CHAPTER 12

Android

After the iPhone changed the smartphone landscape in 2007, the first platform vendor to respond was Google, announcing that Android—its own mobile operating system—would be released the following year. In many ways, it's a very different take on the entire smartphone concept than the iPhone: the operating system and tools are open source, and the environment as a whole is much less tightly regulated. But in other ways—those that matter to a web developer—it's far more similar than different: as a platform, Android is web-enabled every bit as much as iPhone.

So, this chapter will largely mirror the previous one. You'll learn how to install the Android tool set (including the emulator), test and optimize your gadget as a pure web app, repackage it as a native application, and then deploy it to the wider world. Let's get started!

Note Android uses the term *widget* both for internal user interface elements and, more frequently, for home-screen applications (like the Google Search bar in Figure 12-1). Consequently, I'll stay well clear of the word when referring to web gadgets in this chapter.

Introducing Android

Android started life as an independent mobile software company and was acquired by Google in 2005, and the eponymous operating system was announced in late 2007. In contrast to iPhone's near-complete control by a single company (from hardware through OS to applications), Android is backed by the Open Handset Alliance (OHA), a loose affiliation of mobile phone handset manufacturers, software vendors, and carriers. Although Google still maintains a central role in Android development, the source code is open and is used on hardware from a variety of companies both within and without the OHA. This range of devices is beginning to branch out beyond pure mobile phones, presenting a diverse set of opportunities for your web gadget.

Although the overall merits of Android and iPhone are a subject of great debate, when you get into web development on the platform, you'll find that they are essentially equal. Android's browser is again based on the WebKit rendering engine and, apart from a slightly different user interface, is functionally a near twin to Mobile Safari. It's every bit as good at displaying web sites and running web applications in their full glory. The Android browser is also well integrated into the native programming environment—again, just as on the iPhone—and later in the chapter, you'll be exploiting this integration in much the same way to repackage your web gadgets as a full citizen of the Android community.

Installing the Android Emulator

As a platform, Android relies heavily on the Java language. Not only is it the primary means for developing native Android apps, but the software development tools all use Java themselves. Consequently, the first step toward developing for Android is to ensure you have a current version of the Java Development Kit (JDK) on your computer. At least version 5 is required for the examples in this chapter, though later versions should work as well. You can obtain the JDK from `http://java.sun.com/javase/downloads`; follow the instructions appropriate for your host operating system.

Being Java-based, the Android software development kit (SDK) is itself quite cross-platform. It'll happily run on Microsoft Windows (XP or later), Mac OS X 10.2.4+ (although only Intel-based), and most distributions of Linux. The instructions for this chapter tend to favor a Windows installation, but other systems are quite similar.

With the JDK installed, your next step is to download the Android SDK from `http://developer.android.com/sdk`. You'll want to select the current SDK version from that URL; at this writing, it's API Level 3, which equates to the v1.5 of the Android OS (aka Cupcake). The platform is new enough that significant changes are still occurring between versions, and new versions are coming quite quickly, so be aware that these instructions may have changed slightly by the time you read this.

The download itself is a zip file, rather than an executable installation; within this archive, you'll find a single directory containing the entire SDK, with a name like `android-sdk-windows-1.5_r3`. Unzip this directory to a location where you'll be able to find it easily, because I'll be referring to it throughout the chapter. In addition to the emulator, the SDK contains a variety of other tools, examples, and documentation. For the rest of the chapter, I'll be using `[SDK]` to refer to this extracted directory; substitute your local Android SDK path wherever you see `[SDK]`.

With the SDK extracted, the Android emulator is actually installed, but before you can use it, you need to create a *virtual device*. This is essentially a configuration package for an emulator instance; most other platforms (like Symbian S60) prepackage such instances independently, but with Android, you create your own. To do so, open a command prompt (in Windows, select Start → Run, and enter `cmd`), navigate to your `[SDK]/tools` directory, and run the following command:

```
android create avd --target 3 --name my_avd
```

This will create an instance of the default Android virtual device (AVD), named `my_avd`, for the given API release level. If you've installed a different API level than `3`, please enter its number here instead.

You're now ready to run the Android emulator. Back in the command shell (and still in the `[SDK]/tools` directory), enter the following:

```
emulator.exe -avd my_avd
```

The emulator should now start, and when ready, it will look something like Figure 12-1. I recommend creating a shortcut for the emulator command in your host operating system, rather than typing the previous command every time you want to start it.

Figure 12-1. *The default emulator for API Level 3 (Android v1.5)*

Android's is a true emulator (rather than a simulator), meaning that it closely mirrors the environment of a physical device and is capable of running the same executable code. On the downside, this level of abstraction can take a minute or two to boot; contrast this with the iPhone Simulator, which leverages the architecture it shares with the underlying OS to start almost instantly.

As usual, I recommend you take a few minutes now to become familiar with the emulated Android environment.

■ **Tip** Save yourself some annoyance, by disabling the emulator screen's autodim functionality. From the home screen, select Menu → Settings → Sound & display → Screen timeout → Never timeout.

Running Web Applications

By this point in the book, you've probably opened your core HTML gadget on a number of simulated handsets and generally have it working fairly well with them. If so, Android's browser will likely be a bit anticlimactic; the strong odds are that your gadget will work fine the first time out—especially if you've already been through this process with iPhone's Mobile Safari.

Nonetheless, you'll need to confirm this and work through any problems that might arise. Prominently displayed on the emulator's home screen is a Browser shortcut, and not surprisingly, this is the place to start. Open it now, and enter the URL for your core web gadget to see how things look (as shown in Figure 12-2).

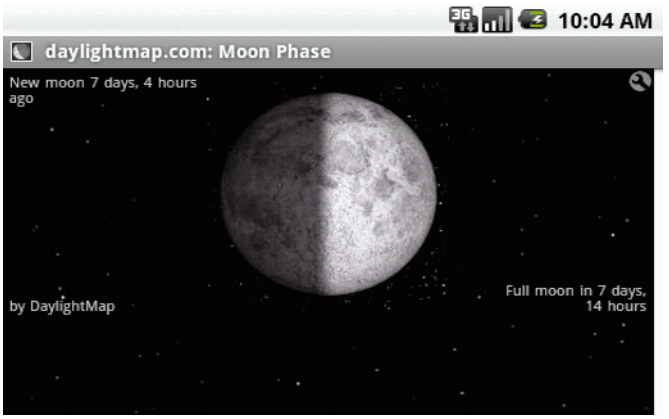

Figure 12-2. *The Moon Phase core gadget in the Android browser*

■ **Tip** To try your gadget on different screen orientations, rotate the emulator between portrait and landscape by pressing Ctrl+F11 on your host computer.

In the event that you do run into trouble, another tool in the SDK can help you. It's called the DDMS, for Dalvik Debug Monitor Service (Dalvik being the Java virtual machine inside Android), and you'll find it at `[SDK]/tools/ddms`. When you run it, you'll see a window similar to Figure 12-3; select your emulator instance in the upper-left pane, and the lower Log pane will show a stream of debug information (Figure 12-3).

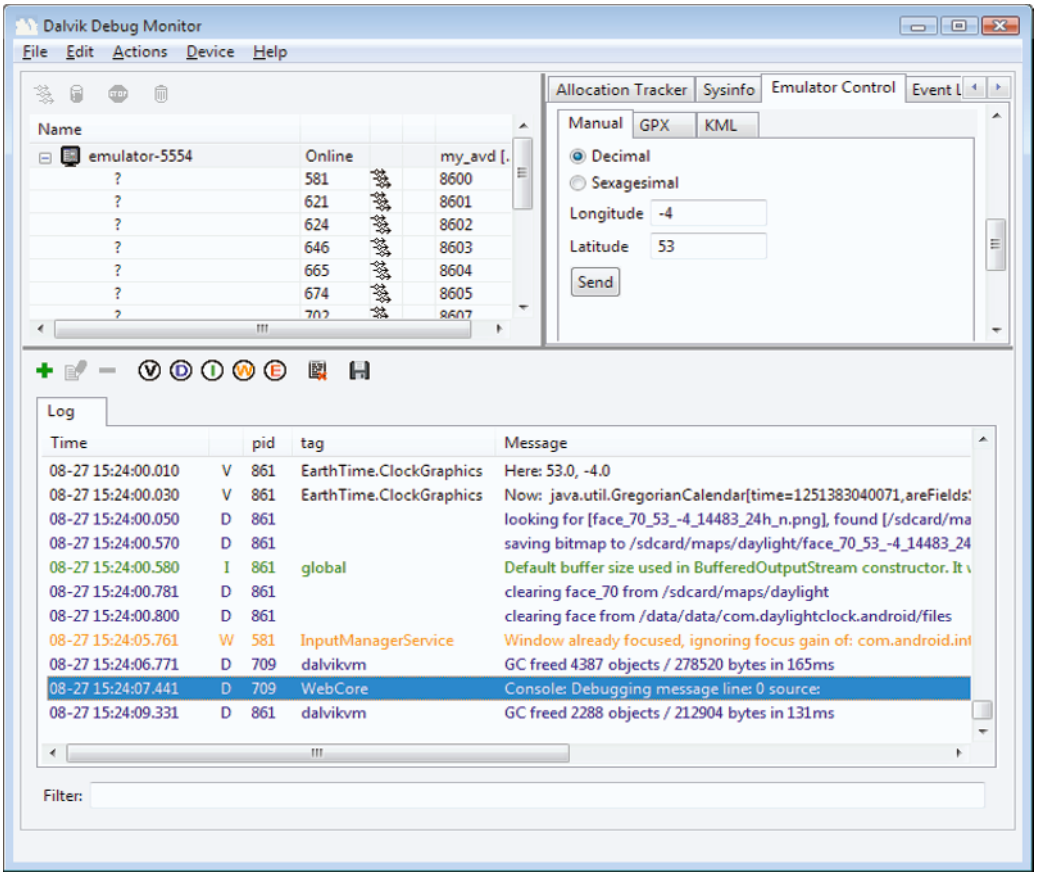

Figure 12-3. *The Dalvik Debug Monitor Service, with JavaScript* `console.log` *output*

Within this log will be any JavaScript errors generated by the browser, and any calls to `console.log` that you place in your own script will appear here as well. In Figure 12-3, for example, I've highlighted the output from a JavaScript statement of `console.log('Debugging message')`. Be aware that DDMS requires `console.log` to be called with a printable string; it doesn't support object logging, like many built-in browser consoles do.

In addition to confirming that your core gadget is ready to serve the growing number of Android users in the wild, I recommend forking your HTML and optimizing a version specifically for Android. This copy will serve as your foundation for the remainder of the chapter, so take the time to ensure that it is working and looking its best, even if that means adding code that isn't appropriate in the core, cross-browser version.

■ **Tip** If you ported your gadget to iPhone in the previous chapter, use that as your starting point for Android. The two browsers' similarity will give you a head start.

Creating Native Android Applications

As another web-enabled smartphone platform, Android is well positioned to implement the same sort of technique that was used in the previous chapter: wrapping a pure web gadget as a "native" application, with all the privileges that will entail. In the rest of this chapter, I'll walk you through the process of creating such a wrapper for your own gadget and then explore what can be done with it after you do.

The architecture of the app, and the process to construct it, will be very similar to the native iPhone wrapper in the previous chapter. You'll generate a simple Android application with a single `WebView` control as its sole user interface element (notice even the control name closely parallels iPhone). You'll then package your gadget resources with the app, loading the core HTML into the `WebView` when it starts up, and you'll be on your way.

So, just as in the previous chapter, you'll need to get your gadget resources—all the static source files and images referred to in your web code—consolidated under a single directory. Obviously, this should be the Android-specific port you developed at the end of the previous section.

You'll also need an application icon for the Android launcher. On this platform, the standard is for 48-pixel-square PNGs, and your integration work will be easier if it's named `icon.png`. Also, I recommend that you create one on an alpha-channel transparent background if possible (such as Figure 12-4). Not only will this look best in the Android app tray, but users can place shortcuts to applications on their home screens, and a good antialiased PNG is definitely your best option here.

Figure 12-4. My Android application `icon.png` for Moon Phase

"NATIVE" ANDROID APPLICATIONS?

The phrase *native application* is a bit slippery on Android. Traditionally, the term refers to applications that are compiled directly for the target operating system and that run as full-fledged members of it. In this chapter, you'll certainly learn how to repackage your web app as a full-fledged Android app, but there's some question about how native it will truly be.

The issue is that you are repackaging in Java, and as you may know, very rarely is Java truly native. It was conceived as a portable language that would run in a virtual machine (VM) on many different host operating systems. By and large, that's still how things stand, and Android is no exception: Java bytecode executes in a VM called Dalvik, running on top of an underlying Linux kernel.

So, Java isn't fully native on Android either, despite being virtually the only language that real-world apps are written in. There actually is a native programming level available, compiling C and C++ code directly for the OS, and it's appropriate for performance-critical areas of functionality. When Android developers use the term *native*, this is usually what they're referring to. But an important point is that this NDK has no access to the user interface; the C/C++ routines must be called from within a larger Java app. So, in a real sense, there are *no* truly native applications on Android.

Of course, this is all a bit academic for a repackaged web app, where the core functionality is written in JavaScript. Nonetheless, knowing about the different definitions of *native* on Android might save you from some confusion at a later date.

Unlike iPhone, there are very few downsides to following the native-app repackaging process through on Android. This is primarily because of the openness of the platform: there is no Apple-like gatekeeper deciding which applications users can install on their phones. By a simple process (which I'll cover in the "Deploying Android Applications" section later in the chapter), any Android user can install any application, from anywhere. This means that the native application that you'll build from your gadget in the next few sections can be widely distributed via channels ranging from Google's official market to your own web site. There's no good reason not to do it.

Setting Up the Environment

It's possible to build Android applications with only the SDK you installed earlier in the chapter, using command-line tools to compile the code and load it onto the simulator. But that's definitely the hard way. Continuing with the cross-platform theme, Google has produced a good Android plug-in for the popular Eclipse IDE, and using it is the course I firmly recommend. It's also the assumption I'll make for the instructions throughout this part of the chapter.

■ **Note** If you are determined not to use Eclipse, you can find a guide to the equivalent command-line utilities at `http://developer.android.com/guide/developing/other-ide.html`.

Installing Eclipse

If you've been doing web development for long, you may have Eclipse already installed, but if not, now is the time. As another open source tool, it's a free download from the Eclipse Foundation (`http://eclipse.org/downloads`). You'll need at least version 3.3 (Europa), though later versions generally work as well, and I recommend at least 3.4. The instructions in this section are specifically for 3.5 (Galileo); other versions are similar, and you can find guidance at `http://developer.android.com/sdk` if you need it.

■ **Tip** If you're installing Eclipse for the first time, choose the package named Eclipse IDE for Java EE (Enterprise Edition). It includes the Java support that Android needs, plus good code editors for HTML, JavaScript, and CSS, which will come in handy sooner or later.

Like the SDK itself, Eclipse comes packaged as a zip file; extract it when the download completes (on Windows, I recommend just placing the `eclipse` folder directly in your `C:\Program Files`). The app itself is `eclipse.exe` in the root directory; again, creating a shortcut somewhere convenient is a good idea.

The first time you run Eclipse, it'll ask you to select a *workspace* folder. This is simply a directory where all your Eclipse projects will reside. If you already have a main directory you use for your programming work, go ahead and use that for your workspace; Eclipse won't harm whatever is already there. Otherwise, Eclipse will create a new folder with the path you specify.

Installing the Android SDK Plug-In

The Eclipse installation in the previous section doesn't yet include the Android tools, so your next task is to install the plug-in:

1. In Eclipse, select Help → Install New Software (see Figure 12-5).
2. In the Install dialog box, enter the following URL in Work With, and click Add:

   `https://dl-ssl.google.com/android/eclipse/`
3. When prompted, give it a name like `Android SDK`.
4. Eclipse will display the plug-ins available at this site. Select Developer Tools, and proceed with the rest of the Install Wizard.

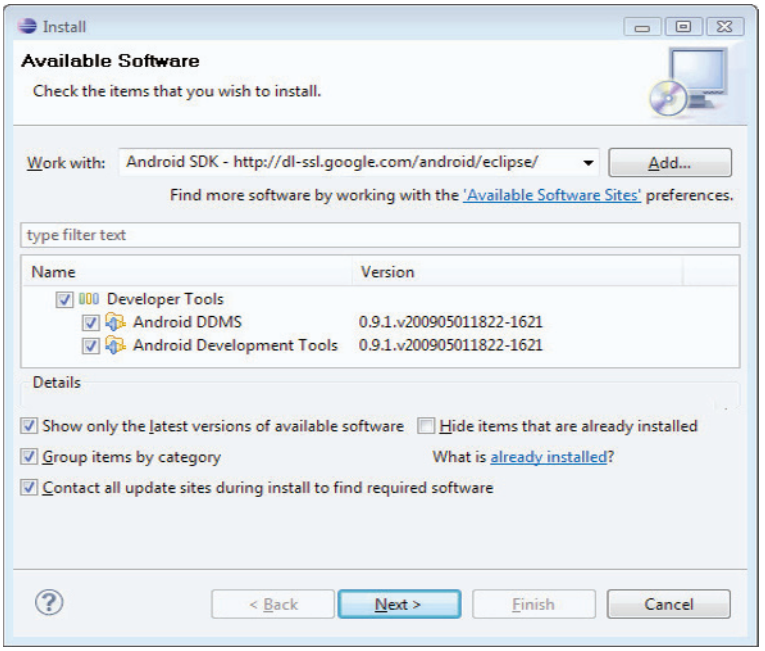

Figure 12-5. *Installing the Android plug-in into Eclipse 3.5*

Eclipse will most likely restart after installing the plug-in. When it has, you have one last task; it needs to know the location where you unzipped your Android SDK. In Eclipse, select Window → Preferences → Android, and then browse to your `[SDK]` directory.

Adapting the Core Gadget

With Eclipse and the Android plug-in installed, you should now be ready to create an Android application. As outlined earlier, your approach will be to create a `WebView` framework, with the gadget resources packaged inside.

Building the Framework

The Android plug-in for Eclipse includes good tools for generating the starting code for a project; the following instructions will walk you through the process of creating an Android app for your gadget:

1. In Eclipse, select File → New → Project. In the dialog box that opens, select Android Project, and click Next.
2. Complete the New Android Project dialog box (Figure 12-6) with the following values, and click Finish. This will create a project folder in your workspace directory.
   a. The project name is an internal name for the project within Eclipse. This will also be the name of the project folder, so you may want to include *Android* in the name, like `MoonPhase Android`.
   b. The application name is the name you want visible to users who install your app, such as `Moon Phase`.
   c. The package name is effectively a namespace for your project, and since this is Java, it should be in reverse-domain format. It also needs to be unique for each Android app you release. Mine is `com.daylightmap.moon.android`.
   d. Select the box for Create Activity, and give it a good short camel-case name for your project, like `MoonPhase` (without spaces). I'll be referring to this string as `[your name]` for the rest of these instructions.
   e. For Min SDK Version, use whatever version you created your AVD for earlier in the chapter (in my previous example, it's `3`).

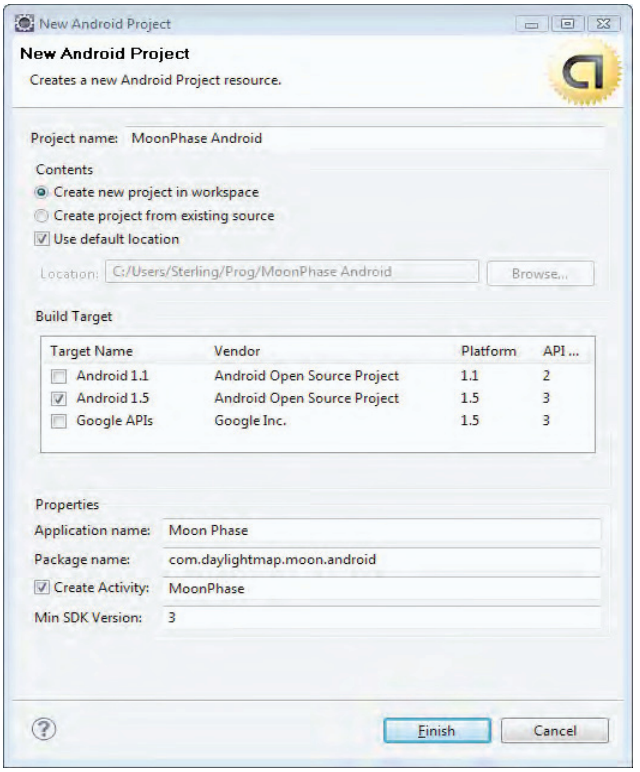

Figure 12-6. *Creating an Android application project*

3. When your project has been created, Eclipse's Package Explorer (usually open on the left side of the IDE) is what you'll use to navigate within it. Double-click `AndroidManifest.xml` to open it, and add the `uses-permission` tag highlighted in Listing 12-1. This element is required if your gadget uses any resources from the Internet; if it's entirely self-contained, you can skip this step.

Listing 12-1. *The Android Framework Application's Manifest,* `AndroidManifest.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.daylightmap.moon.android"
      android:versionCode="1"
      android:versionName="1.0.12">
    <uses-permission android:name="android.permission.INTERNET" />
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".MoonPhase"
                  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>
    <uses-sdk android:minSdkVersion="3" />
</manifest>
```

4. Open `res/layout/main.xml`, and make the changes highlighted in Listing 12-2 (note that the `WebView` element is created as a `TextView` by the project wizard, and you need to change it manually in the XML).

Listing 12-2. *The Layout XML(*`main.xml`*) Defines the Appearance of the Framework App*

```
<?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">
  <WebView
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:scrollbars="none"
      android:id="@+id/main_view" />
</LinearLayout>
```

■ **Note** In Listing 12-2, I'm assuming your gadget will fit within an average HVGA Android screen. If you need to scroll, substitute `android:scrollbars="vertical"` (or `horizontal`) for `none`.

5. Edit [your name].java (under the src directory) to match Listing 12-3; again, the additions you need to make are highlighted in bold. *Important*: use the name of your own gadget's core HTML file instead of phase_android.html.

Listing 12-3. *The Main Java Source File to Make the Framework Application Run*

```
package com.daylightmap.moon.android;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;

public class MoonPhase extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);
        WebView mainView = (WebView) findViewById(R.id.main_view);

        mainView.getSettings().setJavaScriptEnabled(true);
        mainView.addJavascriptInterface(new Boolean(true), "isAndroid");
        mainView.loadUrl("file:///android_asset/phase_android.html");
    }
}
```

■ **Tip** Remember that you don't need to type all of this code yourself—it is available for download at http://sterlingudell.com/pwg/chapter_12.

---

6. Copy your web content into [your name]/assets, including any subdirectories you may be using; see Figure 12-7 for an example. Make sure that your core HTML file name matches the name in the loadUrl call near the end of Listing 12-3 (like phase_android.html).

■ **Note** In Listing 11-3, the URL base for the /assets directory was android_asset (without an "s"). This is correct; android_asset is an SDK constant for the /assets folder of a packaged app.

---

7. Copy your icon.png file into [your name]/res/drawable, replacing any icon.png that may have been automatically placed there by the project wizard.

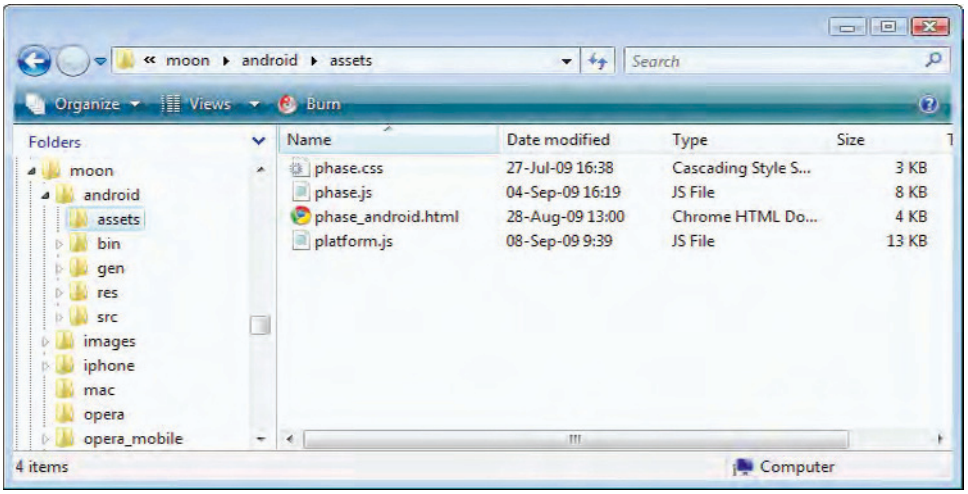

Figure 12-7. *The assets needed for Moon Phase as a native Android app*

Your Android application wrapper should now be ready to run. To run it from Eclipse, select your project folder in the Package Explorer, and then select Run → Run → Android Application or click the green-arrow button on Eclipse's toolbar.

Changes to the HTML

If you optimized a version of your gadget for Android earlier in the chapter, it's extremely likely that it will work just as well in the native-app framework—it is the same browser, after all. If there are some differences, now is the time to work them out.

As suggested earlier, for Moon Phase I've based my Android code from my existing iPhone port, and it turns out that the changes needed are few indeed. Listing 12-4 shows my HTML (with the body content omitted for brevity, as usual). The two changes from Listing 11-3 have been highlighted in bold, and I detail them after the listing.

Listing 12-4. *The HTML for Moon Phase on Android*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html;charset=UTF-8" />
    <meta name="description"
          content="Shows the current phase of the moon on a field of stars." />
    <meta name="viewport" content="width=device-width, height=device-height" />
    <title>Moon Phase</title>
    <link href="moon_32.png" type="image/png" rel="icon" />
    <link href="phase.css"   type="text/css"  rel="stylesheet" />
    <style type="text/css">
```

```
      .moonPhase #main,
      .moonPhase #config {
        font-size: 15px;
      }
      .moonPhase .text {
        width: 40%;
      }
      .moonPhase #last {
        width: 50%;
      }
      .moonPhase #moon {
        position: relative;
      }
      .moonPhase #main_icons {
        opacity: 0.5 !important;
        top: 10px;
        right: 10px;
      }
    </style>
  </head>
  <body class="moonPhase">

    <!-- body content goes here -->

    <script type="text/javascript" src="platform.js"></script>
    <script type="text/javascript" src="phase.js"></script>
    <script type="text/javascript">
      moonPhase.defaultLoad = moonPhase.load;
      moonPhase.load = function () {
        // Set the gadget's size to fill the viewport
        moonPhase.elements.main.style.height = window.innerHeight + 'px';

        // Call the ancestor method
        moonPhase.defaultLoad();

        // Center the moon image within the gadget
        moonPhase.elements.moon.style.top =
          (window.innerHeight - moonPhase.elements.moon.height) / 2 + 'px';
      };
    </script>
  </body>
</html>
```

First, Android doesn't support the additional parameters to the `meta viewport` tag that iPhone does, so I've omitted them (along with the other Apple-specific `head` elements). And second, my preference is for a slightly smaller font size on this platform, so I've dropped it from `17px` to `15px`.

That's it! With these very minor changes, Moon Phase is running well as a native Android app (see Figure 12-8), and I trust your gadget will also.

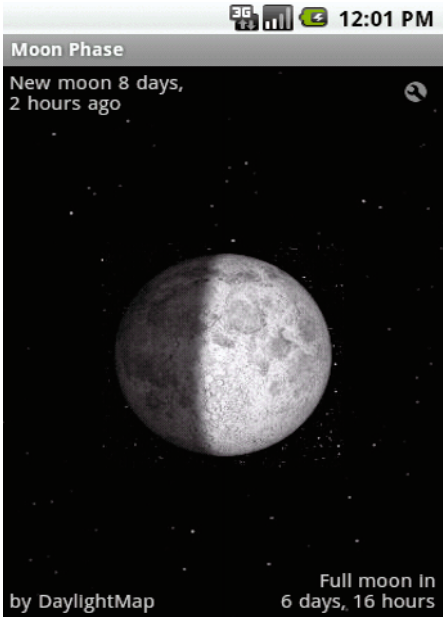

Figure 12-8. *My Moon Phase example gadget running in the Android native-app framework*

Additions to crossPlatform

You may have noticed that I ran my Android app before modifying the `crossPlatform` middleware layer to support it. That's because there are actually *no modifications required*: the Android browser will happily use the default plain-HTML handling already built into `crossPlatform` since Chapter 3. It just works.

However, in case you run into a situation where you do need (or want) to do some Android-specific handling, I have built detection of the framework into `crossPlatform` for your convenience. The JavaScript code is extremely simple:

```
if (!!window.isAndroid) {
  crossPlatform.api = 'android';
  crossPlatform.emptyPref = null;
}
```

It's powered by a technique analogous to what I used for iPhone in the previous chapter. Because the native-app framework isn't a true API, it has no namespaces that my code could detect. Instead, I "push" the `isAndroid` value through to JavaScript with the following line of Java code (also visible in Listing 12-3):

```
mainView.addJavascriptInterface(new Boolean(true), "isAndroid");
```

So if you need to do some special processing for Android in your JavaScript, simply test for `crossPlatform.api === 'android'`, the same as you would for any other API in this book.

Testing and Debugging

The testing situation for the native-application wrapper is really quite similar to what it was when running your gadget directly within the Android browser. JavaScript errors and `console.log` calls will appear in DDMS, as will any native-framework errors if your Java code has gone awry. If your gadget was working well in the browser, the DDMS log should be sufficient to maintain that functionality as a native app.

Deploying Android Applications

The primary reason for repackaging your gadget as a native application is to be able to use the conventional Android distribution channels for it, and in the following sections, I'll talk you through several of these. As mentioned earlier, the openness of the platform means that the deployment opportunities are greater than for any other mobile platform in this book.

The Android Market

Google maintains the official Android Market for the distribution of apps but manages it with a much lighter touch than Apple exerts on the iTunes App Store. There's no prequalification or approval process for applications; Google does occasionally remove apps after they have been listed, but only when they're in violation of the content guidelines (available at `http://android.com/market/terms/developer-content-policy.html`) or are repeatedly flagged as objectionable by the Android Market users themselves. Otherwise, it relies on a crowd-sourced system of ratings and comments to rank applications against each other.

On a practical level, it currently costs US$25 to create a publisher account on the Android Market; once you do so, you can list as many applications as you like. Apps may be free for users, or a purchase price may be set, with transactions processed through Google Checkout. Like Apple, Google takes 30 percent of every sale, though in this case the holdback is distributed to the cellular carriers.

This tie-in to Google Checkout is also one of the disadvantages of the market. Not every Android phone owner has a Google Checkout account, and this hurdle is enough to prevent some from buying any applications at all. Also, Google Checkout does not accept certain payment cards from non-U.S. customers, further limiting its reach.

Publishing Your App

The official guide to publishing on the Android Market is available at `http://developer.android.com/guide/publishing`, and naturally, it will have more complete and up-to-date information than I can provide here. In this section, you'll find a general overview of the publishing process, but please refer to the previous URL as well.

Before you can actually distribute your new app, you'll need to build a signed version from Eclipse. This is a security mechanism; the Android operating system won't replace apps with the same package name but different signatures, thus ensuring that your app can't be supplanted by a (possibly malicious) impostor.

■ **Note** This package/signature security matching also means that you can't directly replace the unsigned app (that you've been using until now) with a signed version. One will need to be uninstalled from your phone or emulator before the other is installed.

As on the iPhone, signing an app will require a cryptographic key, but for Android it's fine to generate one yourself—the keys are neither issued nor controlled by Google or anyone else. Instead, they're generated by the Eclipse plug-in:

1. In Eclipse, right-click the project in the Package Explorer, and select Android Tools → Export Signed Application Package.
2. Confirm that your gadget project is selected, and click Next.
3. Select "Create new keystore," and enter a suitable directory on your disk where your private key will be stored. Important: you can name the keystore whatever you like; make sure you know where to find it later!
4. Enter and confirm a password for your keystore.
5. On the Key Creation page, you'll need to make up an alias name and supply another password, as well as contact information for yourself or your organization. For the validity period, Google recommends a period of 25 years.

■ **Tip** The contact fields on the Key Creation page cannot contain punctuation, such as commas or periods.

---

6. Finally, enter a pathname for your Android package (APK), the file that you will actually distribute. I recommend putting it in your Android project directory and giving it a short but descriptive name, such as `MoonPhase.apk`.

■ **Caution** If you lose your keystore or forget your password, you'll be unable to issue updates to your application.

Keep it secret—keep it safe!

—Gandalf the Grey

Your signed APK is now ready for distribution through the Android Market (or elsewhere); the next step is to create a market account at `http://market.android.com/publish`. A Google Checkout account will be required, both to pay the registration fee and to receive payment for any application sales. Once you've done this, a Developer Console page will be created for you, from which you can upload your APK and make it available for download through the market.

■ **Tip** Your Android market listing will allow you to include screenshots of your app. These can easily be captured from the emulator or a real device with DDMS; select Device → Screen capture.

Maintaining Your Listing

When you have published one or more applications through the Android Market, you'll use the same Developer Console (at `http://market.android.com/publish`) to maintain your account. Basic download statistics will be shown, and modifications to your listings can be made; they'll take effect almost immediately.

One category of information that is not currently available on the Developer Console is user comments. These are free-form text messages that can be created by Android phone owners who download your app; they appear on your market listing page and can be a valuable source of user feedback. To view these comments, you'll either need to browse the market from an actual Android device—it's not available on the simulator—or use a third-party web site that aggregates market listings. One example of the latter is `http://androlib.com`.

When the time comes to publish an update to your application, the process is much the same as the initial release described in the previous section, with one notable exception. Each release of an app through the market must have a unique `versionCode` and `versionName` in its manifest XML (see Listing 12-1). The meaning of these two fields is as follows:

- `versionCode` is an internal number indicating the direct version progression. Usually, you'll use `1` for your first public release, `2` for your second, and so on. It will need to be unique for each update you publish to the Android Market.
- `versionName` is a more conventional version string, such as `1.0` or `2.04.17`. This will be visible to users and appended to your APK name when it's downloaded from the market; use it as you would the version number for any software. Note that `versionName` doesn't need to be numeric; a string such as `1.3 BETA` will also work fine.

Other Deployment Options

Because Android app distribution is not limited to Google's own market, a number of independent marketplaces have sprung up as well. These offer different features than the official Android Market, such as alternative versioning schemes (alpha, beta, release, and so on) and more flexible purchase terms. Also, the main market is available only on so-called Google Experience phones, which are handsets that meet certain criteria required to carry the Google logo. Devices that operate outside this program frequently replace the market with an independent app directory instead.

This third-party distribution system is still relatively new as of this writing and is certain to evolve rapidly as more non-Google handsets are released. Currently, two emerging leaders in this segment are `http://andappstore.com` and `http://mobentoo.com`. I encourage you to investigate the possibilities of distributing your repackaged gadget through these channels as well.

Android applications can also be distributed from your own web site, though this isn't quite as straightforward as with some true gadget platforms. Although all Android devices are technically able to install apps from anywhere, Google Experience handsets come with this capability disabled by default; it needs to be enabled by the user before third-party installations are allowed. From the Android home screen, select Menu → Settings → Applications, and enable Unknown Sources. If you're distributing your own Android app, you'll want to include instructions like this alongside the download link to your signed APK file.

It's also true that the APK will only be useful to visitors browsing your site directly on their Android phone; there's no good route for users to install an APK downloaded to their desk-bound computer. An alternative that has emerged is to include a link to your market listing on your page instead as a QR code (two-dimensional barcode), shown in Figure 12-9. Users who have reader software on their handset can then scan this code from their computer screen and download the app directly—even if they haven't enabled Unknown Sources.

Figure 12-9. *QR code for* `market://search?q=pname:com.daylightmap.moon.android`

To generate a code like this for yourself, the URL you'll need is `market://search?q=pname:[your package name]`, where `[your package name]` is the reverse-domain string you entered in Figure 12-6. With this URL in hand, a number of web sites or software packages will generate a QR code from it; one example is `http://qrcode.kaywa.com`. Simply save the image produced, and place it on your web page.

■ **Note** Because it allows you to leverage the Android Market's payment infrastructure, the QR code is an especially good alternative for nonfree applications.

Learning More About Android

Although Android is an open source project, its community involvement isn't as great as some. At this writing, most of the platform development is still done by Google and then released to public availability at certain milestones. As a result, most of the Android resources online are also still concentrated in Google's hands.

- The main developer portal is `http://developer.android.com`, where you'll find news, download links, programming guides, and a full API reference.
- A family of Google groups contains the majority of Android discussion online. Of most interest to gadget developers are `http://groups.google.com/group/android-discuss` and `http://groups.google.com/group/android-developers`.
- A variety of blogs cover general Android news and information, including `http://androidguys.com` and `http://androidcommunity.com`.

Summary

Google's Android platform is an up-and-coming contender in the modern smartphone segment. It's flexible, web-capable, and supported by the Open Handset Alliance, a consortium whose membership reads like a Who's Who of the mobile phone industry. For the web gadget developer, it is probably the best opportunity in the current smartphone climate, pairing excellent web standards support with a large, accessible distribution ecosystem. In this chapter, you've learned to take full advantage of that opportunity by optimizing your gadget for the Android browser and then repackaging and deploying it as a native Android application.

In the next and final chapter of the book, I'll cover even newer opportunities, with a survey of some emerging standards and technologies important to web gadgets.