2

Getting Started

All you need to start writing your own Android applications is a copy of the Android SDK and the Java development kit. Unless you're a masochist, you'll probably want a Java IDE — Eclipse is particularly well supported — to make development a little easier.

Versions of the SDK, Java, and Eclipse are available for Windows, Mac OS, and Linux, so you can explore Android from the comfort of whatever OS you favor. The SDK includes an emulator for all three OS environments, and because Android applications are run on a virtual machine, there's no advantage to developing from any particular operating system.

Android code is written using Java syntax, and the core Android libraries include most of the features from the core Java APIs. Before they can be run, though, your projects are first translated into Dalvik byte code. As a result, you get the benefits of using Java, while your applications have the advantage of running on a virtual machine optimized for Android devices.

The SDK download includes all the Android libraries, full documentation, and excellent sample applications. It also includes tools to help you write and debug your applications, like the Android Emulator to run your projects and the Dalvik Debug Monitoring Service (DDMS) to help debug them.

By the end of this chapter, you'll have downloaded the Android SDK, set up your development environment, completed two new applications, and run and debugged them using the emulator and DDMS.

If you've developed for mobile devices before, you already know that their small-form factor, limited power, and restricted memory create some unique design challenges. Even if you're new to the game, it's obvious that some of the things you can take for granted on the desktop or the Web aren't going to work on a mobile.

As well as the hardware limitations, the user environment brings its own challenges. Mobile devices are used on the move and are often a distraction rather than the focus of attention, so your applications need to be fast, responsive, and easy to use.

This chapter examines some of the best practices for writing mobile applications to help overcome the inherent hardware and environmental challenges. Rather than try to tackle the whole topic, we'll focus on using the Android SDK in a way that's consistent with good mobile design principles.

Developing for Android

The Android SDK includes all the tools and APIs you need to write compelling and powerful mobile applications. The biggest challenge with Android, as with any new development toolkit, is learning the features and limitations of its APIs.

If you have experience in Java development, you'll find that the techniques, syntax, and grammar you've been using will translate directly into Android, although some of the specific optimization techniques may seem counterintuitive.

If you don't have experience with Java but have used other object-oriented languages (such as C#), you should find the transition straightforward. The power of Android comes from its APIs, not from Java, so being unfamiliar with all the Java specific classes won't be a big disadvantage.

What You Need to Begin

Because Android applications run within the Dalvik virtual machine, you can write them on any platform that supports the developer tools. This currently includes the following:

- ❑ Microsoft Windows (XP or Vista)
- ❑ Mac OS X 10.4.8 or later (Intel chips only)
- ❑ Linux

To get started, you'll need to download and install the following:

- ❑ The Android SDK
- ❑ Java Development Kit (JDK) 5 or 6

You can download the latest JDK from Sun at

```
http://java.sun.com/javase/downloads/index.jsp
```

If you already have a JDK installed, make sure that it meets the version requirements listed above, and note that the Java runtime environment (JRE) is not sufficient.

Downloading and Installing the SDK

The Android SDK is completely open. There's no cost to download and use the API, and Google doesn't charge to allow distribution of your finished programs. You can download the latest version of the SDK for your development platform from the Android development home page at

```
http://code.google.com/android/download.html
```

Unless otherwise noted, the version of the Android SDK used for writing this book was version 1.0 r1.

The SDK is presented as a ZIP file containing the API libraries, developer tools, documentation, and several sample applications and API demos that highlight the use of particular API features. Install it by unzipping the SDK into a new folder. (Take note of this location, as you'll need it later.)

The examples and step-by-step instructions provided are targeted at developers using Eclipse with the Android Developer Tool (ADT) plug-in. Neither is required, though — you can use any text editor or Java IDE you're comfortable with and use the developer tools in the SDK to compile, test, and debug the code snippets and sample applications.

If you're planning to use them, the next sections explain how to set up Eclipse and the ADT plug-in as your Android development environment. Later in the chapter, we'll also take a closer look at the developer tools that come with the SDK, so if you'd prefer to develop without using Eclipse or the ADT plug-in, you'll particularly want to check that out.

The examples included in the SDK are well documented and are an excellent source for full, working examples of applications written for Android. Once you've finished setting up your development environment, it's worth going through them.

Developing with Eclipse

Using Eclipse with the ADT plug-in for your Android development offers some significant advantages.

Eclipse is an open source IDE (integrated development environment) particularly popular for Java development. It's available to download for each of the development platforms supported by Android (Windows, Mac OS, and Linux) from the Eclipse foundation homepage:

```
www.eclipse.org/downloads/
```

There are many variations available when selecting your Eclipse download; the following is the recommended configuration for Android:

- Eclipse 3.3, 3.4 (Ganymede)
  - Eclipse JDT plug-in
  - WST

WST and the JDT plug-in are included in most Eclipse IDE packages.

Installing Eclipse consists of uncompressing the download into a new folder. When that's done, run the `Eclipse` executable. When it starts for the first time, create a new workspace for your Android development.

Using the Eclipse Plug-in

The ADT plug-in for Eclipse simplifies your Android development by integrating the developer tools, including the emulator and .class-to-.dex converter, directly into the IDE. While you don't have to use the ADT plug-in, it does make creating, testing, and debugging your applications faster and easier.

The ADT plug-in integrates the following into Eclipse:

- An Android Project Wizard that simplifies creating new projects and includes a basic application template
- Forms-based manifest, layout, and resource editors to help create, edit, and validate your XML resources
- Automated building of Android projects, conversion to Android executables (`.dex`), packaging to package files (.apk), and installation of packages onto Dalvik virtual machines
- The Android Emulator, including control of the emulator's appearance, network connection settings, and the ability to simulate incoming calls and SMS messages
- The Dalvik Debug Monitoring Service (DDMS), which includes port forwarding; stack, heap, and thread viewing; process details; and screen capture facilities
- Access to the device or emulator's filesystem, allowing you to navigate the folder tree and transfer files
- Runtime debugging, so you can set breakpoints and view call stacks
- All Android/Dalvik log and console outputs

Figure 2-1 shows the DDMS perspective within Eclipse with the ADT plug-in installed.

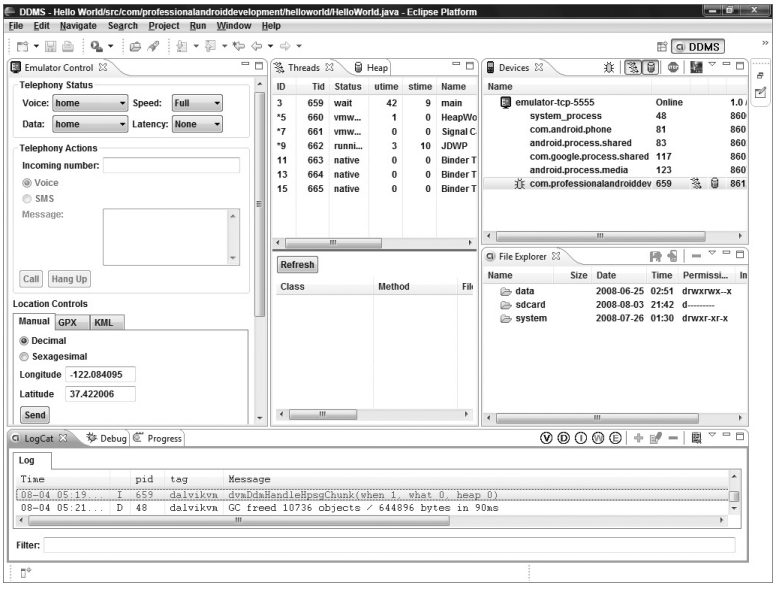

Figure 2-1

Installing the ADT Plug-in

Install the developer tools plug-in by the following steps:

1. Select **Help** ➪ **Software Updates** ➪ **Find and Install** … from within Eclipse.
2. In the resulting dialog box, choose **Search for new features to install**.
3. Select **New Remote Site**, and enter the following address into the dialog box, as shown in Figure 2-2:

```
https://dl-ssl.google.com/android/eclipse/
```

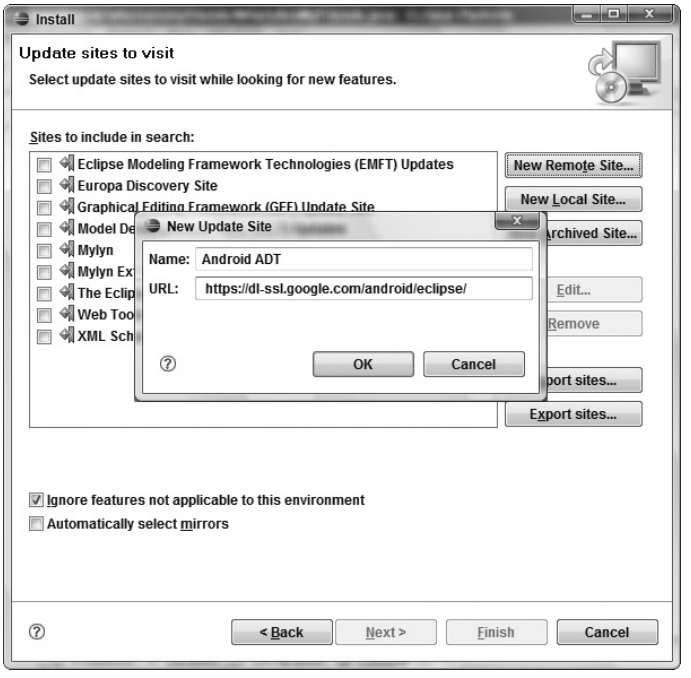

Figure 2-2

4. The new site you entered should now be checked. Click **Finish**.
5. Eclipse will now download the plug-in. When it's finished, select Android **Plugin** ➪ **Developer Tools** from the resulting Search Results dialog box, and click **Next**.
6. Read and then **Accept** the terms of the license agreement, and click **Next** and then **Finish**. As the ADT plug-in is not signed, you'll be prompted before the installation continues.
7. When complete, you'll have to restart Eclipse and update the ADT preferences. Restart and select **Window** ➪ **Preferences** … (or **Eclipse** ➪ **Preferences** for the Mac OS).
8. Then select **Android** from the left panel.
9. Click **Browse** …, and navigate to the folder into which you unzipped the Android SDK, as shown in Figure 2-3; then click **Apply** and **OK**.

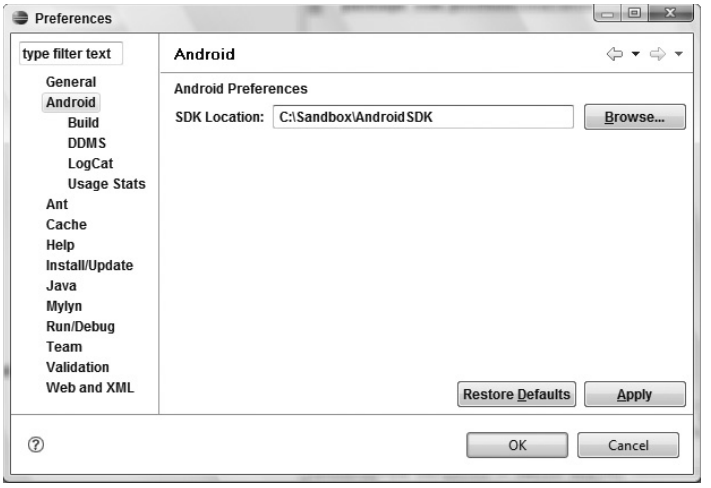

Figure 2-3

If you download a new version of the SDK and place it in a different location, you will need to update this preference to reflect the SDK with which the ADT should be building.

Updating the Plug-in

As the Android SDK matures, there are likely to be frequent updates to the ADT plug-in. In most cases, to update your plug-in, you simply:

1. Navigate to **Help ➪ Software Updates ➪ Find and Install** …
2. Select **Search for updates of the currently installed features**, and click **Finish** …
3. If there are any ADT updates available, they will be presented. Simply select them and choose **Install**.

Sometimes a plug-in upgrade is so significant that the dynamic update mechanism can't be used. In those cases, you may have to remove the previous plug-in completely before installing the newer version as described in the previous section.

Creating Your First Android Activity

You've downloaded the SDK, installed Eclipse, and plugged in the plug-in. You're now ready to start programming for Android. Start by creating a new project and setting up your Eclipse run and debug configurations.

Starting a New Android Project

To create a new Android project using the Android New Project Wizard:

1. Select **File ➪ New ➪ Project**.
2. Select the **Android Project** application type from the Android folder, and click **Finish**.

3. In the dialog that appears (shown in Figure 2-4), enter the details for your new project. The "Project name" is the name of your project file; the "Package name" specifies its package; the "Activity name" is the name of the class that is your initial Activity; and the "Application name" is the friendly name for your application.

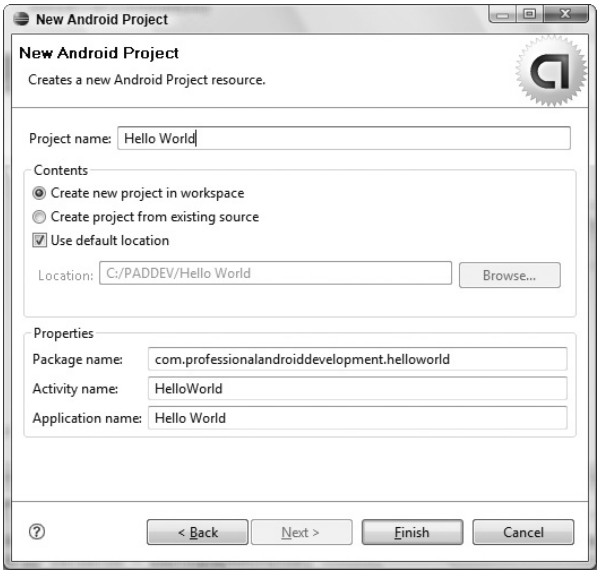

Figure 2-4

4. When you've entered the details, click **Finish**.

The ADT plug-in then creates a new project that includes a new class that extends `Activity`. Rather than being completely empty, the default template implements "Hello World." Before modifying the project, take this opportunity to configure run and debug launch configurations.

Creating a Launch Configuration

Launch configurations let you specify runtime options for running and debugging applications. Using a launch configuration you can specify the following:

- The Project and Activity to launch
- The emulator options to use
- Input/output settings (including console defaults)

You can specify different launch configurations for Run and Debug modes. The following steps show how to create a launch configuration for an Android application:

1. Select **Run** ➪ **Open Run Dialog** … (or **Run** ➪ **Open Debug Dialog** …).
2. Right-click **Android Application** on the project type list, and select **New**.

3. Enter a name for the configuration. You can create multiple configurations for each project, so create a descriptive title that will help you identify this particular setup.
4. Now choose your start-up options. The first (**Android**) tab lets you select the project and Activity that you want to start when you run (or debug) the application. Figure 2-5 shows the settings for the project you created earlier.

Figure 2-5

5. Use the **Target** tab to configure the emulator. There are options to choose the screen size, device skin, and network connection settings. You can also optionally wipe the user data on the emulator and enable or disable the start-up animation. Using the command-line textbox, you can specify additional emulator start-up options if needed.
6. Finally, set any additional properties in the **Common** tab.
7. Click **Apply**, and your launch configuration will be saved.

Running and Debugging Your Android Applications

You've created your first project and created the run and debug configurations for it. Before making any changes, test your installation and configurations by running and debugging the Hello World project.

From the **Run** menu, select **Run** or **Debug** to launch the most recently selected configuration, or select **Open Run Dialog** … or **Open Debug Dialog** … to select a configuration to use.

If you're using the ADT plug-in, running or debugging your application:

- Compiles the current project and converts it to an Android executable (`.dex`).
- Packages the executable and external resources into an Android package (.apk).
- Starts the emulator (if it's not already running).
- Installs your application onto the emulator.
- Starts your application.

If you're debugging, the Eclipse debugger will then be attached, allowing you to set breakpoints and debug your code.

If everything is working correctly, you'll see a new Activity running in the emulator, as shown in Figure 2-6.

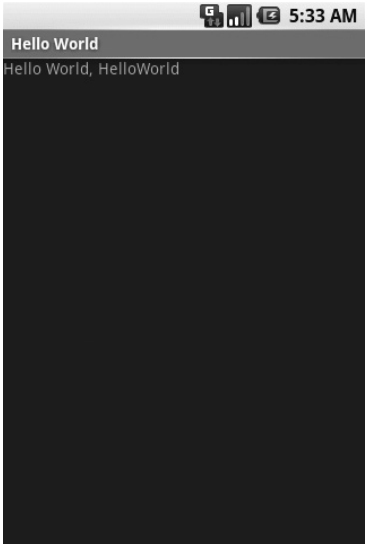

Figure 2-6

Understanding Hello World

With that confirmed, let's take a step back and have a real look at your first Android application.

`Activity` is the base class for the visual, interactive components of your application; it is roughly equivalent to a Form in traditional desktop development. The following snippet shows the skeleton code for an Activity-based class; note that it extends `Activity`, overriding the `onCreate` method.

```
package com.paad.helloworld;

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

public class HelloWorld extends Activity {

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
  }
}
```

What's missing from this template is the layout of the visual interface. In Android, visual components are called *Views*, which are similar to controls in traditional desktop development.

In the Hello World template created by the wizard, the `onCreate` method is overridden to call `setContentView`, which lays out the user interface by inflating a layout resource, as highlighted below:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.main);
}
```

The resources for an Android project are stored in the `res` folder of your project hierarchy, which includes `drawable`, `layout`, and `values` subfolders. The ADT plug-in interprets these XML resources to provide design time access to them through the `R` variable as described in Chapter 3.

The following code snippet shows the UI layout defined in the `main.xml` file created by the Android project template:

```
<?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="fill_parent"
    android:layout_height="wrap_content"
    android:text="Hello World, HelloWorld"
  />
</LinearLayout>
```

Defining your UI in XML and inflating it is the preferred way of implementing your user interfaces, as it neatly decouples your application logic from your UI design.

To get access to your UI elements in code, you add identifier attributes to them in the XML definition. You can then use the `findViewById` method to return a reference to each named item. The following XML snippet shows an ID attribute added to the TextView widget in the Hello World template:

```
<TextView
  android:id="@+id/myTextView"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="Hello World, HelloWorld"
/>
```

And the following snippet shows how to get access to it in code:

```
TextView myTextView = (TextView)findViewById(R.id.myTextView);
```

Alternatively (although it's not considered good practice), if you need to, you can create your layout directly in code as shown below:

```
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  LinearLayout.LayoutParams lp;
```

```
    lp = new LinearLayout.LayoutParams(LayoutParams.FILL_PARENT,
                                       LayoutParams.FILL_PARENT);

    LinearLayout.LayoutParams textViewLP;
    textViewLP = new LinearLayout.LayoutParams(LayoutParams.FILL_PARENT,
                                               LayoutParams.WRAP_CONTENT);

    LinearLayout ll = new LinearLayout(this);
    ll.setOrientation(LinearLayout.VERTICAL);
    TextView myTextView = new TextView(this);
    myTextView.setText("Hello World, HelloWorld");
    ll.addView(myTextView, textViewLP);
    this.addContentView(ll, lp);
}
```

All the properties available in code can be set with attributes in the XML layout. As well as allowing easier substitution of layout designs and individual UI elements, keeping the visual design decoupled from the application code helps keep the code more concise.

> *The Android web site (*`http://code.google.com/android/documentation.html`*) includes several excellent step-by-step guides that demonstrate many of the features and good practices you will be using as an Android developer. They're easy to follow and give a good idea of how Android applications fit together.*

Types of Android Applications

Most of the applications you create in Android will fall into one of the following categories:

- **Foreground Activity** An application that's only useful when it's in the foreground and is effectively suspended when it's not visible. Games and map mashups are common examples.
- **Background Service** An application with limited interaction that, apart from when being configured, spends most of its lifetime hidden. Examples of this include call screening applications or SMS auto-responders.
- **Intermittent Activity** Expects some interactivity but does most of its work in the background. Often these applications will be set up and then run silently, notifying users when appropriate. A common example would be a media player.

Complex applications are difficult to pigeonhole into a single category and include elements of all three. When creating your application, you need to consider how it's likely to be used and then design it accordingly. Let's look more closely at some of the design considerations for each application type described above.

Foreground Activities

When creating foreground applications, you need to consider the Activity life cycle (described in Chapter 3) carefully so that the Activity switches seamlessly between the foreground and the background.

Applications have no control over their life cycles, and a backgrounded application, with no Services, is a prime candidate for cleanup by Android's resource management. This means that you need to save the state of the application when the Activity becomes invisible, and present the exact same state when it returns to the foreground.

It's also particularly important for foreground Activities to present a slick and intuitive user experience.

You'll learn more about creating well-behaved and attractive foreground Activities in Chapter 3.

Background Services

These applications run silently in the background with very little user input. They often listen for messages or actions caused by the hardware, system, or other applications, rather than rely on user interaction.

It's possible to create completely invisible services, but in practice, it's better form to provide at least some sort of user control. At a minimum, you should let users confirm that the service is running and let them configure, pause, or terminate it as needed.

Services, the powerhouse of background applications, are covered in depth in Chapter 8.

Intermittent Activities

Often you'll want to create an application that reacts to user input but is still useful when it's not the active foreground Activity. These applications are generally a union of a visible controller Activity with an invisible background Service.

These applications need to be aware of their state when interacting with the user. This might mean updating the Activity UI when it's visible and sending notifications to keep the user updated when it's in the background, as seen in the section on Notifications and Services in Chapter 8.

Developing for Mobile Devices

Android does a lot to simplify mobile-device software development, but it's still important to understand the reasons behind the conventions. There are several factors to account for when writing software for mobile and embedded devices, and when developing for Android, in particular.

> *In this chapter, you'll learn some of the techniques and best practices for writing efficient Android code. In later examples, efficiency is sometimes compromised for clarity and brevity when introducing new Android concepts or functionality. In the best traditions of "Do as I say, not as I do," the examples you'll see are designed to show the simplest (or easiest-to-understand) way of doing something, not necessarily the best way of doing it.*

Hardware-Imposed Design Considerations

Small and portable, mobile devices offer exciting opportunities for software development. Their limited screen size and reduced memory, storage, and processor power are far less exciting, and instead present some unique challenges.

Compared to desktop or notebook computers, mobile devices have relatively:

- Low processing power
- Limited RAM
- Limited permanent storage capacity

- Small screens with low resolution
- Higher costs associated with data transfer
- Slower data transfer rates with higher latency
- Less reliable data connections
- Limited battery life

It's important to keep these restrictions in mind when creating new applications.

Be Efficient

Manufacturers of embedded devices, particularly *mobile* devices, value small size and long battery life over potential improvements in processor speed. For developers, that means losing the head start traditionally afforded thanks to Moore's law. The yearly performance improvements you'll see in desktop and server hardware usually translate into smaller, more power-efficient mobiles without much improvement in processor power.

In practice, this means that you always need to optimize your code so that it runs quickly and responsively, assuming that hardware improvements over the lifetime of your software are unlikely to do you any favors.

Since code efficiency is a big topic in software engineering, I'm not going to try and capture it here. This chapter covers some Android-specific efficiency tips below, but for now, just note that efficiency is particularly important for resource-constrained environments like mobile devices.

Expect Limited Capacity

Advances in flash memory and solid-state disks have led to a dramatic increase in mobile-device storage capacities (although people's MP3 collections tend to expand to fill the available space). In practice, most devices still offer relatively limited storage space for your applications. While the compiled size of your application is a consideration, more important is ensuring that your application is polite in its use of system resources.

You should carefully consider how you store your application data. To make life easier, you can use the Android databases and Content Providers to persist, reuse, and share large quantities of data, as described in Chapter 6. For smaller data storage, such as preferences or state settings, Android provides an optimized framework, as described in Chapter 6.

Of course, these mechanisms won't stop you from writing directly to the filesystem when you want or need to, but in those circumstances, always consider how you're structuring these files, and ensure that yours is an efficient solution.

Part of being polite is cleaning up after yourself. Techniques like caching are useful for limiting repetitive network lookups, but don't leave files on the filesystem or records in a database when they're no longer needed.

Design for Small Screens

The small size and portability of mobiles are a challenge for creating good interfaces, particularly when users are demanding an increasingly striking and information-rich graphical user experience.

Write your applications knowing that users will often only glance at the (small) screen. Make your applications intuitive and easy to use by reducing the number of controls and putting the most important information front and center.

Graphical controls, like the ones you'll create in Chapter 4, are an excellent way to convey dense information in an easy-to-understand way. Rather than a screen full of text with lots of buttons and text entry boxes, use colors, shapes, and graphics to display information.

If you're planning to include touch-screen support (and if you're not, you should be), you'll need to consider how touch input is going to affect your interface design. The time of the stylus has passed; now it's all about finger input, so make sure your Views are big enough to support interaction using a finger on the screen. There's more information on touch-screen interaction in Chapter 11.

Of course, mobile-phone resolutions and screen sizes are increasing, so it's smart to design for small screens, but also make sure your UIs scale.

Expect Low Speeds, High Latency

In Chapter 5, you'll learn how to use Internet resources in your applications. The ability to incorporate some of the wealth of online information in your applications is incredibly powerful.

The mobile Web unfortunately isn't as fast, reliable, or readily available as we'd often like, so when you're developing your Internet-based applications, it's best to assume that the network connection will be slow, intermittent, and expensive. With unlimited 3G data plans and city-wide Wi-Fi, this is changing, but designing for the worst case ensures that you always deliver a high-standard user experience.

This also means making sure that your applications can handle losing (or not finding) a data connection.

The Android Emulator lets you control the speed and latency of your network connection when setting up an Eclipse launch configuration. Figure 2-7 shows the emulator's network connection speed and latency set up to simulate a distinctly suboptimal EDGE connection.

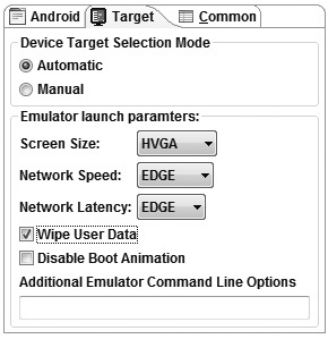

Figure 2-7

Experiment to ensure responsiveness no matter what the speed, latency, and availability of network access. You might find that in some circumstances, it's better to limit the functionality of your application or reduce network lookups to cached bursts, based on the network connection(s) available. Details

on how to detect the kind of network connections available at run time, and their speeds, are included in Chapter 10.

At What Cost?

If you're a mobile owner, you know all too well that some of the more powerful features on your mobile can literally come at a price. Services like SMS, GPS, and data transfer often incur an additional tariff from your service provider.

It's obvious why it's important that any costs associated with functionality in your applications are minimized, and that users are aware when an action they perform might result in them being charged.

It's a good approach to assume that there's a cost associated with any action involving an interaction with the outside world. Minimize interaction costs by the following:

- Transferring as little data as possible
- Caching data and GPS results to eliminate redundant or repetitive lookups
- Stopping all data transfers and GPS updates when your activity is not visible in the foreground if they're only being used to update the UI
- Keeping the refresh/update rates for data transfers (and location lookups) as low as practicable
- Scheduling big updates or transfers at "off peak" times using alarms as shown in Chapter 8

Often the best solution is to use a lower-quality option that comes at a lower cost.

When using the location-based services described in Chapter 7, you can select a location provider based on whether there is an associated cost. Within your location-based applications, consider giving users the choice of lower cost or greater accuracy.

In some circumstances, costs are hard to define, or they're different for different users. Charges for services vary between service providers and user plans. While some people will have free unlimited data transfers, others will have free SMS.

Rather than enforcing a particular technique based on which seems cheaper, consider letting your users choose. For example, when downloading data from the Internet, you could ask users if they want to use any network available or limit their transfers to only when they're connected via Wi-Fi.

Considering the Users' Environment

You can't assume that your users will think of your application as the most important feature of their phones.

Generally, a mobile is first and foremost a phone, secondly an SMS and e-mail communicator, thirdly a camera, and fourthly an MP3 player. The applications you write will most likely be in a fifth category of "useful mobile tools."

That's not a bad thing — it's in good company with others including Google Maps and the web browser. That said, each user's usage model will be different; some people will never use their mobiles

to listen to music, and some phones don't include a camera, but the multitasking principle inherent in a device as ubiquitous as it is indispensable is an important consideration for usability design.

It's also important to consider when and how your users will use your applications. People use their mobiles all the time — on the train, walking down the street, or even while driving their cars. You can't make people use their phones appropriately, but you can make sure that your applications don't distract them any more than necessary.

What does this mean in terms of software design? Make sure that your application:

- **Is well behaved** Start by ensuring that your Activities suspend when they're not in the foreground. Android triggers event handlers when your Activity is suspended or resumed so you can pause UI updates and network lookups when your application isn't visible — there's no point updating your UI if no one can see it. If you need to continue updating or processing in the background, Android provides a Service class designed to run in the background without the UI overheads.
- **Switches seamlessly from the background to the foreground** With the multitasking nature of mobile devices, it's very likely that your applications will regularly switch into and out of the background. When this happens, it's important that they "come to life" quickly and seamlessly. Android's nondeterministic process management means that if your application is in the background, there's every chance it will get killed to free up resources. This should be invisible to the user. You can ensure this by saving the application state and queuing updates so that your users don't notice a difference between restarting and resuming your application. Switching back to it should be seamless with users being shown the exact UI and application state they last saw.
- **Is polite** Your application should never steal focus or interrupt a user's current activity. Use Notifications and Toasts (detailed in Chapter 8) instead to inform or remind users that their attention is requested if your application isn't in the foreground. There are several ways for mobile devices to alert users. For example, when a call is coming in, your phone rings; when you have unread messages, the LED flashes; and when you have new voice mail, a small "mail" icon appears in your status bar. All these techniques and more are available through the notification mechanism.
- **Presents a consistent user interface** Your application is likely to be one of several in use at any time, so it's important that the UI you present is easy to use. Don't force users to interpret and relearn your application every time they load it. Using it should be simple, easy, and obvious — particularly given the limited screen space and distracting user environment.
- **Is responsive** Responsiveness is one of the most important design considerations on a mobile device. You've no doubt experienced the frustration of a "frozen" piece of software; the multifunction nature of a mobile makes it even more annoying. With possible delays due to slow and unreliable data connections, it's important that your application use worker threads and background services to keep your activities responsive and, more importantly, stop them from preventing other applications from responding in a timely manner.

Developing for Android

Nothing covered so far is specific to Android; the design considerations above are just as important when developing applications for any mobile. In addition to these general guidelines, Android has some particular considerations.

To start with, it's worth taking a few minutes to read Google's Android design philosophy at `http://code.google.com/android/toolbox/philosophy.html`.

The Android design philosophy demands that applications be:

- Fast
- Responsive
- Secure
- Seamless

Being Fast and Efficient

In a resource-constrained environment, being fast means being efficient. A lot of what you already know about writing efficient code will be just as effective in Android, but the limitations of embedded systems and the use of the Dalvik VM mean you can't take things for granted.

The smart bet for advice is to go to the source. The Android team has published some specific guidance on writing efficient code for Android, so rather than rehash their advice, I suggest you visit `http://code.google.com/android/toolbox/performance.html` and take note of their suggestions.

> *You may find that some of these performance suggestions contradict established design practices — for example, avoiding the use of internal setters and getters or preferring virtual over interface. When writing software for resource-constrained systems like embedded devices, there's often a compromise between conventional design principles and the demand for greater efficiency.*

One of the keys to writing efficient Android code is to not carry over assumptions from desktop and server environments to embedded devices.

At a time when 2 to 4 GB of memory is standard for most desktop and server rigs, even advanced smartphones are lucky to feature 32 MB of RAM. With memory such a scarce commodity, you need to take special care to use it efficiently. This means thinking about how you use the stack and heap, limiting object creation, and being aware of how variable scope affects memory use.

Being Responsive

Android takes responsiveness very seriously.

Android enforces responsiveness with the Activity Manager and Window Manager. If either service detects an unresponsive application, it will display the unambiguous Application unresponsive (AUR) message, as shown in Figure 2-8.

This alert is modal, steals focus, and won't go away until you hit a button or your application starts responding — it's pretty much the last thing you ever want to confront a user with.

Android monitors two conditions to determine responsiveness:

- An application must respond to any user action, such as a key press or screen touch, within 5 seconds.
- A Broadcast Receiver must return from its `onReceive` handler within 10 seconds.

Figure 2-8

The most likely culprits for causing unresponsiveness are network lookups, complex processing (such as calculating game moves), and file I/O. There are a number of ways to ensure that these actions don't exceed the responsiveness conditions, in particular, using services and worker threads, as shown in Chapter 8.

The AUR dialog is a last resort of usability; the generous 5-second limit is a worst-case scenario, not a benchmark to aim for. Users will notice a regular pause of anything more than half a second between key press and action. Happily, a side effect of the efficient code you're already writing will be faster, more responsive applications.

Developing Secure Applications

Android applications have direct hardware access, can be distributed independently, and are built on an open source platform featuring open communication, so it's not particularly surprising that security is a big concern.

For the most part, users will take responsibility for what applications they install and what permissions they grant them. The Android security model restricts access to certain services and functionality by forcing applications to request permission before using them. During installation, users then decide if the application should be granted the permissions requested. You can learn more about Android's security model in Chapter 11 and at `http://code.google.com/android/devel/security.html`.

This doesn't get you off the hook. You not only need to make sure your application is secure for its own sake, but you also need to ensure that it can't be hijacked to compromise the device. You can use several techniques to help maintain device security, and they'll be covered in more detail as you learn the technologies involved. In particular, you should:

- Consider requiring permissions for any services you create or broadcasts you transmit.
- Take special care when accepting input to your application from external sources such as the Internet, SMS messages, or instant messaging (IM). You can find out more about using IM and SMS for application messaging in Chapter 9.
- Be cautious when your application may expose access to lower-level hardware.

For reasons of clarity and simplicity, many of the examples in this book take a fairly relaxed approach to security. When creating your own applications, particularly ones you plan to distribute, this is an area that should not be overlooked. You can find out more about Android security in Chapter 11.

Ensuring a Seamless User Experience

The idea of a seamless user experience is an important, if somewhat nebulous, concept. What do we mean by *seamless*? The goal is a consistent user experience where applications start, stop, and transition instantly and without noticeable delays or jarring transitions.

The speed and responsiveness of a mobile device shouldn't degrade the longer it's on. Android's process management helps by acting as a silent assassin, killing background applications to free resources as required. Knowing this, your applications should always present a consistent interface, regardless of whether they're being restarted or resumed.

With an Android device typically running several third-party applications written by different developers, it's particularly important that these applications interact seamlessly.

Use a consistent and intuitive approach to usability. You can still create applications that are revolutionary and unfamiliar, but even they should integrate cleanly with the wider Android environment.

Persist data between sessions, and suspend tasks that use processor cycles, network bandwidth, or battery life when the application isn't visible. If your application has processes that need to continue running while your activity is out of sight, use a Service, but hide these implementation decisions from your users.

When your application is brought back to the front, or restarted, it should seamlessly return to its last visible state. As far as your users are concerned, each application should be sitting silently ready to be used but just out of sight.

You should also follow the best-practice guidelines for using Notifications and use generic UI elements and themes to maintain consistency between applications.

There are many other techniques you can use to ensure a seamless user experience, and you'll be introduced to some of them as you discover more of the possibilities available in Android in the coming chapters.

To-Do List Example

In this example, you'll be creating a new Android application from scratch. This simple example creates a new to-do list application using native Android View controls. It's designed to illustrate the basic steps involved in starting a new project.

> *Don't worry if you don't understand everything that happens in this example. Some of the features used to create this application, including ArrayAdapters, ListViews, and KeyListeners, won't be introduced properly until later chapters, where they're explained in detail. You'll also return to this example later to add new functionality as you learn more about Android.*

1. Start by creating a new Android project. Within Eclipse, select **File ➪ New ➪ Project** …, then choose **Android** (as shown in Figure 2-9) before clicking **Next**.

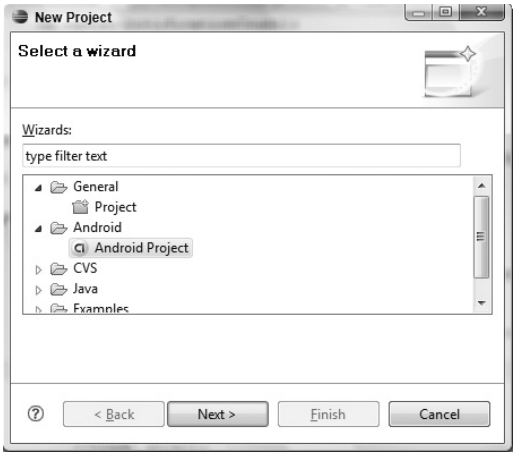

Figure 2-9

2. In the dialog box that appears (shown in Figure 2-10), enter the details for your new project. The "Application name" is the friendly name of your application, and the "Activity name" is the name of your Activity subclass. With the details entered, click **Finish** to create your new project.

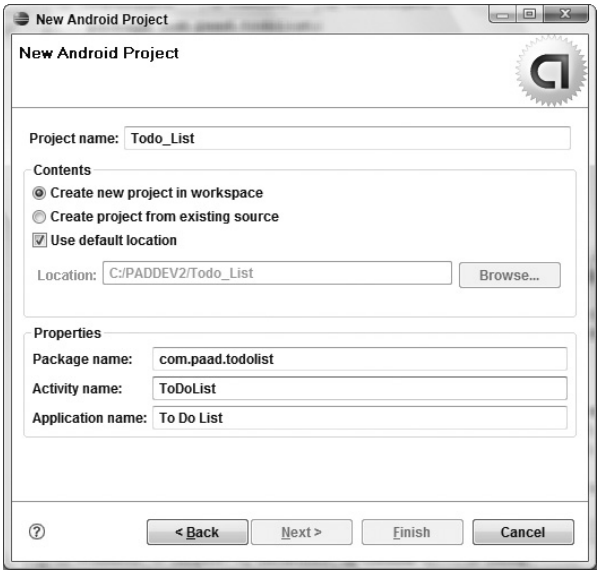

Figure 2-10

3. Take this opportunity to set up debug and run configurations by selecting **Run ➪ Open Debug Dialog** … and then **Run ➪ Open Run Dialog** …, creating a new configuration for each, specifying the `Todo_List` project. You can leave the launch actions as **Launch Default Activity** or explicitly set them to launch the new `ToDoList` Activity, as shown in Figure 2-11.

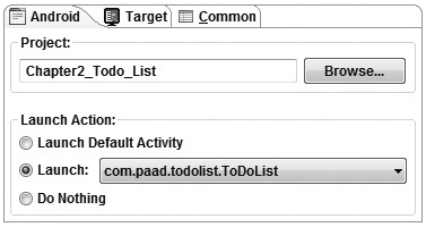

Figure 2-11

4. Now decide what you want to show the users and what actions they'll need to perform. Design a user interface that will make this as intuitive as possible.

In this example, we want to present users with a list of to-do items and a text entry box to add new ones. There's both a list and a text entry control (View) available from the Android libraries. You'll learn more about the Views available in Android and how to create new ones in Chapter 4.

The preferred method for laying out your UI is using a layout resource file. Open the main.xml layout file in the res/layout project folder, as shown in Figure 2-12.

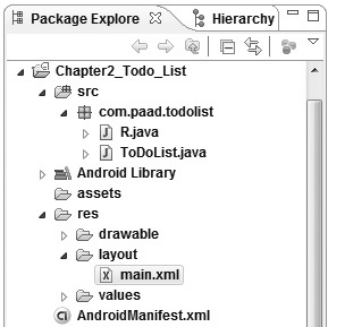

Figure 2-12

5. Modify the main layout to include a `ListView` and an `EditText` within a `LinearLayout`. It's important to give both the `EditText` and `ListView` controls IDs so you can get references to them in code.

```
<?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">
  <EditText
    android:id="@+id/myEditText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="New To Do Item"
  />
  <ListView
```

```
    android:id="@+id/myListView"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```

6. With your user interface defined, open the `ToDoList.java` Activity class from your project's source folder. In this example, you'll make all your changes by overriding the `onCreate` method. Start by inflating your UI using `setContentView` and then get references to the `ListView` and `EditText` using `findViewById`.

```
public void onCreate(Bundle icicle) {
  // Inflate your view
  setContentView(R.layout.main);

  // Get references to UI widgets
  ListView myListView = (ListView)findViewById(R.id.myListView);
  final EditText myEditText = (EditText)findViewById(R.id.myEditText);
}
```

7. Still within `onCreate`, define an `ArrayList` of Strings to store each to-do list item. You can bind a `ListView` to an `ArrayList` using an `ArrayAdapter`, so create a new `ArrayAdapter` instance to bind the to-do item array to the `ListView`. We'll return to `ArrayAdapters` in Chapter 5.

```
public void onCreate(Bundle icicle) {
  setContentView(R.layout.main);

  ListView myListView = (ListView)findViewById(R.id.myListView);
  final EditText myEditText = (EditText)findViewById(R.id.myEditText);

  // Create the array list of to do items
  final ArrayList<String> todoItems = new ArrayList<String>();
  // Create the array adapter to bind the array to the listview
  final ArrayAdapter<String> aa;
  aa = new ArrayAdapter<String>(this,
                                android.R.layout.simple_list_item_1,
                                todoItems);
  // Bind the array adapter to the listview.
  myListView.setAdapter(aa);
}
```

8. The final step to make this to-do list functional is to let users add new to-do items. Add an `onKeyListener` to the `EditText` that listens for a "D-pad center button" click before adding the contents of the `EditText` to the to-do list array and notifying the `ArrayAdapter` of the change. Then clear the `EditText` to prepare for another item.

```
public void onCreate(Bundle icicle) {
  setContentView(R.layout.main);

  ListView myListView = (ListView)findViewById(R.id.myListView);
  final EditText myEditText = (EditText)findViewById(R.id.myEditText);

  final ArrayList<String> todoItems = new ArrayList<String>();
  final ArrayAdapter<String> aa;
  aa = new ArrayAdapter<String>(this,
```

```
                                        android.R.layout.simple_list_item_1,
                                        todoItems);
        myListView.setAdapter(aa);

        myEditText.setOnKeyListener(new OnKeyListener() {
          public boolean onKey(View v, int keyCode, KeyEvent event) {
            if (event.getAction() == KeyEvent.ACTION_DOWN)
              if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER)
              {
                todoItems.add(0, myEditText.getText().toString());
                aa.notifyDataSetChanged();
                myEditText.setText("");
                return true;
              }
            return false;
          }
        });
      }
```

9. Run or debug the application, and you'll see a text entry box above a list, as shown in Figure 2-13.

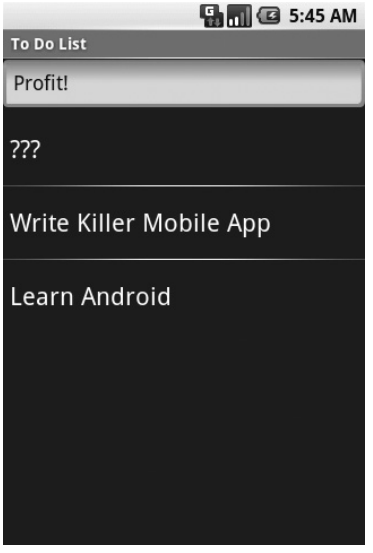

Figure 2-13

10. You've now finished your first "real" Android application. Try adding breakpoints to the code to test the debugger and experiment with the DDMS perspective.

As it stands, this to-do list application isn't spectacularly useful. It doesn't save to-do list items between sessions, you can't edit or remove an item from the list, and typical task list items like due dates and task priority aren't recorded or displayed. On balance, it fails most of the criteria laid out so far for a good mobile application design.

You'll rectify some of these deficiencies when you return to this example in later chapters.

Android Development Tools

The Android SDK includes several tools and utilities to help you create, test, and debug your projects. A detailed examination of each developer tool is outside the scope of this book, but it's worth briefly reviewing what's available. For more detail than is included here, check out the Android documentation at:

```
http://code.google.com/android/intro/tools.html
```

As mentioned earlier, the ADT plug-in conveniently incorporates most of these tools into the Eclipse IDE, where you can access them from the DDMS perspective, including:

- **The Android Emulator** An implementation of the Android virtual machine designed to run on your development computer. You can use the emulator to test and debug your android applications.
- **Dalvik Debug Monitoring Service (DDMS)** Use the DDMS perspective to monitor and control the Dalvik virtual machines on which you're debugging your applications.
- **Android Asset Packaging Tool (AAPT)** Constructs the distributable Android package files (.apk).
- **Android Debug Bridge (ADB)** The *ADB* is a client-server application that provides a link to a running emulator. It lets you copy files, install compiled application packages (.apk), and run shell commands.

The following additional tools are also available:

- **SQLite3** A database tool that you can use to access the SQLite database files created and used by Android
- **Traceview** Graphical analysis tool for viewing the trace logs from your Android application
- **MkSDCard** Creates an SDCard disk image that can be used by the emulator to simulate an external storage card.
- **dx** Converts Java `.class` bytecode into Android `.dex` bytecode.
- **activityCreator** Script that builds Ant build files that you can then use to compile your Android applications without the ADT plug-in

Let's take a look at some of the more important tools in more detail.

The Android Emulator

The emulator is the perfect tool for testing and debugging your applications, particularly if you don't have a real device (or don't want to risk it) for experimentation.

The emulator is an implementation of the Dalvik virtual machine, making it as valid a platform for running Android applications as any Android phone. Because it's decoupled from any particular hardware, it's an excellent baseline to use for testing your applications.

A number of alternative user interfaces are available to represent different hardware configurations, each with different screen sizes, resolutions, orientations, and hardware features to simulate a variety of mobile device types.

Full network connectivity is provided along with the ability to tweak the Internet connection speed and latency while debugging your applications. You can also simulate placing and receiving voice calls and SMS messages.

The ADT plug-in integrates the emulator into Eclipse so that it's launched automatically when you run or debug your projects. If you aren't using the plug-in or want to use the emulator outside of Eclipse, you can telnet into the emulator and control it from its console. For more details on controlling the emulator, check the documentation at `http://code.google.com/android/reference/emulator.html`.

> *At this stage, the emulator doesn't implement all the mobile hardware features supported by Android, including the camera, vibration, LEDs, actual phone calls, the accelerometer, USB connections, Bluetooth, audio capture, battery charge level, and SD card insertion/ejection.*

Dalvik Debug Monitor Service (DDMS)

The emulator lets you see how your application will look, behave, and interact, but to really see what's happening under the surface, you need the DDMS. The Dalvik Debug Monitoring Service is a powerful debugging tool that lets you interrogate active processes, view the stack and heap, watch and pause active threads, and explore the filesystem of any active emulator.

The DDMS perspective in Eclipse also provides simplified access to screen captures of the emulator and the logs generated by LogCat.

If you're using the ADT plug-in, the DDMS is fully integrated into Eclipse and is available from the DDMS perspective. If you aren't using the plug-in or Eclipse, you can run DDMS from the command line, and it will automatically connect to any emulator that's running.

The Android Debug Bridge (ADB)

The *Android debug bridge (ADB)* is a client-service application that lets you connect with an Android Emulator or device. It's made up of three components: a daemon running on the emulator, a service that runs on your development hardware, and client applications (like the DDMS) that communicate with the daemon through the service.

As a communications conduit between your development hardware and the Android device/emulator, the ADB lets you install applications, push and pull files, and run shell commands on the target device. Using the device shell, you can change logging settings, and query or modify SQLite databases available on the device.

The ADT tool automates and simplifies a lot of the usual interaction with the ADB, including application installation and update, log files, and file transfer (through the DDMS perspective).

To learn more about what you can do with the ADB, check out the documentation at `http://code.google.com/android/reference/adb.html`.

Summary

This chapter showed you how to download and install the Android SDK; create a development environment using Eclipse on Windows, Mac OS, or Linux platforms; and how to create run and debug configurations for your projects. You learned how to install and use the ADT plug-in to simplify creating new projects and streamline your development cycle.

You were introduced to some of the design considerations for developing mobile applications, particularly the importance of optimizing for speed and efficiency when increasing battery life and shrinking sizes are higher priorities than increasing processor power.

As with any mobile development, there are considerations when designing for small screens and mobile data connections that can be slow, costly, and unreliable.

After creating an Android to-do list application, you were introduced to the Android Emulator and the developer tools you'll use to test and debug your applications.

Specifically in this chapter, you:

- ❑ Downloaded and installed the Android SDK.
- ❑ Set up a development environment in Eclipse and downloaded and installed the ADT plug-in.
- ❑ Created your first application and learned how it works.
- ❑ Set up run and debug launch configurations for your projects.
- ❑ Learned about the different types of Android applications.
- ❑ Were introduced to some mobile-device design considerations and learned some specific Android design practices.
- ❑ Created a to-do list application.
- ❑ Were introduced to the Android Emulator and the developer tools.

The next chapter focuses on Activities and application design. You'll see how to define application settings using the Android manifest and how to externalize your UI layouts and application resources. You'll also find out more about the Android application life cycle and Android application states.