Chapter 1

Introducing the Android Computing Platform

Computing is more accessible than ever before. Handheld devices have transformed into computing platforms. Be it a phone or a tablet, the mobile device is now so capable of general-purpose computing that it's becoming the *real* personal computer (PC). Every traditional PC manufacturer is producing devices of various form factors based on the Android OS. The battles between operating systems, computing platforms, programming languages, and development frameworks are being shifted and reapplied to mobile devices.

We are also seeing a surge in mobile programming as more and more IT applications start to offer mobile counterparts. In this book, we'll show you how to take advantage of your Java skills to write programs for devices that run on Google's Android platform (`http://developer.android.com/index.html`), an open source platform for mobile and tablet development.

NOTE: We are excited about Android because it is an advanced Java-based platform that introduces a number of new paradigms in framework design (even with the limitations of a mobile platform).

In this chapter, we'll provide an overview of Android and its SDK, give a brief overview of key packages, introduce what we are going to cover in each chapter, show you how to take advantage of Android source code, and highlight the benefits of programming for the Android platform.

A New Platform for a New Personal Computer

The Android platform embraces the idea of general-purpose computing for handheld devices. It is a comprehensive platform that features a Linux-based operating system stack for managing devices, memory, and processes. Android's Java libraries cover telephony, video, speech, graphics, connectivity, UI programming, and a number of other aspects of the device.

NOTE: Although built for mobile- and tablet-based devices, the Android platform exhibits the characteristics of a full-featured desktop framework. Google makes this framework available to Java programmers through a Software Development Kit (SDK) called the Android SDK. When you are working with the Android SDK, you rarely feel that you are writing to a mobile device because you have access to most of the class libraries that you use on a desktop or a server—including a relational database.

The Android SDK supports most of the Java Platform, Standard Edition (Java SE), except for the Abstract Window Toolkit (AWT) and Swing. In place of AWT and Swing, Android SDK has its own *extensive modern UI framework*. Because you're programming your applications in Java, you could expect that you need a Java Virtual Machine (JVM) that is responsible for interpreting the runtime Java byte code. A JVM typically provides the necessary optimization to help Java reach performance levels comparable to compiled languages such as C and C++. Android offers its own optimized JVM to run the compiled Java class files in order to counter the handheld device limitations such as memory, processor speed, and power. This virtual machine is called the Dalvik VM, which we'll explore in a later section, "Delving into the Dalvik VM."

NOTE: The familiarity and simplicity of the Java programming language, coupled with Android's extensive class library, makes Android a compelling platform to write programs for.

Figure 1–1 provides an overview of the Android software stack. (We'll provide further details in the section "Understanding the Android Software Stack.")

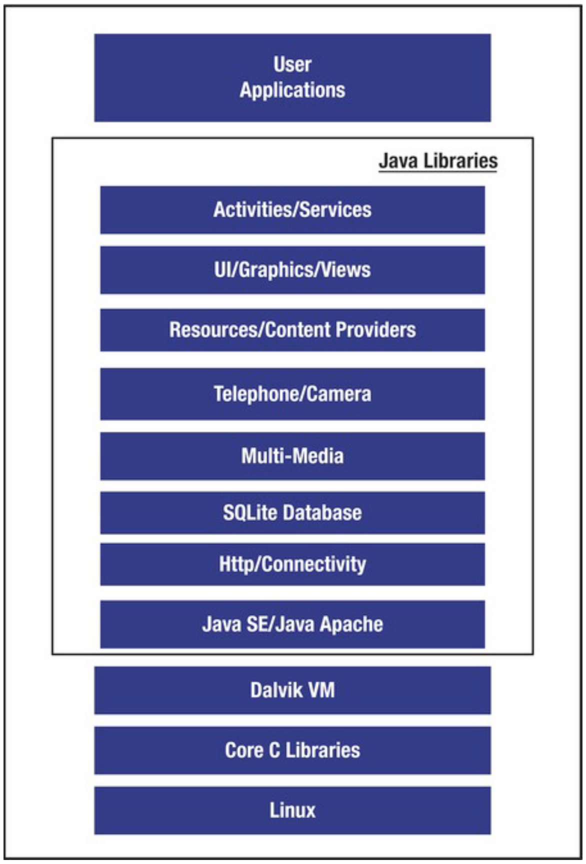

Figure 1–1. *High-level view of the Android software stack*

Early History of Android

Mobile phones use a variety of operating systems, such as Symbian OS, Microsoft's Windows Phone OS, Mobile Linux, iPhone OS (based on Mac OS X), Moblin (from Intel), and many other proprietary OSs. So far, no single OS has become the de facto standard. The available APIs and environments for developing mobile applications are too restrictive and seem to fall behind when compared to desktop frameworks. In contrast, the Android platform promised openness, affordability, open source code, and, more important, a high-end, all-in-one-place, consistent development framework.

Google acquired the startup company Android Inc. in 2005 to start the development of the Android platform (see Figure 1–2). The key players at Android Inc. included Andy Rubin, Rich Miner, Nick Sears, and Chris White.

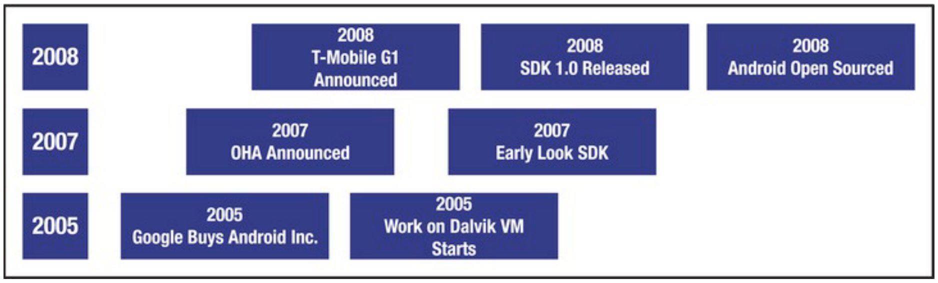
2008
2008 T-Mobile G1 Announced
2008 SDK 1.0 Released
2008 Android Open Sourced
2007
2007 OHA Announced
2007 Early Look SDK
2005
2005 Google Buys Android Inc.
2005 Work on Dalvik VM Starts

Figure 1–2. *Android early timeline*

The Android SDK was first issued as an “early look” release in November 2007. In September 2008, T-Mobile announced the availability of T-Mobile G1, the first smartphone based on the Android platform. Since then we have seen the SDKs 2.0, 3.0, and now 4.0, roughly one every year. The devices that run Android started out as a trickle but now are a torrent.

One of Androids key architectural goals is to allow applications to interact with one another and reuse components from one another. This reuse applies not only to services, but also to data and the user interface (UI).

Android has attracted an early following and sustained the developer momentum because of its fully developed features to exploit the cloud-computing model offered by web resources and to enhance that experience with local data stores on the handset itself. Android’s support for a relational database on the handset also played a part in early adoption.

In releases 1.0 and 1.1 (2008) Android did not support soft keyboards, requiring the devices to carry physical keys. Android fixed this issue by releasing the 1.5 SDK in April 2009, along with a number of other features, such as advanced media-recording capabilities, widgets, and live folders.

In September 2009 came release 1.6 of the Android OS and, within a month, Android 2.0 followed, facilitating a flood of Android devices in time for the 2009 Christmas season. This release introduced advanced search capabilities and text to speech.

In Android 2.3, the significant features include remote wiping of secure data by administrators, the ability to use camera and video in low-light conditions, Wi-Fi hotspot, significant performance improvements, improved Bluetooth functionality, installation of applications on the SD card optionally, OpenGL ES 2.0 support, improvements in backup, improvements in search usability, Near Field Communications support for credit card processing, much improved motion and sensor support (similar to Wii), video chat, and improved Market.

Android 3.0 is focused on tablet-based devices and much more powerful dual core processors such as NVIDIA Tegra 2. The main features of this release include support to use a larger screen. A significantly new concept called fragments has been introduced. Fragments permeate the 3.0 experience. More desktop-like capabilities, such as the action bar and drag-and-drop, have been introduced. Home-screen widgets have been significantly enhanced, and more UI controls are now available. In the 3D space, OpenGL has been enhanced with Renderscript to further supplement ES 2.0. It is an exciting introduction for tablets.

However, the 3.0 experience is limited to tablets. At the time of the 3.0 release, the 2.x branch of Android continued to serve phones while 3.x branches served the tablets. Starting with 4.0, Android has merged these branches and forged a single SDK. For phone users, the primary UI difference is that the tablet experience is brought to phones as well.

The key aspects of the 4.0 user experience are as follows:

- A new type face called Roboto to provide crispness on high-density screens.
- A better way to organize apps into folders on home pages.
- Ability to drag apps and folders into the favorites tray that is always present at the bottom of the device.
- Optimization of notifications based on device type. For small devices, they show up on the top, and for larger devices they show up in the bottom system bar.
- Resizable, scrollable widgets.
- A variety of ways to unlock screens.
- Spell checker.
- Improved voice input with a “speak continuously” option.
- More controls to work with network data usage.
- Enhanced Contacts application with a personal profile much like social networks.
- Enhancements to the calendar application.
- Better camera app: continuous focus, zero shutter lag, face detection, tap to focus, and a photo editor.
- Live effects on pictures and videos for silly effects.
- A quick way to take and share screen shots.
- Browser performance that is twice as fast.
- Improved e-mail.
- A new concept called Android beaming for NFC-based sharing.
- Support for Wi-Fi Direct to promote P2P services.
- Bluetooth health device profile.

Key aspects of developer support for 4.0 include

- Revamped animation based on changing properties of objects, including views
- Fixed number of list-based widget behaviors from 3.0
- Much more mature action bar with integrated search
- Support for a number of mobile standards: Advanced Audio Distribution Profile (A2DP: the ability to use external speakers), Real-time Transport Protocol RTP: to stream audio/video over IP), Media Transfer Protocol (MTP), Picture Transfer Protocol (PTP: for hooking up to computers to download photos and media), and Bluetooth Headset Profile (HSP)
- Full device encryption
- Digital Rights Management (DRM)
- Encrypted storage and passwords
- Social API involving personal profiles
- Enhanced Calendar API
- Voice Mail API

Delving Into the Dalvik VM

As part of Android, Google has spent a lot of time thinking about optimizing designs for low-powered handheld devices. Handheld devices lag behind their desktop counterparts in memory and speed by eight to ten years. They also have limited power for computation. The performance requirements on handsets are severe as a result, requiring handset designers to optimize everything. If you look at the list of packages in Android, you'll see that they are fully featured and extensive.

These issues led Google to revisit the standard JVM implementation in many respects. The key figure in Google's implementation of this JVM is Dan Bornstein, who wrote the Dalvik VM—Dalvik is the name of a town in Iceland. Dalvik VM takes the generated Java class files and combines them into one or more Dalvik Executable (`.dex`) files. The goal of the Dalvik VM is to find every possible way to optimize the JVM for space, performance, and battery life.

The final executable code in Android, as a result of the Dalvik VM, is based not on Java byte code but on `.dex` files instead. This means you cannot directly execute Java byte code; you have to start with Java class files and then convert them to linkable `.dex` files.

Understanding the Android Software Stack

So far we've covered Android's history and its optimization features, including the Dalvik VM, and we've hinted at the Java programming stack available. In this section, we will cover the development aspect of Android. Figure 1–3 shows the Android software stack from a developer's perspective.

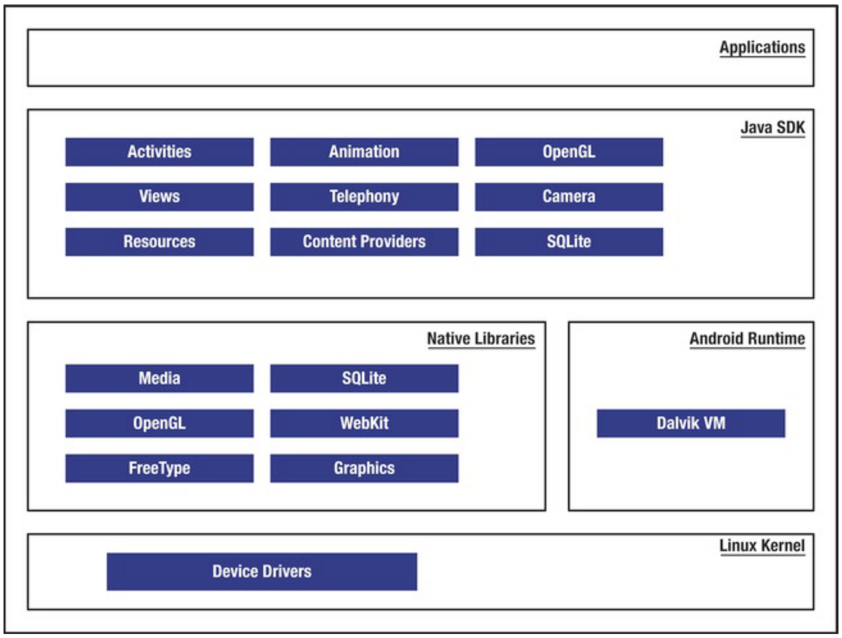

Figure 1–3. *Detailed Android SDK software stack*

At the core of the Android platform is a Linux kernel responsible for device drivers, resource access, power management, and other OS duties. The supplied device drivers include Display, Camera, Keypad, Wi-Fi, Flash Memory, Audio, and inter-process communication (IPC). Although the core is Linux, the majority—if not all—of the applications on an Android device such as a Motorola Droid are developed in Java and run through the Dalvik VM.

Sitting at the next level, on top of the kernel, are a number of C/C++ libraries such as OpenGL, WebKit, FreeType, Secure Sockets Layer (SSL), the C runtime library (`libc`), SQLite, and Media. The system C library based on Berkeley Software Distribution (BSD) is tuned (to roughly half its original size) for embedded Linux-based devices. The media libraries are based on PacketVideo's (`www.packetvideo.com/`) OpenCORE. These libraries are responsible for recording and playback of audio and video formats. A library called Surface Manager controls access to the display system and supports 2D and 3D.

NOTE: These core libraries are subject to change because they are all internal implementation details of Android and not directly exposed to the published Android API. We have indicated these core libraries just to inform you of the nature of the underbelly of Android. Refer to the Android developer site for updates and future insight.

The WebKit library is responsible for browser support; it is the same library that supports Google Chrome and Apple's Safari. The FreeType library is responsible for font support. SQLite (`www.sqlite.org/`) is a relational database that is available on the device itself. SQLite is also an independent open source effort for relational databases and not directly tied to Android. You can acquire and use tools meant for SQLite for Android databases as well.

Most of the application framework accesses these core libraries through the Dalvik VM, the gateway to the Android platform. As we indicated in the previous sections, Dalvik is optimized to run multiple instances of VMs. As Java applications access these core libraries, each application gets its own VM instance.

The Android Java API's main libraries include telephony, resources, locations, UI, content providers (data), and package managers (installation, security, and so on). Programmers develop end-user applications on top of this Java API. Some examples of end-user applications on the device include Home, Contacts, Phone, and Browser.

Android also supports a custom Google 2D graphics library called Skia, which is written in C and C++. Skia also forms the core of the Google Chrome browser. The 3D APIs in Android, however, are based on an implementation of OpenGL ES from the Khronos group (`www.khronos.org`). OpenGL ES contains subsets of OpenGL that are targeted toward embedded systems.

From a media perspective, the Android platform supports the most common formats for audio, video, and images. From a wireless perspective, Android has APIs to support Bluetooth, EDGE, 3G, Wi-Fi, and Global System for Mobile Communication (GSM) telephony, depending on the hardware.

Developing an End-User Application with the Android SDK

In this section, we'll introduce you to the high-level Android Java APIs that you'll use to develop end-user applications on Android. We will briefly talk about the Android emulator, Android foundational components, UI programming, services, media, telephony, animation, and more.

Android Emulator

The Android SDK ships with an Eclipse plug-in called Android Development Tools (ADT). You will use this Integrated Development Environment (IDE) tool for developing, debugging, and testing your Java applications. (We'll cover ADT in depth in Chapter 2.) You can also use the Android SDK without using ADT; you'd use command-line tools instead. Both approaches support an emulator that you can use to run, debug, and test your applications. You will not even need the real device for 90% of your application development. The full-featured Android emulator mimics most of the device features. The emulator limitations include USB connections, camera and video capture, headphones, battery simulation, Bluetooth, Wi-Fi, NFC, and OpenGL ES 2.0.

The Android emulator accomplishes its work through an open source “processor emulator” technology called QEMU, developed by Fabrice Bellard (`http://wiki.qemu.org/Index.html`). This is the same technology that allows emulation of one operating system on top of another, regardless of the processor. QEMU allows emulation at the CPU level.

With the Android emulator, the processor is based on Advanced RISC Machine (ARM). ARM is a 32-bit microprocessor architecture based on Reduced Instruction Set Computing (RISC), in which design simplicity and speed is achieved through a reduced number of instructions in an instruction set. The emulator runs the Android version of Linux on this simulated processor.

ARM is widely used in handhelds and other embedded electronics where lower power consumption is important. Much of the mobile market uses processors based on this architecture.

You can find more details about the emulator in the Android SDK documentation at `http://developer.android.com/guide/developing/tools/emulator.html`.

The Android UI

Android uses a UI framework that resembles other desktop-based, full-featured UI frameworks. In fact, it’s more modern and more asynchronous in nature. The Android UI is essentially a fourth-generation UI framework, if you consider the traditional C-based Microsoft Windows API the first generation and the C++-based Microsoft Foundation Classes (MFC) the second generation. The Java-based Swing UI framework would be the third generation, introducing design flexibility far beyond that offered by MFC. The Android UI, JavaFX, Microsoft Silverlight, and Mozilla XML User Interface Language (XUL) fall under this new type of fourth-generation UI framework, in which the UI is declarative and independently themed.

NOTE: In Android, you program using a modern user interface paradigm, even though the device you’re programming for may be a handheld.

Programming in the Android UI involves declaring the interface in XML files. You then load these XML view definitions as windows in your UI application. This is very much like HTML-based web pages. Much as in HTML, you find (get hold of) the individual controls through their IDs and manipulate them with Java code.

Even menus in your Android application are loaded from XML files. Screens or windows in Android are often referred to as *activities*, which comprise multiple views that a user needs in order to accomplish a logical unit of action. *Views* are Android’s basic UI building blocks, and you can further combine them to form composite views called *view groups*.

Views internally use the familiar concepts of canvases, painting, and user interaction. An activity hosting these composite views, which include views and view groups, is the logical replaceable UI component in Android.

Android 3.0 introduced a new UI concept called *fragments* to allow developers to chunk views and functionality for display on tablets. Tablets provide enough screen space for multipane activities, and fragments provide the abstraction for the panes.

One of the Android framework's key concepts is the life cycle management of activity windows. Protocols are put in place so that Android can manage state as users hide, restore, stop, and close activity windows. You will get a feel for these basic ideas in Chapter 2, along with an introduction to setting up the Android development environment.

The Android Foundational Components

The Android UI framework, along with many other parts of Android, relies on a new concept called an *intent*. An intent is an intra- and interprocess mechanism to invoke components in Android.

A component in Android is a piece of code that has a well defined life cycle. An activity representing a window in an Android application is a component. A service that runs in an Android process and serves other clients is a component. A receiver that wakes up in response to an event is another example of a component in Android.

While serving this primary need of invoking components, an intent exhibits the characteristics parallel to those of windowing messages, actions, publish-and-subscribe models, and interprocess communications. Here is an example of using the `Intent` class to invoke or start a web browser:

```
public static void invokeWebBrowser(Activity activity)
{
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(Uri.parse("http://www.google.com"));
    activity.startActivity(intent);
}
```

In this example, through an intent, we are asking Android to start a suitable window to display the content of a web site. Depending on the list of browsers that are installed on the device, Android will choose a suitable one to display the site. You will learn more about intents in Chapter 5.

Android has extensive support for *resources*, which include such things as strings and bitmaps, as well as some not-so-familiar items like XML-based view (layout like HTML) definitions. The Android framework makes use of resources in a novel way so their usage is easy, intuitive, and convenient. Here is an example where resource IDs are automatically generated for resources defined in XML files:

```
public final class R {
    //All string resources will have constants auto generated here
    public static final class string {
        public static final int hello=0x7f070000;
    }
    //All image files will have unique ids generated here
    public static final class drawable {
        public static final int myanimation=0x7f020001;
        public static final int numbers19=0x7f02000e;
    }
    //View ids are auto generated based on their names
    public static final class id {
        public static final int textViewId1=0x7f080003;
    }
    //The following are two files (like html) that define layout
    //auto generated from the filenames in respective sub directories.
    public static final class layout {
        public static final int frame_animations_layout=0x7f030001;
        public static final int main=0x7f030002;
    }
}
```

Each auto-generated ID in this class corresponds to either an element in an XML file or a whole file itself. Wherever you would like to use those XML definitions, you will use these generated IDs instead. This indirection helps a great deal when it comes to specializing resources based on locale, device size, and so on. (Chapter 3 covers the `R.java` file and resources in more detail.)

Another new concept in Android is the *content provider*. A content provider is an abstraction of a data source that makes it look like an emitter and consumer of RESTful services. The underlying SQLite database makes this facility of content providers a powerful tool for application developers. We will cover content providers in Chapter 4. In Chapters 3, 4, and 5, we'll discuss how intents, resources, and content providers promote openness in the Android platform.

Advanced UI Concepts

XML page-layout definitions (similar to HTML web pages) play a critical role in describing the Android UI. Let's look at an example of how an Android layout XML file does this for a simple layout containing a text view:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- place it in /res/layout/sample_page1.xml -->
<!-- will auto generate an id called: R.layout.sample_page1 -->
<LinearLayout ..some basic attributes..>
<TextView android:id="@+id/textViewId"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    />
</LinearLayout>
```

You will use an ID generated for this XML file to load this layout into an activity window. (We'll cover this process in Chapter 6.) Android also provides extensive support for menus (more on that in Chapter 7), from standard menus to context menus. You'll find it convenient to work with menus in Android, because they are also loaded as XML files and because resource IDs for those menus are auto-generated. Here's how you would declare menus in an XML file:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- This group uses the default category. -->
    <group android:id="@+id/menuGroup_Main">
        <item android:id="@+id/menu_clear"
            android:orderInCategory="10"
            android:title="clear" />
        <item android:id="@+id/menu_show_browser"
            android:orderInCategory="5"
            android:title="show browser" />
    </group>
</menu>
```

Android supports dialogs, and all dialogs in Android are asynchronous. These asynchronous dialogs present a special challenge to developers accustomed to the synchronous modal dialogs in some windowing frameworks. We'll address menus in Chapter 7 and dialogs in Chapter 9.

Android offers extensive support for animation. There are three fundamental ways to accomplish animation. You can do frame-by-frame animation. Or you can provide tweening animation by changing view transformation matrices (position, scale, rotation, and alpha). Or you can also do tweening animation by changing properties of objects. The property-based animation is introduced in 3.0 and is the most flexible and recommended way to accomplish animation. All of these animations are covered in Chapter 21.

Moreover, Android allows you to define these animations in an XML resource file. Check out this example, in which a series of numbered images is played in frame-by-frame animation:

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
        android:oneshot="false">
    <item android:drawable="@drawable/numbers11" android:duration="50" />
   ……
    <item android:drawable="@drawable/numbers19" android:duration="50" />
 </animation-list>
```

Android also supports 3D graphics through its implementation of the OpenGL ES 1.0 and 2.0 standards. OpenGL ES, like OpenGL, is a C-based flat API. The Android SDK, because it's a Java-based programming API, needs to use Java binding to access the OpenGL ES. Java ME has already defined this binding through Java Specification Request (JSR) 239 for OpenGL ES, and Android uses the same Java binding for OpenGL ES in its implementation. If you are not familiar with OpenGL programming, the learning curve is steep. Due to space limitations, we are not able to include the OpenGL coverage in the fourth edition of the book. However we have over 100 pages of coverage in our third edition.

Android has a number of new concepts that revolve around *information at your fingertips* using the home screen. The first of these is *live folders*. Using live folders, you can publish a collection of items as a folder on the homepage. The contents of this collection change as the underlying data changes. This changing data could be either on the device or from the Internet. Due to space limitations, we are not able to cover live folders in the fourth edition of the book. However, the third edition includes extensive coverage.

The second homepage-based idea is the *home screen widget*. Home screen widgets are used to paint information on the homepage using a UI widget. This information can change at regular intervals. An example could be the number of e-mail messages in your e-mail store. We describe home screen widgets in Chapter 25. The home screen widgets are enhanced in 3.0 to include list views that can get updated when their underlying data changes. These enhancements are covered in Chapter 26.

Integrated Android Search is the third homepage-based idea. Using integrated search, you can search for content both on the device and also across the Internet. Android search goes beyond search and allows you to fire off commands through the search control. Due to space limitations, we can't include coverage of the search API in the fourth edition of the book. However, it is covered in the third edition.

Android also supports touchscreen and gestures based on finger movements on the device. Android allows you to record any random motion on the screen as a named gesture. This gesture can then be used by applications to indicate specific actions. We cover touchscreens and gestures in Chapter 27.

Sensors are now becoming a significant part of the mobile experience. We cover sensors in Chapter 29.

Another necessary innovation required for a mobile device is the dynamic nature of its configurations. For instance, it is very easy to change the viewing mode of a handheld between portrait and landscape. Or you may dock your handheld to become a laptop. Android 3.0 has introduced a concept called fragments to deal with these variations effectively. Chapter 8 is dedicated to fragments, and Chapter 12 talks about how to deal with configuration changes.

We also cover the 3.0 feature (which is much enhanced in 4.0) of action bars in Chapter 10. Action bars bring Android up to par with a desktop menu bar paradigm.

Drag-and-drop is introduced for tablets in 3.0. This feature is now available to phones as well. We cover drag-and-drop in Chapter 28.

Handheld devices are fully aware of a cloud-based environment. To make server-side HTTP calls, it is important to understand the threading model to avoid Application Not Responding messages. We cover the mechanisms available for asynchronous processing in Chapter 18.

Outside of the Android SDK, there are a number of independent innovations taking place to make development exciting and easy. Some examples are XML/VM, PhoneGap, and Titanium. Titanium allows you to use HTML technologies to program the WebKit-based Android browser. We covered Titanium in the second edition of this book. However, due to time and space limitations, we are not covering Titanium in this edition.

Android Service Components

Security is a fundamental part of the Android platform. In Android, security spans all phases of the application life cycle—from design-time policy considerations to runtime boundary checks. We cover security and permissions in Chapter 14.

In Chapter 15, we'll show you how to build and consume services in Android, specifically HTTP services. This chapter will also cover interprocess communication (communication between applications on the same device).

Location-based service is another of the more exciting components of the Android SDK. This portion of the SDK provides application developers with APIs to display and manipulate maps, as well as obtain real-time device-location information. We'll cover these ideas in detail in Chapter 22.

Android Media and Telephony Components

Android has APIs that cover audio, video, and telephony components. Chapter 23 will address the telephony API. We'll cover the audio and video APIs extensively in Chapter 24.

Starting with Android 2.0, Android includes the Pico Text-to-Speech engine. Due to space limitations, we are not able to include Text-to-Speech coverage in the fourth edition of the book. The third edition does cover the Text-to-Speech API.

Last but not least, Android ties all these concepts into an application by creating a single XML file that defines what an application package is. This file is called the application's *manifest file* (`AndroidManifest.xml`). Here is an example:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.ai.android.HelloWorld"
      android:versionCode="1"
      android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".HelloWorld"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The Android manifest file is where activities are defined, where services and content providers are registered, and where permissions are declared. Details about the manifest file will emerge throughout the book as we develop each idea.

Android Java Packages

One way to get a quick snapshot of the Android platform is to look at the structure of Java packages. Because Android deviates from the standard JDK distribution, it is important to know what is supported and what is not. Here's a brief description of the important packages that are included in the Android SDK:

`android.app`: Implements the Application model for Android. Primary classes include `Application`, representing the start and stop semantics, as well as a number of activity-related classes, fragments, controls, dialogs, alerts, and notifications. We work with most of these classes through out this book.

`android.app.admin`: Provides the ability to control the device by folks such as enterprise administrators.

`android.accounts`: Provides classes to manage accounts such as Google, Facebook, and so on. The primary classes are `AccountManager` and `Account`. We cover this API briefly in Chapter 30 when we discuss the Contacts API.

`android.animation`: Hosts all the new property animation classes. These clases are extensively covered in Chapter 21.

`android.app.backup`: Provides hooks for applications to back up and restore their data when folks switch their devices.

`android.appwidget`: Provides functionality for home screen widgets. This package is covered extensively in Chapter 25 and Chapter 26 when we talk about home screen widgets, including list-based widgets.

`android.bluetooth`: Provides a number of classes to work with Bluetooth functionality. The main classes include `BluetoothAdapter`, `BluetoothDevice`, `BluetoothSocket`, `BluetoothServerSocket`, and `BluetoothClass`. You can use `BluetoothAdapter` to control the locally installed Bluetooth adapter. For example, you can enable it, disable it, and start the discovery process. `BluetoothDevice` represents the remote Bluetooth device that you are connecting with. The two Bluetooth sockets are used to establish communication between the devices. A Bluetooth class represents the type of Bluetooth device you are connecting to.

`android.content`: Implements the concepts of content providers. Content providers abstract out data access from data stores. This package also implements the central ideas around intents and Android Uniform Resource Identifiers (URIs). These classes are covered in Chapter 4.

android.content.pm: Implements package manager–related classes. A package manager knows about permissions, installed packages, installed providers, installed services, installed components such as activities, and installed applications.

android.content.res: Provides access to resource files, both structured and unstructured. The primary classes are `AssetManager` (for unstructured resources) and `Resources`. Some of the classes from this package are covered in Chapter 3.

android.database: Implements the idea of an abstract database. The primary interface is the `Cursor` interface. Some of the classes from this package are covered in Chapter 4.

android.database.sqlite: Implements the concepts from the `android.database` package using SQLite as the physical database. Primary classes are `SQLiteCursor`, `SQLiteDatabase`, `SQLiteQuery`, `SQLiteQueryBuilder`, and `SQLiteStatement`. However, most of your interaction is going to be with classes from the abstract `android.database` package.

android.drm: Classes related to Digital Rights Management.

android.gesture: Houses all the classes and interfaces necessary to work with user-defined gestures. Primary classes are `Gesture`, `GestureLibrary`, `GestureOverlayView`, `GestureStore`, `GestureStroke`, and `GesturePoint`. A `Gesture` is a collection of `GestureStrokes` and `GesturePoints`. Gestures are collected in a `GestureLibrary`. Gesture libraries are stored in a `GestureStore`. Gestures are named so that they can be identified as actions. Some of the classes from this package are covered in Chapter 27.

android.graphics: Contains the classes `Bitmap`, `Canvas`, `Camera`, `Color`, `Matrix`, `Movie`, `Paint`, `Path`, `Rasterizer`, `Shader`, `SweepGradient`, and `TypeFace`.

android.graphics.drawable: Implements drawing protocols and background images, and allows animation of drawable objects.

android.graphics.drawable.shapes: Implements shapes including `ArcShape`, `OvalShape`, `PathShape`, `RectShape`, and `RoundRectShape`.

android.hardware: Implements the physical Camera-related classes. The Camera represents the hardware camera, whereas `android.graphics.Camera` represents a graphical concept that's not related to a physical camera at all.

android.hardware.usb: Lets you talk to USB devices from Android.

android.location: Contains the classes Address, GeoCoder, Location, LocationManager, and LocationProvider. The Address class represents the simplified Extensible Address Language (XAL). GeoCoder allows you to get a latitude/longitude coordinate given an address, and vice versa. Location represents the latitude/longitude. Some of the classes from this package are covered in Chapter 22.

android.media: Contains the classes MediaPlayer, MediaRecorder, Ringtone, AudioManager, and FaceDetector. MediaPlayer, which supports streaming, is used to play audio and video. MediaRecorder is used to record audio and video. The Ringtone class is used to play short sound snippets that could serve as ringtones and notifications. AudioManager is responsible for volume controls. You can use FaceDetector to detect people's faces in a bitmap. Some of the classes from this package are covered in Chapter 24.

android.media.audiofx: Provides audio effects.

android.media.effect: Provides video effects.

android.mtp: Provides the ability to interact with cameras and music devices.

android.net: Implements the basic socket-level network APIs. Primary classes include Uri, ConnectivityManager, LocalSocket, and LocalServerSocket. It is also worth noting here that Android supports HTTPS at the browser level and also at the network level. Android also supports JavaScript in its browser.

android.net.rtp: Supports streaming protocols.

android.net.sip: Provides support for VOIP.

android.net.wifi: Manages Wi-Fi connectivity. Primary classes include WifiManager and WifiConfiguration. WifiManager is responsible for listing the configured networks and the currently active Wi-Fi network.

android.net.wifi.p2p: Supports P2P networks with Wi-Fi Direct.

android.nfc: Lets you interact with devices in close proximity to enable touchless commerce such as credit card processing at sales counters.

android.opengl: Contains utility classes surrounding OpenGL ES 1.0 and 2.0 operations. The primary classes of OpenGL ES are implemented in a different set of packages borrowed from JSR 239. These packages are javax.microedition.khronos.opengles, javax.microedition.khronos.egl, and javax.microedition.khronos.nio. These packages are thin wrappers around the Khronos implementation of OpenGL ES in C and C++.

android.os: Represents the OS services accessible through the Java programming language. Some important classes include `BatteryManager`, `Binder`, `FileObserver`, `Handler`, `Looper`, and `PowerManager`. `Binder` is a class that allows interprocess communication. `FileObserver` keeps tabs on changes to files. You use `Handler` classes to run tasks on the message thread and `Looper` to run a message thread.

android.preference: Allows applications to have users manage their preferences for that application in a uniform way. The primary classes are `PreferenceActivity`, `PreferenceScreen`, and various preference-derived classes such as `CheckBoxPreference` and `SharedPreferences`. Some of the classes from this package are covered in Chapter 13 and Chapter 25.

android.provider: Comprises a set of prebuilt content providers adhering to the `android.content.ContentProvider` interface. The content providers include `Contacts`, `MediaStore`, `Browser`, and `Settings`. This set of interfaces and classes stores the metadata for the underlying data structures. We cover many of the classes from the Contacts provider package in Chapter 30.

android.sax: Contains an efficient set of Simple API for XML (SAX) parsing utility classes. Primary classes include `Element`, `RootElement`, and a number of `ElementListener` interfaces.

android.speech.:* Provides support for converting text to speech. The primary class is `TextToSpeech`. You will be able to take text and ask an instance of this class to queue the text to be spoken. You have access to a number of callbacks to monitor when the speech has finished, for example. Android uses the Pico Text-to-Speech (TTS) engine from SVOX.

android.telephony: Contains the classes `CellLocation`, `PhoneNumberUtils`, and `TelephonyManager`. `TelephonyManager` lets you determine cell location, phone number, network operator name, network type, phone type, and Subscriber Identity Module (SIM) serial number. Some of the classes from this package are covered in Chapter 23.

android.telephony.gsm: Allows you to gather cell location based on cell towers and also hosts classes responsible for SMS messaging. This package is called GSM because Global System for Mobile Communication is the technology that originally defined the SMS data-messaging standard.

android.telephony.cdma: Provides support for CDMA telephony.

android.test, *android.test.mock*, *android.test.suitebuilder:* Packages to support writing unit tests for Android applications.

android.text: Contains text-processing classes.

android.text.method: Provides classes for entering text input for a variety of controls.

android.text.style: Provides a number of styling mechanisms for a span of text.

android.utils: Contains the classes `Log`, `DebugUtils`, `TimeUtils`, and `Xml`.

android.view: Contains the classes `Menu`, `View`, and `ViewGroup`, and a series of listeners and callbacks.

android.view.animation: Provides support for tweening animation. The main classes include `Animation`, a series of interpolators for animation, and a set of specific animator classes that include `AlphaAnimation`, `ScaleAnimation`, `TranslationAnimation`, and `RotationAnimation`.Some of the classes from this package are covered in Chapter 21.

android.view.inputmethod: Implements the input-method framework architecture.

android.webkit: Contains classes representing the web browser. The primary classes include `WebView`, `CacheManager`, and `CookieManager`.

android.widget: Contains all of the UI controls usually derived from the `View` class. Primary widgets include `Button`, `Checkbox`, `Chronometer`, `AnalogClock`, `DatePicker`, `DigitalClock`, `EditText`, `ListView`, `FrameLayout`, `GridView`, `ImageButton`, `MediaController`, `ProgressBar`, `RadioButton`, `RadioGroup`, `RatingButton`, `Scroller`, `ScrollView`, `Spinner`, `TabWidget`, `TextView`, `TimePicker`, `VideoView`, and `ZoomButton`.

com.google.android.maps: Contains the classes `MapView`, `MapController`, and `MapActivity`, essentially classes required to work with Google maps.

These are some of the critical Android-specific packages. From this list, you can see the depth of the Android core platform.

> **NOTE:** In all, the Android Java API contains more than 50 packages and more than 1,000 classes, and it keeps growing with each release.

In addition, Android provides a number of packages in the `java.*` namespace. These include `awt.font`, beans, `io`, `lang`, `lang.annotation`, `lang.ref`, `lang.reflect`, `math`, `net`, `nio`, `nio.channels`, `nio.channels.spi`, `nio.charset`, `security`, `security.acl`, `security.cert`, `security.interfaces`, `security.spec`, `sql`, `text`, `util`, `util.concurrent`, `util.concurrent.atomic`, `util.concurrent.locks`, `util.jar`, `util.logging`, `util.prefs`, `util.regex`, and `util.zip`.

Android comes with these packages from the `javax` namespace: `crypto`, `crypto.spec`, `microedition.khronos.egl`, `microedition.khronos.opengles`, `net`, `net.ssl`, `security.auth`, `security.auth.callback`, `security.auth.login`, `security.auth.x500`, `security.cert`, `sql`, `xml`, and `xmlparsers`.

In addition to these, it contains a lot of packages from `org.apache.http.*` as well as `org.json`, `org.w3c.dom`, `org.xml.sax`, `org.xml.sax.ext`, `org.xml.sax.helpers`, `org.xmlpull.v1`, and `org.xmlpull.v1.sax2`. Together, these numerous packages provide a rich computing platform to write applications for handheld devices.

Taking Advantage of Android Source Code

Android documentation is a bit wanting in places. Android source code can be used to fill the gaps.

The source code for Android and all its projects is managed by the Git source code control system. Git (`http://git-scm.com/`) is an open source source-control system designed to handle large and small projects with speed and convenience. The Linux kernel and Ruby on Rails projects also rely on Git for version control.

The details of the Android source distribution are published at `http://source.android.com`. The code was made available as open source around October 2008. One of the Open Handset Alliance's goals was to make Android a free and fully customizable mobile platform.

Browsing Android Sources Online

Prior to Android 4.0, the Android source distribution was made available at `http://android.git.kernel.org/`. Android now is hosted on its own Git site at `https://android.googlesource.com`. However, this is not browsable online as of this writing. There are some posts online indicating that online browsing may be available soon.

Another frequently visited site to browse Android sources online is at

`www.google.com/codesearch/p?hl=en#uX1GffpyOZk/core/java/android/`

However, there are rumors that the Code Search project may be in the process of getting shut down. Even if it is not, this site does not search the Android 4.0 code yet. For example, we were not able to find the new Contact APIs here.

Another useful site is

`www.grepcode.com/search/?query=google+android&entity=project`

There seems to be a 4.01 branch of Android available here.

We hope both these sites will continue to have the most recent releases so that you can browse the sources online.

Using Git to Download Android Sources

If all else fails, you may have to install Git on your computer and download the sources yourself. If you have a Linux distribution, you can follow the instructions at `http://source.android.com` to get the latest sources.

If you are on a Windows platform, this gets to be a challenge. You will have to install Git first and then use it to get the Android packages you want.

NOTE: Our research notes on using Git to download Android can be found at `http://androidbook.com/item/3919`.

Installing Git

Use the following URL to install the `msysGit` package on Windows:

```
http://code.google.com/p/msysgit/downloads/list
```

Once you have installed it, you will see a directory called `C:\git` (assuming you have installed it under `c:\`).

Testing the Git Installation

The key directory is `C:\git\bin`. To see if it is working, you can use the following command to clone a public repository:

```
git clone git://git.kernel.org/pub/scm/git/git.git
```

This should clone the repository to your local drive.

Downloading Android Repositories

Run this command to discover how many Android Git repositories there are:

```
git clone https://android.googlesource.com/platform/manifest.git
```

This will bring down a directory called `manifest`. Look for a file called `manifest\default.xml`.

This file will have many of the names for the Android repositories. Here are a couple of lines from that file:

```
<project path="frameworks/base"
          name="platform/frameworks/base" />
<project path="frameworks/compile/libbcc"
            name="platform/frameworks/compile/libbcc" />
```

You can see this full file for 4.0 at `http://androidbook.com/item/3920`, where we have posted the contents of the file for quick review. Keep in mind that it is not updated with the latest information.

Now you can get the base `android.jar` source code by using the command

```
git clone https://android.googlesource.com/platform/frameworks/base.git
```

Using the same logic, you can get the contacts provider package by typing

```
git clone https://android.googlesource.com/platform/packages/providers/ContactsProvider
```

The Sample Projects in this Book

In this book, you will find many, many working sample projects. At the end of each chapter is a "References" section that contains a URL to download sample projects for that chapter. All of these sample projects can be accessed from

```
http://androidbook.com/proandroid4/projects
```

If you have any issues downloading or compiling these projects, please contact us by e-mail: `satya.komatineni@gmail.com` or `davemac327@gmail.com`.

We are continuously updating the `androidbook.com` supporting site with what we are learning. It is well worth our efforts if we are able to further contribute to your learning.

Summary

In this chapter, we wanted to pique your curiosity about Android. If you are a Java programmer, you have a great opportunity to profit from this exciting, capable, general-purpose computing platform. We welcome you to journey through the rest of the book for a methodical and in-depth understanding of the Android SDK.