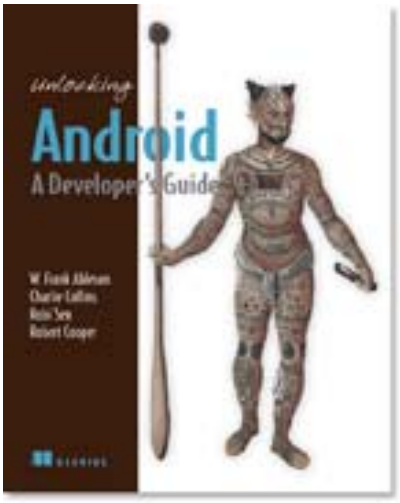
MANNING PUBLICATIONS

Targeting Android

Green paper from



Unlocking Android

EARLY ACCESS EDITION

A Developer's Guide
W. Frank Ableson, Charlie Collins, Robi Sen, and Robert Cooper

MEAP Release: March 2008
Softbound print: March 2009 (est.) | 435 pages
ISBN: 1933988673

This green paper is taken from the forthcoming book Unlocking Android from Manning Publications. It explains the essential concepts of the Android platform, describes the features of the Android stack, and provides practical examples to illustrate how it works. For the table of contents, the Author Forum, and other resources, go to http://manning.com/ableson/.

You've heard about Android. You've read about Android. Now it is time to unlock Android.

Android is the software platform from Google and the Open Handset Alliance that has the potential to revolutionize the global cell phone market. This paper introduces Android – what it is, and importantly, what it is not. After reading this paper you will have an understanding of how Android is constructed, how it compares with other offerings in the market, Android's foundational technologies, and a preview of Android application architecture. The paper concludes with a simple Android application to get things started quickly.

This introductory paper answers some of the basic questions about what Android is and where it fits. While there are code examples in this paper, they are not very in-depth – just enough to get a taste for Android application development and to convey the key concepts introduced.

Introducing Android

Android is the first open source mobile application platform that has the potential to make significant inroads in many markets. When examining Android there are a number of dimensions to consider, both technical and market related. This first section introduces the platform and provides some context to help better understand Android and where it fits in the global cell phone scene.

Android is the product of primarily Google, but more appropriately, the Open Handset Alliance. The Open Handset Alliance is an organization of approximately 30 organizations committed to bringing a "better" and "open" mobile phone to market. A quote taken from their website says it best, "Android was built from the ground up with the explicit goal to be the first open, complete, and free platform created specifically for mobile devices." As discussed in this section, open is good, complete is good, however "free" may be turn out to be an ambitious goal. There are many examples of "free" in the computing market that are free from licensing, but of course there is a

"cost of ownership" when taking support and hardware costs into account. And of course, "free" cell phones come tethered to two year contracts, plus tax. No matter the way some of the details play out, the introduction of Android is a market moving event and Android is likely to prove an important player in the mobile software landscape.

With this background of who is behind Android and the basic ambition of the Open Handset Alliance, it is time to understand a little more about the platform itself and how it fits in the mobile marketplace.

1.1.1 The Android Platform

Android is a software environment built for mobile devices. It is *not* a hardware platform. Android includes a Linux kernel based operating system, a rich User Interface, end-user applications, code libraries, application frameworks, multimedia support and much more. And, yes, even telephone functionality is included! While components of the underlying operating system are written in C or C++, user applications are built for Android using the Java programming language. Even the built-in applications are written in Java.

One of the most powerful features of the Android platform is that there is "no difference" between the built-in applications shipped on the device and user applications created with the SDK. This means that powerful applications can be written to tap into the resources available on the device. Figure 1 demonstrates the relationship between Android and the hardware it runs on. Perhaps the most notable feature of Android is that it is an open source platform, therefore missing elements can and will be provided by the global developer community. For example, Android's Linux kernel based operating system does not come with a sophisticated shell environment, but because the platform is open, shells can be written and installed on a device. Likewise, multimedia codecs can be supplied by third party developers and do not need to rely on Google or anyone else to provide new functionality. That is the power of an open source platform brought to the mobile market.

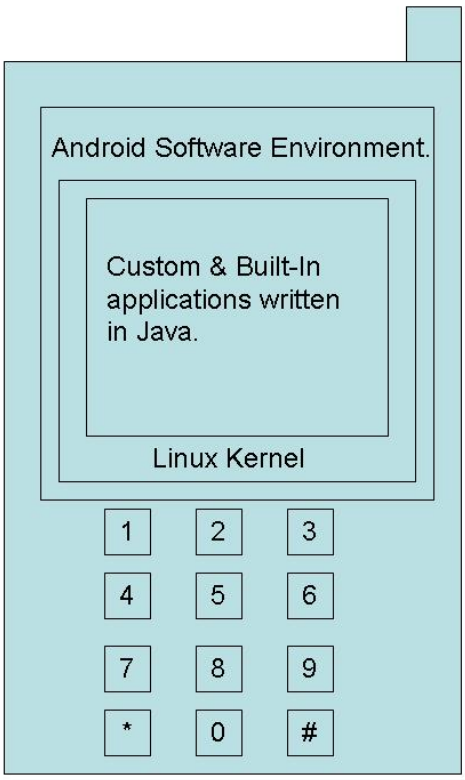


Figure 1. Android is software only. Leveraging its Linux kernel to interface with the hardware, Android can be expected to run on many different devices from multiple cell phone manufacturers. Applications are written in the Java programming language.

Platform vs. Device

Android capable devices are just hitting the market, though there are often references to the "device" in this paper. The term *device* here represents the future family of mobile phones capable of running Android. Throughout the paper, wherever code must be tested or exercised on a "device", a software based emulator is employed.

The term *platform* refers to Android itself—the software--including all of the binaries, code libraries, and tool chains. This paper is focused on the Android *platform*. The Android emulators available in the SDK are simply one of many components of the Android platform.

The mobile market is a rapidly changing landscape with many players who often have diverging goals. For example, consider the often at-odds relationship between mobile operators, mobile device manufacturers and software vendors. Mobile operators want to lock down their networks, controlling and of course metering traffic. Device manufacturers want to differentiate themselves with features, reliability and price-points. Software vendors want unfettered access to the metal to deliver cutting edge applications. Then layer on to that a demanding user base, both consumer and corporate, that has become addicted to the "free phone", and operators who reward churn and not customer loyalty. The mobile market becomes not only a confusing array of choices, but also a dangerous fiscal exercise for the participants, such as the cell phone retailer who sees the underbelly of the industry and just wants to stay alive in an endless sea of change. What users come to expect on a mobile phone

has evolved rapidly. For example, examine Figure 2 which provides a glimpse of the way we view "mobile technology" and how it has matured in a few short years.

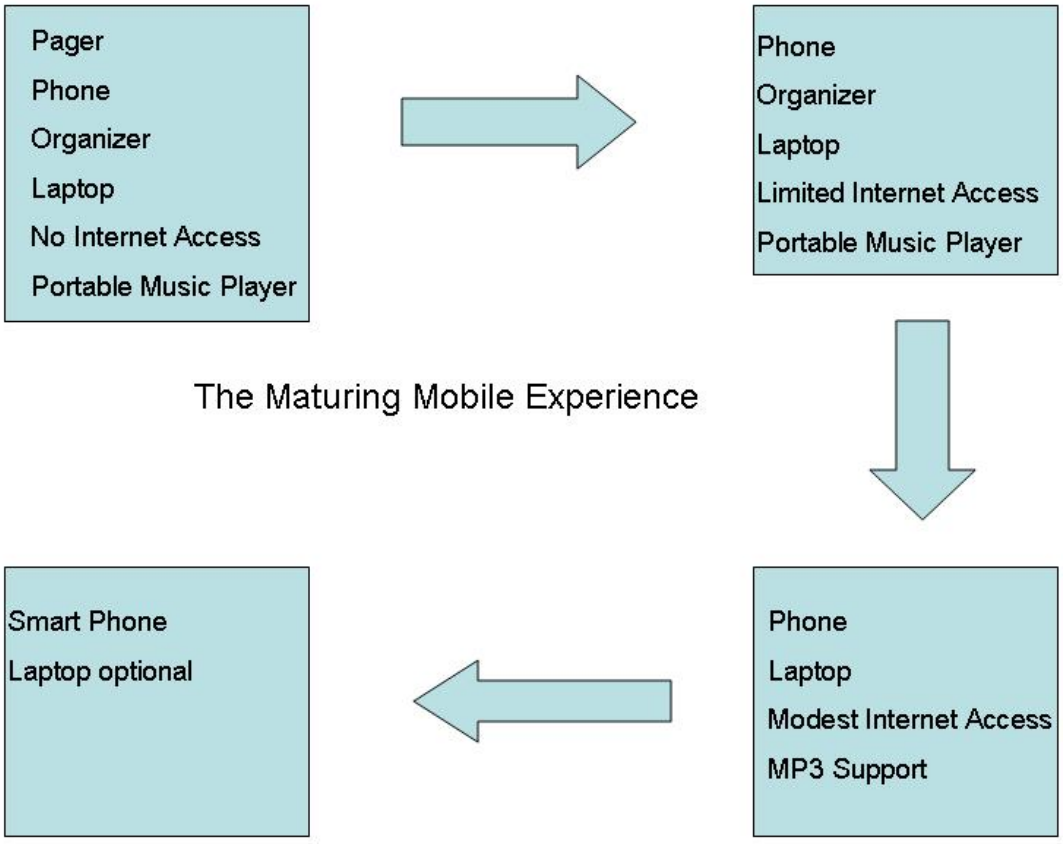


Figure 2: The mobile worker can be pleased with the reduction in devices that need to be toted. Mobile device functionality has converged at a very rapid pace. The laptop computer is becoming an optional piece of travel equipment.

With all of that as a back-drop, creating a successful mobile platform is clearly a non-trivial task involving numerous players, so Android is an ambitious undertaking, even for Google, a company of seemingly boundless resources and moxy. If anyone has the clout to move the mobile market, it is Google and its entrant to the mobile marketplace, Android.

The next section begins and ends the "Why and Where of Android" to provide some context and set the backdrop for Android's introduction to the marketplace. After that, it's on to exploring and exploiting the platform itself!

In the market for an Android?

Android promises to have something for everyone. Android looks to support a variety of hardware devices, not just high-end devices typically associated with expensive "Smart Phones". Of course, Android will run better on a more powerful device, particularly considering it is sporting a comprehensive set of computing features. The real question is how well can Android scale up and down to a variety of markets and gain market- and mind-share.

This section provides some conjecture on Android from the perspective of a few existing “players” in the market place. When talking about the cellular market, the place to start is “at the top”, with the carriers or as they are sometimes referred, Mobile Operators.

Mobile Operators

Mobile operators are in the business of selling subscriptions to their services, first and foremost. Share holders want a return on their investment and it is hard to imagine an industry where there is a larger investment than in a network that spans such broad geographic territory. To the mobile operator, cell phones are all at once a conduit for services, a drug to entice subscribers, and an annoyance to support and lock down.

The optimistic view of the mobile operator’s response to Android is that Android is embraced with open arms as a platform to drive new data services across the excess capacity operators have built into their networks. Data services represent high premium services and high-margin revenues for the operator. If Android can help drive those revenues for the mobile operator, then all the better.

The pessimistic view of the mobile operator’s response to Android is that the operator feels threatened by Google and the potential of “free wireless”, driven by advertising revenues and an upheaval of their market. The other threat from mobile operators is that they have the final say on what services are enabled across their network. Historically, one of the complaints of handset manufacturers is that their devices are handicapped and not exercising all of the features designed into them due to the mobile operator’s lack of capability or lack of willingness to support those features. An encouraging sign is that there are mobile operators involved in the Open Handset Alliance.

Enough conjecture, let’s move on to a quick comparison of Android and existing cell phones on the market today.

Android vs. The Feature-Phones

The overwhelming majority of cell phones on the market are the consumer ”flip phones” and ”feature phones”. These are the phones consumers get when they walk into the retailer and ask what can be had for “free”, or the “I just want a phone” customer. This customer’s primary interest is in a phone for voice communications and perhaps an address book and maybe even a camera. Many of these phones have more capabilities such as mobile web browsing, but due to a relatively limited user experience, these features are not employed heavily. The one exception to that is text messaging which is a dominant application, no matter the classification of device. Another increasingly in-demand category is location based services.

Android’s challenge is to scale down to this market. Some of the bells and whistles in Android can be left out to “fit” into lower end hardware. One of the big functionality gaps on these lower-end phones is the web experience. Part of this is due to screen size, but equally challenging is the browser technology itself which often struggles to match the rich web experience of the desktop computer. Android features the market-leading WebKit browser engine, which brings desktop compatible browsing to the mobile arena. Figure 3 demonstrates the WebKit in action on Android. If this can be effectively scaled down to the feature phones, it would go a long way towards penetrating this end of the market.

Figure 3 Android's built-in browser technology is based on Webkit.org's browser engine.

NOTE

The WebKit browser engine is an open source project which powers the browser found in Macs (Safari) and is the engine behind Mobile Safari, the browser found on the iPhone. It is not a stretch to say that the browser experience is what makes the iPhone popular, so its inclusion in Android is a strong plus for Android's architecture.

Software at this end of the market generally falls into one of two camps:

- **Qualcomm's BREW environment**. BREW stands for Binary Runtime Environment for Wireless. For a high volume example of BREW technology, consider Verizon's "Get it Now" capable devices which run on this platform. The challenge to the software developer desiring to gain access to this market is

that the bar is very high to get an application on this platform as everything is managed by the mobile operator, with expensive testing and revenue sharing fee structures. The upside to this platform is that the mobile operator collects the money and disburses it to the developer after the sale, and often these sales are recurring monthly. Just about everything else is a challenge to the software developer, however. Android's open application environment is much more accessible than BREW.

- **J2ME**, or Java Micro Edition, is a very popular platform for this class of device. The barrier to entry is much lower for software developers. J2ME developers will find a "same but different" environment in Android. Android is not strictly a J2ME compatible platform, however the Java programming environment is a plus for J2ME developers. Also, as Android matures, it is very likely that J2ME support will be added in some fashion.

Gaming, a better browser, and anything to do with texting or social applications present fertile territory for Android at this end of the market.

While the masses carry feature phones as described in this section, Android's capabilities will put Android-capable devices into the next market segment with the higher end devices as discussed in the next section.

Android vs. The Smart Phones

The market leaders in the smart phone race are Windows Mobile, Windows SmartPhone, Blackberry, with Symbian (huge in non-US markets), iPhone and Palm rounding out the market. While we could focus on market share and pros vs. cons of each of the smart phone platforms, one of the major concerns of this market is a platform's ability to synchronize data and access Enterprise Information Systems for corporate users. Device management tools are also an important factor in the Enterprise market. The browser experience is respectable, largely due to larger displays and more intuitive input methods such as a touch screen or a jog-dial.

Android's opportunity in this market is that it promises to deliver more performance on the same hardware and at a lower software acquisition cost. The challenge Android faces is the same challenge faced by Palm – scaling the Enterprise walls. Blackberry is dominant due to its intuitive email capabilities and the Microsoft platforms are compelling because of tight integration to the desktop experience and overall familiarity for Windows users.

The next section poses an interesting question: can Android, the open source mobile platform, succeed as an open source project?

Android vs. Itself – The Challenge of Open Source in the Consumer Market

Perhaps the biggest challenge of all is Android's commitment to open source. Coming from the lineage of Google, Android will likely always be an "open source" project, but in order to succeed in the mobile market, it must sell millions of units. Android is not the first 'open source phone', but it is the first from a player with the market-moving weight of Google leading the charge.

Open source is a double-edged sword. On one hand, the power of many talented people and companies working around the globe and around the clock to push the ball up the hill and deliver desirable features is a force to be reckoned with, particularly in comparison with a traditional, commercial approach to software development – this is really a trite topic unto itself by now as the benefits of open source development are well documented. However, the other side of the open source equation is that without a centralized code base that has some stability to it, Android could splinter and not gain the critical mass it needs to penetrate the mobile market. For example, look at the Linux platform as an alternative to the "incumbent" Windows operating system. As a kernel, Linux has enjoyed tremendous success as it is found in many operating systems, appliances such as routers and switches, and a host of embedded and mobile platforms, such as Android. There are numerous "Linux distributions" available for the desktop – and ironically, the plethora of choice has held it back as a desktop alternative to Windows. Linux is arguably the most successful Open Source project, however as a desktop alternative to Windows, it has become splintered and that has hampered its market penetration from a product perspective. As an example of the diluted Linux market, consider the abridged list of Linux distributions:

- Ubunto
- openSUSE
- Fedora (Red Hat)

- Debian
- Mandriva (formerly Mandrake)
- PCLinuxOS
- MEPIS
- Slackware
- Gentoo
- Knoppix

The list contains just a sampling of the more popular Linux desktop software distributions. How many people do you know that use Linux as their primary desktop OS, and if so, do they all use the same version? Open source alone is not enough, Android must stay focused as a product and not get diluted in order to penetrate the market in a meaningful way. This is the classic challenge of the intersection between commercialization and open source. This is Android's challenge, among others, as Android needs to demonstrate staying power and the ability scale from the mobile operator to the software vendor, and even at the grass-roots level to the retailer. Becoming diluted into many "distributions" is not a recipe for success for such a consumer product as a cell phone.

The licensing model of open source projects can be sticky. Some software licenses are more restrictive than others. Some of those restrictions pose a challenge to the "open source" label. At the same time, Android licensees need to protect their investment, so licensing is an important topic for the commercialization of Android.

Licensing Android

Android is released under two different open source licenses. The Linux kernel is released under the GPL, as is required for anyone licensing the open source operating system kernel. The Android platform, excluding the kernel, is licensed under the Apache Software License (ASL). While both licensing models are open-source oriented, the major difference is that the Apache license is considered to be more friendly towards commercial use. Some open source purists will find fault with anything but complete open-ness, source code sharing, and non-commercialization, however the ASL attempts to balance the open source goals with commercial market forces. If there is not a financial incentive to deliver Android capable devices to the market, devices will never appear in the meaningful volumes required to launch Android.

The high-level, touchy-feely portion of this paper has now concluded! The remainder of this paper is focused on Android application development. Any technical discussion of a software environment must include a review of the layers that comprise the environment, sometimes referred to as a "stack" because of the layer upon layer construction. The next section begins a high-level break down of the components of the Android stack.

Stacking up Android

The Android stack includes an impressive array of features for mobile applications. In fact, looking at the architecture alone, without the context of Android being a platform designed for mobile environments, it would be easy to confuse Android with a general computing environment. All of the major components of a computing platform are here and read like a Who's Who of the open source community. Here is a quick run-down of some of the prominent components of the Android Stack:

- A Linux Kernel provides a foundational hardware abstraction layer as well as core services such as process, memory and file system management. The kernel is where hardware specific drivers are implemented – capabilities such as WiFi and Bluetooth are found here. The Android stack is designed to be flexible, with many optional components. The optional components largely rely on the availability of specific hardware on a given device. These include features like touch-screens, cameras, GPS receivers and accelerometers.
- Prominent code libraries include:
    - Browser technology from WebKit - the same open source engine powering Mac's Safari and the iPhone's Mobile Safari browser.

    - Database support via SQLite an easy to use SQL database
    - Advanced Graphics support, including 2D, 3D, animation from SGL and OpenGL|ES
    - Audio and Video media support from Packet Video's OpenCore
    - SSL capabilities from the apache project
- An array of "Managers" providing services for:
    - Activities and Views
    - Telephony
    - Windows
    - Resources
    - Location Based Services
- The Android Runtime provides
    - Core Java Packages for a nearly full featured Java programming environment. It should be noted that this is *not* a J2ME environment.
    - The Dalvik Virtual Machine employs services of the Linux based kernel to provide an environment to host Android applications.

Both core applications and third-party applications (such as the ones built in this paper) run in the Dalvik Virtual Machine, atop the components just introduced. The relationship between these layers can be seen in Figure 4.

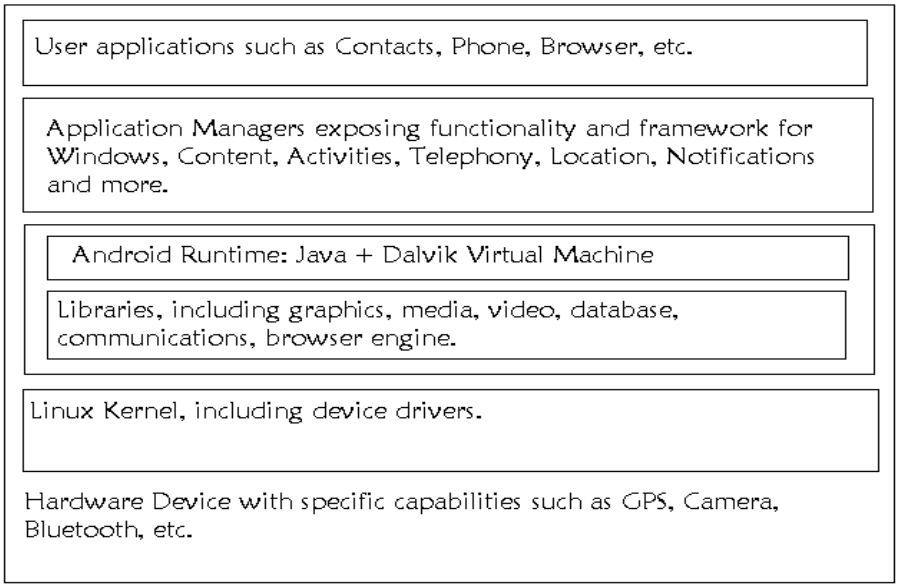


Figure 4: The Android Stack offers an impressive array of technologies and capabilities.

TIP

Android development requires Java programming skills, without question. To get the most out of this paper, please be sure to brush up on your Java programming knowledge. There are many Java references on the

Internet and there is no shortage of excellent Java books on the market. An excellent source of Java titles can be found at http://manning.com/catalog/java.

Now that the obligatory stack diagram is shown and the layers introduced, let's look further at the run time technology that underpins Android.

Probing Android's Foundation

Android is built upon a Linux kernel and an advanced, optimized Virtual Machine for its Java applications. Both of these technologies are crucial to Android. The Linux kernel component of the Android stack promises agility and portability to take advantage of numerous hardware options for future Android equipped phones. Android's Java environment is key because it makes the Android environment very accessible to programmers due to both the number of Java software developers and the rich environment that Java programming has to offer. Other mobile platforms that have relied on less accessible programming environments have seen stunted adoption due to a lack of applications as developers have shied away from the platform.

Building on the Linux Kernel

Why Linux for a phone? Using a full featured platform such as the Linux kernel provides tremendous power and capabilities for Android. Using an open source foundation unleashes the capabilities of numerous talented individuals and companies to move the platform forward. This is particularly important in the world of mobile devices where the products change so rapidly. The rate of change in the mobile market makes the general computer market look slow and plodding. And of course, the Linux kernel is also a proven core platform. Reliability is more important than performance when it comes to a mobile phone, because voice communication is the primary use of a phone. Consumers and business customers want the cool data features and will purchase a device based on those features, but they demand voice reliability. Linux can help meet this requirement.

Speaking to the rapid rate of phone turnover and accessories hitting the market, another advantage of using Linux as the foundation of the Android platform stack is that it provides a hardware abstraction layer, letting the upper levels remain unchanged despite changes in the underlying hardware. Of course, good coding practices demand that user applications fail gracefully in the event of a resource being unavailable, such as a camera not being present in a particular handset model. As new accessories appear on the market, drivers can be written at the Linux level to provide support just as on other Linux platforms.

User applications, as well as core Android applications are written in the Java programming language and are compiled into "byte codes". Byte codes are interpreted at runtime by an interpreter known as a virtual machine.

Running in the Dalvik Virtual Machine

The Dalvik Virtual Machine is an example of the needs of efficiency, the desire for a rich programming environment, and perhaps even some Intellectual Property constraints colliding with creative innovation occurring as a result. Android's Java environment provides a rich application platform and is very accessible due to the popularity of the Java language. Also, application performance, particularly in a low memory setting such as is found in a mobile phone, is paramount for the mobile market. However this is not the only issue at hand.

Android is not a J2ME platform. Without commenting on whether this is ultimately good or bad for Android, there are other forces at play here. There is a matter of Java Virtual Machine licensing from Sun Microsystems. From a very high level, Android's code environment is Java, which is compiled to Java byte codes and then subsequently translated to a similar but different representation called dex files. These files are logically equivalent to Java byte codes, but permit Android to run its applications in its own Virtual Machine that is both (arguably) free from Sun's licensing clutches and is an open platform upon which Google, and potentially the open source community, can improve as necessary. At the time of this writing the Dalvik Virtual Machine has not been released as an open source project.

NOTE

It is too early to tell whether there will be a big battle between the Open Handset Alliance and Sun over the use of Java in Android. From the mobile application developer's perspective, Android is a Java environment, however the runtime is not strictly a Java virtual machine. This accounts for the incompatibilities between Android and "proper" Java environments and libraries.

The important thing to know about the Dalvik Virtual Machine is that Android applications run inside of it and that it relies on the Linux kernel for services such as process, memory and file system management.

After this discussion of the foundational technologies in Android, it is now time to focus on Android application development.

Booting Android Development

This section jumps right into the fray of Android development to focus on an important component of the Android platform and then expands out to take a broader view of how Android applications are constructed. An important and recurring theme of Android development is the *Intent*. An Intent in Android describes "what you want to do". This may look like "I want to lookup a contact record", or "Please launch this website", or "Show the Order Confirmation Screen". Intents are important because they not only facilitate navigation in an innovative way as discussed next, but they also represent arguably the most important aspect of Android coding. *Understand the Intent, Understand Android.*

NOTE:

The code examples in this paper are primarily for illustrative purposes. Classes are referenced and introduced without necessarily naming specific java packages.

The next section provides foundational information about why Intents are important, and then goes on to describe how Intents work. Beyond the introduction of the Intent, the remainder of this paper describes the major elements of Android application development leading up to and including the first complete application.

Android's good Intent-ions

The power of Android's application framework lies in the way in which it brings a web mindset to mobile applications. This doesn't mean the platform simply has a powerful browser and is limited to clever Javascript and server-side resources, but rather it goes to the core of both how the Android platform itself works and how the user of the platform interacts with the mobile device. The power of the Internet, should one be so bold to reduce it to a single statement, is that everything is just a click away. Those clicks are known to the user as Uniform Resource Locator, URLs, or alternatively, Uniform Resource Identifiers, or URIs. The use of effective URIs permits easy and quick access to the information users need and want every day. "Send me the link", says it all.

Beyond being an effective way to get access to data, why is this URI topic important, and what does it have to do with Intents? The answer to that question is a non-technical, but crucial response: *the way in which a mobile user navigates on the platform is crucial to its commercial success.* Platforms which replicate the desktop experience on a mobile device are acceptable to only a small percentage of hard-core power users. Deep menus, multiple taps and clicks are generally not well received in the mobile market. The mobile application, more than in any other market, demands intuitive ease of use. While a consumer may purchase a device based on all of the cool features enumerated in the marketing materials, instruction manuals are almost never touched. The ease of use of the User Interface (UI) of a computing environment is highly correlated with its market penetration. User Interfaces are also a reflection of the platform's data access model, so if the navigation and data models are clean and intuitive, the UI will follow suit. This section introduces the concept of *Intents* and *IntentFilters*, Android's innovative navigation and triggering mechanism. Intents and IntentFilters bring the "click on it" paradigm to the core of mobile application use (and development!) for the Android platform.

- An Intent is a declaration of need.
- An IntentFilter is a declaration of capability and interest in offering assistance to those in need.
- An Intent is made up of a number of pieces of information describing the desired action or service. This section examines the requested action, and generically, the data that accompanies the requested action.
- An IntentFilter may be generic or specific with respect to which Intents it offers to service.

The action attribute of an Intent is typically a verb, for example: VIEW, PICK, or EDIT. There are a number of built-in Intent actions defined as members of the Intent class. Application developers can create new actions as well. To view a piece of information, an application would employ the following Intent action:

```
android.content.Intent.ACTION_VIEW
```

The data component of an Intent is expressed in the form of a URI and can be virtually any piece of information such as a Contact Record, a website location, or perhaps a reference to a media clip. Table 1 lists some example URI possibilities:

Table 1: Intents employ URIs, with some of the commonly employed URIs in Android listed here.

Type of information	URI data
Contact Lookup	content://contacts/people
Map Lookup/Search	Geo:0,0?q=*23+Route+206+Stanhope+NJ*
Browser Launch to a specific website	http://google.com

The IntentFilter defines the relationship between the Intent and the application. IntentFilters can be specific to the data portion of the Intent, the action portion, or both. IntentFilters also contain a field known as a category. A category helps classify the action. For example, the category named CATEGORY_LAUNCHER instructs Android that the Activity containing this IntentFilter should be visible in the main application launcher/shell.

When an Intent is dispatched, the system evaluates the available Activities, Services, and registered BroadcastReceivers (more on these in the next section) and dispatches the Intent to the most appropriate recipient. Figure 5 depicts this relationship between Intents, IntentFilters and BroadcastReceivers.

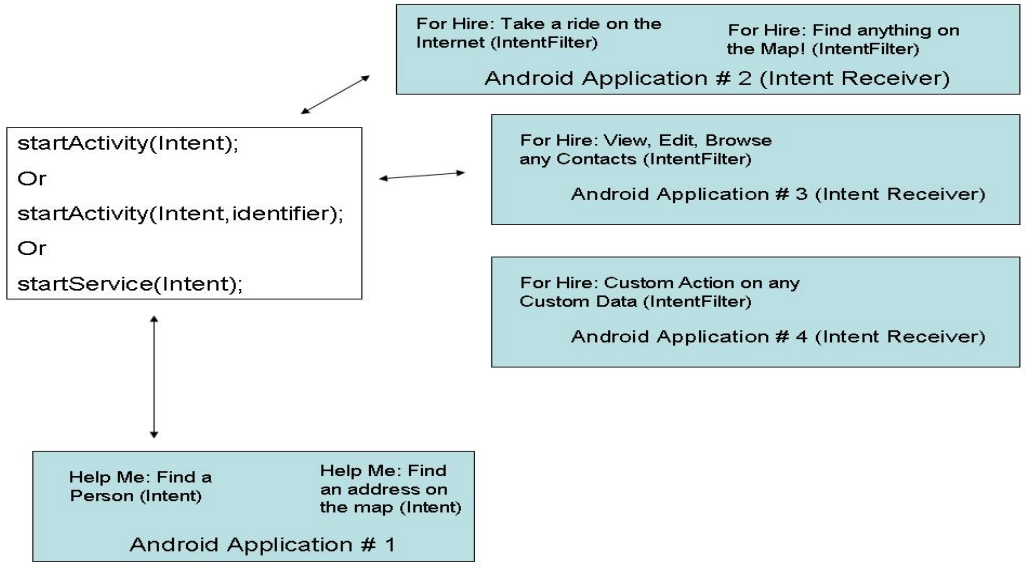


Figure 5: Intents are distributed to various Android applicaions which register themselves by way of the IntentFilter, typically in the AndroidManifest.xml file.

IntentFilters are often defined in an application's AndroidManifest.xml with the <intent-filter> tag. The AndroidManfest.xml file is essentially an application descriptor file, which is discussed later in this paper.

A common task on a mobile device is the lookup of a specific contact record for the purpose of initiating a call, sending an SMS, or perhaps looking up a snail-mail address when you are standing in line at the neighborhood

Pack-and-Ship store. A user may desire to view a specific piece of information, say a Contact Record for user 1234. In this case, the action is ACTION_VIEW and the data is a specific Contact Record identifier. This is accomplished by creating an Intent with a the action set to ACTION_VIEW and a URI which represents the specific person of interest.

Here is an example of the URI for use with the android.content.Intent.ACTION_VIEW action:

```
content://contacts/people/1234
```

Here is an example of the URI for obtaining a list of all contacts, the more generalized URI of

```
content://contacts/people
```

Here is a snippet of code demonstrating the PICKing of a contact record:

```
Intent myintent = new
Intent(Intent.ACTION_PICK,Uri.parse("content://contacts/people"));

startActivity(myintent);
```

This Intent is evaluated and passed to the most appropriate handler. In this case, the recipient would likely be a built-in Activity named com.google.android.phone.Dialer. However, the best recipient of this Intent may be an Activity contained in the same custom Android Application (ie, the one you build!), a built-in application as in this case, or it may be a 3rd party application on the device. Applications can leverage existing functionality in other applications by creating and dispatching an Intent requesting existing code to handle the Intent rather than writing code from scratch. One of the great benefits of employing Intents in this manner is that it leads to the same user interfaces being used frequently, creating familiarity for the user. *This is particularly important for mobile platforms where the user is often non tech-savvy, nor interested in learning multiple ways to accomplish the same task such as looking up a contact on their phone*.

The Intents we have discussed thus far are known as "implicit" Intents, which rely on the IntentFilter and the Android environment to dispatch the Intent to the appropriate recipient. There are also "explicit" Intents where we can specify the exact class we desire to handle the Intent. This is helpful when we know exactly which Activity we want to handle the Intent and do not want to leave anything up to "chance" in terms of what code is executed. To create an explicit Intent, use the overloaded Intent constructor which takes a class as an argument as shown here:

```
public void onClick(View v)
{
 try
 {
 startActivityForResult(new Intent(v.getContext(),RefreshJobs.class),0);
 }
 catch (Exception e)
 {
       …
 }
}
```

These examples show how an Android application creates an Intent and asks for it to be handled. Similarly, an Android application can be deployed with an IntentFilter indicating that it responds to Intents already created on the system, thereby publishing new functionality for the platform. This facet alone should bring joy to Independent Software Vendors (ISVs) who have made a living by offering better Contact Manager and To-Do List Management software titles for other mobile platforms.

Intent resolution, or dispatching, takes place at Runtime, as opposed to when the application is compiled, so specific Intent handling features can be added to a device which may provide an upgraded or more desirable set of functionality than the original shipping software. This Runtime dispatching is also referred to as "late binding".

The Power and the Complexity of Intents

It is not hard to imagine that an absolutely unique user experience is possible with Android because of the variety of Activities with specific IntentFilters installed on any given device. It is architecturally feasible to upgrade various aspects of an Android installation to provide sophisticated functionality and customization. While this may be a desirable characteristic for the user, it can be a bit troublesome for someone providing tech support and having to navigate through a number of components and applications to troubleshoot a problem.

This discussion of Intents has focused on Intents which cause User Interface elements to be displayed for the user. There are also Intents which are more "event-driven' rather than "task-oriented" as the earlier Contact Record example described. For example, the Intent class is also used to notify applications that a text message has arrived. Intents are a very central element to Android.

Now that Intents have been briefly introduced as the catalyst for navigation and event flow on Android, let's jump back out to a broader view and discuss Android Application life cycle and the key components that make Android tick.

Activating Android

This section builds on the knowledge of the Intent and IntentFilter classes introduced in the previous section and explores the four primary components of Android applications as well as their relation to the Android process model. Code snippets are included to provide a taste of Android application development.

NOTE

A particular Android application may not contain all of these elements, but it will have at least one of these elements, and could in fact have all of them.

ACTIVITY

An application may or may not have a User Interface. If it has a user interface, it will have one or more Activity.

The easiest way to think of an Android Activity is to relate a visible screen to an Activity, as more often than not, there is a one-to-one relationship between an Activity and a user interface screen. An Android application will often contain more than one Activity. Each Activity displays a user interface and responds to system and user initiated events. The Activity employs one or more Views to present the actual User Interface elements to the user. The Activity class is extended by user classes, as seen in the code listing 1.

Listing 1: A very basic Activity in an Android application.

```
package com.msi.manning.chapter1;

import android.app.Activity;                                    1

import android.os.Bundle;

public class activity1 extends Activity                         2
{
    @Override
    public void onCreate(Bundle icicle)
    {
        super.onCreate(icicle);
        setContentView(R.layout.main);                          3
    }
}
```

1. The Activity class is part of the `android.app` java package, found in the Android runtime. The Android runtime is deployed in the android.jar file.

2. The class activity1 extends the class Activity.

3. One of the primary tasks an Activity performs is the display of User Interface elements which are implemented as Views and described in XML layout files.

Moving from one Activity to another is accomplished with the `startActivity()` method or the `startActivityForResult()` method when a "synchronous" call/result paradigm is desired. The argument to these methods is the Intent.

You say Intent, I say Intent.

The Intent class is used in similar sounding, but very different scenarios.

There are Intents used to assist in navigation from one activity to the next such as the example given earlier of VIEWing a Contact Record. Activities are the targets of these kinds of Intents used with the startActivity or startActivityForResult methods.

Services can be started by passing an Intent to the startService method.

BroadcastReceivers receive Intents when responding to system-wide events such as the phone ringing or an incoming text message.

The Activity represents a very visible application component within Android. With assistance from the View class, the Activity is the most common type of Android application. The next topic of interest is the Service, which runs in the background and does not generally present a direct User Interface.

SERVICE

If an application is to have a long life cycle it should be put into a Service. For example a background data synchronization utility running continuously should be implemented as a Service.

Like the Activity, a Service is a class provided in the Android runtime which should be extended, as seen in Listing 2, which sends a message to the Android log periodically:

Listing 2: A simple example of an Android Service.

```
package com.msi.manning.chapter1;

import android.app.Service;                                          1
import android.os.IBinder;
import android.util.Log;                                             2

public class service1 extends Service implements Runnable            3
{
        public static final String tag = "service1";
        private int myiteration = 0;
    @Override
    protected void onCreate()                                        4
    {
          super.onCreate();
           Thread mythread = new Thread (this);
          mythread.start();
    }

    public void run()
    {
              while (true)
              {
                     try
                     {
                            Log.i(tag,"service1 firing : # " + myiteration++);
                            Thread.sleep(10000);
                     }
                     catch(Exception ee)
                     {
                            Log.e(tag,ee.getMessage());
                     }
              }
    }
```

```
        @Override
        public IBinder onBind(Intent intent)                                    5
        {
                // TODO Auto-generated method stub
                return null;
        }

}
```

1. This example requires that the package android.app.Service be imported. This package contains the Service class.

2. This example also demonstrates Android's Logging mechanism, which is useful for debugging purposes.

3. The service1 class extends Service. It also implements Runnable to perform its main task on a separate Thread. This is a common paradigm, but it is not required to implement Runnable.

4. The onCreate method of the Service class permits the application to perform initialization type tasks.

Services are started with the startService(Intent) method of the abstract Context class. Note that again, the Intent is used to initiate a desired result on the platform.

Now that the application has a user interface in an Activity and a means to have a long-running task in a Service, it is time to explore the BroadcastReceiver, another form of Android application which is dedicated to processing Intents.

BroadcastReceiver

If an application desires to receive and respond to a global event, such as the phone ringing or an incoming text message, it must register itself as an BroadcastReceiver. An application registers to receive Intents in a couple of different manners:

- The application implements a <receiver> element in the AndroidManfest.xml file which describes the BroadcastReceiver's class name and enumerates its IntentFilters. Remember, the IntentFilter is a descriptor of the Intent an application desires to process. If the receiver is registered in the AndroidManifest.xml file, it does not have to be running in order to be triggered when the event occurs as the application is started automatically upon the triggering event. All of this house-keeping is managed by Android.
- An application registers itself at runtime via the Context class's registerReceiver method.

Like Services, BroadcastReceivers do not have a User Interface. Of even more importance, the code running in the onReceive method of a BroadcastReceiver should make no assumptions about persistence or long-running operations. If the BroadcastReceiver requires more than a trivial amount of code execution, it is recommended that the code initiate a request to a Service to complete the requested functionality.

NOTE

The familiar Intent class is used in the triggering of BroadcastReceivers, however the use of these Intents are mutually exclusive from the Intents used to start an Activity or a Service as previously discussed.

An BroadcastReceiver implements the abstract method onReceive to process incoming Intents. The arguments to the method are a Context and an Intent. The method returns void, but there are a handful of methods useful for passing back "results", including setResult which passes back to the invoker an integer return code, a String return value, and a Bundle value, which can contain any number of objects.

Listing 3 is an example of an BroadcastReceiver triggering upon an incoming Text Message.

Listing 3: A sample IntentReceiver.

```
package com.msi.manning.unlockingandroid;

import android.content.Context;
import android.content.Intent;
import android.content.IntentReceiver;
import android.util.Log;
```

```
public class MySMSMailBox extends BroadcastReceiver                                    1
{
    public static final String tag = "MySMSMailBox";                                   2

    @Override
    public void onReceive(Context context, Intent intent)                          3
    {

        Log.i(tag,"onReceive");
      if (intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED"))   4
        {
            Log.i(tag,"Found our Event!");                                              5
        }
    }

}
```

1. The class MySMSMailBox extends the BroadcastReceiver class. This is the most straight-forward way to employ an BroadcastReceiver. Note the class name of "MySMSMailBox", as it will be used in the AndroidManifest.xml file, shown in code listing 4.

2. The tag variable is used in conjunction with the logging mechanism to assist in labeling messages sent to the console log on the emulator.

3. The onReceive method is where all of the work takes place in an BroadcastReceiver. The BroadcastReceiver is required to implement this method. Note that a given BroadcastReceiver can register multiple IntentFilters and can therefore be instantiated for an arbitrary number of Intents.

4. It is important to make sure to handle the appropriate Intent by checking the action of the incoming Intent as shown.

5. Once the desired Intent is received, carry out the specific functionality required. A common task in an SMS receiving application would be to parse the message and display it to the user via a Notification Manager display.

In order for this BroadcastReceiver to "fire" and receive this Intent, it must be listed in the AndroidManifest.xml file as seen in listing 4. This listing contains the elements required to respond to an incoming text message.

Listing 4: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.msi.manning.unlockingandroid">
        <uses-permission android:name="android.permission.RECEIVE_SMS" />                  1
    <application android:icon="@drawable/icon">
        <activity android:name=".chapter1" android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name=".MySMSMailBox" >
        2
                <intent-filter>
        3
                        <action android:name="android.provider.Telephony.SMS_RECEIVED" />
                </intent-filter>
        </receiver>
    </application>
</manifest>
```

1. Note that certain tasks on Android require the application to have a designated privilege. To give an application the required permissions, the <uses-permission> tag is used. This is discussed in further detail later in this paper in the AndroidManifest.xml section.

2. The <receiver> tag contains the class name of the class implementing the BroadcastReceiver. Note that in this example, the class name is “MySMSMailBox” from the package named “com.msi.manning.unlockingandroid”.

3. The IntentFilter is defined in the <intent-filter> tag. The desired action is “android.provider.Telephony.SMS_RECEIVED”. The Android SDK enumerates the available actions for the standard Intents. In addition, remember that user applications can define their own Intents as well as listen for them.

Testing SMS

The emulator has a built in set of tools for manipulating certain telephony behavior to simulate a variety of conditions such as in and out of network coverage, placing phone calls, and as was demonstrated in this section’s example, sending sms messages.

To send a sms message to the emulator, telnet to port 5554 (may vary on your system) which will connect to the emulator and issue the following command at the prompt:

sms send <sender’s phone number> <body of text message>

Type help from the prompt to learn about other commands available.

Now that Intents and the various Android classes which process or handle Intents have been introduced, it is time to explore the next major Android application topic of this paper, the Content Provider, Android’s preferred data publishing mechanism.

CONTENT PROVIDER

If an application manages data and needs to expose that data to other applications running in the Android environment, a ContentProvider should be implemented. Alternatively, if an application component (Activity, Service, or BroadcastReceiver) needs to access data from another application, the other application’s ContentProvider is used to access that data. The Content Provider implements a standard set of methods to permit applications access to a data store. The access may be for read and/or write operations. A ContentProvider may provide data to an Activity or Service in the same containing application as well as an Activity or Service contained in other applications.

A ContentProvider may use any form of data storage mechanisms available on the Android platform including files, SQLite databases, or even a memory based Hash Map if data persistence is not required. In essence, the Content Provider is a data layer providing data abstraction for its clients and centralizing storage and retrieval routines in a single place.

Directly sharing files or databases is discouraged on the Android platform and is further enforced by the Linux security system which prevents ad-hoc file access from one application space to another without appropriate permissions.

Data stored in a Content Provider may be of traditional data types such as integers and strings. Content Providers can also manage binary data such as image data. When binary data is retrieved, suggested practice is to return a string representing the filename containing the binary data. In the event that a filename is returned as part of a Content Provider query, the file should not be access directly, but rather use the helper class ContentResolver’s openInputStream method to access the binary data. This approach avoids Linux process/security hurdles as well as keeps all data access normalized through the Content Provider. Figure 6 outlines the relationship between ContentProviders, data stores, and their “clients”.

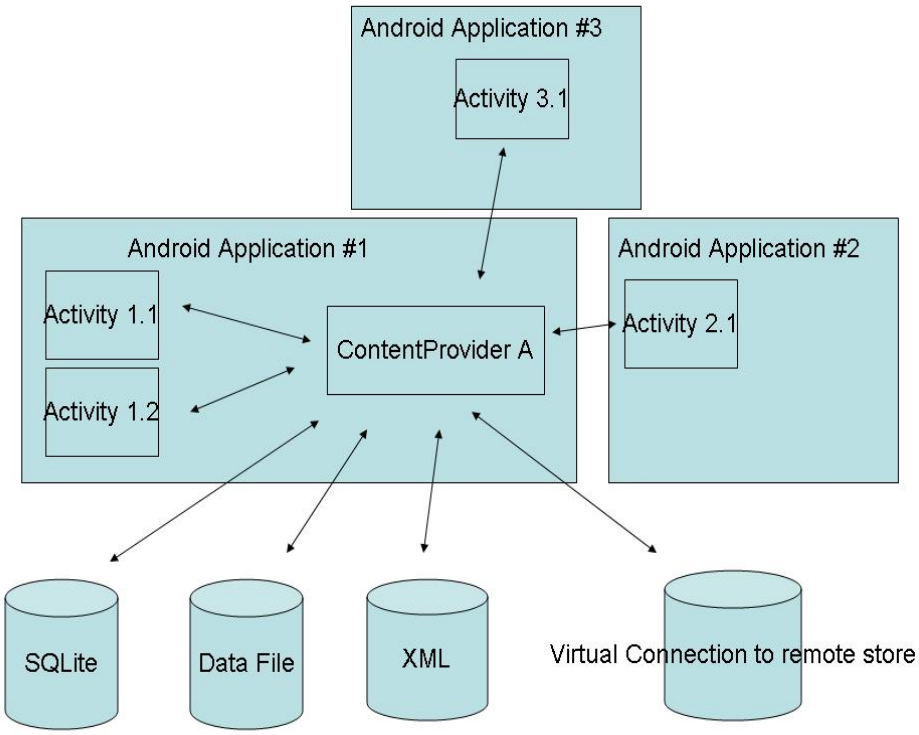


Figure 6 The Content Provider is the "data tier" for Android applications and is the prescribed manner in which data is accessed and shared on the device.

A ContentProvider's data is accessed through the familiar Content URI. A ContentProvider defines this as a public static final String, for example an application might have a data store managing material safety data sheets. The Content URI for this Content Provider might look like:

```
public static final Uri CONTENT_URI =
Uri.parse("content://com.msi.manning.provider.unlockingandroid/datasheets");
```

From this point, accessing a Content Provider is similar to using Structured Query Language in other platforms, though a complete SQL statement is not employed. A query is submitted to the Content Provider including the columns desired, and optional "Where" and "Order By" clauses. For those familiar with parameterized queries in SQL, parameter substitution is even supported. Results are returned in the Cursor class, of course.

NOTE

In many ways, a Content Provider acts like a database server. While an application could contain only a Content Provider and in essence be a database server, it is important to note that a Content Provider is typically a component of a larger Android application which hosts one or more Activity, Service and/or BroadcastReceiver as well.

This concludes the brief introduction to the major Android application classes. Gaining an understanding of these classes and how they work together is an important aspect of Android development. Getting application components to work together can be a daunting task at times. For example, have you ever had a piece of software

that just didn't seem to work properly on your computer? Perhaps it was copied and not "installed" properly. Every software platform has "environmental" concerns, though they vary by platform. For example, when connecting to a remote resource such as a database server or ftp server, which user name and password are used? What about the necessary libraries to run your application? These are all topics related to software deployment. An Android application requires a file named AndroidManifest.xml, which ties together the necessary pieces to run an Android application.

AndroidManifest.xml

The previous sections introduced the common elements of an Android application. To restate an important comment, an Android application will contain one or more Activity, Service, BroadcastReceiver, and ContentProvider. Some of these elements will advertise the Intents they are interested in processing via the IntentFilter mechanism. All of these pieces of information need to be tied together in order for an Android application to execute. The mechanism for this task is the AndroidManifest.xml file.

The AndroidManifest.xml file exists in the root of an application directory and contains all of the design-time relationships of a specific application and Intents. Listing 5 is an example of a very simple AndroidManifest.xml file:

Listing 5: AndroidManifest.xml file for a very basic Android application. AndroidManfest.xml files act as deployment descriptors for Android applications.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.unlockingandroid">                                    1
    <application android:icon="@drawable/icon">
        <activity android:name=".chapter1" android:label="@string/app_name">       2
            <intent-filter>
     3
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

1. The manifest element contains the obligatory namespace as well as the Java package name containing this application.

2. This application contains a single Activity, with a class name of "chapter1". Note also the @string syntax. Any time an @ symbol is used in an AndroidManifest.xml file, it is referencing information stored in one of the resource files. For example, in this case, the label attribute is obtained from the *app_name* string resource defined elsewhere in the application.

3. The Activity contains a single IntentFilter definition. The IntentFilter seen here is the most common IntentFilter seen in Android applications. The action android.intent.action.MAIN indicates that this is an entry point to the application. The category android.intent.category.LAUNCHER places this Activity in the launcher window. Note that it is possible to have multiple Activity elements in a manifest file (and thereby an application), with more than one of them visible in the launcher window, as seen in Figure 7.

Figure 7 Applications are listed in the launcher based on their Intent Filter. In this example, the application titled “Where Do You Live” is available in the LAUNCHER category.

In addition to the elements seen in this sample manifest file, other common tags include:

- The <service> tag represents a Service. The attributes of the service tag include its class and label. A Service may also include the <intent-filter> tag.
- The <receiver> tag represents a BroadcastReceiver, which may or may not have an explicit <intent-filter> tag.
- The <uses-permission> tag tells Android that this application requires certain security privileges. For example, if an application requires access to the Contacts on a device, it requires the following tag in its AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

Now that there is a basic understanding of the Android application and the AndroidManifest.xml file which describes its components, it is time to discuss how and where it actually executes. The next section discusses the relationship between Android applications and its Linux and Dalvik Virtual Machine runtime.

Mapping Applications to Processes

Android applications run each in a single Linux process. Android relies on Linux for process management and the application itself runs in an instance of the Dalvik Virtual Machine. The operating system may need to unload, or even kill, an application from time to time to accommodate resource allocation demands. There is a hierarchy or sequence the system uses to select the victim of a resource shortage. In general, the rules are as follows:

- Visible, running activities have top priority
- Visible, non running activities are important, because they are recently paused and are likely to be resumed shortly.
- A running service is next in priority
- The most likely candidates for termination are processes which are empty (loaded perhaps for performance caching purposes) or processes which have dormant Activities.

ps -a

The Linux environment is complete with process management. It is possible to launch and kill applications directly from the "shell" on the Android platform. However, this is largely a developer's debugging task, not something the average Android handset user is likely to be carrying out. It is a real nice-to-have for troubleshooting application issues. It is unheard of on commercially available mobile phones to touch the "metal" in this fashion.

It is time to wrap up this paper with a simple Android application.

An Android Application

This section presents a simple Android application demonstrating a single Activity, with one View. The Activity collects data, a street address to be specific, and creates an Intent to find this address. The Intent is ultimately dispatched to Google Maps. Figure 8 is a screen shot of the application running on the emulator. The name of the application is "Where Do You Live".

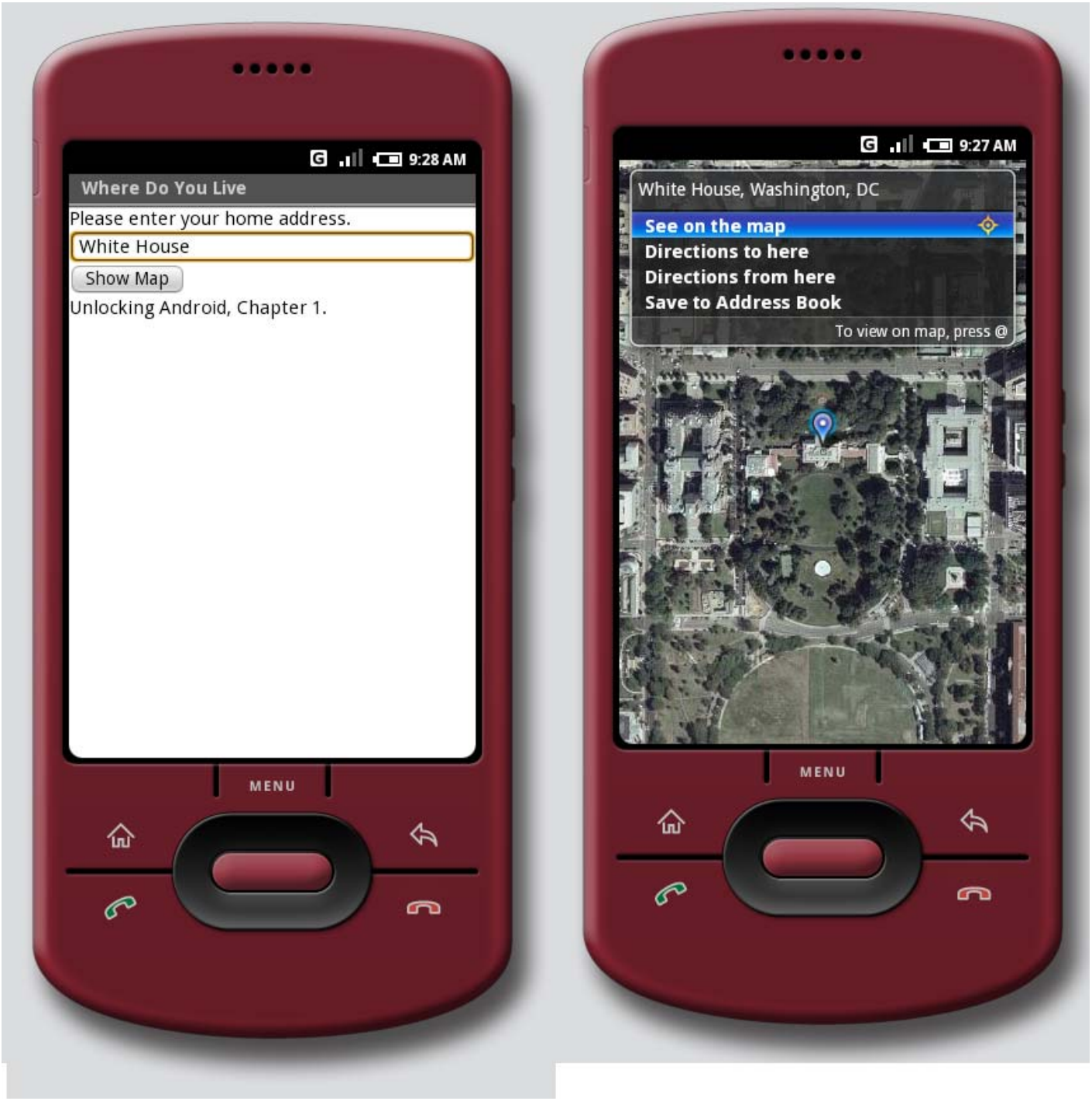


Figure 8 This Android application demonstrates a simple Activity and Intent.

As previously introduced, the AndroidManifest.xml file contains the descriptors for the high level classes of the application. This application contains a single Activity named AWhereDoYouLive. The application's AndroidManifest.xml file is see in Listing 6.

Listing 6: AndroidManifest.xml for the Where Do You Live Application.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.unlockingandroid">
    <application android:icon="@drawable/icon">
        <activity android:name=".AWhereDoYouLive" 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 sole Activity is implemented in the file AWhereDoYouLive.java, presented in Listing 7.

Listing 7: Implementing the Android Activity for this paper's sample application with AWhereDoYouLive.java implements

```
package com.msi.manning.unlockingandroid;

import com.msi.manning.unlockingandroid.R;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class AWhereDoYouLive extends Activity
{
    @Override
    public void onCreate(Bundle icicle)
    {
        super.onCreate(icicle);
        setContentView(R.layout.main);
1


        final EditText addressfield = (EditText) findViewById(R.id.address);          2

        final Button button = (Button) findViewById(R.id.launchmap);                 3
        button.setOnClickListener(new Button.OnClickListener()
        {
            public void onClick(View v)
            {
                    try
                    {
                            String address = addressfield.getText().toString();   4
                            address = address.replace(' ', '+');
                            Intent myIntent = new Intent(android.content.Intent.ACTION_VIEW,
Uri.parse("geo:0,0?q=" + address));
5
                    startActivity(myIntent);
6
                    }
                    catch (Exception e)
                    {
                            …
                    }
                    }
        });
    }
}
```

1. The setContentView method creates the primary User Interface, which is a layout defined in main.xml in the /res/layout directory

2. The EditText View collects information. It is a “text box” or “edit box” in generic programming parlance. The findViewById method connects the resource identified by “R.id.address” to an instance of the EditText class.

3. A Button object is connected to the launchmap user interface element.

4. The address is retrieved from the user interface element.

5. An Intent is created, requesting the ACTION_VIEW. The URI is a string representing a Geographic Search.

6. The Intent is initiated with a call to startActivity.

Resources are pre-compiled into a special class known as the “R” class, as seen in Listing 8. The final members of this class represent user interface elements. Note that you should never modify the R.java file manually as it is automatically built every time the underlying resources change.

Listing 8: R.java contains the R class, which has User Interface element identifiers. .

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */

package com.msi.manning.unlockingandroid;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class id {
        public static final int address=0x7f050000;
        public static final int launchmap=0x7f050001;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
    }
}
```

The primary screen of this application is defined as a LinearLayout View as seen in Listing 9. It is a single layout containing one label, one text entry element and one button control.

Listing 9: Main.xml defines the user interface elements for this paper’s sample application.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Please enter your home address."
    />
<EditText
    android:id="@+id/address"                                          1
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
     android:autoText="true"
     />
<Button
    android:id="@+id/launchmap"                                        2
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Show Map"
        />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Unlocking Android, Chapter 1."
        />
    </LinearLayout>
```

1 & 2. Note the use of the '@' symbol in this resource's id attribute. This causes the appropriate entries to be made in to the R class via the automatically generated R.java file. These R class members are used in the calls to findViewById() as seen previously to tie the UI elements to an instance of the appropriate class.

A strings file and icon round out the resources in this simple application as seen in Listing 10. The strings.xml file is used to localize string content.

Listing 10: strings.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Where Do You Live</string>
</resources>
```

This concludes our first Android application.

Summary

This paper has introduced the Android platform, briefly touched on market positioning and what Android is up against as a newcomer to the mobile marketplace. Android is such a new platform that there are sure to be changes and announcements as the platform matures and actual hardware hits the market. New platforms need to be adopted and flexed to identify the strengths and expose the weaknesses so they can be improved. Perhaps the biggest challenge for Android is to navigate the world of the mobile operators and convince them that Android is good for business. Fortunately with Google behind it, Android should have some ability to flex its muscles and we'll see some devices sooner than later.

In this paper the Android stack was examined and its relationship with Linux and Java was discussed. With Linux at its core, Android is a formidable platform, especially for the mobile space. While Android development is done in the Java programming language, the runtime is executed in the Dalvik Virtual Machine, as an alternative to the Java Virtual Machine from Sun. Regardless of the VM, Java coding skills are an important aspect of Android development. The big question is the degree to which existing Java libraries can be leveraged.

We also examined the Android Intent class. The Intent is what makes Android tick. It is responsible for how events flow, which code handles them, and provides a mechanism for delivering specific functionality to the platform, enabling 3rd party developers to deliver innovative solutions and products for Android. The main application classes of Activity, Service and BroadcastReceiver were all introduced with a simple code snippet example for each. Each of these application classes interacts with Intents in a slightly different manner, but the core facility of using Intents and using Content URIs to access functionality and data combine to create the innovative and flexible Android environment.

The AndroidManifest.xml descriptor file ties all of the details together for an Android application. It includes all of the information necessary for the application to run, what Intents it can handle and what permissions the application requires.

Finally, this paper provided a taste of Android application development with a very simple example tying a simple user interface, an Intent and Google Maps into one seamless user experience. This is just scratching the surface of what Android can do.