Build Variant cell for the app module should now list the four build variants:

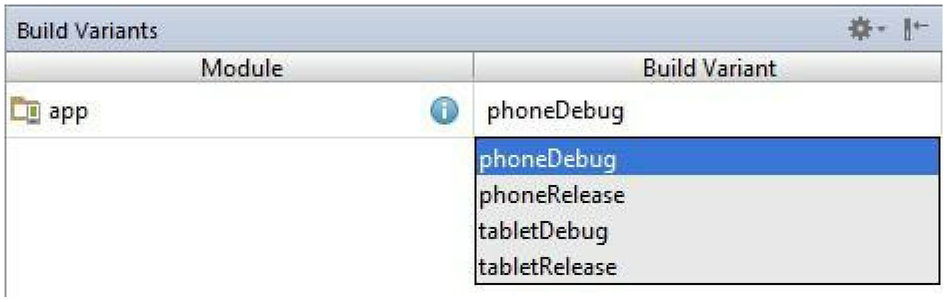

Figure 77-2

Now that the build flavors have been added to the project the project structure needs to be extended to provide support for these two new flavors.

77.3 **Adding the Flavors to the Project Structure**

So far in this book we have been using the Android Studio Project tool window in *Android* mode. This mode presents a less cluttered view of the directory structure of a project. When working with build variants, however, it will be necessary to switch the window into *Project* mode so that we can gain access to all of the directory levels in the project. To switch mode, click on the button indicated by the arrow in Figure 77-3.

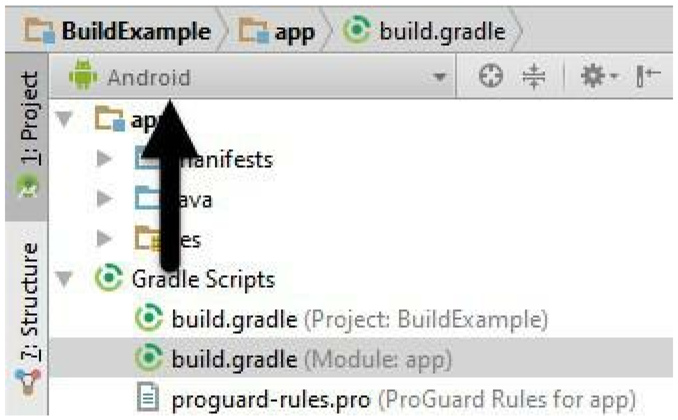
BuildExample
app
build.gradle
Android
1: Project
7: Structure
app
manifests
java
res
Gradle Scripts
build.gradle (Project: BuildExample)
build.gradle (Module: app)
proguard-rules.pro (ProGuard Rules for app)

Figure 77-3

From the drop down menu, select the *Project* option (Figure 77-4):

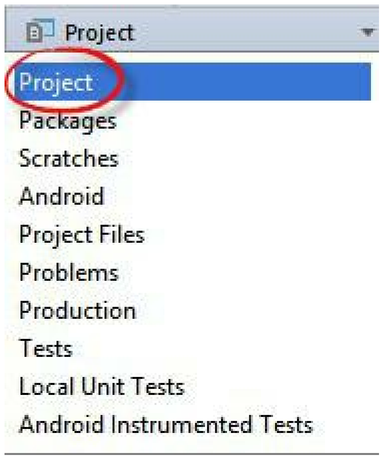

Figure 77-4

With the Project tool window now in Project mode, right-click on the *BuildExample -> app -> src* directory, select the *New -> Directory* menu option and create a new directory named *phone/res/layout*. Right-click once again on the src directory, this time adding a new directory named *phone/res/values*.

Repeat these steps, this time naming the new directories *tablet/res/layout* and *tablet/res/values*. Once complete, the module section of the project structure should resemble that of Figure 77-5:

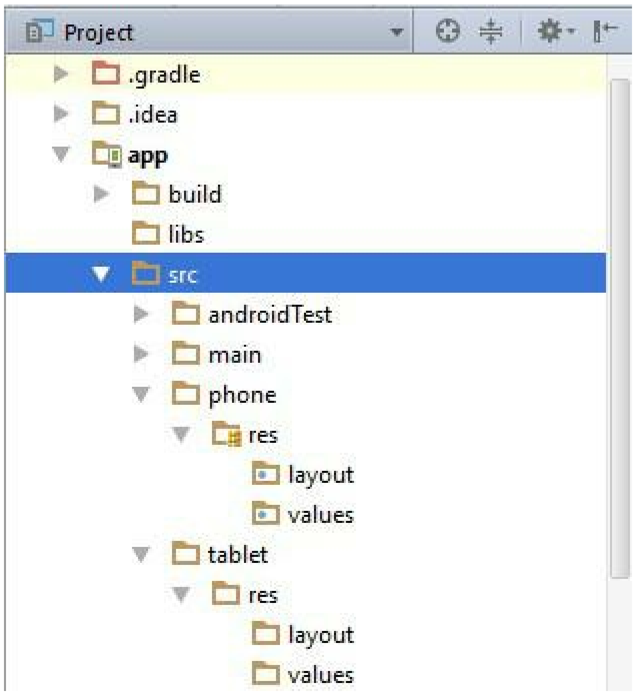

Figure 77-5

77.4 **Adding Resource Files to the Flavors**

Each flavor is going to need to contain a copy of the activity's *activity_build_example.xml* and *strings.xml* resource files. Each copy will then be modified to meet the requirements of the respective flavor.

Within the Project tool window, navigate to the *app -> src -> main -> res -> layout -> activity_build_example.xml* file, right-click on it and select the *Copy* menu option. With the file copied, right-click on the *src -> phone -> res -> layout* folder and select the *Paste* menu option to add a copy of the file to the folder. Within the *Copy* dialog click on *OK* to accept the defaults. Once copied, modify the phone layout variant so that the TextView widget is positioned in the top left-hand corner of the layout.

Using the same technique, add a copy of the *src -> main -> res -> values –> strings.xml* file to the *src -> phone -> res -> values* folder.

Once the resource files have been added, edit the *strings.xml* file in the phone flavor to change the *variant_text* string resource as follows:

```
<resources>
    <string name="app_name">BuildExample</string>
    <string name="variant_text">This is the phone flavor</string>
</resources>
```

Change the Build Variant setting to *tabletDebug* and copy and paste the phone *activity_build_example.xml* and *strings.xml* files to the corresponding locations in the tablet *res -> layout* and *res -> values* folders respectively.

Edit the tablet flavor *strings.xml* file so that the *variant_text* string resource reads "This is the tablet flavor".

With the changes made, the flavor section of the project structure should now match that of Figure 77-

6:

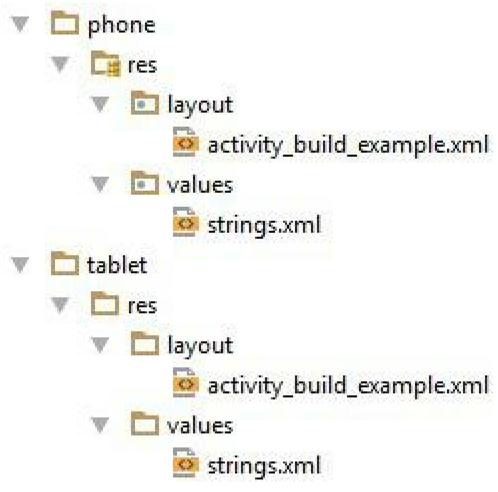

Figure 77-6

77.5 Testing the Build Flavors

At this point two flavors have been configured, each with different string and layout resources. Before moving on to the next step, it is important to check that the two build variants work as expected.

Within the Build Variants tool window, change the Build Variant setting for the *app* module to *phoneDebug* before running the application on a device or emulator. Once running, the phone flavor of the user interface should be displayed with the "This is the phone flavor" message displayed on the TextView object located in the top left-hand corner of the screen.

Stop the running app, change the build variant to *tabletDebug* and wait while Gradle rebuilds the project for the new selection. Once the build completes, run the application once again, noting that this time the tablet flavor has been built with the "This is the tablet flavor" message positioned in the center of the display.

77.6 Build Variants and Class Files

As the project currently stands, the two flavors of the application share the main BuildExample activity class and all of the flavor changes have been made through resource files. While much can be achieved through resource file modifications, it is inevitable that in many instances changes in the source code will be necessary from one flavor to the next. In the remainder of this chapter the main Activity class will be moved into the flavor variants so that different code bases can be used for each flavor.

77.7 Adding Packages to the Build Flavors

From this point on, each of the build flavors will have its own activity class file which can be customized to meet the requirements of the two different build targets.

Within the Build Variants tool window, begin by selecting the *phoneDebug* variant. Move to the Project tool window, right-click on the *phone* entry and select the *New -> Directory* menu option. Name the new directory *java* before clicking on the *OK* button.

Right-click on the new *java* directory, this time selecting the *New -> Package* menu option and naming the new package *com.ebookfrenzy.buildexample*.

Finally, find the *BuildExampleActivity* class which is located in the *src -> main -> java ->*

com.ebookfrenzy.buildexample folder, right-click on it and select the *Copy* menu option. Right-click on the new package added to the phone variant and select the *Paste* menu option to copy the Activity source file into the package. In the resulting Copy Class dialog click on the *OK* button to accept the default settings.

The phone variant now has its own version of the activity class. Using the Build Variants tool window, change the build variant to *tabletDebug* and repeat the above steps to add instances of the package and Activity class file to the tablet flavor of the build.

At this point, Android Studio will most likely have started to complain about duplicate instances of the BuildExampleActivity class. This is because in addition to having the class declared within the flavor variants, the original instance still exists within the main build folder. Locate, therefore, and delete the *BuildExampleActivity* entry listed under *src -> main -> java -> com.ebookfrenzy.buildexample.*

77.8 **Customizing the Activity Classes**

With *phoneDebug* selected in the Build Variants tool window, load the *phone -> java -> com.ebookfrenzy.buildexample -> BuildExampleActivity* class file into the editing window and modify the *onCreate* method to change the background color of the ConstraintLayout to red:

```
package com.ebookfrenzy.buildexample;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.graphics.Color;
import android.support.constraint.ConstraintLayout;

public class BuildExampleActivity extends AppCompatActivity {

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

        ConstraintLayout myLayout =
            (ConstraintLayout)
                findViewById(R.id.activity_build_example);
        myLayout.setBackgroundColor(Color.RED);
    }
.
.
.
}
```

Change the build variant to tabletDebug and modify the *tablet -> java -> com.ebookfrenzy.buildexample -> BuildExampleActivity* class file to change the background color of the layout to green:

```
package com.ebookfrenzy.buildexample;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.graphics.Color;
import android.support.constraint.ConstraintLayout;
```

```
public class BuildExampleActivity extends AppCompatActivity {

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

        ConstraintLayout myLayout =
            (ConstraintLayout)
                findViewById(R.id.activity_build_example);
        myLayout.setBackgroundColor(Color.GREEN);
    }
.
.
.
}
```

Compile and run the application using each variant and note that the background color changes to indicate that each flavor is using its own activity class file in addition to different resource files.

77.9 **Summary**

The Android market now comprises a diverse range of devices all of which have different hardware capabilities and screen sizes. While there is much that can be achieved with careful use of layout managers such as the ConstraintLayout and defensive coding, there will inevitably be situations where some target devices will need a separate application package to be built. In recognition of this fact, Android Studio introduces the concept of build variants and flavors designed specifically to make the task of building multiple variations of an application project easier to manage.