UNIT-1

An Overview of the Android Platform: **A Brief History of Mobile Software Development-** Way Back When, "The Brick", Wireless Application Protocol (WAP), Proprietary Mobile Platforms, **The Android Platform-**Android's Underlying Architecture, Security and Permissions, Exploring Android Applications, **Building Your First Android Application-** Creating and Configuring a New Android Project, Core Files and Directories of the Android Application, Creating an AVD for Your Project, Running Your Android Application in the Emulator, Debugging Your Android Application in the Emulator, Adding Logging Support to Your Android Application, Android Application Basics: **Understanding the Anatomy of an Android Application**-Mastering Important Android Terminology, Performing Application Tasks with Activities-The Lifecycle of an Android Activity, Managing Activity Transitions with Intents - Transitioning between Activities with Intents, Organizing Application Navigation with Activities and Intents,

Defining Your Application Using the Android Manifest File: **Configuring Android Applications Using the Android Manifest File**- Editing the Android Manifest File.

An Overview of the Android Platform:

Android is, without a doubt, a darling of the mobile development fraternity and is considered one of the premier operating systems. Companies today invest more time in Android app development to maintain a close connection with mobile users. Android remains relevant for mobile developers to cultivate incredible application experiences. Indeed, both manufacturers of handsets and mobile operators advertise actively in Android since it creates new forms of interaction with consumers.

Due to the relative openness of the Android framework, Android is well-poised to address new trends in the mobile landscape. Android grows not only in the regions different from technophile and high-end smart devices, but it also transforms the concept of mobility.

A Brief History of Mobile Software Development

To understand what makes Android so compelling, we must examine how mobile development has evolved and how Android differs from competing platforms.

Way Back When

Remember way back when a phone was just a phone? When we relied on fixed landlines? When we ran for the phone instead of pulling it out of our pocket? When we lost our friends at a crowded ball game and waited around for hours hoping to reunite? When we forgot the grocery list (see Figure 1.1) and had to find a pay phone or drive back home again? Those days are long gone. Today, commonplace problems such as these are easily solved with a one-button speed dial or a simple text message such as "WRU?" or "20?" or "Milk and?"

Figure 1.1 Mobile phones have become a crucial shopping accessory.

Our mobile phones keep us safe and connected. Now we roam around freely, relying on our phones not only to keep us in touch with friends, family, and coworkers, but also to tell us where to go, what to do, and how to do it. Even the simplest events seem to involve a mobile phone these days.

Motorola DynaTAC 8000X: The First Portable Cell Phone
Introduced in 1983
Dimensions: 13 × 1.75 × 3.5 inches
Weight: Approximately 2.5 pounds
Talk time: Slightly over half an hour
Retail price: $3,995 (plus monthly service fees and per-minute charges)
Nicknamed "The Brick"
Primarily used by traveling business executives, security personnel, and the wealthy Early mobile phones were expensive, with service charges that could bankrupt the average person Limited features (mainly calls and basic contacts). The first-generation mobile phones were developed by handset manufacturers in a competitive environment with closely guarded trade secrets. The first-generation mobile phones were designed and developed by the handset manufacturers. Competition was fierce and trade secrets were closely guarded. Manufacturers didn't want to expose the internal workings of their handsets, so they usually

Figure 1.2 The first commercially available mobile phone: the Motorola DynaTAC.

developed the phone software in-house. As a developer, if you weren't part of this inner circle, you had no opportunity to write applications for the phones. It was during this period that we saw the first "time-waster" games begin to appear. Nokia was famous for putting the 1970s video game Snake on some of its earliest monochrome phones. Other manufacturers followed suit, adding games such as Pong, Tetris, and Tic-Tac-Toe.

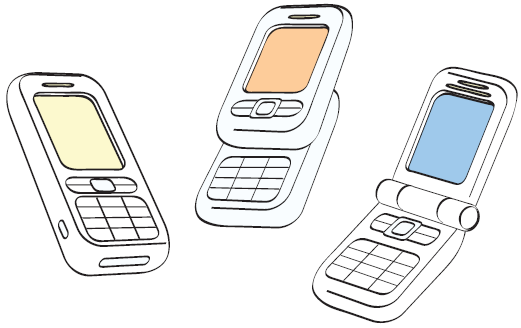


Figure 1.3 Various mobile phone form factors: the candy bar, the slider, and the clamshell.

These early devices were flawed, but they did something important—they changed the way people thought about communication. As mobile device prices dropped, batteries improved, and reception areas grew, more and more people began carrying these handy devices. Soon mobile devices were more than just a novelty. Customers began pushing for more features and more games. But there was a problem. The handset manufacturers didn't have the motivation or the resources to build every application users wanted. They needed some way to provide a portal for entertainment and information services without allowing direct access to the handset. What better way to provide these services than the Internet?

Wireless Application Protocol (WAP)

Wireless Application Protocol (WAP) was introduced to allow direct phone access to the Internet, but it struggled for mobile devices due to their low-resolution screens and limited storage and processing power. Professional websites were full color, using JavaScript, Flash, and other technologies to enhance user experience. The first clamshell phone, Motorola StarTAC, and Nokia's slider phone, the 8110, were limited in data-intensive operations. The Wireless Application Protocol (WAP) standard was created to address issues with traditional web browsers. WAP was a stripped-down version of HTTP, designed to run within phone memory and bandwidth constraints. Third-party WAP sites served up pages written in Wireless Markup Language (WML) for users to navigate on their phones. This solution was beneficial for handset manufacturers and mobile operators, as they could provide a custom WAP portal and charge high data charges. However, some developers struggled to develop content for phone users, leading to limited consumer traction. Early WAP sites were extensions of popular branded websites, such as CNN.com and ESPN.com. Commercializing WAP applications was challenging, and there was no built-in billing mechanism. Popular commercial WAP applications included simple wallpaper and ringtone catalogs, payment, and verification through premium delivery mechanisms like SMS, EMS, MMS, and WAP Push. WAP, a web application, failed to meet commercial expectations in some markets, like Japan, and in the United States. Its user experience was ruined by small screens and charging for downloading, leading to criticism as "Wait and Pay." Mobile operators also restricted access to WAP sites, discouraged third-party developers from creating applications.

Proprietary Mobile Platforms

Writing robust applications with WAP, such as graphics-intensive video games, was nearly impossible. The traditional desktop application developer was suddenly a player in the embedded device market, especially with smartphone technologies such as Windows Mobile, which they found familiar. A variety of different proprietary platforms emerged for traditional handsets. Some smartphone devices ran Palm OS (later known as WebOS) and RIM BlackBerry OS. Other platforms, such as Symbian OS, were developed by handset manufacturers such as Nokia, Sony Ericsson, Motorola, and Samsung. Many of these platforms

operate associated developer programs. These programs are often required, and developers even pay to participate.
Of course, developers love to debate about which platform is "the best. Some platforms are best suited for commercializing games and making millions—if your company has brand backing.
Other platforms are more open and suitable for the hobbyist or vertical market applications.
No mobile platform is best suited for all possible applications. As a result, the mobile phone market has become increasingly fragmented, with all platforms sharing parts of the pie. For manufacturers and mobile operators, handset product lines quickly became complicated. Platform market penetration varies greatly by region and user demographic. Instead of choosing just one platform, manufacturers and operators have been forced to sell phones for all the different platforms to compete in the market. (For instance, BlackBerry 10 phones provide a runtime to support Android applications). The mobile developer community has become as fragmented as the market. The platform development requirements vary greatly.It`s a nightmare for the ACME Company that wants a mobile application. Should it develop an application for BlackBerry? As a result, many wonderful applications have not reached their desired users, and many other great ideas have not been developed at all.

The Android Platform

Android is an operating system and a software platform upon which applications are developed. A core set of applications for everyday tasks, such as Web browsing and email, are included on Android devices. As a product of the OHA's vision for a robust and open-source development environment for wireless, Android is an emerging mobile development platform. The platform was designed for the sole purpose of encouraging a free and open market that users might want to have and software developers might want to develop for.

Android's Underlying Architecture

The Android platform is designed to be more fault tolerant than many of its predecessors. The device runs a Linux operating system upon which Android applications are executed in a secure fashion. Each Android application runs in its own virtual machine (see Figure 1.7). Android applications are managed code; therefore, they are much less likely to cause the device to crash, leading to fewer instances of device corruption (also called "bricking" the device, or rendering it useless). The Linux Operating System

The Linux 3.4 kernel handles core system services and acts as a hardware abstraction layer (HAL) between the physical hardware of the device and the Android software stack.

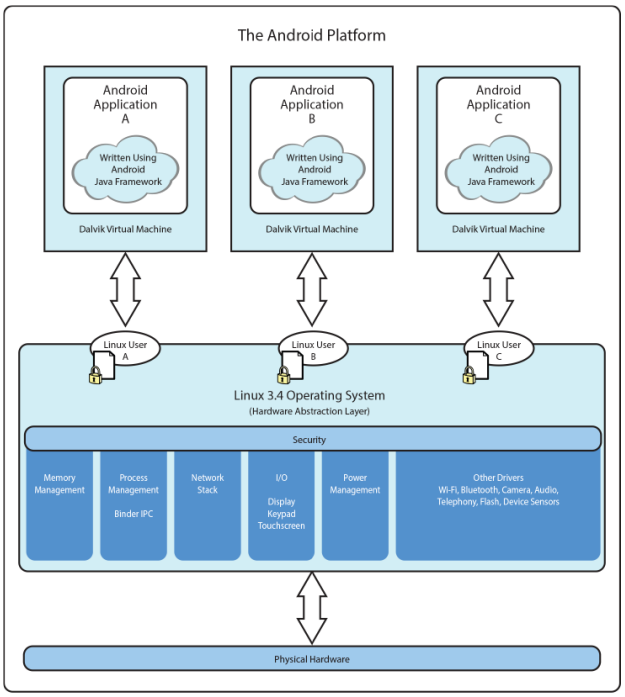


Figure 1.7 Diagram of the Android platform architecture.

Some of the core functions the kernel handles include
■ Enforcement of application permissions and security
■ Low-level memory management
■ Process management and threading
■ The network stack
■ Display, keypad input, camera, Wi-Fi, Flash memory, audio, binder inter-process communication (IPC), and power management driver access

Android Application Runtime Environment

Each Android application runs in a separate process, with its own instance of the Dalvik virtual machine (VM). Based on the Java VM, the Dalvik design has been optimized for mobile devices. The Dalvik VM has a small memory footprint and optimized application loading, and multiple instances of the Dalvik VM can run concurrently on a device.

Security and Permissions

These measures help ensure that the user`s data is secure and that the device is not subjected to malware or misuse. Each application runs as a different user, with its own private files on the file system, a user ID, and a secure operating environment. Explicitly Defined Application Permissions To access shared resources on the system, Android applications register for the specific privileges they require. Some of these privileges enable the application to use device functionality to make calls, access the network, and control the camera and other hardware sensors. Applications also require permission to access shared data containing private and personal information, such as user preferences, the user`s location, and contact information.

This is done using ad hoc granting and revoking of access to specific resources using Uniform Resource Identifiers (URIs). Let`s say we have an application that keeps track of the user`s public and private birthday wish lists. If this application wanted to share its data with other applications, the application could grant URI permissions for the public wish list, allowing another application to access this list without explicitly having to ask the user for it. Application Signing for Trust Relationships All Android application packages are signed with a certificate, so users know that the application is authentic.

It also enables the developer to control which applications can grant access to one another on the system. Multiple Users and Restricted Profiles Android 4.

2 (API Level 17) brought support for multiple user accounts on shareable Android devices such as tablets.

3 (API Level 18), primary device users are now able to create restricted profiles for limiting a user profile`s access to particular applications.

Developers may also leverage restricted profile capabilities in their applications to provide primary users the ability to further prohibit particular device users from accessing specific in-app content.

Exploring Android Applications

The Android SDK provides an extensive set of APIs that are both modern and robust. Android device core system services are exposed and accessible to all applications. When granted the appropriate permissions, Android applications can share data with one another and access shared resources on the system securely.

Android Programming Language Choices

Android applications are written in Java (see Figure 1.8). For now, the Java language is the developer's only choice for accessing the entire Android SDK. This book focuses on developing mobile Web applications for Android devices, which can be accessed through an Android browser or embedded WebView control within a native Android application. Adobe's AIR support for Android allows users to load compatible applications using the Adobe AIR application. Developers can also build applications using scripting languages like Python. However, developing SL4A applications is outside the scope of this book.

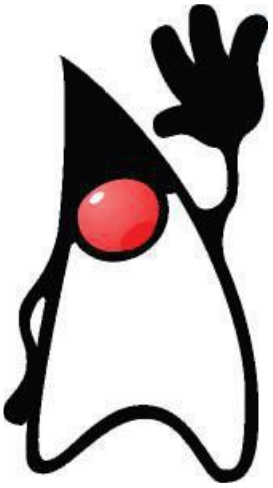


Figure 1.8 Duke, the Java mascot.

No Distinctions Made between Native and Third-Party Applications

Unlike other mobile development platforms, the Android platform makes no distinction between native applications and developer-created applications. Provided they are granted the appropriate permissions, all applications have the same access to core libraries and the underlying hardware interfaces. Android devices come with native applications like a web browser and contact manager, which can be integrated with, extended, or replaced with third-party applications using the same APIs. Google has used undocumented APIs in some cases, but has warned developers that using them may result in incompatibilities in future SDK versions. Android packages include support for various user interface controls, layouts, integration capabilities, secure networking features, XML support, structured storage, relational databases, powerful graphics, multimedia frameworks, audio and visual media formats, and optional hardware like location-based services and hardware sensors.

Commonly Used Packages

With Android, mobile developers no longer have to reinvent the wheel. Instead, developers use familiar class libraries exposed through Android's Java packages to perform common tasks involving graphics, database access, network access, secure communications, and utilities. The Android packages include support for the following:

- A wide variety of user interface controls (Buttons, Spinners, Text input)
- A wide variety of user interface layouts (Tables, Tabs, Lists, Galleries)
- Integration capabilities (Notifications, Widgets)
- Secure networking and Web browsing features (SSL, WebKit)
- XML support (DOM, SAX, XML Pull Parser)

■ Structured storage and relational databases (App Preferences, SQLite)

■ Powerful 2D and 3D graphics (including SGL, OpenGL ES, and RenderScript)

■ Multimedia frameworks for playing and recording standalone or network streaming

(MediaPlayer, JetPlayer, SoundPool, AudioManager)

■ Extensive support for many audio and visual media formats (MPEG4, H.264, MP3,

AAC, AMR, JPG, and PNG)

■ Access to optional hardware such as location-based services (LBS), USB, Wi-Fi,

Bluetooth, and hardware sensors

Android Application Framework

The Android application framework provides everything necessary to implement an average application. The Android application lifecycle involves the following key components:

■ Activities are functions that the application performs.

■ Groups of views define the application's layout.

■ Intents inform the system about an application's plans

■ Services allow for background processing without user interaction.

■ Notifications alert the user when something interesting happens.

■ Content providers facilitate data transmission among different applications.

Android Platform Services

Android applications can interact with the operating system and underlying hardware using a collection of managers. Each manager is responsible for keeping the state of some underlying system service. For example:

■ The LocationManager facilitates interaction with the location-based services available on the device.

■ The ViewManager and WindowManager manage display and user interface fundamentals related to the device.

■ The AccessibilityManager manages accessibility events, facilitating device support for users with physical impairments.

■ The ClipboardManager provides access to the global clipboard for the device, for cutting and pasting content.

■ The DownloadManager manages HTTP downloads in the background as a system service.

■ The FragmentManager manages the fragments of an activity.

■ The AudioManager provides access to audio and ringer controls.

Google Services

Google provides APIs for integrating with many different Google services. Prior to the addition of many of these services, developers would need to wait for mobile operators and device manufacturers to upgrade Android on their devices in order to take advantage of many common features such as maps or location-based services. Now developers are able to integrate the latest and greatest updates of these services by including the required SDKs in their application projects. Some of these Google services include

- Maps
- Location-based services
- Game Services
- Authorization APIs
- Google Plus
- Play Services
- In-app Billing
- Google Cloud Messaging
- Google Analytics
- Google AdMob ads

Building Your First Android Application

Now it's time to write your first Android application from scratch. To get your feet wet, you will start with a simple "Hello World" application and build upon it to explore some of the features of the Android platform in more detail.

Creating and Configuring a New Android Project

You can create a new Android application in much the same way that you added the Snake application to your Android IDE workspace.

The first thing you need to do is create a new project in your Android IDE workspace. The Android Application Project creation wizard creates all the required files for an

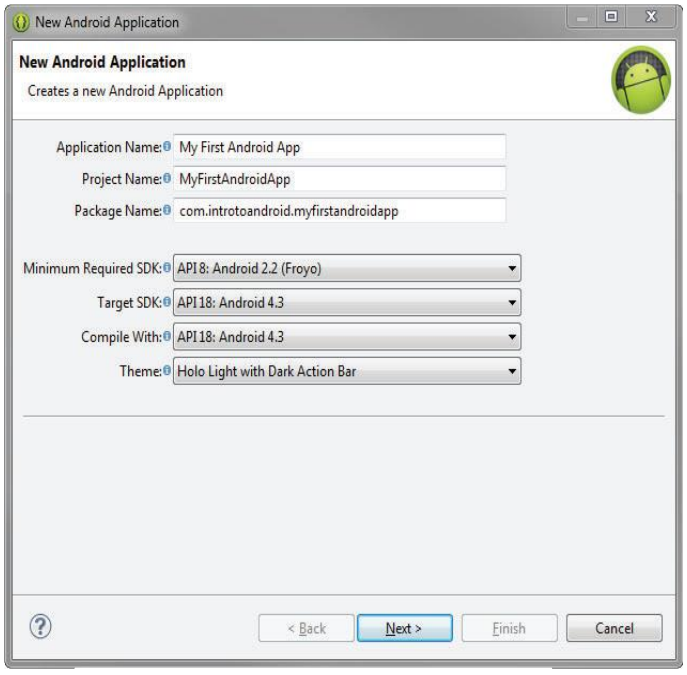


Figure 3.15 Configuring a new Android project.

Android application. Follow these steps within the Android IDE to create a new project:

1. Choose File, New, Android Application Project on the Android IDE toolbar.

2. Choose an application name as shown in Figure 3.15. The application name is the "friendly" name of the application and the name shown with the icon on the application launcher. Name the application My First Android App. This will automatically create a project name of MyFirstAndroidApp, but you are free to change this to a name of your choosing.

3. We should also change the package name, using reverse domain name notation (http://en.wikipedia.org/wiki/Reverse_domain_name_notation),tocom.introtoandroid.myfirst androidapp. The Minimum Required SDK version should be the first SDK API level you plan to target. Because our application will be compatible with just about any Android device, you can set this number low (such as to 4 to represent Android 1.6) or at the target API level to avoid any warnings in the Android IDE. Make sure you set the minimum SDK version to encompass any test devices you have available so you can successfully install the application on them. The default options are just fine for our example. Click Next.

4. Keep the rest of the New Android Application settings at their defaults, unless you want to change the directory of where the source files will be stored. Click Next (see Figure 3.16).

5. Leave the Configure Launcher Icon settings at their defaults. This option screen would allow us to define how our application launcher icon appears, but for this example, we will use the standard icon set included with the Android SDK. Choose Next (see Figure 3.17).

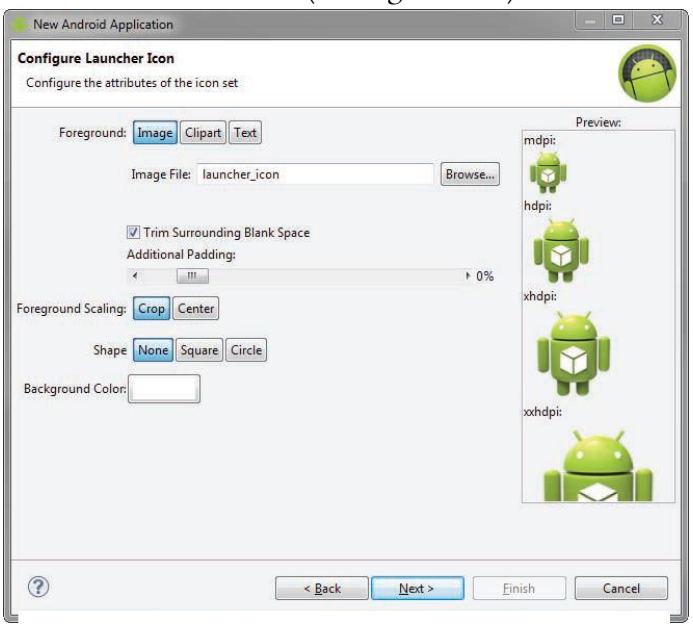


Figure 3.17 Configuring the launcher icon for our Android project.

6. The Create Activity wizard allows us to include a default launch activity by type.We will leave the settings as is and choose Next (see Figure 3.18).

7. Choose an Activity Name. Call this Activity class MyFirstAndroidApp Activity. The Layout Name should automatically change to a name resembling what you just entered. Finally, click the Finish button (see Figure 3.19) to create the application.

8. The Android IDE should now display our first application created using the wizard with our layout file open and ready for editing (see Figure 3.20).

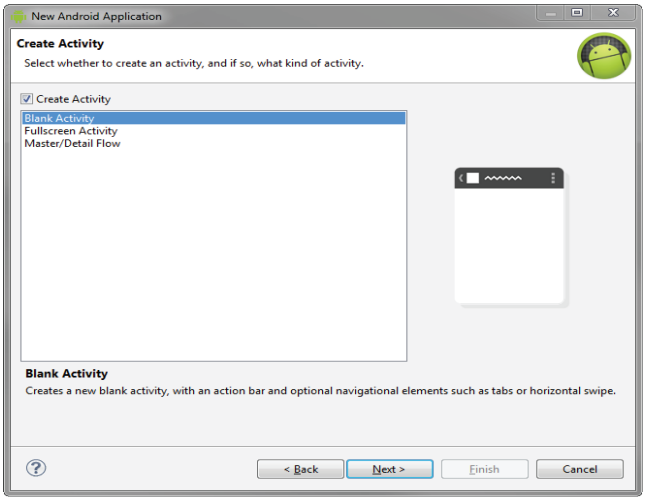


Figure 3.18 Creating an `Activity` for our Android project.

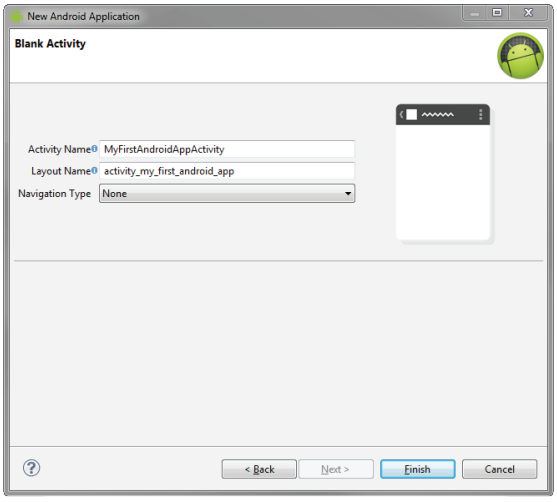


Figure 3.19 Choosing an `Activity Name`.

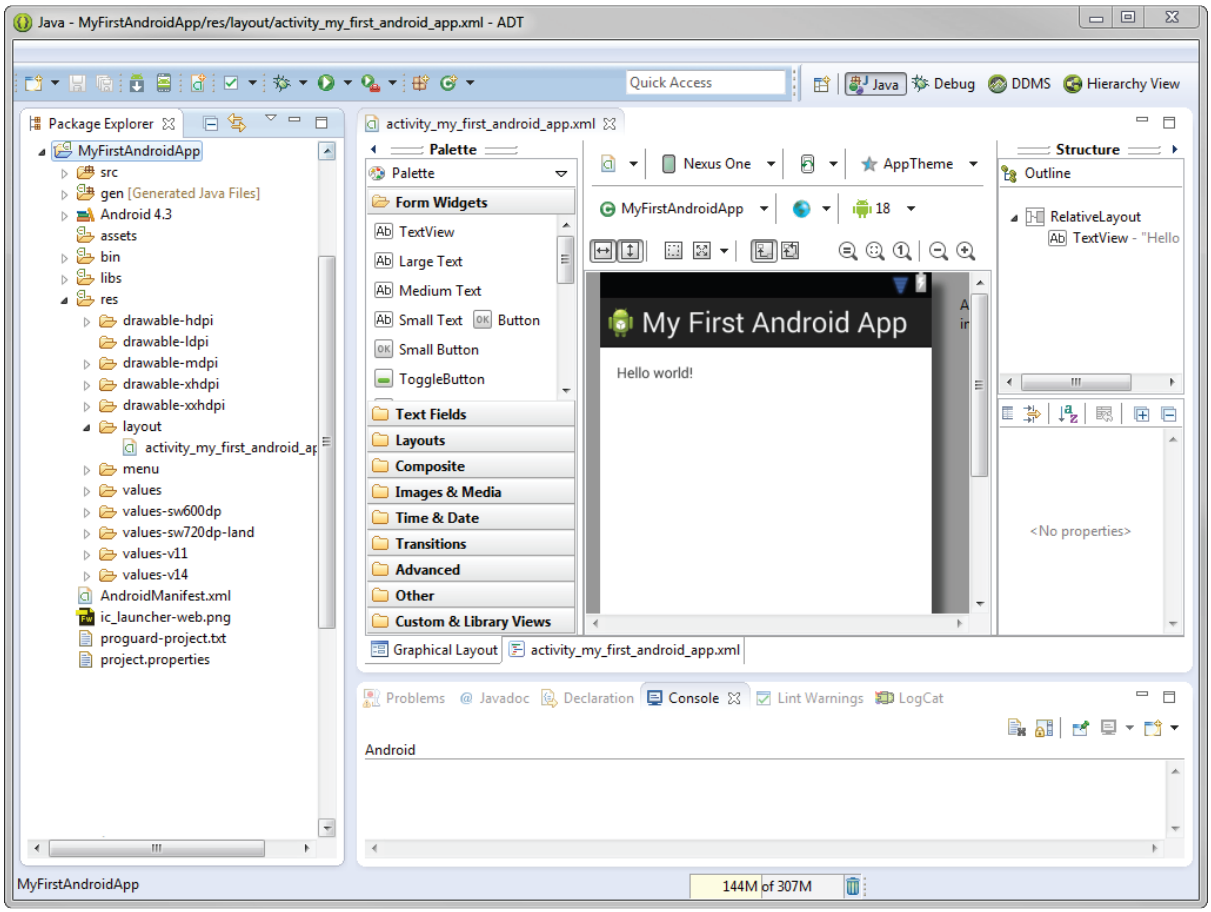


Figure 3.20 Our first application created with the wizard.

Core Files and Directories of the Android Application

Every Android application has a set of core files that are created and used to define the functionality of the application. The following files are created by default with a new Android application:

■ AndroidManifest.xml—the central configuration file for the application. It defines your application's capabilities and permissions as well as how it runs.

■ ic_launcher-web.png—This is a high-resolution 32-bit 512 x 512 PNG application icon that is required and used for your application listing in the Google Play

store. The size of this icon should not exceed 1024KB.

■ proguard-project.txt—a generated build file used by the Android IDE and Pro-Guard. Edit this file to configure your code optimization and obfuscation settings for release builds.

■ project.properties—a generated build file used by the Android IDE. It defines your application's build target and other build system options, as required. Do not edit this file.

A number of other files are saved on disk as part of the Android IDE project in the workspace. However, the files and resource directories included in the list here are the important project files you will use on a regular basis.

Creating an AVD for Your Project

Create an AVD for the Snake application, which describes the device type to emulate. This AVD can be used for multiple applications or with different data, such as different applications installed and SD card contents.

Creating a Launch Configuration for Your Project

Next, you must create a Run and Debug launch configuration in the Android IDE to configure the circumstances under which the MyFirstAndroidApp application builds and launches. The launch configuration is where you configure the emulator options to use and the entry point for your application. You can create Run configurations and Debug configurations separately, with different options for each. Begin by creating a Run configuration for the application. Follow these steps to create a basic Run configuration for the MyFirstAndroidApp application:

1. Choose Run, Run Configurations... (or right-click the project and choose Run As).

2. Double-click Android Application.

3. Name your configuration MyFirstAndroidAppRunConfig.

4. Choose the project by clicking the Browse button and choosing the MyFirstAndroidApp project.

5. Switch to the Target tab and set the Deployment Target Selection Mode to Always prompt to pick device.

6. Click Apply and then click Close.

Now create a Debug configuration for the application. This process is similar to creating a Run configuration. Follow these steps to create a basic Debug configuration for the MyFirstAndroidApp application:

1. Choose Run, Debug Configurations... (or right-click the project and choose Debug As).

2. Double-click Android Application.

3. Name your configuration MyFirstAndroidAppDebugConfig.

4. Choose the project by clicking the Browse button and choosing the MyFirstAndroidApp project.

5. Switch to the Target tab and set the Deployment Target Selection Mode to Always prompt to pick device.

6. Click Apply and then click Close.

You now have a Debug configuration for your application

Running Your Android Application in the Emulator

Now you can run the MyFirstAndroidApp application using the following

Now you can run the MyFirstAndroidApp application using the following steps:

1. Choose the Run As icon drop-down menu on the toolb ( )

2. Pull the drop-down menu and choose the Run configuration you created. (If you do not see it listed, choose the Run Configurations... item and select the appropriate configuration. The Run configuration shows up on this drop-down list the next time you run the configuration.)

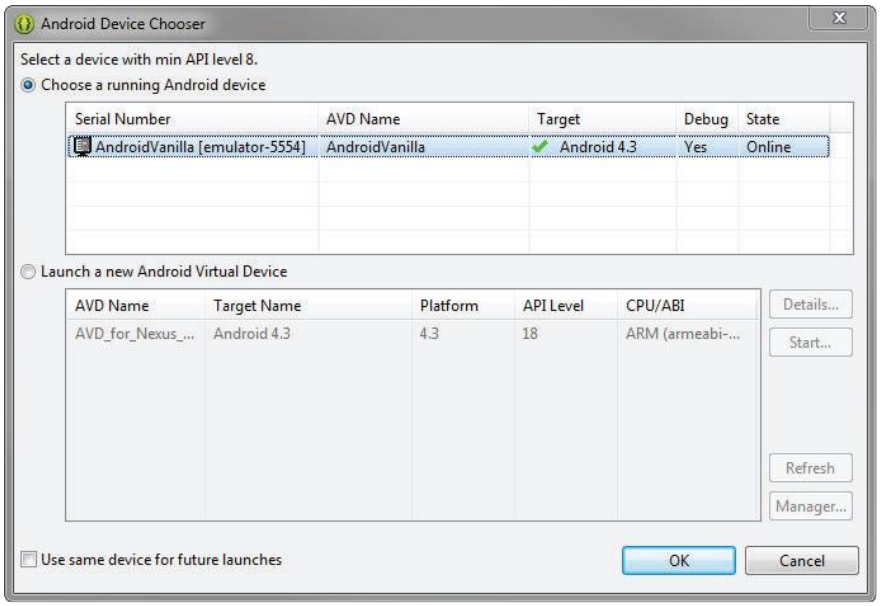


Figure 3.21 Manually choosing a deployment target selection mode.

3. Because you chose the Always prompt to pick device selection mode, you are now prompted for your emulator instance. Change the selection to Launch a New Android Virtual Device and then select the AVD you created. Here, you can choose from an already-running emulator or launch a new instance with an AVD that is compatible with the application settings, as shown in Figure 3.21.

4. The Android emulator starts up, which might take a moment.

5. Click the Menu button or push the slider to the right to unlock the emulator.

6. The application starts, as shown in Figure 3.22.

7. Click the Back button in the emulator to end the application, or click Home to suspend it.

8. Click the All Apps button (see Figure 3.23) found in the Favorites tray to browse all installed applications from the All Apps screen.

9. Your screen should now look something like Figure 3.24. Click the My First Android App icon to launch the application again.

Figure 3.22 `My First Android App` running in the emulator

Figure 3.23 The `All Apps` button.

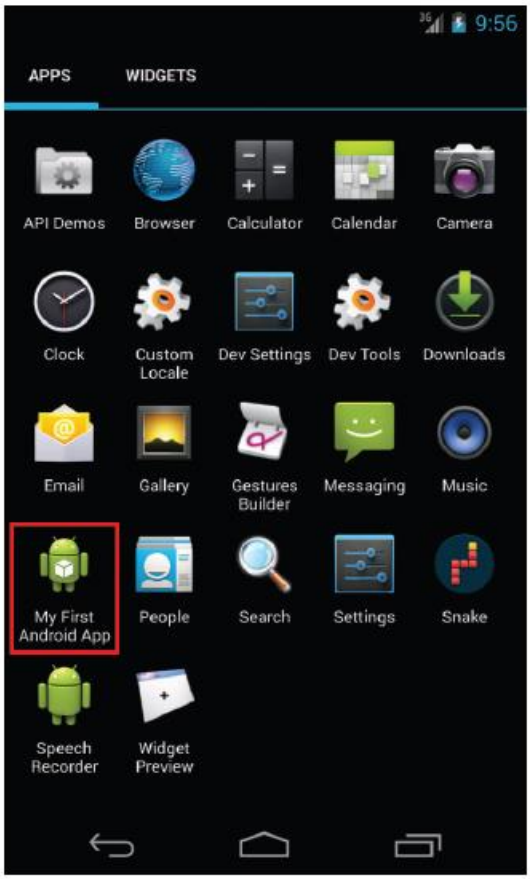


Figure 3.24 The `My First Android App` icon shown in the `All Apps` screen.

Debugging Your Android Application in the Emulator

Before going any further, you need to become familiar with debugging in the emulator. To illustrate some useful debugging tools, let's manufacture an error in the My First Android App. In your project, edit the source file called MyFirstAndroidAppActivity.java. Create a new method called forceError() in your class and make a call to this method in your Activity class's onCreate() method. The forceError() method forces a new unhandled error in your application.

The forceError() method should look something like this:

```
public void forceError() {

if(true) {

throw new Error("Whoops");

}

}
```

It's probably helpful at this point to run the application and watch what happens. Do this using the Run configuration first. In the emulator, you see that the application has stopped unexpectedly. You are prompted by a dialog that enables you to force the application to close, as shown in Figure 3.25.

Figure 3.25 `My First Android App` crashing gracefully.

Shut down the application but keep the emulator running. Now it's time to debug.

You can debug the MyFirstAndroidApp application using the following steps:

1. Choose the Debug As icon drop-down menu on the toolbar.

2. Pull the drop-down menu and choose the Debug configuration you created. (If you do not see it listed, choose the Debug Configurations... item and select the appropriate configuration. The Debug configuration shows up on this drop-down list the next time you run the configuration.)

3. Continue as you did with the Run configuration and choose the appropriate AVD, and then launch the emulator again, unlocking it if needed. It takes a moment for the debugger to attach. If this is the first time you've debugged an Android application, you may need to click through some dialogs, such as the one shown in Figure 3.26, the first time your application attaches to the debugger.

In the Android IDE, use the Debug perspective to set breakpoints, step through code, and watch the LogCat logging information about your application. This time, when the application fails, you can determine the cause using the debugger. You might need to click through several dialogs as you set up to debug within the Android IDE. If you allow the application to continue after throwing the exception, you can examine the results in the Debug perspective of the Android IDE. If you examine the LogCat logging pane, you see that your application was forced to exit due to an unhandled exception (see Figure 3.27).

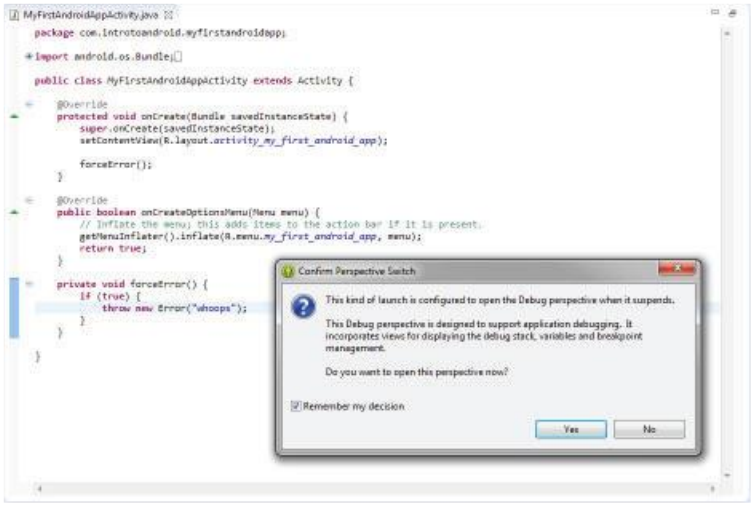


Figure 3.26 Switching to `Debug` perspective for Android emulator debugging.

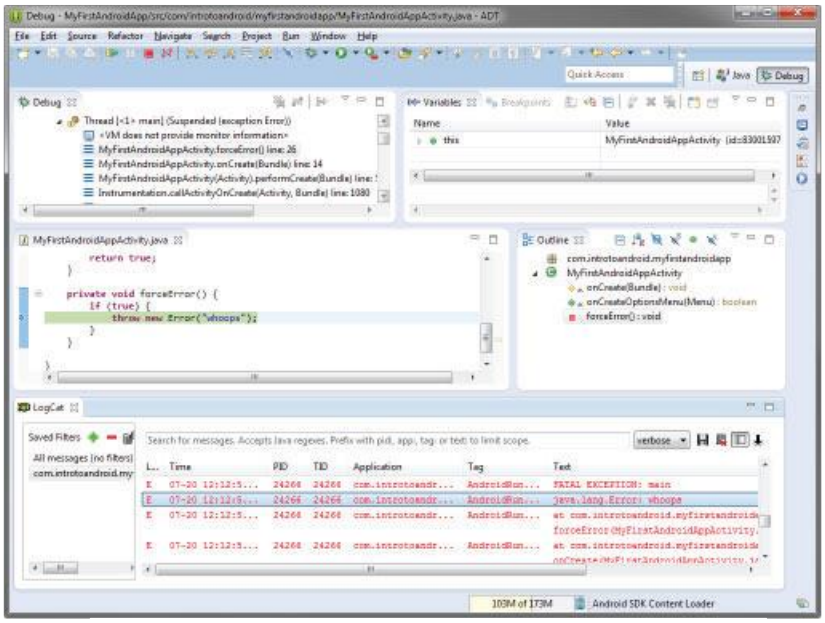


Figure 3.27 Debugging `MyFirstAndroidApp` in the Android IDE.

Specifically, there's a red AndroidRuntime error: java.lang.Error: whoops. Back in the emulator, click the Force Close button. Now set a breakpoint on the forceError() method by right-clicking the left side of the line of code and choosing Toggle Breakpoint (or double-clicking).

In the emulator, restart your application and step through your code. You see that your application has thrown the exception, and then the exception shows up in the Variable Browser pane of the Debug perspective. Expanding its contents shows that it is the "Whoops" error. This is a great time to crash your application repeatedly and get used to the controls. While you're at it, switch over to the DDMS perspective. Note that the emulator has a list of processes running on the device, such as system_process and com.android.phone. If you launch MyFirstAndroidApp, you see com.introtoandroid.myfirstandroidapp show up as a process on the emulator listing. Force the app to close because it crashes, and note that it disappears from

the process list. You can use DDMS to kill processes, inspect threads and the heap, and access the phone file system.

Adding Logging Support to Your Android Application

Before you start diving into the various features of the Android SDK, you should familiarize yourself with logging, a valuable resource for debugging and learning Android. Android logging features are in the Log class of the android.util package. See Table 3.1 for some helpful methods in the android.util.Log class.

Table 3.1 **Commonly Used Logging Methods**

Method	Purpose
`Log.e()`	Log errors
`Log.w()`	Log warnings
`Log.i()`	Log informational messages
`Log.d()`	Log debug messages
`Log.v()`	Log verbose messages

To add logging support to MyFirstAndroidApp, edit the file MyFirstAndroidApp .java. First, you must add the appropriate import statement for the Log class:

import android.util.Log;

Next, within the MyFirstAndroidApp class, declare a constant string that you use to tag all logging messages from this class. You can use the LogCat utility within the Android IDE to filter your logging messages to this DEBUG_TAG tag string:

private static final String DEBUG_TAG= "MyFirstAppLogging";

Now, within the onCreate() method, you can log something informational:

Log.i(DEBUG_TAG,

"In the onCreate() method of the MyFirstAndroidAppActivity Class");

While you're here, you must comment out your previous forceError() call so that your application doesn't fail. Now you're ready to run MyFirstAndroidApp. Save your work and debug it in the emulator. Notice that your logging messages appear in the LogCat listing, with the Tag field MyFirstAppLogging (see Figure 3.28).

Understanding the Anatomy of an Android Application

Mastering Important Android Terminology

Some of the important terms used in Android Application are:

■ Context: The context is the central command center for an Android application. Most application-specific functionality can be accessed or referenced through the context. The Context class (android.content.Context) is a fundamental building block of any Android application and provides access to application-wide features such as the application's private files and device resources as well as system-wide services. The application-wide Context object is instantiated as an Application object (android.app.Application).

■ Activity: An Android application is a collection of tasks, each of which is called an activity. Each activity within an application has a unique task or purpose. The Activity class (android.app.Activity) is a fundamental building block of any Android application, and most applications are made up of several activities. Typically, the purpose is to handle the display of a single screen, but thinking only in terms of "an activity is a screen" is too simplistic. An Activity class extends the Context class, so it also has all of the functionality of the Context class

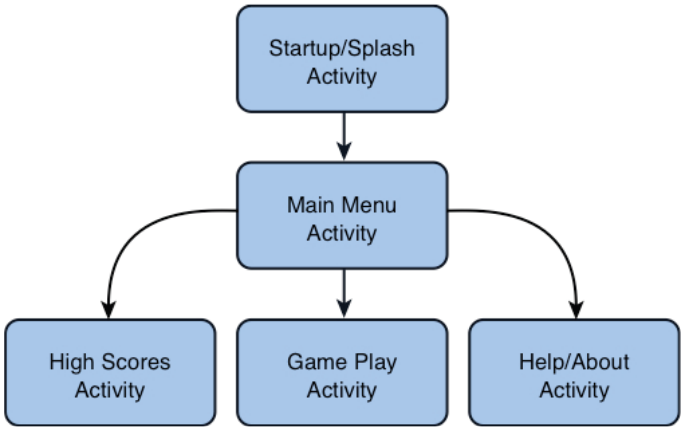
■ Fragment: An activity has a unique task or purpose, but it can be further componentized; each component is called a fragment. Each fragment within an application has a unique task or purpose within its parent activity. The Fragment class (android .app.Fragment) is often used to organize activity functionality in such a way as to allow a more flexible user experience across various screen sizes, orientations, and aspect ratios. A fragment is commonly used to hold the code and screen logic for placing the same user interface component in multiple screens, which are represented by multiple Activity classes.

■ Intent: The Android operating system uses an asynchronous messaging mechanism to match task requests with the appropriate activity. Each request is packaged as an intent. You can think of each such request as a message stating an intent to do something. Using the Intent class (android.content.Intent) is the primary method by which application components such as activities and services communicate with one another.

■ Service: Tasks that do not require user interaction can be encapsulated in a service. A service is most useful when the operations are lengthy (offloading time-consuming processing) or need to be done regularly (such as checking a server for new mail). Whereas activities run in the foreground and generally have a user interface, the Service class (android.app.Service) is used to handle background operations related to an Android application. The Service class extends the Context class.

Performing Application Tasks with Activities

The Android Activity class (android.app.Activity) is core to any Android application. Much of the time, you define and implement an Activity class for each screen in your application. For example, a simple game application might have the following five activities, as shown in Figure:



■ A startup or splash screen: This activity serves as the primary entry point to the application. It displays the application name and version information and transitions to the main menu after a short interval.

■ A main menu screen: This activity acts as a switch to drive the user to the core activities of the application. Here, the users must choose what they want to do within the application.

■ A game play screen: This activity is where the core game play occurs.

■ A high scores screen: This activity might display game scores or settings.

■ A Help/About screen: This activity might display the information the user might need to play the game.

The Lifecycle of an Android Activity

Android applications can be multiprocess, and the Android operating system allows multiple applications to run concurrently, provided memory and processing power are available. Applications can have background behavior, and applications can be interrupted and paused when events such as phone calls occur.

Only one active application can be visible to the user at a time—specifically, a single application Activity is in the foreground at any given time. The Android operating system keeps track of all Activity objects running by placing them on an Activity stack (see Figure 4.2). This Activity stack is referred to as the "back stack." When a new Activity starts, the Activity on the top of the stack (the current foreground Activity) pauses, and the new Activity pushes onto the top of the stack. When that Activity finishes, it is removed from the Activity stack, and the previous Activity in the stack
Resumes

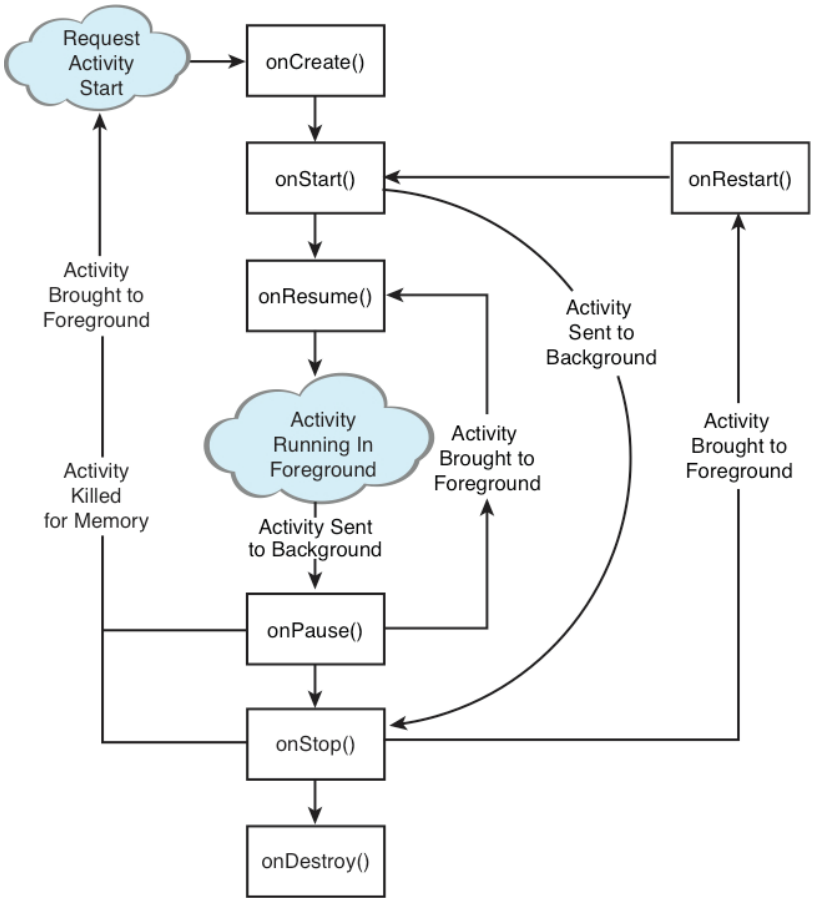
Android applications are responsible for managing their state and their memory, resources, and data. They must pause and resume seamlessly. Understanding the different states within the Activity lifecycle is the first step in designing and developing robust Android applications

Using Activity Callbacks to Manage Application State and Resources

Different important state changes within the Activity lifecycle are punctuated by a series of important method callbacks. These callbacks are shown in Figure 4.3.

Here are the method stubs for the most important callbacks of the Activity class:

```
public class MyActivity extends Activity {
        protected void onCreate(Bundle savedInstanceState);
        protected void onStart();
        protected void onRestart();
        protected void onResume();
        protected void onPause();
        protected void onStop();
        protected void onDestroy();
 }
```



Initializing Static Activity Data in onCreate() When an Activity first starts, the onCreate() method is called. The onCreate() method has a single parameter, a Bundle, which is null if this is a newly started Activity. If this Activity was killed for memory reasons and is now restarted, the Bundle contains the previous state information for this Activity so that it can reinitiate. It is appropriate to perform any setup, such as layout and data binding, in the onCreate() method. This includes calls to the setContentView() method.

Initializing and Retrieving Activity Data in onResume()
When the Activity reaches the top of the Activity stack and becomes the foreground process, the onResume() method is called. Although the Activity might not be visible yet to the user, this is the most appropriate place to retrieve any instances of resources (exclusive or otherwise) that the Activity needs to run. Often, these resources are the most process intensive, so we keep them around only while the Activity is in the foreground.

Stopping, Saving, and Releasing Activity Data in onPause()
When another Activity moves to the top of the Activity stack, the current Activity is informed that it is being pushed down the Activity stack by way of the onPause() method. Here, the Activity should stop any audio, video, and animations it started in the onResume() method. This is also where you must deactivate resources such as database Cursor objects or other objects that should be cleaned up should your Activity be terminated. The onPause() method may be the last chance for the Activity to clean up and release any resources it does not need while in the background. You need to save any uncommitted data here, in case your application does not resume. The system reserves the right to kill an Activity without further notice after the call

on onPause(). The Activity can also save state information to Activity-specific preferences or application-wide preferences. We talk more about preferences in Chapter 11, "Using Android Preferences." The Activity needs to perform anything in the onPause() method in a timely fashion, because the new foreground Activity is not started until the onPause() method returns

Destroying Static Activity Data in onDestroy()

When an Activity is being destroyed in the normal course of operation, the onDestroy() method is called. The onDestroy() method is called for one of two reasons: the Activity has completed its lifecycle voluntarily, or the Activity is being killed by the Android operating system because it needs the resources but still has the time to gracefully destroy the Activity (as opposed to terminating it without calling the onDestroy() method)
onStart()
When activity start getting visible to user then onStart() will be called. This calls just after the onCreate() at first time launch of activity.When activity launch, first onCreate() method call then onStart() and then onResume(). If the activity is in onPause() condition i.e. not visible to user and if user again launch the activity then onStart() method will be called.
onStop()
The onStop() function is called when the application enters the stopped state. In this state, the activity is no longer visible to the user. This may happen for the following reasons, the user performing some action that invoked another activity to come to the foreground or the activity finished. In this state, the resources must be managed or released as the activity is no longer visible to the user. The activity either comes back to interact with the user after running the onRestart() or finishes by executing the onDestroy() callback.

Example:

activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.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="example.javatpoint.com.activitylifecycle.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

MainActivity.java:

```
package example.javatpoint.com.activitylifecycle;
import android.app.Activity;
```

```
import android.os.Bundle;
import android.util.Log;
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d("lifecycle","onCreate invoked");
    }
    @Override
    protected void onStart() {
        super.onStart();
        Log.d("lifecycle","onStart invoked");
    }
    @Override
    protected void onResume() {
        super.onResume();
        Log.d("lifecycle","onResume invoked");
    }
    @Override
    protected void onPause() {
        super.onPause();
        Log.d("lifecycle","onPause invoked");
    }
    @Override
    protected void onStop() {
        super.onStop();
        Log.d("lifecycle","onStop invoked");
    }
    @Override
    protected void onRestart() {
        super.onRestart();
        Log.d("lifecycle","onRestart invoked");
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.d("lifecycle","onDestroy invoked");
    }
}
```

Managing Activity Transitions with Intents

In the course of the lifetime of an Android application, the user might transition between a number of different Activity instances. At times, there might be multiple Activity instances on the Activity stack. Developers need to pay attention to the lifecycle of each Activity during these transitions.

Some Activity instances—such as the application splash/startup screen—are shown and then permanently discarded when the main menu screen Activity takes over. The user cannot return to the splash screen Activity without relaunching the application. In this case, use the

startActivity() and appropriate finish() methods. Other Activity transitions are temporary, such as a child Activity displaying a dialog and then returning to the original Activity (which was paused on the Activity stack and now resumes). In this case, the parent Activity launches the child Activity and expects a result. For this, use the startActivityForResult() and onActivityResult() methods.

Transitioning between Activities with Intents

Android applications can have multiple entry points. A specific Activity can be designated as the main Activity to launch by default within the AndroidManifest.xml file;

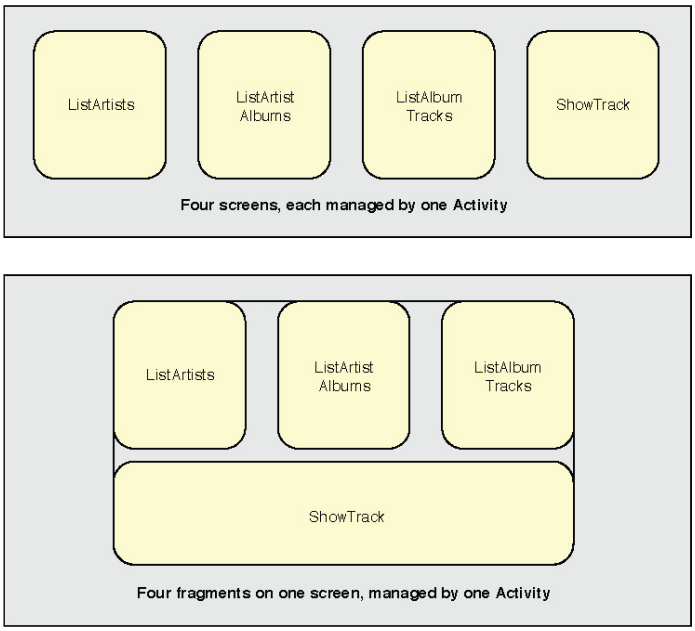


Figure 4.4 How fragments can improve application workflow flexibility

we talk more about this file in Chapter 5, "Defining Your Application Using the Android Manifest File." Other activities might be designated to launch under specific circumstances. For example,
a music application might designate a generic Activity to launch by default from the Application menu but also define specific alternative entry-point activities for accessing specific music playlists by playlist ID or artists by name.

Launching a New Activity by Class Name

You can start activities in several ways. The simplest method is to use the application Context object to call the startActivity() method, which takes a single parameter, an Intent.
An Intent (android.content.Intent) is an asynchronous message mechanism used by the Android operating system to match task requests with the appropriate Activity or Service (launching it, if necessary) and to dispatch broadcast Intent events to the system at large.

For now, though, we focus on the Intent object and how it is used with activities. The following line of code calls the startActivity() method with an explicit Intent. This Intent requests the

launch of the target Activity named MyDrawActivity by its class. This class is implemented elsewhere within the package. startActivity(new Intent(getApplicationContext(), MyDrawActivity.class)); This line of code might be sufficient for some applications, which simply transition from one Activity to the next. However, you can use the Intent mechanism in a much more robust manner. For example, you can use the Intent structure to pass data between activities.

Creating Intents with Action and Data

You've seen the simplest case of using an `Intent` to launch a class by name. Intents need not specify the component or class they want to launch explicitly. Instead, you can create an intent filter and register it within the Android manifest file. An intent filter is used by activities, services, and broadcast receivers to specify which intents each is interested in receiving (and filter out the rest). The Android operating system attempts to resolve the `Intent` requirements and launch the appropriate `Activity` based on the filter criteria. The guts of the `Intent` object are composed of two main parts: the action to be performed and, optionally, the data to be acted upon. You can also specify action/data pairs using `Intent` action types and `Uri` objects. As you saw in Chapter 3, "Writing Your First Android Application," a `Uri` object represents a string that gives the location and name of an object. Therefore, an `Intent` is basically saying "do this" (the action) to "that" (the URI describing to what resource to do the action).
The most common action types are defined in the `Intent` class, including `ACTION_MAIN` (describes the main entry point of an `Activity`) and `ACTION_EDIT` (used in conjunction with a URI to the data edited). You also find action types that generate integration points with activities in other applications, such as the browser or Phone Dialer.

Launching an Activity Belonging to Another Application

Initially, your application might be starting only activities defined within its own package. However, with the appropriate permissions, applications might also launch external activities within other applications. For example, a customer relationship management (CRM) application might launch the Contacts application to browse the Contacts database, choose a specific contact, and return that contact's unique identifier to the CRM application for use.

Here is an example of how to create a simple `Intent` with a predefined action (`ACTION_DIAL`) to launch the Phone Dialer with a specific phone number to dial in the form of a simple `Uri` object:

```
Uri number = Uri.parse("tel:5555551212");
Intent dial = new Intent(Intent.ACTION_DIAL, number);
startActivity(dial);
```

You can find a list of commonly used Google application intents at *http://d.android.com/guide/appendix/g-app-intents.html*. Also available is the developer-managed Registry of Intents protocols at OpenIntents, found at *http://openintents.org/en/intentstable*. A growing list of intents is available from third-party applications and those within the Android SDK.

Passing Additional Information Using Intents

You can also include additional data in an `Intent`. The `Extras` property of an `Intent` is stored in a `Bundle` object. The `Intent` class also has a number of helper methods for getting and setting name/value pairs for many common data types.
For example, the following `Intent` includes two extra pieces of information—a string value and a `boolean`:

```
Intent intent = new Intent(this, MyActivity.class);
intent.putExtra("SomeStringData","Foo");
intent.putExtra("SomeBooleanData",false);
startActivity(intent);
```

Then in the `onCreate()` method of the `MyActivity` class, you can retrieve the extra data sent as follows:

```
Bundle extras = getIntent().getExtras();
if (extras != null) {
String myStr = extras.getString("SomeStringData");
Boolean myBool = extras.getBoolean("SomeBooleanData");
}
```

Organizing Application Navigation with Activities and Intents

As previously mentioned, your application likely has a number of screens, each with its own `Activity`. There is a close relationship between activities and intents, and application navigation. You often see a kind of menu paradigm used in several different ways for application navigation:

- **Main menu or list-style screen:** acts as a switch in which each menu item launches a different `Activity` in an application, for instance, menu items for launching the Play Game `Activity`, the High Scores `Activity`, and the Help `Activity`.
- **Drill-down-list-style screen:** acts as a directory in which each menu item launches the same `Activity`, but each item passes in different data as part of the `Intent` (for example, a menu of all database records). Choosing a specific item might launch the Edit Record `Activity`, passing in that particular item's unique identifier.
- **Click actions:** Sometimes you want to navigate between screens in the form of a wizard. You might set the click handler for a user interface control, such as a "Next" button, to trigger a new `Activity` to start and the current one to finish.
- **Options menus:** Some applications like to hide their navigational options until the user needs them. The user can then click the `Menu` button on the device and launch an options menu, where each option listed corresponds to an `Intent` to launch a different `Activity`. Options menus are no longer recommended for use, as action bars are now the preferred method for presenting options.
- **Action-bar-style navigation:** Action bars are functional title bars with navigational button options, each of which spawns an `Intent` and launches a specific `Activity`. To support action bars on devices running Android versions all the way back to 2.1 (API Level 7), you should use the `android-support-v7-appcompat` Support Library that comes packaged with the SDK.

Defining Your Application Using the Android Manifest File

Android projects use a special configuration file called the Android manifest file to determine application settings—settings such as the application name and version, as well as what permissions the application requires to run and what application components it is composed of.

Configuring Android Applications Using the Android Manifest File

The Android application manifest file is a specially formatted XML file that must accompany each Android application. This file contains important information about the application's identity. Here, you define the application's name and version information as well as what application components the application relies upon, what permissions the application requires to run, and other application configuration information. The Android manifest file is named `AndroidManifest.xml` and must be included at the top level of any Android project. The information in this file is used by the Android system to

- Install and upgrade the application package
- Display the application details, such as the application name, description, and icon, to users
- Specify application system requirements, including which Android SDKs are supported, what device configurations are required (for example, D-pad navigation), and which platform features the application relies upon (for example, multitouch capabilities)
- Specify what features are required by the application for market-filtering purposes
- Register application activities and when they should be launched
- Manage application permissions
- Configure other advanced application component configuration details, including defining services, broadcast receivers, and content providers
- Specify intent filters for your activities, services, and broadcast receivers
- Enable application settings such as debugging and configuring instrumentation for application testing

Editing the Android Manifest File

The manifest resides at the top level of your Android project. You can edit the Android manifest file by using the Android IDE manifest file resource editor, which is a feature of the ADT Bundle, or by manually editing the XML.

Editing the Manifest File Using the Android IDE

You can use the Android IDE manifest file resource editor to edit the project manifest file. The Android IDE manifest file resource editor organizes the manifest information into categories:

- The `Manifest` tab
- The `Application` tab
- The `Permissions` tab
- The `Instrumentation` tab
- The `AndroidManifest.xml` tab

Let's take a closer look at a sample Android manifest file. We've chosen a more complex sample project to illustrate a number of different characteristics of the Android manifest file, as opposed to the very simple default manifest file you configured for the `MyFirstAndroidApp` project. The application manifest we will be discussing is for an application named `SimpleMultimedia`.

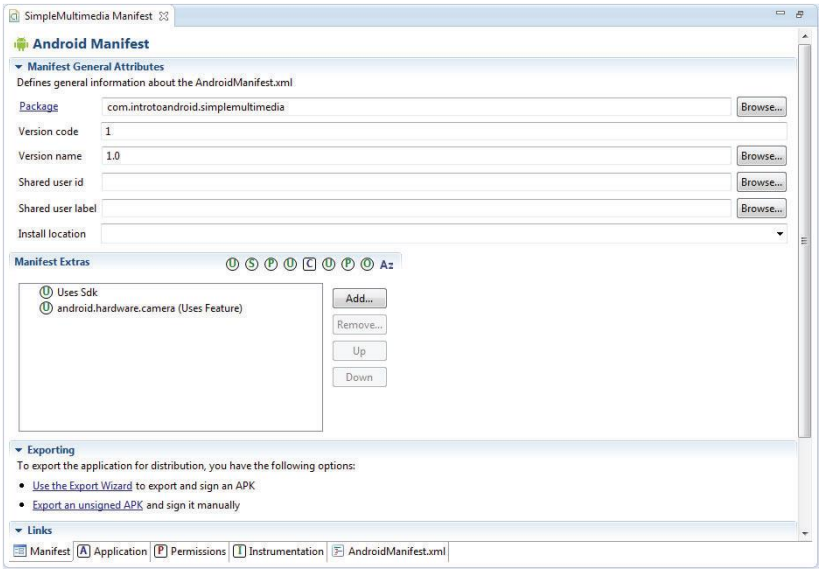


Figure 5.1 The `Manifest` tab of the Android IDE manifest file resource editor.

Configuring Package-Wide Settings Using the **Manifest** Tab

The `Manifest` tab (see Figure 5.1) contains package-wide settings, including the package name, version information, and supported Android SDK information. You can also set any hardware or feature requirements here.

Managing Application and Activity Settings Using the Application Tab

The `Application` tab (see Figure 5.2) contains application-wide settings, including the application label and icon, as well as information about the application components, such as activities, and other application components, including configuration for services, intent filters, and content providers.

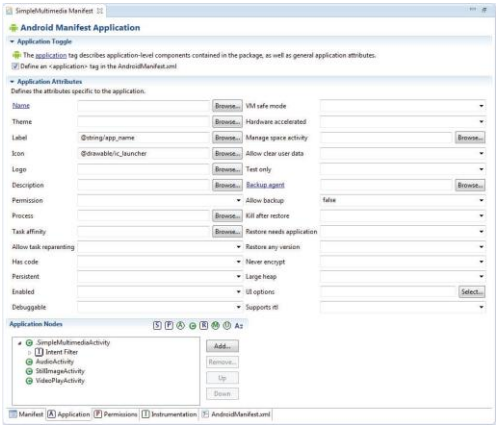


Figure 5.2 The `Application` tab of the Android IDE manifest file resource editor.

Enforcing Application Permissions Using the Permissions Tab

The `Permissions` tab (see Figure 5.3) contains any permission rules required by your application. This tab can also be used to enforce custom permissions created for the application.

Managing Test Instrumentation Using the Instrumentation Tab

The `Instrumentation` tab (seen in Figure 5.4) allows the developer to declare any instrumentation classes for monitoring the application. We talk more about instrumentation and testing in Chapter 18, "Testing Android Applications."

Editing the Manifest File Manually

The Android manifest file is a specially formatted XML file. You can edit the XML manually by clicking the `AndroidManifest.xml` tab.

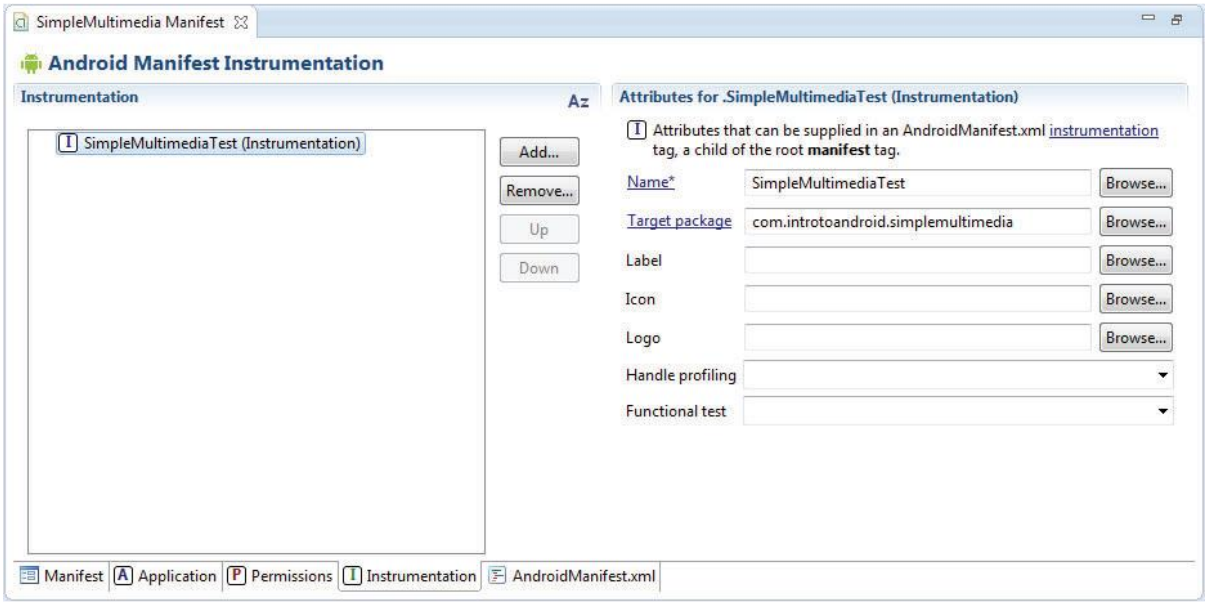


Figure 5.4 The `Instrumentation` tab of the Android IDE manifest file resource editor.

Android manifest files generally include a single `<manifest>` tag with a single `<application>` tag. The following is a sample `AndroidManifest.xml` file for an application called `SimpleMultimedia`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
package="com.introtoandroid.simplemultimedia"
android:versionCode="1"
android:versionName="1.0">
<application android:icon="@drawable/ic_launcher"
android:label="@string/app_name"
android:debuggable="true">
<activity android:name=".SimpleMultimediaActivity"
android:label="@string/app_name">
<intent-filter>
<action
android:name="android.intent.action.MAIN" />
<category
android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
<activity android:name="AudioActivity" />
<activity android:name="StillImageActivity" />
<activity android:name="VideoPlayActivity" />
</application>
<uses-sdk
android:minSdkVersion="10"
android:targetSdkVersion="18" />
<uses-permission
android:name="android.permission.WRITE_SETTINGS" />
<uses-permission
android:name="android.permission.RECORD_AUDIO" />
```

```
<uses-permission
android:name="android.permission.SET_WALLPAPER" />
<uses-permission
android:name="android.permission.CAMERA" />
<uses-permission
android:name="android.permission.INTERNET" />
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-feature
android:name="android.hardware.camera" />
</manifest>
```

Here is a summary of what this file tells us about the `SimpleMultimedia` application:

- The application uses the package name `com.introtoandroid.simplemultimedia`.
- The application version name is 1.0.
- The application version code is 1.
- The application name and label are stored in the resource string called `@string/app_name` within the `/res/values/strings.xml` resource file.
- The application is `debuggable` on an Android device.
- The application icon is the graphics file called `ic_launcher` (which could be a PNG, JPG, or GIF) stored within the `/res/drawable-*` directory (there are actually multiple versions for different pixel densities).
- The application has four activities (`SimpleMultimediaActivity`, `AudioActivity`, `StillImageActivity`, and `VideoPlayActivity`).
- `SimpleMultimediaActivity` is the primary entry point for the application because it handles the action `android.intent.action.MAIN`. This `Activity` shows in the application launcher, because its category is `android.intent.category.LAUNCHER`.
- The application requires the following permissions to run: the ability to write settings, the ability to record audio, the ability to set the wallpaper on the device, the ability to access the built-in camera, the ability to communicate over the Internet, and the ability to write to external storage.
- The application works from any API level from 10 to 18; in other words, Android SDK 2.3.3 is the lowest supported platform version, and the application was written to target Jelly Bean MR2 (for example, Android 4.3).
- Finally, the application requests to use the camera with the `<uses-feature>` tag.