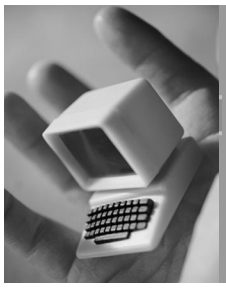

CHAPTER 3

Programming Strategies for Small Devices

The limitations that are inherent in small devices require you to change the way in which you code your applications. Before we move on to discussing specific Java implementations that are aimed at small devices, let's take some time to develop some general programming strategies for small devices.

If in Doubt, Do Not Use Java

The first strategy to consider is simple, if perhaps a bit heretical. You should avoid using Java until you are sure that it meets your application requirements—not only in features, but in performance. This statement might seem obvious, but becoming blinded by all of the hype that accompanies Java and the enthusiasm that it generates is easy. A programmer who codes in Java on desktop or server systems is understandably reluctant to abandon Java when moving to smaller devices, but this reluctance is detrimental to the project.

The reality is that Java on small devices is still an immature technology with a lot of room to improve and evolve, as we will see later in this book when we discuss the individual specifications and implementations associated with the Java 2 Micro Edition (J2ME). Possibly, the Micro Edition meets your needs today, or perhaps it does not. There may also be new licensing issues to deal with if you need to include a Java runtime environment with your application.

If you absolutely want to use Java, start with a simple, non-critical project. Writing your mission-critical software in Java is not the way to experiment with a device's Java support. You should instead bite the bullet and write your software by using C/C++.

We are not saying that Java is not a viable programming language. Rather, the technology is so new that not all devices will support it—and device limitations on central processing unit (CPU) speed and memory capacity might not make it possible to write Java applications that run (or run with acceptable performance) within those limits. For example, the initial beta releases of the KVM—which is a small Java interpreter that we will discuss later—is a key part of the Micro Edition and could not deal with programs larger than 64K when running on Palm devices. This situation made it difficult to write serious applications that could run in that environment.

Move Computation to the Server

Once you have decided to take the plunge with Java, the next strategy is almost as simple: avoid running computationally intensive tasks on the device. Instead, let a server computer run them for you. The alternative is to tie up the device (potentially even making its user interface unresponsive) for several seconds or minutes—delays that your users will find unacceptable. In many ways, this process is similar to deploying a thin-client Web application where most of the logic is in the Web server, leaving the Web browser to handle the user interface.

Finding the right balance between what to do with the device and what to do on the server is tricky and depends on both the application and the device's connectivity. Obviously, a device that has a wireless radio can connect to a server more often than a device that has only cradle-based communication. But the cradle-based communication is faster

and essentially free, while wireless communication can be slow and expensive. Therefore, even if you can connect to the server on an as-needed basis, it might be time-prohibitive or cost-prohibitive to download large amounts of data.

Letting the server do some of the work does not have to be complicated. Even simple things that are done on the server can make a big difference in your application's responsiveness. For example, rather than downloading data from the server and sorting it on the device, let the server sort it for you. The download time will not improve, but you will save the time that your application spent sorting the data after the download. Later, after changes have been made to the data, the application can resort the data—and you will benefit because the data will be in near-sorted condition already.

Of course, in many cases you have no choice but to partition the application between the device and a server. Often, there is just too much data to put on the device itself, or else the data is too sensitive to leave on an unsecured machine.

Simplify the Application

After moving as much as you can to the server, the next step is to simplify the application. This step is best done during application design, of course, and as usual, some work and prototyping ahead of the actual coding will save you time and work later.

The most obvious simplification for your application is to remove unnecessary features. Consider each feature of your application carefully. Is this feature really needed, or could users get by without it? Removing code is the simplest way to reduce the size of an application.

If you need a feature only occasionally, consider moving it and similar features into a second, auxiliary application. Users can then remove the auxiliary application if they do not need those features.

Internationalized programs are often large because of all of their resources—text strings, bitmaps, and other locale-sensitive data—which are required to support various national, linguistic, and cultural scenarios. For desktop programming, it is common to ship a single version of the application that can handle the different locales transparently.

Performing this task in Java is quite easy because of its support for resource bundles, formatters, and other locale-sensitive classes. The resources are often located and loaded dynamically, based on the current locale. As an optimization, the application installer often installs the files for a single locale, because this procedure cuts down on the disk space that the application requires. While a desktop system's total storage capacity is large, it is not infinite. To save space, then, you will want to use a similar strategy and build separate versions of your application for each locale, instead of relying on automatic locale detection and adaptation.

After removing unnecessary features, the next step is to reuse the user interface wherever possible. A large portion of an interactive application is spent dealing with the user interface. Look for opportunities to reuse parts of the user interface whenever possible. Not only does this reuse make the application smaller, but it makes it easier for the user to learn the application.

Finally, try to provide a single-action path for each feature. Use simple, consistent, and unique ways to invoke particular features. This approach is less confusing for your users, and there is less code for you to write. Also, this approach forces you to look carefully at your overall user-interface design. You should always group the commonly used features so that they are quickly accessible by users with as few button/key presses or pen strokes as possible.

Do not forget, though, that you will have different kinds of users. As you simplify your application, be sure to leave in the shortcuts and features that your power users will appreciate. If you think that this advice seems contradictory, you are right. Everything is a delicate balancing act. You cannot remove everything from your application, after all. It is really just a matter of fine-tuning its features for smaller devices.

Build Smaller Applications

Another step that you can take is building smaller applications. A smaller application takes up less memory on the device and requires less time to install. This type of application will usually require a shorter startup time as well, especially with a language such as Java

where a significant part of the startup time is spent verifying and otherwise preparing the individual classes that an application requires. Smaller applications are also cheaper to install if you happen to be downloading it onto the device via a wireless network.

For installation purposes, you want to package your Java applications as compressed Java Archive (JAR) files whenever possible. This technique is the most obvious way to build a smaller application, but it does increase the load time for the application because of the time that is required to decompress the individual class files. You might have no choice, however. A JAR file might be the only packaging that is acceptable to the runtime environment. (If startup time is an issue, try building an uncompressed JAR file and determine whether that helps. It might be worth the extra cost in terms of memory footprint and installation/download time.)

To further reduce the size of your application, consider using a tool called an *obfuscator*. An obfuscator makes the code and the symbolic information in your class files harder to read by converting identifiers to short, undistinguished character sequences and by performing other tricks that still result in a legal class file. A side effect of the obfuscation is a reduction in the size of the final class file. Most obfuscators will also remove unnecessary and unused methods and classes, which of course leads to further savings in class-file size. You will find some freely available obfuscators on the book's companion Web site.

Remove the Public Members

As a rule, obfuscators will not remove public members (fields or methods) of a class, because other classes might call those members. If you think about it, the only members that are safe to remove are private members. If you are building a complete application that safely stands by itself, however, you can usually tell the obfuscator to consider the classes that your application uses to be a closed set of classes. You can also tell it to treat any public, package, or protected members as if no one outside the closed set will refer to them. This technique enables the obfuscator to remove more code than it would normally be capable of removing. Still, your best bet is to make as many members as you can private. The fewer public members you have, the easier it is to optimize the code.

Of course, reducing the number of classes that your application uses is really the simplest way to reduce its memory footprint. Remember that every class or interface that you create generates a separate class file, including nested, inner, or (especially) anonymous classes.

Use Less Memory at Run Time

As we have discussed, the runtime memory capacity of a small computing device can be quite limited. Sometimes these limits are not obvious, however. For example, the Palm operating system (OS) defines two kinds of memory: dynamic and storage. Dynamic memory stores the application's runtime data—in particular, its stack and its runtime memory heap. Storage memory is write-protected, persistent memory. The amount of dynamic memory available to an application varies from 32K to 256K, and the remainder of the device's RAM is storage memory. Even if a device has 8MB of RAM, the 256K limit on dynamic memory is important. If more memory is required, the application must use storage memory, which is slower to access due to the write protection.

What follows are some simple tips and examples of how to reduce the amount of runtime memory that your Java applications use.

Use Scalar Types

Each object that you use must be allocated from the runtime memory heap. There is no way to declare objects that are allocated on the stack. The object's constructor runs as part of that allocation process. Therefore, each object that you allocate impacts your application's performance as well as the amount of memory that it requires. To reduce the number of objects that are allocated, consider using scalar types—the non-object types such as `int` and `boolean`—in place of objects whenever possible.

Consider the methods of `java.awt.Component` as an example. These methods define two variants of the `setSize` method:

```
public void setSize( int width, int height );
public void setSize( Dimension size );
```

The first variant takes only scalar types, while the second one requires you to allocate a `Dimension` object. When you call the first variant, you generate a bit more code—but you avoid an object allocation. Besides, the second variant is defined as follows:

```
public void setSize( Dimension size ){
    setSize( size.width, size.height );
}
```

Although a call to the second variant is a bit cheaper in terms of bytecode generation, you end up executing more code in addition to incurring the object-allocation expense.

Do Not Depend on the Garbage Collector

As we saw in the previous chapter, the garbage collector is an important part of the execution engine. Without it, Java programmers would have to explicitly free allocated objects, which adds complexity to programs and requires the use of object-management schemes such as reference counting. The garbage collector frees you from having to worry about who is using an object and about determining if the memory can be reclaimed by the system.

Although a garbage collector is handy, it does not excuse you entirely from the memory-management process. If you allocate too many objects too quickly, the garbage collector might have trouble keeping up and collecting unreferenced objects. Your application might pause at inopportune times as the garbage collector works to catch up. Also, as the peak memory usage increases, the execution engine will grab more and more memory from the system in order to allocate objects. When the garbage collector finally catches up, that memory is rarely (if ever) returned to the system, even if most of it is not required by the application anymore. This memory is now effectively unavailable for use by other applications.

Help the Garbage Collector

Be sure to help the garbage collector do its work by setting object references to `null` whenever you are finished with them. For example,

you might define a `deinitialize` method for a class in order to explicitly clear out its members:

```
public class MyClass {
    private Object someObject;

    public MyClass( Object obj ){
        someObject = obj;
    }

    public void deinitialize(){
        someObject = null;
    }
}
```

By clearing out object references, you make it easier for the garbage collector to find and reclaim unreferenced objects. This technique is simple and does not add too much code to your classes.

Another way to help the garbage collector is to use weak references, which premiered in Java 2. Weak references are instances of special system classes that reference objects that you would like to keep active and in memory but that can be garbage collected by the system if necessary in order to free some memory. Weak references are not usually supported on small devices, however, so you should not depend on their presence.

Watch out for garbage-collection bugs. Sun's garbage collector in Java 1.1, for example, prematurely frees singleton objects (objects that are supposed to be created only once and that are stored in a static class member) if the singletons are not referenced from an active thread. The workaround in this case is to start a thread that does nothing except keep a reference to the singleton.

Use Lazy Instantiation

Another technique to reduce overall and peak memory usage is to only allocate objects as they are needed. We usually refer to this technique as lazy instantiation. In this approach, you check for a null object reference (remember that Java guarantees that the members of a class are always initialized to a default value, which makes it easy to perform this kind of check):

```
public class LazyClass {
    private Vector v; // defaults to null

    public Vector getVector(){
        if( v == null ) v = new Vector();
        return v;
    }
}
```

If you are using multiple threads, however, be sure to use the double check technique described later to ensure that only one object ever gets created.

Release Resources Early

Whether you are writing for small devices or for desktop systems, it makes sense to release resources—database connections, network connections, files, and so on—as soon as possible. Do not hang onto them longer than necessary. Not only does this procedure free the resource for use by another application, but it also enables the system to free any memory associated with that resource.

In particular, do not depend on finalizers to free resources. Finalizers might never run, and as we will see later, they are not even supported by some Java interpreters. Always provide methods for explicitly freeing the resources, and document their use. If your Java platform does support finalizers, however, it is still a good idea to define finalizers for each resource-using class just to ensure that the resources are really freed (in case the programmer forgot to release them).

Reuse Objects

One technique that can pay good dividends is to reuse objects instead of continually allocating and abandoning them. Basically, what you want to do is provide methods to initialize and deinitialize an object and provide a way to cache unused objects. Consider the following class, for example:

```
public class ObjHolder
{
    private Object _value;
```

```
    public ObjHolder( Object value )
    {
        _value = value;
    }

    public Object getValue()
    {
        return _value;
    }
}
```

This class is trivial but is enough of an example for our purposes. First, we modify it by adding initialization and deinitialization methods:

```
    private void initialize( Object value )
    {
        _value = value;
    }

    private void deinitialize()
    {
        _value = null;
    }
```

Then, we modify the constructor by making it private and by having it call the `initialize` method:

```
    private ObjHolder( Object value )
    {
        initialize( value );
    }
```

We could also choose to keep the constructor public or simply define a no-argument version that does not call the `initialize` method. Because we have chosen to make the constructor private, we have to provide a static function to create one for us. This static function will also keep a cache of objects for us, which for simplicity, we will limit to a single object at a time:

```
    private static ObjHolder _cache; // single object cache

    public static synchronized ObjHolder allocate( Object value )
    {
        ObjHolder holder = _cache;
        if( holder == null ){
            holder = new ObjHolder( value );
        } else {
```

```
            holder.initialize( value );
        }
        _cache = null;
        return holder;
    }
```

Also, of course, we will define an equivalent function to place objects back into the cache:

```
    public static synchronized void deallocate( ObjHolder holder )
    {
        holder.deinitialize();
        if( _cache == null ){
            _cache = holder;
        }
    }
```

Objects of this class are now allocated and freed by using this sequence:

```
ObjHolder holder = ObjHolder.allocate( somevalue );

..... // use it

ObjHolder.deallocate( holder ); // free it!
```

Be careful with the size of your cache. If it is too large, then your class will be holding references to objects whose memory could otherwise be reclaimed. The size of the cache depends on the application, and you might not be able to determine an appropriate size except by running the application and gathering some statistics.

If your cache needs to be thread-safe, as in the previous example, you will also pay a time penalty because of the synchronization primitives that you need to use. Therefore, if you can guarantee that a single thread will use the cache, you will be better off and will be able to avoid the synchronization overhead.

Avoid Exceptions

Java's built-in support for exception handling is extremely convenient. Exceptions are sometimes overused, however. In general, you want to reserve exceptions for unusual or unexpected (exceptional) situations.

Errors that are expected to occur in the normal course of running an application should be handled through other means. By avoiding exceptions, you can reduce the size of the class files and also reduce the number of objects that are allocated (because each exception throws an exception object).

Code with Performance in Mind

With a small device, performance is critical. Always write your code with performance in mind. Here are a few suggestions to explore. You can also consult any book on good coding practices for more tips, because a lot of these books are independent of any particular programming language. Also, do not forget to use the Java compiler's optimization options for producing tighter code.

Use Local Variables

It is generally slower to access class members than to access local variables. If you are using the same class member over and over, such as within a loop, it might make sense to assign the value to a temporary variable stored on the stack and to use that temporary variable in place of the class member. Again, be careful when dealing with data that is shared by multiple threads. This optimization might not make sense.

Using local variables also makes sense when dealing with arrays. Each time an array element is accessed, the Java interpreter performs a bounds check in order to ensure that the array index is valid. If you access the same array element more than once, store it in a local variable and access it from there instead. For example, instead of:

```
Char[] buf = ....; // get an array somehow

for( int i = 0; i < buf.length; ++i ){
    if( buf[i] >= '0' && buf[i] <= '9' ){
        ....
    } else if( buf[i] == '\r' || buf[i] == '\n' ){
        ....
    }
}
```

Use a local variable to reduce the number of references to `buf[i]`:

```
for( int i = 0; i < buf.length; ++i ){
    Char ch = buf[i];
    if( ch >= '0' && ch <= '9' ){
        ....
    } else if( ch == '\r' || ch == '\n' ){
        ....
    }
}
```

Avoid String Concatenation

Java makes it easy to build strings by concatenation. In other words, it is natural to do this kind of coding:

```
public String indent( String line, int spaces ){
    String out = "";
    for( int i = 0; i < spaces; ++i ){
        out += " ";
    }
    return out;
}
```

From a performance viewpoint, however, this coding is extremely poor. String concatenation involves creating a new `StringBuffer` object, calling its `append` method, and then calling its `toString` method in order to obtain the final string. Concatenation inside a loop (as shown earlier) can lead to the creation of many short-lived `String` and `StringBuffer` objects, which not only affects performance but can also increase the application's peak memory usage. The better solution is to do most of the work yourself, as in this example:

```
public String indent( String line, int spaces ){
    StringBuffer out = new StringBuffer();
    for( int i = 0; i < spaces; ++i ){
        out.append( ' ' );
    }
    return out.toString();
}
```

This simple change can drastically reduce the number of objects that are created.

Use Threads, but Avoid Synchronization

Threads are an important part of Java, and your application should take advantage of them whenever possible. The usual rule is that any operation that will take more than a tenth of a second to run should run on a separate thread so that it will not block the user interface. User interface responsiveness is extremely important on a small device—even more so than on a desktop system—because users are less forgiving. Also, the instant-on capability of the device is one of its important features.

If you do use threads, though, you have to control access to shared data. You can perform this task with the `synchronized` keyword, but there is added overhead (quite significant overhead in most cases) when obtaining a lock on the object in question. Sometimes it is hard to avoid this overhead. Many of the common data structures, such as `java.util.Vector` and `java.util.Hashtable`, use synchronized methods to be thread-safe. If you can ensure that only a single thread at a time will ever access the data in a class, then you can do away with synchronization. You can perform this task by writing your own versions of these classes or by using the unsynchronized Java 2 equivalents like `ArrayList` or `HashMap` if your platform supports them. You can gain a bit of performance that way.

To use lazy instantiation in a multithreaded situation, you must be sure to perform a double check as follows:

```
private Vector v = null;

public Vector getVector() {
    if( v == null ){ // first check
        synchronized( this ){
            if( v == null ){ // second check
                v = new Vector();
            }
        }
    }

    return v;
}
```

Because the `Vector` class is already thread-safe, there is really no need to make `getVector` a synchronized method. If the vector has already been created, we just return it and let it deal with any synchro-

nization issues. This action is what the first check for `null` performs. The only thing we have to worry about is the creation of the vector. If the first check fails, we immediately synchronize on an object that is handy (typically, the object's `this` reference for instance methods or else the `Class` object for static methods), then perform a second check. If two or more threads make it past the first check, only the first one will fail the second check and create the object. The other threads will pass the second check and return the newly created object.

Separate the Model

When you write an application for small devices, you have to adapt your user interface to each device's form factor. A common technique for dealing with this problem is to separate the logic of the application from the code that controls the presentation—a technique formally referred to as the model-view-controller (MVC) technique.

An Introduction to MVC

The MVC technique factors an application into three different parts: a model, a view, and a controller. The model is the data held by the application and the code that deals specifically with the data. In a text editor, for example, the model is the text being edited. The view is a representation of the model, typically drawn on a screen but not necessarily limited to display output. Although there is only one model, there can be more than one view—and each view can represent the model in different ways. A spreadsheet, for example, can display data by using a grid-based tabular format or as a chart. Each is a different view of the same basic data. The controller interprets external input, usually from a user, to modify the model or the view. In a three-dimensional rendering application, for example, the controller can interpret the user's mouse movements in order to rotate a rendered image back and forth.

MVC originates from the Smalltalk language and has been the subject of numerous academic discussion papers. One of the criticisms leveled against it is that it is too hard to separate the view from the controller, because the input and the output are too closely linked in most systems. This concern led to the development of a simplified MVC that combines the view and the controller into a presentation, keeping only

the model separate. Instead of having multiple views, an application has multiple presentations. This simplified form of MVC is often referred to as user-interface delegation. (If the concept seems familiar to you, it is because the Swing user-interface classes make extensive use of user-interface delegation.)

Why Separate the Model?

Separating models and presentations takes some effort. Why go through the trouble, especially if your application only uses a single view/presentation?

As we saw in the first chapter, small devices come in a variety of different form factors and support a wide variety of different output and input methods. If you separate the model and the presentation, you will find it easier to adapt—or possibly rewrite—your user interface code to work on different devices. It is not necessarily a given that you will need to rework the user interface, but it is certainly a possibility. Separating the model means that you will not have to rewrite the entire application.

How to Build a Model

You build a model by creating a separate class to hold the data. That class then exposes methods that the other parts of the application (the presentation) use to access and modify the data. The class also triggers events to notify presentations whenever the external state—the data that the presentations can see and use—of the model changes.

You should note that the model must not directly expose any data members; otherwise, it cannot track changes to the data. In other words, do not define a class in this way:

```
public class Model {
    public int x;
    public int y;
}
```

Instead, define accessor methods that can then trigger events to broadcast changes to the data:

```
public class Model {
    private int x;
    private int y;

    public int getX() { return x; }
    public int getY() { return y; }

    public void setX( int x ){
        this.x = x;
        notifyListeners(); // broadcast the change
    }

    public void setY( int y ){
        this.y = y;
        notifyListeners(); // broadcast the change
    }

    // other methods for listener registration, notification, etc.
}
```

Presentations are notified of changes by using whichever event model is convenient. Normally, you would use an event listener model similar to the Abstract Windowing Toolkit (AWT) event model introduced in Java 1.1. A listener interface defines the methods that the model invokes in order to trigger an event:

```
public interface ModelListener {
    void modelChanged( Model m );
}
```

Any presentation that is interested in receiving updates from the model then implements the interface and registers itself with the model:

```
public class Presentation implements ModelListener {
    public Presentation( Model model ){
        model.addListener( this );
    }

    public void modelChanged( Model m ){
        // do something in response to the event
    }
}
```

The model defines a public registration method and a private notification method:

```
public class Model {

    ..... // other methods previously discussed

    private Vector listeners = new Vector();

    public addListener( ModelListener l ){
        listeners.addElement( l );
    }

    private void notifyListeners(){
        Enumeration e = listeners.elements();
        while( e.hasMoreElements() ){
            ((ModelListener) e.nextElement()).modelChanged( this );
        }
    }
}
```

Whenever the model changes, it calls its private notification method(s). There might be several different events that can be triggered, depending on how you design the listener. This method(s) notifies the presentations, which will usually update their own states by using the new data.

Think carefully about multithreading and performance issues when designing your event model, and apply all of the techniques that we have discussed so far to make sure that the event dispatching does not become a bottleneck in your application's performance.

The Tic-Tac-Toe Example

Later in Part 3 of this book, we will build a simple tic-tac-toe game by using different Java implementations. Tic-tac-toe is a simple grid-based game where two players take turns filling in cells with their own color or character (usually X and O characters). The first player to completely fill a row (horizontal, vertical, or diagonal) with his or her color/character wins. If all of the cells are filled before either player wins, the game ends in a tie.

Tic-tac-toe is really a children's game, because the game almost always ends in a tie if played by two adults. The game is complicated enough, however, to serve as a useful example of small-device Java programming. Tic-tac-toe also enables us to demonstrate a real-world, model-presentation, separation scenario. The game logic is encapsulated into

a single class (the `TicTacToeModel` class),which we will reuse in each version of the application. What follows are highlights of the tic-tac-toe model, which you will need to understand in order to follow the game as we write it. Refer to Appendix A for a complete listing of the class. You will also find it on the CD-ROM that accompanies this book.

First, we define the constructor for our class:

```
public TicTacToeModel(int size){
newGame (sized)}
```

In the spirit of generalization, we will define the tic-tac-toe model by using a variable grid size. The minimum value that we will accept is 3, and the maximum is 6. This minimum and maximum will be controlled by the `newGame` method that we will define later.

The model uses an array in order to represent the cells:

```
private byte[] cells;
```

This array is allocated at run time by `newGame`. Notice that we avoid using a two-dimensional array and thereby avoid the extra overhead of managing the array of arrays and accessing the individual elements. Each cell in the grid is mapped to an element in the `cells` array, starting at the top of the grid and going left to right and top to bottom. Thus, in a 3-by-3 grid (nine cells total), the top-left cell has index 0; the top-right cell has index 2; the bottom-left cell has index 6; and the bottom-right cell has index 8.

What goes in each cell? The answer is a value that indicates whether the cell is empty or filled by one of the two players. We therefore define some constants in order to indicate each state:

```
public static final int NO_PLAYER = 20;
public static final int PLAYER_1 = 1;
public static final int PLAYER_2 = -1;
```

Why define the values this way? By summing the values in a sequence of cells (which can be six cells at most), we can quickly determine whether there are any empty cells in the sequence or whether one of the players has completely filled each cell in the sequence. Take a grid of size 4 as an example. Player 1 wins the game if any sequence sums to 4 (`4 * PLAYER_1`), while Player 2 wins if any sequence sums to -4

(4 * PLAYER_2). Also, if the sum is greater than 4, then the sequence contains at least one empty cell.

The sums are themselves stored in an array called `totals`, which is also allocated by `newGame`:

```
private int[] totals;
```

The size of the `totals` array is always $2*n+2$, where n is the game size (because there are n horizontal rows, n vertical rows, and two diagonal rows in any game). By storing the sums, we only need to recalculate them every time a cell value is changed (by using the `calculateTotals` method):

```
private void calculateTotals(){
    int index = 0;
    for( int i = 0; i < numTotals; ++i ){
        int total = 0;
        for( int j = 0; j < cellsPerRow; ++j ){
            total += cells[ rowIndices[ index++ ] ];
        }
        totals[i] = total;
    }
}
```

The `numTotals` and `cellsPerRow` fields are defined as part of the `newGame` initialization. The initialization also creates a one-dimensional array called `rowIndices` that stores the cell indices for each of the $2*n+2$ rows:

```
public void newGame(){
    newGame( 3 );
}

public void newGame( int size ){
    if( size < 3 || size > 6 ){
        size = 3;


    endGame();

    cellsPerRow = size;
    numCells = size * size;
    numTotals = 2 * size + 2;
    player1Wins = size * PLAYER_1;
    player2Wins = size * PLAYER_2;
```

```
    if( cells == null || cells.length < numCells ){
        cells = new byte[ numCells ];
    }

    int i;

    for( i = 0; i < numCells; ++i ){
        cells[i] = NO_PLAYER;
    }

    if( totals == null || totals.length < numTotals ){
        totals = new int[ numTotals ];
    }

    int initial = cellsPerRow * NO_PLAYER;

    for( i = 0; i < numTotals; ++i ){
        totals[i] = initial;
    }

    calculateIndices();
    gameStarted = true;
    notifyGameStarted();
}
```

The actual cell indices are calculated by `calculateIndices`, and we will leave the details of this process for Appendix A. `newGame` also defines `numCells`, `player1Wins`, and `player2Wins` and sets `gameStarted` to true in order to indicate that a game is in progress.

Of course, `calculateTotals` will never be called unless we provide methods to set the value of a cell:

```
public void setCellState( int row, int col, int value ){
    setCellState( row * cellsPerRow + col, value );
}

public void setCellState( int index, int value ){
    if( !gameStarted || index < 0 || index >= numCells ) return;
    if( cells[index] != NO_PLAYER ) return;
    cells[index] = (byte) value
    calculateTotals();
    notifyGameUpdated( index, value );
    if( isGameOver() ){
        endGame();
    }
}
```

For convenience, we enable cell values to be set by row and column or by index. Only empty cells can have their values set. After a cell is set, we notify any listeners about the new value and check to see whether the game is over. Listeners implement the `TicTacToeModelListener` interface, which defines these three methods:

```
void gameStarted( TicTacToeModel model );
void gameUpdated( TicTacToeModel model, int index, int value );
void gameOver( TicTacToeModel model );
```

The model class itself defines methods for notifying the listeners, as with the `notifyGameOver` method:

```
private void notifyGameOver(){
    Enumeration e = listeners.elements();
    while( e.hasMoreElements() ){
        ((TicTacToeModelListener) e.nextElement()).notifyGameOver( this
);
    }
}
```

Also, of course, `TicTacToeModel` defines an `addListener` method just as we did previously, except with the additional feature that if a game has already started, the newly registered listener is immediately notified of this fact via the `gameStarted` method.

Finally, the model class defines methods such as `isGameOver`, `getCellState`, `getWinner`, and `getWinningCells` that listeners and other interested parties can use to determine the status of a game.

Optimizing Event Notification

The `TicTacToeModel` class notifies listeners in a straight-forward fashion by building an enumeration of registered listeners and cycling through that enumeration. This process works well for our situation, because not many events are triggered. If you are dealing with events that are triggered frequently, however, you might want to change the notification code to use something other than an enumeration and to perform other optimizations based on the number of registered listeners. As always, be sure to keep multithreaded programs in mind when performing these optimizations; otherwise, an event that is triggered while another thread is adding itself as a listener could cause an exception (because the set of listeners is in flux).

The complexity of this model might surprise you. Perhaps tic-tac-toe is not quite as simple as you thought. You will see the benefits of the model when we write the actual game in Part 3.

Chapter Summary

In this chapter, we looked at some general strategies for small-device programming, which finishes this first part of the book. The next part introduces us to the J2ME and the specifications that define this application.