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 effi cient Android code. In later examples, effi ciency 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 Effi cient

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

Expect Limited Capacity

Advances in fl ash 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 fi ll 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 fi lesystem when you want or need to, but in those circumstances, always consider how you're structuring these fi les, and ensure that yours is an effi cient solution.

Part of being polite is cleaning up after yourself. Techniques like caching are useful for limiting repetitive network lookups, but don't leave fi les on the fi lesystem 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 fi nger input, so make sure your Views are big enough to support interaction using a fi nger 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.

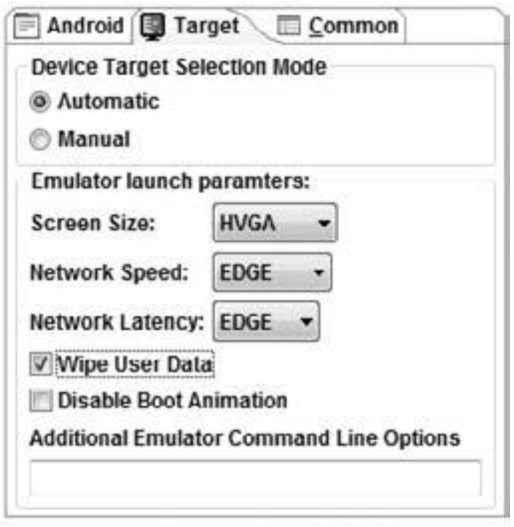
Expect Low Speeds, High Latency

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 fi nding) a data connection.

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


Android | Target | Common

Device Target Selection Mode
Automatic [✓]
Manual

Emulator launch paramters:
Screen Size: HVGA
Network Speed: EDGE
Network Latency: EDGE
[✓] Wipe User Data
[ ] Disable Boot Animation
Additional Emulator Command Line Options

Figure 2-7

Experiment to ensure responsiveness no matter what the speed, latency, and availability of network access. You might fi nd 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 defi ne, 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 fi rst 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 fi fth 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 Notifi cations 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 fl ashes; 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 notifi - cation 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 specifi c 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 Effi cient

In a resource-constrained environment, being fast means being effi cient. A lot of what you already know about writing effi cient 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 specifi c guidance on writing effi cient 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 fi nd 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 effi ciency.

One of the keys to writing effi cient 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 effi ciently. 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

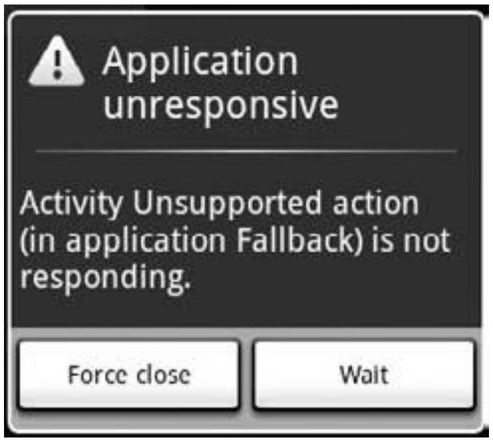
detects an unresponsive application, it will display the unambiguous Application unresponsive (AUR) message, as shown in Figure 2-8.

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.


Application unresponsive

Activity Unsupported action (in application Fallback) is not responding.

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 effi cient 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 fi nd 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 fi nd 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 Notifi cations 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 E xample

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 Array Adapters, 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**.

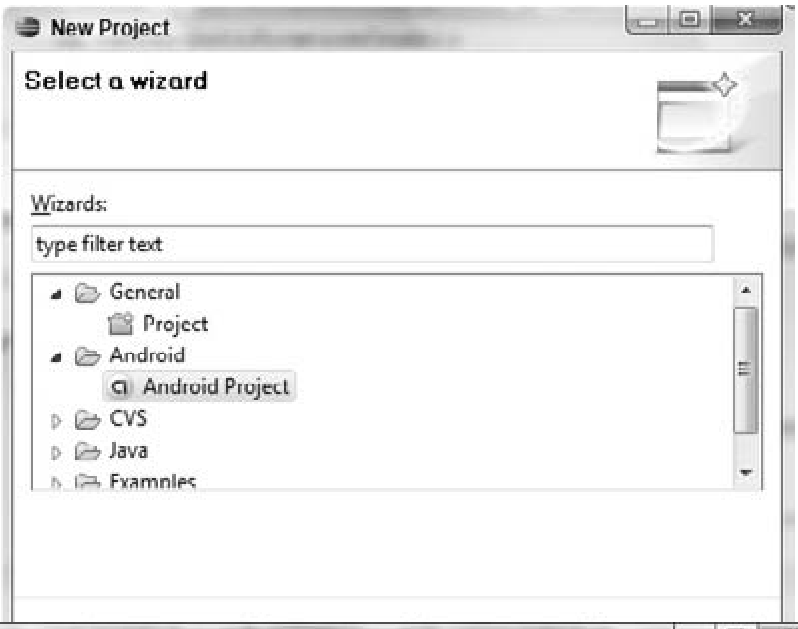
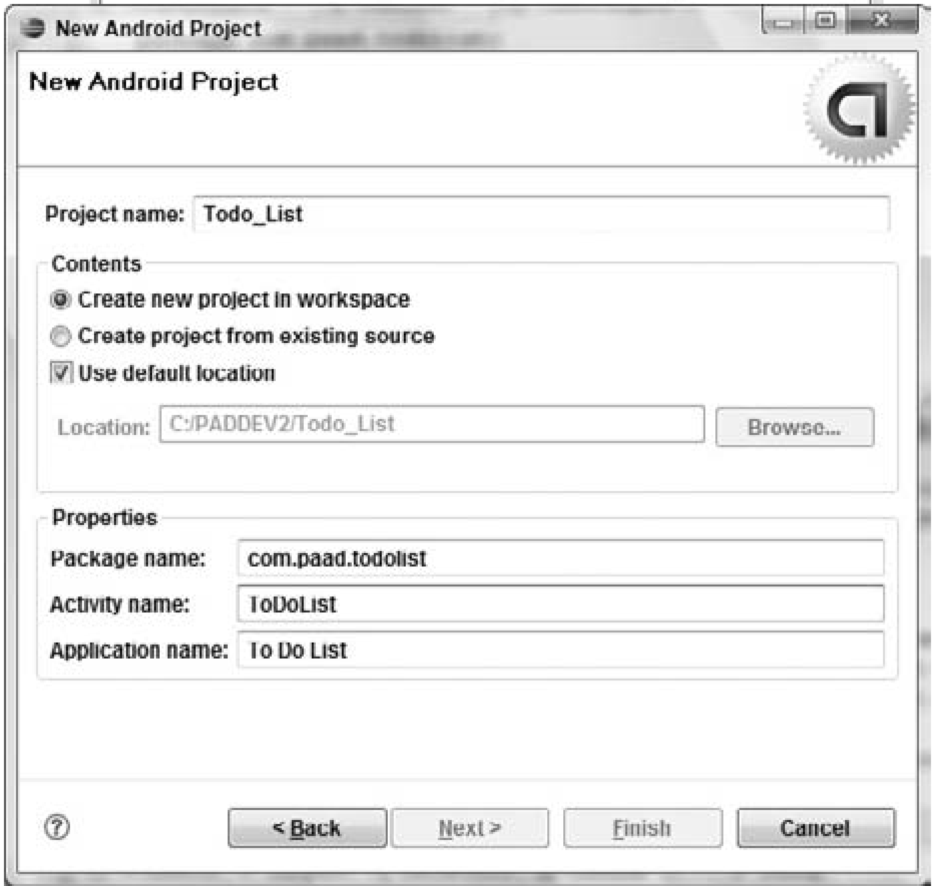


Figure 2-9

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.


New Android Project

Project name: Todo_List

Contents
- Create new project in workspace (selected)
- Create project from existing source
- [✓] Use default location

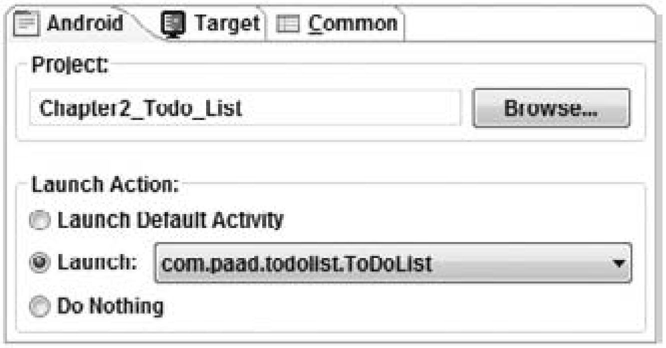
Location: C:/PADDEV2/Todo_List

Properties

Package name:	com.paad.todolist
Activity name:	ToDoList
Application name:	To Do List

Figure 2-10

3. Take this opportunity to set up debug and run confi gurations by selecting **Run ➪ Open Debug Dialog** … and then **Run ➪ Open Run Dialog** …, creating a new confi guration 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.



Project: Chapter2_Todo_List

Launch Action:
- Launch Default Activity
- [✓ Launch: com.paad.todolist.ToDoList]
- Do Nothing

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 fi le. Open the main.xml layout file in the res/layout project folder, as shown in Figure 2-12.

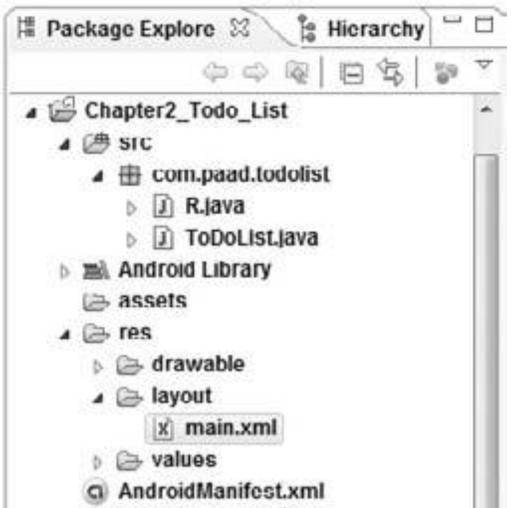


Figure 2-12

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 defi ned, 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 infl ating your UI using setContentView and then get references to the ListView and EditText using findViewById.

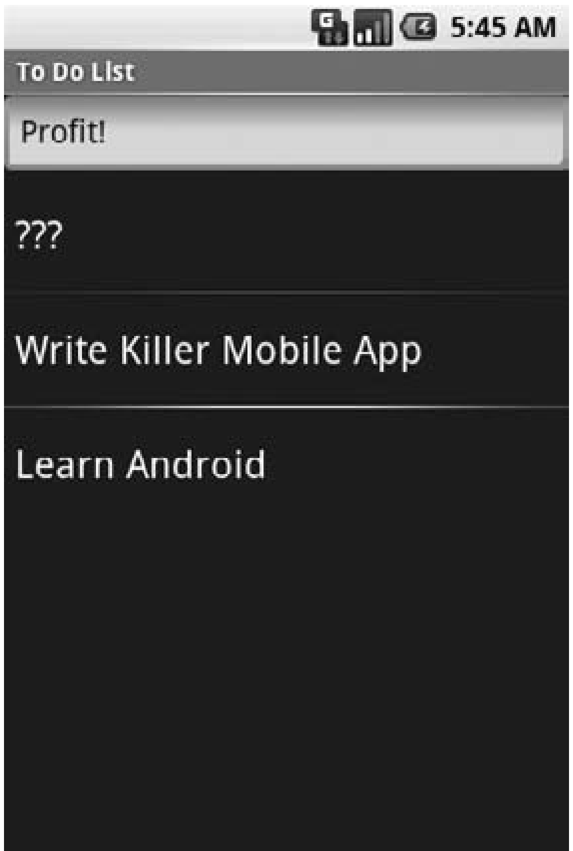
```
public void onCreate(Bundle icicle) {
// Inflate your view
setContentView(R.layout.main);
// Get references to UI widgets
ListView myListView = (ListView)findViewById(R.id.myListView);
final EditText myEditText = (EditText)findViewById(R.id.myEditText);
}
```

7. Still within onCreate, defi ne 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 fi nal 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 fi nished your fi rst "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 defi ciencies when you return to this example in later chapters.