Chapter 2

Key Concepts

Now that you have an idea of what Android is, let's take a look at how it works. Some parts of Android may be familiar, such as the Linux kernel, OpenGL, and the SQL database. Others will be completely foreign, such as Android's idea of the application life cycle.

You'll need a good understanding of these key concepts in order to write well-behaved Android applications, so if you read only one chapter in this book, read this one.

2.1 The Big Picture

Let's start by taking a look at the overall system architecture—the key layers and components that make up the Android open source software stack. In Figure 2.1, on the next page, you can see the "20,000-foot" view of Android. Study it closely—there will be a test tomorrow.

Each layer uses the services provided by the layers below it. Starting from the bottom, the following sections highlight the layers provided by Android.

Linux Kernel

Android is built on top of a solid and proven foundation: the Linux kernel. Created by Linus Torvalds in 1991 while he was a student at the University of Helsinki, Linux can be found today in everything from wristwatches to supercomputers. Linux provides the hardware abstraction layer for Android, allowing Android to be ported to a wide variety of platforms in the future.

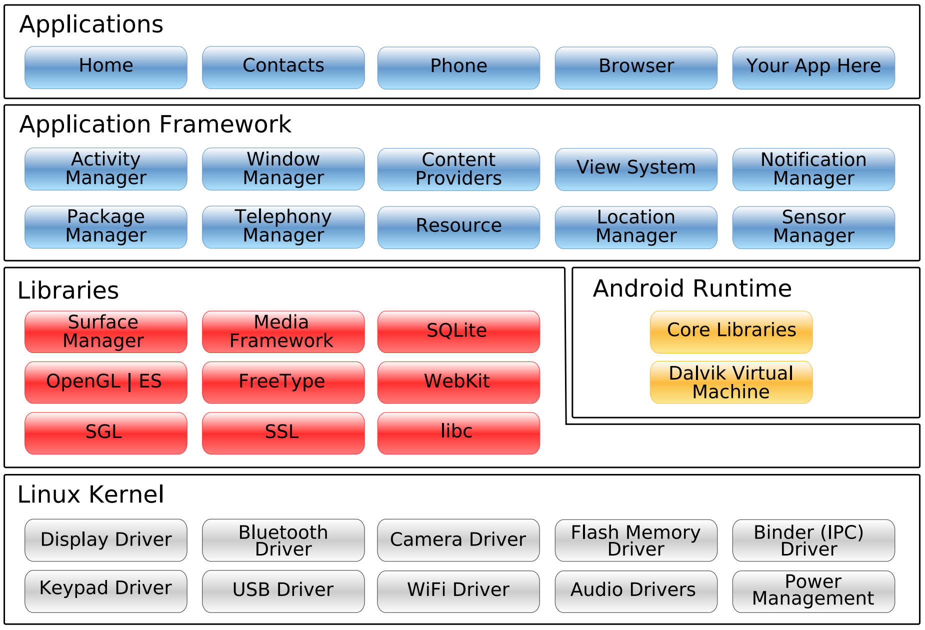

Figure 2.1: Android system architecture

Internally, Android uses Linux for its memory management, process management, networking, and other operating system services. The Android phone user will never see Linux and your programs will not make Linux calls directly. As a developer, though, you'll need to be aware it's there.

Some utilities you need during development interact with Linux. For example, the adb shell command[1] will open a Linux shell in which you can enter other commands to run on the device. From there you can examine the Linux file system, view active processes, and so forth.

Native Libraries

The next layer above the kernel contains the Android native libraries. These shared libraries are all written in C or C++, compiled for the particular hardware architecture used by the phone, and preinstalled by the phone vendor.

1. http://d.android.com/guide/developing/tools/adb.html

Some of the most important native libraries include the following:

- *Surface Manager*: Android uses a compositing window manager similar to Vista or Compiz, but it's much simpler. Instead of drawing directly to the screen buffer, your drawing commands go into offscreen bitmaps that are then combined with other bitmaps to form the display the user sees. This lets the system create all sorts of interesting effects such as see-through windows and fancy transitions.
- *2D and 3D graphics*: Two- and three-dimensional elements can be combined in a single user interface with Android. The library will use 3D hardware if the device has it or a fast software renderer if it doesn't. See Chapter 4, *Exploring 2D Graphics*, on page 71 and Chapter 10, *3D Graphics in OpenGL*, on page 193.
- *Media codecs*: Android can play video and record and play back audio in a variety of formats including AAC, AVC (H.264), H.263, MP3, and MPEG-4. See Chapter 5, *Multimedia*, on page 102 for an example.
- *SQL database*: Android includes the lightweight SQLite database engine,[2] the same database used in Firefox and the Apple iPhone. You can use this for persistent storage in your application. See Chapter 9, *Putting SQL to Work*, on page 173 for an example.
- *Browser engine*: For the fast display of HTML content, Android uses the WebKit library.[3] This is the same engine used in the Google Chrome browser, Apple's Safari browser, the Apple iPhone, and Nokia's S60 platform. See Chapter 7, *The Connected World*, on page 128 for an example.

Android Runtime

Also sitting on top of the kernel is the Android runtime, including the Dalvik virtual machine and the core Java libraries.

The Dalvik VM is Google's implementation of Java, optimized for mobile devices. All the code you write for Android will be written in Java and run within the VM.

2. http://www.sqlite.org
3. http://www.webkit.org

Joe Asks...
What's a Dalvik?

Dalvik is a virtual machine (VM) designed and written by Dan Bornstein at Google. Your code gets compiled into machine-independent instructions called *bytecodes*, which are then executed by the Dalvik VM on the mobile device.

Although the bytecode formats are a little different, Dalvik is essentially a Java virtual machine optimized for low memory requirements. It allows multiple VM instances to run at once and takes advantage of the underlying operating system (Linux) for security and process isolation.

Bornstein named Dalvik after a fishing village in Iceland where some of his ancestors lived.

Dalvik differs from traditional Java in two important ways:

- The Dalvik VM runs .dex files, which are converted at compile time from standard .class and .jar files. .dex files are more compact and efficient than class files, an important consideration for the limited memory and battery-powered devices that Android targets.
- The core Java libraries that come with Android are different from both the Java Standard Edition (Java SE) libraries and the Java Mobile Edition (Java ME) libraries. There is a substantial amount of overlap, however. In Appendix A, on page 217, you'll find a comparison of Android and standard Java libraries.

Application Framework

Sitting above the native libraries and runtime, you'll find the Application Framework layer. This layer provides the high-level building blocks you will use to create your applications. The framework comes preinstalled with Android, but you can also extend it with your own components as needed.

The most important parts of the framework are as follows:

- *Activity manager*: This controls the life cycle of applications (see Section 2.2, *It's Alive!*, on page 33) and maintains a common "backstack" for user navigation.

Embrace and Extend

One of the unique and powerful qualities of Android is that all applications have a level playing field. What I mean is that the system applications have to go through the same public API that you use. You can even tell Android to make your application replace the standard applications if you want.

- *Content providers*: These objects encapsulate data that needs to be shared between applications, such as contacts. See Section 2.3, *Content Providers*, on page 38.
- *Resource manager*: Resources are anything that goes with your program that is not code. See Section 2.4, *Using Resources*, on page 38.
- *Location manager*: An Android phone always knows where it is. See Chapter 8, *Locating and Sensing*, on page 156.
- *Notification manager*: Events such as arriving messages, appointments, proximity alerts, alien invasions, and more can be presented in an unobtrusive fashion to the user.

Applications

The highest layer in the Android architecture diagram is the Applications layer. Think of this as the tip of the Android iceberg. End users will see only these applications, blissfully unaware of all the action going on below the waterline. As an Android developer, however, you know better.

When someone buys an Android phone, it will come prepackaged with a number of standard system applications, including the following:

- Phone dialer
- Email
- Contacts
- Web browser
- Android Market

Using the Android Market, the user will be able to download new programs to run on their phone. That's where you come in. By the time you finish this book, you'll be able to write your own killer applications for Android.

Now let's take a closer look at the life cycle of an Android application. It's a little different from what you're used to seeing.

2.2 It's Alive!

On your standard Linux or Windows desktop, you can have many applications running and visible at once in different windows. One of the windows has keyboard focus, but otherwise all the programs are equal. You can easily switch between them, but it's your responsibility as the user to move the windows around so you can see what you're doing and close programs you don't need anymore.

Android doesn't work that way.

In Android, there is one foreground application, which typically takes over the whole display except for the status line. When the user turns on their phone, the first application they see is the Home application (see Figure 2.2, on the following page). This program typically shows the time, a background image, and a scrollable list of other applications the user can invoke.

When the user runs an application, Android starts it and brings it to the foreground. From that application, the user might invoke another application, or another screen in the same application, and then another and another. All these programs and screens are recorded on the *application stack* by the system's Activity Manager. At any time, the user can press the Back button to return to the previous screen on the stack. From the user's point of view, it works a lot like the history in a web browser. Pressing Back returns them to the previous page.

Process != Application

Internally, each user interface screen is represented by an Activity class (see Section 2.3, *Activities*, on page 37). Each activity has its own life cycle. An application is one or more activities plus a Linux process to contain them. That sounds pretty straightforward, doesn't it? But don't get comfortable yet; I'm about to throw you a curve ball.

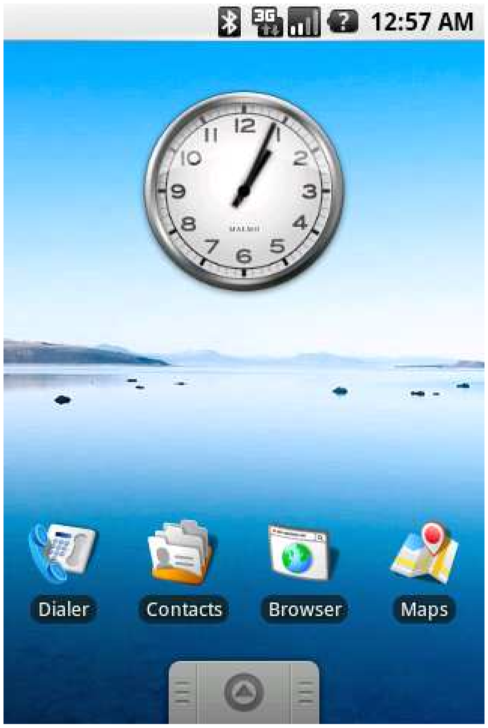

Figure 2.2: The Home application

In Android, an application can be "alive" even if its process has been killed. Put another way, the activity life cycle is not tied to the process life cycle. Processes are just disposable containers for activities. This is probably different from every other system you're familiar with, so let's take a closer look before moving on.

Life Cycles of the Rich and Famous

During its lifetime, each activity of an Android program can be in one of several states, as shown in Figure 2.3, on the following page. You, the developer, do not have control over what state your program is in. That's all managed by the system. However, you do get notified when the state is about to change through the on*XX*() method calls.

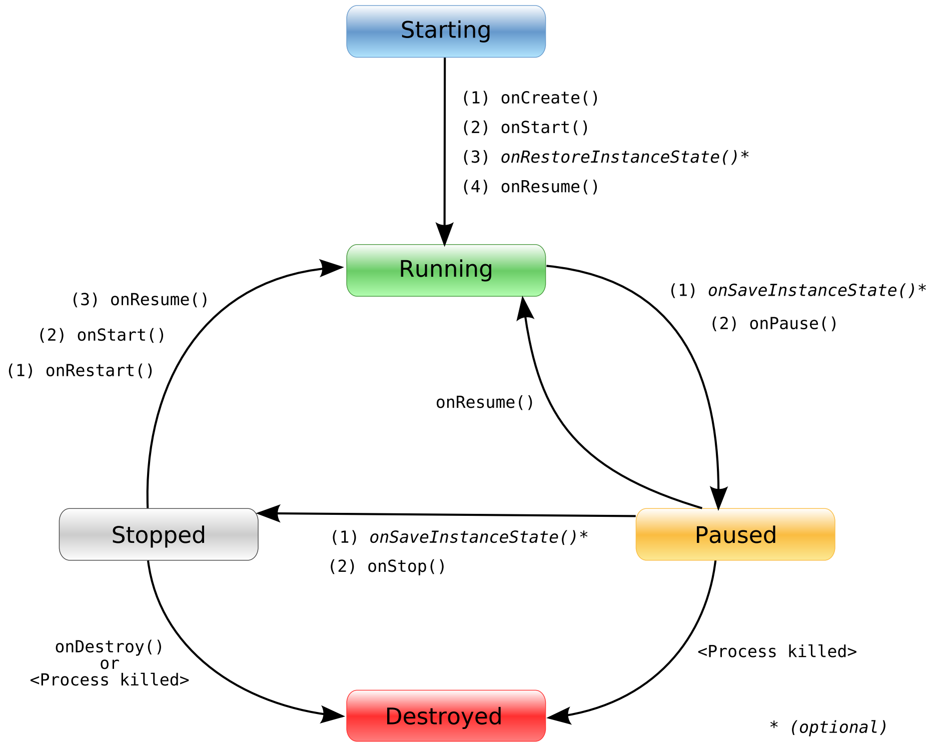

Figure 2.3: Life cycle of an Android activity

You override these methods in your Activity class, and Android will call them at the appropriate time:

- onCreate(Bundle): This is called when the activity first starts up. You can use it to perform one-time initialization such as creating the user interface. onCreate() takes one parameter that is either **null** or some state information previously saved by the onSaveInstanceState() method.
- onStart(): This indicates the activity is about to be displayed to the user.
- onResume(): This is called when your activity can start interacting with the user. This is a good place to start animations and music.

- onPause(): This runs when the activity is about to go into the background, usually because another activity has been launched in front of it. This is where you should save your program's persistent state, such as a database record being edited.
- onStop(): This is called when your activity is no longer visible to the user and it won't be needed for a while. If memory is tight, onStop() may never be called (the system may simply terminate your process).
- onRestart(): If this method is called, it indicates your activity is being redisplayed to the user from a stopped state.
- onDestroy(): This is called right before your activity is destroyed. If memory is tight, onDestroy() may never be called (the system may simply terminate your process).
- onSaveInstanceState(Bundle): Android will call this method to allow the activity to save per-instance state, such as a cursor position within a text field. Usually you won't need to override it because the default implementation saves the state for all your user interface controls automatically.[4]
- onRestoreInstanceState(Bundle): This is called when the activity is being reinitialized from a state previously saved by the onSaveInstanceState() method. The default implementation restores the state of your user interface.

Activities that are not running in the foreground may be stopped or the Linux process that houses them may be killed at any time in order to make room for new activities. This will be a common occurrence, so it's important that your application be designed from the beginning with this in mind. In some cases, the onPause() method may be the last method called in your activity, so that's where you should save any data you want to keep around for next time.

In addition to managing your program's life cycle, the Android framework provides a number of building blocks that you use to create your applications. Let's take a look at those next.

4. Before version 0.9_beta, onSaveInstanceState() was called onFreeze(), and the saved state was called an icicle. You may still see the old names in some documentation and examples.

Flipping the Lid

Here's a quick way to test that your state-saving code is working correctly. In current versions of Android, an orientation change (between portrait and landscape modes) will cause the system to go through the process of saving instance state, pausing, stopping, destroying, and then creating a new instance of the activity with the saved state. On the T-Mobile G1 phone, for example, flipping the lid on the keyboard will trigger this, and on the Android emulator pressing `Ctrl+F11` or the `7` or `9` key on the keypad will do it.

2.3 Building Blocks

A few objects are defined in the Android SDK that every developer needs to be familiar with. The most important ones are activities, intents, services, and content providers. You'll see several examples of them in the rest of the book, so I'd like to briefly introduce them now.

Activities

An *activity* is a user interface screen. Applications can define one or more activities to handle different phases of the program. As discussed in Section 2.2, *It's Alive!*, on page 33, each activity is responsible for saving its own state so that it can be restored later as part of the application life cycle. See Section 3.3, *Creating the Opening Screen*, on page 44 for an example.

Intents

An *intent* is a mechanism for describing a specific action, such as "pick a photo," "phone home," or "open the pod bay doors." In Android, just about everything goes through intents, so you have plenty of opportunities to replace or reuse components. See Section 3.5, *Implementing an About Box*, on page 55 for an example of an intent.

For example, there is an intent for "send an email." If your application needs to send mail, you can invoke that intent. Or if you're writing a new email application, you can register an activity to handle that intent and replace the standard mail program. The next time somebody tries to send an email, they'll get your program instead of the standard one.

Services

A *service* is a task that runs in the background without the user's direct interaction, similar to a Unix daemon. For example, consider a music player. The music may be started by an activity, but you want it to keep playing even when the user has moved on to a different program. So, the code that does the actual playing should be in a service. Later, another activity may bind to that service and tell it to switch tracks or stop playing. Android comes with many services built in, along with convenient APIs to access them.

Content Providers

A *content provider* is a set of data wrapped up in a custom API to read and write it. This is the best way to share global data *between applications*. For example, Google provides a content provider for contacts. All the information there—names, addresses, phone numbers, and so forth—can be shared by any application that wants to use it. See Section 9.5, *Using a ContentProvider*, on page 186 for an example.

2.4 Using Resources

A *resource* is a localized text string, bitmap, or other small piece of noncode information that your program needs. At build time all your resources get compiled into your application.

You will create and store your resources in the res directory inside your project. The Android resource compiler (aapt)[5] processes resources according to which subfolder they are in and the format of the file. For example, PNG and JPG format bitmaps should go in the res/drawable directory, and XML files that describe screen layouts should go in the res/layout directory.

The resource compiler compresses and packs your resources and then generates a class named R that contains identifiers you use to reference those resources in your program. This is a little different from standard Java resources, which are referenced by key strings. Doing it this way allows Android to make sure all your references are valid and saves space by not having to store all those resource keys. Eclipse uses a similar method to store and reference the resources in Eclipse plug-ins.

5. http://d.android.com/guide/developing/tools/aapt.html

We'll see an example of the code to access a resource in Chapter 3, *Designing the User Interface*, on page 42.

2.5 Safe and Secure

As mentioned earlier, every application runs in its own Linux process. The hardware forbids one process from accessing another process's memory. Furthermore, every application is assigned a specific user ID. Any files it creates cannot be read or written by other applications.

In addition, access to certain critical operations are restricted, and you must specifically ask for permission to use them in a file named AndroidManifest.xml. When the application is installed, the Package Manager either grants or doesn't grant the permissions based on certificates and, if necessary, user prompts. Here are some of the most common permissions you will need:

- INTERNET: Access the Internet.
- READ_CONTACTS: Read (but don't write) the user's contacts data.
- WRITE_CONTACTS: Write (but don't read) the user's contacts data.
- RECEIVE_SMS: Monitor incoming SMS (text) messages.
- ACCESS_COARSE_LOCATION: Use a coarse location provider such as cell towers or wifi.
- ACCESS_FINE_LOCATION: Use a more accurate location provider such as GPS.

For example, to monitor incoming SMS messages, you would specify this in the manifest file:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.google.android.app.myapp" >
   <uses-permission android:name="android.permission.RECEIVE_SMS" />
</manifest>
```

Android can even restrict access to entire parts of the system. Using XML tags in AndroidManifest.xml, you can restrict who can start an activity, start or bind to a service, broadcast intents to a receiver, or access the data in a content provider. This kind of control is beyond the scope of this book, but if you want to learn more, read the online help for the Android security model.[6]

6. http://d.android.com/guide/topics/security/security.html

2.6 Fast-Forward >>

The rest of this book will use all the concepts introduced in this chapter. In Chapter 3, *Designing the User Interface*, on page 42, we'll use activities and life-cycle methods to define a sample application. Chapter 4, *Exploring 2D Graphics*, on page 71 will use some of the graphics classes in the Android native libraries. Media codecs will be explored in Chapter 5, *Multimedia*, on page 102, and content providers will be covered in Chapter 9, *Putting SQL to Work*, on page 173.