Extracted from:

Hello, Android

Introducing Google's Mobile Development Platform, 3rd Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Updated for Kindle Fire

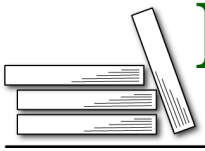
Hello, Android

Introducing Google's Mobile Development Platform

Third Edition

Ed Burnette

Edited by Susannah Davidson Pfalzer



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Gesture icons in Chapter 11 courtesy of GestureWorks (www.gestureworks.com).

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Seth Maislin (indexer)
Kim Wimpsett (copyeditor)
David Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-56-2
Printed on acid-free paper.
Book version: P8.0—January 2012

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 9, *Android system architecture*, on page 6, 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, 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.

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, subject to security restrictions.

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.

Some of the most important native libraries include the following:

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

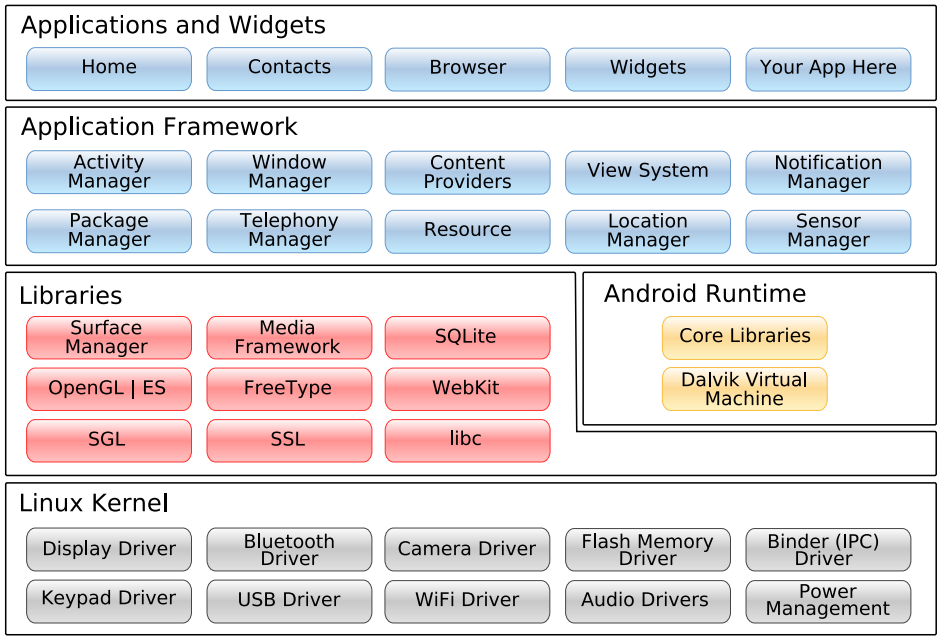


Figure 9—Android system architecture

- *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 off-screen 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 ? and Chapter 10, *3D Graphics in OpenGL*, on page ?.
- *Media codecs*: Android can play video and record and play back audio in various formats including AAC, AVC (H.264), H.263, MP3, and MPEG-4. See Chapter 5, *Multimedia*, on page ? for an example.

- *SQL database*: Android includes the lightweight SQLite database engine,[2] the same database used in Firefox and the Apple iPhone.[3] You can use this for persistent storage in your application. See Chapter 9, *Putting SQL to Work*, on page ? for an example.
- *Browser engine*: For the fast display of HTML content, Android uses the WebKit library.[4] 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 ? for an example.

These libraries are not applications that stand by themselves. They exist only to be called by higher-level programs. Starting in Android 1.5, you can write and deploy your own native libraries using the Native Development Toolkit (NDK). Native development is beyond the scope of this book, but if you're interested, you can read all about it online.[5]

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. 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 1, *Java vs. the Android Language and APIs*, on page ?, you'll find a comparison of Android and standard Java libraries.

2. http://www.sqlite.org
3. See http://www.zdnet.com/blog/burnette/iphone-vs-android-development-day-1/682 for a comparison of iPhone and Android development.
4. http://webkit.org
5. http://d.android.com/sdk/ndk

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.

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 9) and maintains a common "backstack" for user navigation.
- *Content providers*: These objects encapsulate data that needs to be shared between applications, such as contacts. See *Content Providers*, on page ?.
- *Resource manager*: Resources are anything that goes with your program that is not code. See Section 2.4, *Using Resources*, on page ?.
- *Location manager*: An Android phone always knows where it is. See Chapter 8, *Locating and Sensing*, on page ?.
- *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 and Widgets

The highest layer in the Android architecture diagram is the Applications and Widgets layer. Think of this as the tip of the Android iceberg. End users will

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.

see only these programs, blissfully unaware of all the action going on below the waterline. As an Android developer, however, you know better.

Applications are programs that can take over the whole screen and interact with the user. On the other hand, widgets (which are sometimes called *gadgets*), operate only in a small rectangle of the Home screen application.

The majority of this book will cover application development, because that's what most of you will be writing. Widget development is covered in Chapter 12, *There's No Place Like Home*, on page ?.

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.

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 10, *The Home application*, on page 11).

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 *Activities*, on page ?). 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.

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 11, *Life cycle of an Android activity*, on page 12. 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.

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.

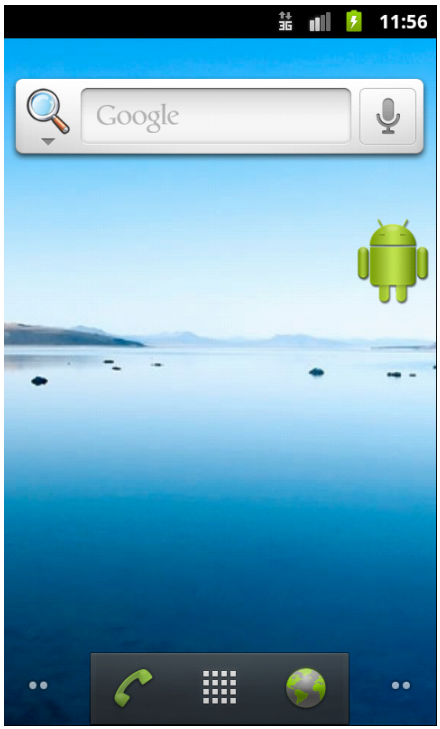


Figure 10—The Home application

- `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).

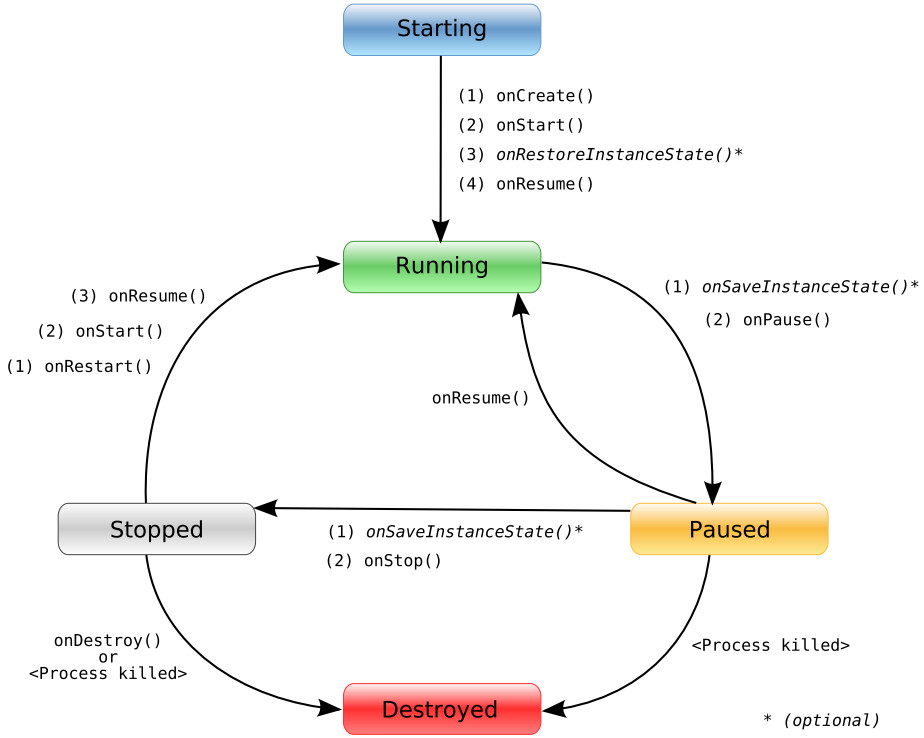


Figure 11—Life cycle of an Android activity

- 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.
- 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.

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.

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.