Extracted from:

Hello, Android

Introducing Google's Mobile Development Platform, Fourth Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Updated for Android 5

Hello, Android

Introducing Google's Mobile Development Platform

Fourth Edition

Ed Burnette

Edited by Susannah Davidson Pfalzer

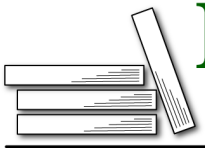
Hello, Android

Introducing Google's Mobile Development Platform, Fourth Edition

Ed Burnette

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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 *https://pragprog.com*.

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-037-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—May 2015

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 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.

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 the figure 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 user will never see Linux, and your programs will not usually 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 Android device, and preinstalled by the vendor.

1. http://d.android.com/tools/help/adb.html

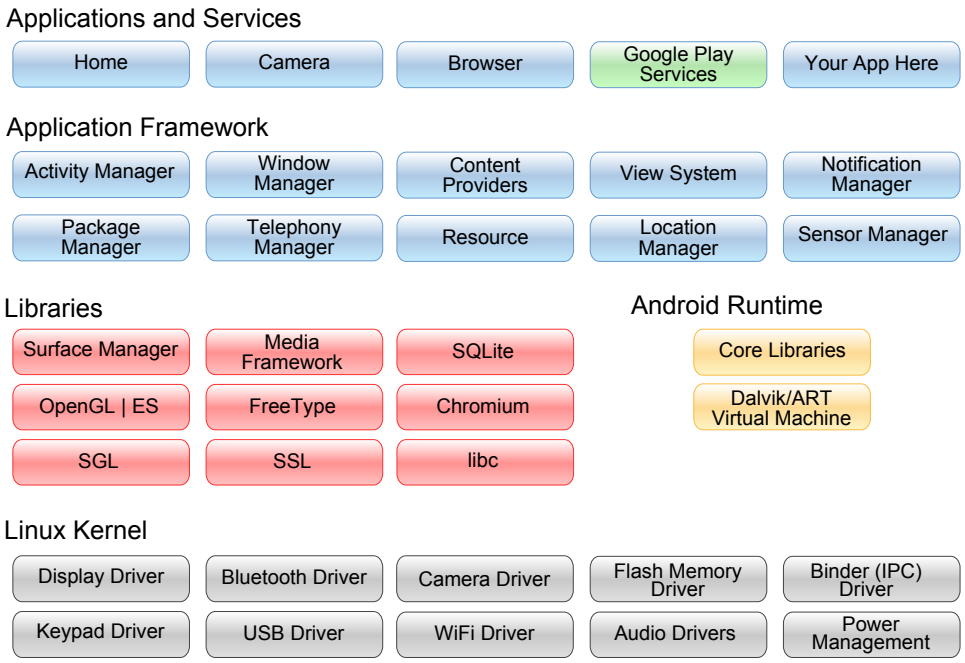


Figure 1—Android system architecture

Some of the most important native libraries include the following:

- *Surface Manager*: Instead of drawing directly to the screen, your drawing commands are saved into lists that are then combined with lists from other windows and are then composited 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. Everything is converted into 3D drawing lists and rendered by hardware for the fastest possible experience.
- *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.
- *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.

2. http://www.sqlite.org

- *Browser engine*: For the fast display of HTML content, Android uses the Chromium library.[3] This is the same engine used in the Google Chrome browser, and it's a close cousin of the one used in Apple's Safari browser and the Apple iPhone.

These libraries aren't applications that stand by themselves. They exist only to be called by higher-level programs. 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.[4]

Android Runtime

Also sitting on top of the kernel is the Android runtime, including the runtime environment and the core Java libraries. Depending on the version of Android, the environment uses either Dalvik or ART.

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.

ART (Android Runtime) is an ahead-of-time compiler that replaced Dalvik in Android 5.0 (Lollipop). When an application is installed onto your Android device, ART compiles it into machine code. Compared to Dalvik, this makes programs run faster at the expense of a longer install time.

Dalvik and ART are Google's semi-compatible implementation of Java, optimized for mobile devices. All the code you write for Android will be written in Java and run by Dalvik or ART.

Note that 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. A substantial amount of overlap exists, however. In Appendix 1, *Java vs. the Android Language and APIs*, on page ?, 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'll use to create your applications. The framework comes preinstalled with Android, but you can also extend it with your own components as needed.

3. http://www.chromium.org
4. http://d.android.com/tools/sdk/ndk

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.

The most important parts of the framework are as follows:

- *Activity manager:* This controls the life cycle of applications (see *It's Alive!*, on page 12) 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 11.
- *Resource manager:* Resources are anything that goes with your program that is not code. See *Using Resources*, on page 11.
- *Location manager:* An Android device always knows where it is. See Chapter 12, *Using Google Play Services*, 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 Services

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

Applications are programs that can take over the whole screen and interact with the user. On the other hand, services operate invisibly to extend the application framework. The majority of this book will cover application development, because that's what most of you will be writing.

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

- Phone dialer
- Email
- Camera
- Web browser

- Google Play Store

Using the Play Store, users 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 awesome applications for Android.

The Android framework provides a number of building blocks that you use to create your applications. Let's take a look at those next.

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, fragments, views, intents, services, and content providers. You'll see examples of most of these in the rest of the book, so I'd like to briefly introduce them now.

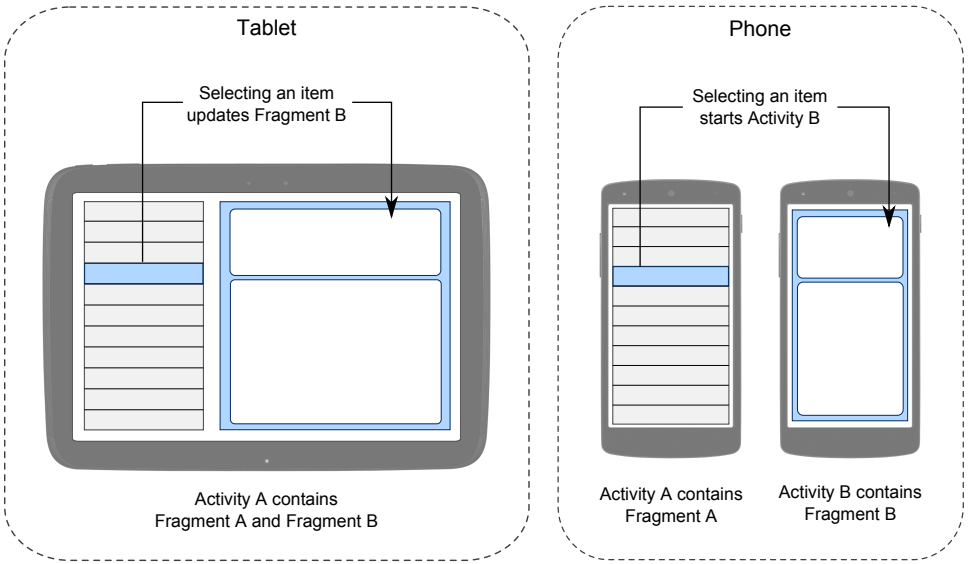
Activities

An *activity* is a user interface screen. Applications can define one or more activities to handle different phases of the program. As discussed in *It's Alive!*, on page 12, each activity is responsible for saving its own state so that it can be restored later as part of the application life cycle. See *Creating the Main Screen*, on page ? for an example. Activities extend the `Context` class, so you can use them to get global information about your application.

Fragments

A *fragment* is a component of an activity. Usually they're displayed on the screen, but they don't have to be. Fragments were introduced in Android 3.0 (Honeycomb), but if you need to target older versions of Android you can use a compatibility library.

If you consider an email program, there's one part of the app that displays the list of all the mail you have, and another part that displays the text of one email. These could be (and probably are) implemented as two different fragments. Using fragments allows you to more easily adapt to different-sized screens (see the following diagram).



Views

A *view* is the smallest level of the user interface. Views are contained directly by activities or by fragments inside activities. They can be created by Java code, or preferably, by XML layouts. Each view has a series of attributes, or properties, that control what it does, how it acts, and what it displays.

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 *Browsing by Intent*, on page ? for an example of an intent.

For example, there's 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, that person will get the option to use 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. Google also provides optional services for extra functionality (see Chapter 12, *Using Google Play Services*, on page ?).

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 *Using a ContentProvider*, on page ? for an example.

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. This is useful for internationalization and for supporting multiple device types (see *Specifying Alternate Resources*, on page ?).

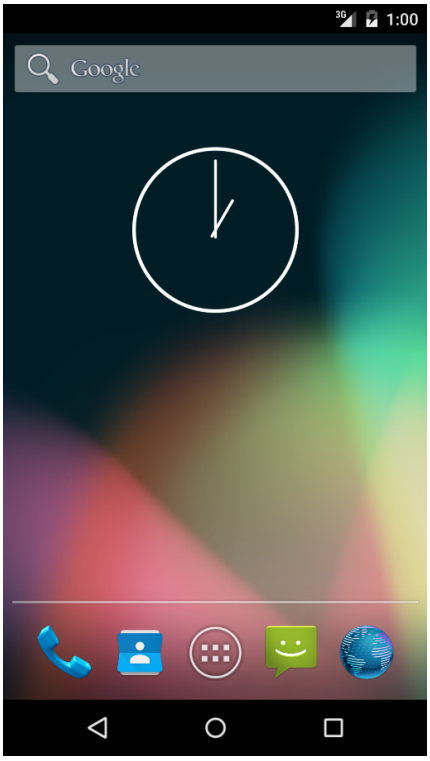
You'll create and store your resources in the `res` directory inside your project. The Android resource compiler (`aapt`)[5] processes resources according to which subfolder they're in and the format of the file. For example, PNG and JPG format bitmaps should go in a directory starting with `res/drawable`, and XML files that describe screen layouts should go in a directory starting with `res/layout`. You can add suffixes for particular languages, screen orientations, pixel densities, and more (see *All Screens Great and Small*, on page ?).

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. We'll see an example of the code to access a resource in Chapter 3, *Opening Moves*, on page ?.

5. http://d.android.com/tools/building

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.

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's one foreground application, which typically takes over the whole display except for the status line. When users turn on their phone or tablet, the first application they see is the Home application (see the figure).

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, users can press the Back button to return to the previous screen on the stack. From the users' point of view, it works a lot like the history in a web browser. Pressing Back returns them to the previous page.

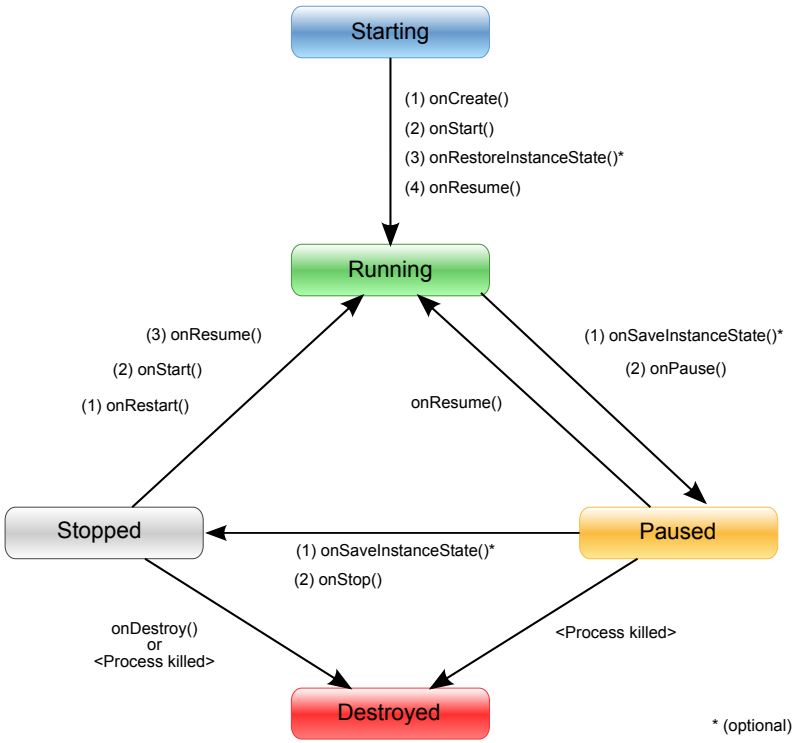
Process != Application

Internally, each user interface screen is represented by an Activity class (see *Activities*, on page 9). 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 isn't tied to the process life cycle. Processes are just disposable containers for activities.

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 the following figure. You, the developer, don't 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:

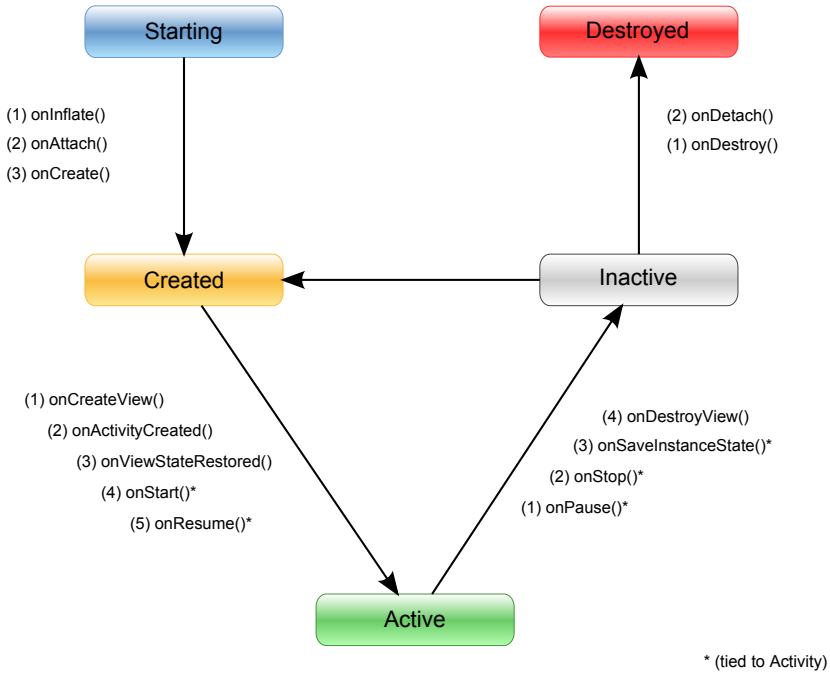
- onCreate(Bundle): This is called when the activity first starts up. You can use it to perform one-time initialization such as creating the user interface. onCreate() takes one parameter that is either null or some state information previously saved by the onSaveInstanceState() method.
- onStart(): This indicates the activity is about to be displayed to the user.
- 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.
- 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 aren't 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.

Starting with Android 3.0 (Honeycomb), Google introduced another twist in the story of application life cycles: *fragments*.

Better Living Through Fragments

Fragments represent a component of your application. They're contained within activities (see *Fragments*, on page 9). and have a life cycle very similar to activities. In fact, many of the life-cycle methods for fragments are called by the methods of the Activity (for example, Fragment.onResume() is called indirectly by Activity.onResume()). See the following diagram for details:



Fragments can outlive the activities that contain them. For example, if you rotate the screen while an app is running, the activity will usually be destroyed and re-created so that it can adjust to the new screen dimensions. However, the fragments will usually keep on going. This lets you keep heavyweight objects such as a network connection alive during the transition.