Taming Java Threads

ALLEN HOLUB

Apress™

Taming Java Threads
Copyright ©2000 by Allen I. Holub

ISBN (pbk): 1-893115-10-0

Printed and bound in the United States of America 12345678910

Copy Editor: Tom Robinson, TSI Graphics

Artist: Warren Fischbach, TSI Graphics

Compositor and Pretty Good Vorpal Player: Susan Glinert

Indexer: Nancy Guenther

Project Manager: Grace Wong

Cover and Interior Design: Derek Yee Design

Distributed to the book trade worldwide by Springer-Verlag New York, Inc. 175 Fifth Avenue, New York, NY 10010

In the United States, phone 1-800-SPRINGER; orders@springer-ny.com

www.springer-ny.com

For information on translations, please contact Apress directly: Apress, 901 Grayson Street, Suite 204, Berkeley, CA 94710

Phone: 510-549-5930, Fax: 510-549-5939, info@apress.com, www.apress.com

CHAPTER 10

If I Were King: Fixing Java's Threading Problems

IN A WAY, THE PREVIOUS CHAPTERS are a litany of everything wrong with the Java threading model and a set of Band-Aid solutions to those problems. I call the classes Band-Aids because the problems addressed by the classes should really be part of the syntax of the Java language. Using syntactic methods rather than libraries can give you better code in the long run since the compiler and JVM working together can perform optimizations that would be difficult or impossible with a library approach.

In this chapter, I want to approach the threading problem in a more positive light by suggesting a few changes to Java that would provide solutions to those problems. These proposals are very tentative—they are just one person's thoughts on the matter, and they would need a lot of work and peer review before being viable. But they're a start. The proposals are also rather bold. Several people have suggested subtle, and minimal, changes to the Java-Language Specification (JLS) to fix currently ambiguous JVM behavior, but I want more sweeping improvement.

On a practical note, many of my proposals involve the introduction of new keywords to the language. Though the usual requirement that you don't want to break existing code is certainly valid, if the language is not to stagnate and thus become obsolete, it must be possible to introduce keywords. In order to introduce keywords that won't conflict with existing identifiers, I've deliberately used a character (`$`) which is illegal in an identifier. (For example, `$task` rather than `task`). A compiler command-line switch could perhaps enable variants on these keywords that would omit the dollar sign.

The Task

The fundamental problem with Java's threading model is the fact that it is not in the least bit object oriented. A thread is effectively nothing but a procedure [`run()`] which calls other procedures. Notions of objects, asynchronous versus synchronous messages, and the like, are simply not addressed.

One solution to this problem is the `Active_object` class presented in Chapter 9, but a better solution would be to modify the language itself to support asynchronous messaging directly. The asynchronous messages running on an Active Object are effectively synchronous with respect to each other. Consequently, you can eliminate much of the synchronization hassles required to program in a more procedural model by using an Active Object.

My first proposal, then, is to incorporate Active Objects into the language itself by incorporating the notion of a *task* into Java. A task has a built-in Active-Object dispatcher, and takes care of all the mechanics of handling asynchronous messages automatically. You would define a task exactly as you would a class, except that the `asynchronous` modifier could be applied to methods of the task to indicate that those methods should execute in the background on the Active-Object dispatcher. To see the parallels with the class-based approach discussed in Chapter 9, consider the following file I/O class, which uses my `Active_object` to implement an asynchronous write operation:

```
interface Exception_handler
{   void handle_exception( Throwable e );
}

class File_io_task
{   Active_object dispatcher = new Active_object();

    final OutputStream      file;
    final Exception_handler handler;

    File_io_task( String file_name, Exception_handler handler )
                                            throws IOException
    {   file = new FileOutputStream( file_name );
        this.handler = handler;
    }

    public void write( final byte[] bytes )
    {
        dispatcher.dispatch
        (   new Runnable()
            {   public void run()
                {
                    try
                    {   byte[] copy new byte[ bytes.length ];
                        System.arrayCopy(   bytes,  0,
                                            copy,   0,
                                            bytes.length );
                        file.write( copy );
                    }
                    catch( Throwable problem )
                    {   handler.handle_exception( problem );
                    }
                }
            }
        );
    }
}
```

All write requests are queued up on the Active-Object's input queue with a `dispatch()` call. Any exceptions that occur while processing the asynchronous message in the background are handled by the `Exception_handler` object that's passed into the `File_io_task`'s constructor. You would write to the file like this:

```
File_io_task io =   new File_io_task
                    ( "foo.txt"
                        new Exception_handler
                        {   public void handle( Throwable e )
                            {   e.printStackTrace();
                            }
                        }
                    );
//...
io.write( some_bytes );
```

Introducing the `$task` and `$asynchronous` keywords to the language lets you rewrite the previous code as follows:

```
$task File_io $error{ $.printStackTrace(); }
{
    OutputStream file;

    File_io( String file_name ) throws IOException
    {   file = new FileOutputStream( file_name );
    }

    asynchronous public write( byte[] bytes )
    {   file.write( copy );
    }
}
```

Note that asynchronous methods don't specify return values because the handler returns immediately, long before the requested operation completes. Consequently, there is no reasonable value that could be returned. The `$task` keyword should work exactly like `class` with respect to the derivation model: A `$task` could implement interfaces and extend classes and other tasks. Methods marked with the `asynchronous` keyword are handled by the `$task` in the background. Other methods would work synchronously, just as they do in classes.

The `$task` keyword can be modified with an optional `$error` clause (as shown), which specifies a default handler for any exceptions that are not caught by the asynchronous methods themselves. I've used `$` to represent the thrown exception object. If no `$error` clause is specified, then a reasonable error message (and probably stack trace) is printed.

Note that the arguments to the asynchronous method must be immutable to be thread safe. The run-time system should take care of any semantics required to guarantee this immutability. (A simple copy is often not sufficient.) All task objects would have to support a few pseudo-messages as well:

`some_task.close()`	Any asynchronous messages sent after this call is issued would throw a `TaskClosedException`. Messages waiting on the Active-Object queue will be serviced, however.
`some_task.join()`	The caller blocks until the task is closed and all pending requests are processed.

In addition to the usual modifiers (such as `public`), the `task` keyword would accept the `$pooled(n)` modifier, which would cause the `task` to use a thread pool rather than a single thread to run the asynchronous requests. The `n` specifies the desired pool size; the pool can grow if necessary, but should deflate to the original size when the additional threads aren't required. The server-side socket handler that I used as an example of thread-pool usage in Chapter 8 could be rewritten in the proposed syntax as follows:

```
abstract $pooled(10) $task Socket_server
{
    public asynchronous listen(ServerSocket server, Client_handler client)
    {   while(true)
        {   client.handle( server.accept() );
        }
    }

    interface Client_handler
    {   asynchronous void handle( Socket s );
    }
}

//...

Socket_server listener = new Socket_server();

listener.listen (   new ServerSocket(some_port),
                    new Socket_server.Client_handler()
                    { public asynchronous void handle( Socket s )
                      { // client-handling code goes here.
                      }
                    }
                  );
```

The arguments to `listen()` can just as easily be constructor arguments, of course.

Improvements to `synchronized`

Though a `$task` eliminates the need for synchronization in many situations, all multithreaded systems cannot be implemented solely in terms of tasks. Consequently, the existing threading model needs to be updated as well. The `synchronized` keyword has several flaws:

1. You cannot specify a timeout value.
2. You cannot interrupt a thread that is waiting to acquire a lock.
3. You cannot safely acquire multiple locks. (Multiple locks should always be acquired in the same order.)

You can solve these problems by extending the syntax of `synchronized` both to support a list of multiple arguments and to accept a timeout specification (specified in brackets, next): Here's the syntax that I'd like:

`synchronized(x && y && z)`	Acquire the locks on the `x`, `y`, `and z` objects.
`synchronized(x || y || z)`	Acquire the locks on the `x`, `y`, `or z` objects.
`synchronized( (x && y ) || z)`	Obvious extension of the previous code.
`synchronized (. . .) [1000]`	Acquire designated locks with a one-second timeout.
`synchronized [1000] f() {. . .}`	Acquire the lock for `this`, but with a one-second timeout.

A timeout is necessary, but not sufficient, for making the code robust. You also need to be able to terminate the wait to acquire the lock externally. Consequently, the `interrupt()` method, when sent to a thread that is waiting to acquire a lock, should break the waiting thread out of the acquisition block by tossing a `SynchronizationException` object. This exception should be a `RuntimeException` derivative so that the it would not have to be handled explicitly.

The main problem with these proposed modifications to the `synchronized` syntax is that they would require changes at the byte-code level, which currently implements `synchronized` using enter-monitor and exit-monitor instructions. These instructions take no arguments, so the byte-code definition would have to be extended to support the acquisition of multiple locks. This change is no more serious than the changes added to the JVM in Java 2, however, and would be backward compatible with existing Java code.

Improvements to wait() and notify()

The `wait()/notify()` system also has problems:

1. There is no way to detect whether `wait()` has returned normally or because of a timeout.

2. There is no way to implement a traditional condition variable that remains in a "signaled" state.

3. Nested-monitor lockout can happen too easily

The timeout-detection problem is easily solved by redefining `wait()` to return a `boolean` (rather than `void`). A `true` return value would indicate a normal return, `false` would indicate a timeout.

The notion of a state-based condition variable is an important one: The variable can be set to a `false` state in which waiting threads will block until the variable entered a `true` state, and any thread that waits on a `true` condition variable is released immediately. [The `wait()` call won't block in this case.] You can support this functionality by extending the syntax of `notify()` as follows:

`notify();`	Waiting threads without changing the state of the underlying condition variable.
`notify(true);`	Set the condition variable's state to `true` and release any waiting threads. Subsequent calls to `wait()` won't block.
`notify(false);`	Set the condition variable's state to `false` (subsequent calls to `wait()` or `wait(true)` will block.

The nested-monitor-lockout problem is thornier, and I don't have an easy solution. One possible solution is for `wait()` to release *all* the locks that the current thread has acquired in the opposite order of acquisition and then reacquire them in the original acquisition order when when the wait is satisfied. I can imagine that code that leveraged this behavior would be almost impossible for a human being to figure out, however, so I don't think that this is really a viable solution. If anybody has any ideas, send me email (*aih@holub.com*).

I'd also like to be able to wait for complex conditions to become true. For example:

```
[a && [b || c]].wait();
```

where `a`, `b`, and `c` are any `Object`. I've used brackets rather than parentheses to make it easier for the compiler to distinguish this sort of expression from an arithmetic expression, but I can think of alternative syntaxes that would serve the same purpose.

Fixing the Thread Class

The ability to support both preemptive and cooperative threads is essential in some server applications, especially if you intend to squeeze maximum performance out of the system. I suggest that Java has gone too far in simplifying the threading model, and that the Posix/Solaris notion of a "green thread" and a "lightweight process" that I discussed in Chapter 1 should be supported by Java. This means, of course, that some JVM implementations (such as NT implementations) would have to simulate cooperative threads internally, and other JVMs would have to simulate preemptive threading, but adding these extensions to the JVM is reasonably easy to do.

A Java thread, then, should always be preemptive. That is, a Java thread should work much like a Solaris lightweight process. The `Runnable` interface can be used to define a a Solaris-like green thread that must explicitly yield control to other green threads running on the same lightweight process. For example, the current syntax of:

```
class My_thread implements Runnable
{   public void run(){ /*...*/ }
}

new Thread( new My_thread );
```

would effectively create a green thread for the `Runnable` object and bind that green thread to the lightweight process represented by the `Thread` object. This change in implementation is transparent to existing code since the effective behavior is the same as at present.

By thinking of `Runnable` objects as green threads, you can expand Java's existing syntax to support multiple green threads bound to a single lightweight processes simply by passing several `Runnable` objects to the `Thread` constructor. (The green threads would be cooperative with respect to each other, but could be preempted by other green threads [`Runnable` objects] running on other lightweight processes [`Thread` objects]). For example, the following code would create a green thread for each of the `Runnable` objects, and these green threads would share the lightweight process represented by the `Thread` object.

```
new Thread( new My_runnable_object(), new My_other_runnable_object() );
```

The existing idiom of overriding `Thread` and implementing `run()` should still work, but it should map to a single green thread bound to a lightweight process. (The default `run()` method in the `Thread` class would effectively create a second `Runnable` object internally.)

Inter-Thread Coordination

More facilities should be added to the language to support inter-thread communication. Right now, the `PipedInputStream` and `PipedOutputStream` classes can be used for this purpose, but they are much too inefficient for most applications. I propose the following additions to the `Thread` class:

1. Add a `wait_for_start()` method that blocks until a thread's `run()` method starts up. (It would be okay if the waiting thread was released just before `run()` was called.) This way one thread could create one or more auxiliary threads, and then be assured that the auxiliary threads were running before the creating thread continued with some operation.

2. Add (to the `Object` class) `$send(Object o)` and `Object=$receive()` methods that would use an internal blocking queue to pass objects between threads. The blocking queue would be created automatically as a side effect of the first `$send()` call. The `$send()` call would enqueue the object; the `$receive()` call would block until an object was enqueued, and then return that object. Variants on these methods would support timeouts on both the enqueue and dequeue operations: `$send(Object o, long timeout)` and `$receive(long timeout)`.

Internal Support for Reader/Writer Locks

The notion of a reader/writer lock should be built into Java. To remind you, a reader-writer lock enforces the rule that multiple threads can simultaneously access an object, but only one thread at a time can modify the object, and modifications cannot go on while accesses are in progress. The syntax for a reader/writer lock can be borrowed from that of the `synchronized` keyword:

```
static Object global_resource;

//...

public void f()
{
    $reading( global_resource )
    {   // While in this block, other threads requesting read
        // access to global_resource will get it, but threads
        // requesting write access will block.
    }
}
```

```
public void g()
{
    $writing( global_resource )
    {   // Blocks until all ongoing read or write operations on
        // global_resource are complete. No read or write
        // operation or global_resource can be initiated while
        // within this block.
    }
}

public $reading void f()
{   // just like $reading(this)...
}

public $writing void g()
{   // just like $writing(this)...
}
```

Access to Partially Constructed Objects Should Be Illegal

The JLS currently permits access to a partially created object. For example, a thread created within a constructor can access the object being constructed, even though that object might not be fully constructed. The behavior of the following code is undefined:

```
class Broken
{   private long x;

    Broken()
    {   new Thread()
        {   public void run()
            {   x = -1;
            }
        }.start();

        x = 0;
    }
}
```

The thread that sets `x` to –1 can run in parallel to the thread that sets `x` to 0. Consequently, the value of `x` is unpredictable.

One possible solution to this problem is to require that the `run()` methods of threads started from within a constructor not execute until that constructor returns, even if the thread created by the constructor is of higher priority than the one that called `new`. That is, the `start()` request must be deferred until the constructor returns.

Alternatively, Java could permit the synchronization of constructors. In other words, the following code (which is currently illegal) would work as expected:

```
class Illegal
{   private long x;

    synchronized Broken()
    {   new Thread()
        {   public void run()
            {   synchronized( Illegal.this )
                {
                    x = -1;
                }
            }
        }.start();

        x = 0;
    }
}
```

I think that the first approach is cleaner than the second one, but it is admittedly harder to implement.

Volatile Should Always Work as Expected

The compiler and JVM are both permitted to shuffle around your code, provided that the semantics of the code doesn't change. For example, in the following source code the assignment to `first` might be made *after* the assignment to `second`, in which case the `g()` method, which uses a test on `first` to decide what to pass to `some_method()` won't work correctly. The value `false` might be passed to `some_method()`.

```
class Broken
{
    volatile boolean first  = false;;
    volatile boolean second = false;;

    public void f()
    {   first  = true;
        second = true;
    }

    public void g()
    {   if( second )
            Some_class.some_method( first );
    }
}
```

This code movement is desirable because some optimizations (such as loop-invariant code motion) require it. Nonetheless, it makes the non-`synchronized` use of `volatile` rather risky. One possible solution is to require that if a method accesses several `volatile` fields in sequence, that the fields always be accessed in declaration order. This is a really tough one to implement, but would probably be worth the trouble.

Access Issues

The lack of good access control makes threading more difficult than necessary. Often, methods don't have to be thread safe if you can guarantee that they be called only from synchronized subsystems. I'd tighten up Java's notion of access privilege as follows:

1. Require explicit use of the `package` keyword to specify package access. I think that the very existence of a default behavior is a flaw in any computer language, and I am mystified that a default access privilege even exists (and am even more mystified that the default is "package" rather than "private"). Java doesn't use defaults anywhere else. Even though the requirement for an explicit `package` specifier would break existing code, it would make that code a lot easier to read and could eliminate whole classes of potential bugs (if the the access privilege had been omitted in error rather than deliberately, for example).

2. Reintroduce `private protected`, which works just like `protected` does now, but does not permit package access.

3. Permit the syntax `private private` to specify "implementation access:" `private` to all outside objects, even objects of the same class as the current object. The only reference permitted to the left of the (implicit or explicit) dot is `this`.

4. Extend the syntax of `public` to grant access to specific classes. For example, the following code would permit objects of class `Fred` to call `some_method()`, but the method would be private with respect to all other classes of objects.

   ```
   public(Fred) void some_method()
   {
   }
   ```

   This proposal is different from the C++ "friend" mechanism, which grants a class full access to *all* private parts of another class. Here, I'm suggesting a tightly controlled access to a limited set of methods. This way one class could define an interface to another class that would be invisible to the rest of the system.

5. Require all field definitions to be `private` unless they reference truly immutable objects or define `static final` primitive types. Directly accessing the fields of a class violates two basic principles of OO design: abstraction and encapsulation. From the threading perspective, allowing direct access to

fields just makes it easier to inadvertently have non-synchronized access to a field.

6. Add the `$property` keyword: Objects tagged in this way are accessible to a "bean box" application that is using the introspection APIs defined in the `Class` class, but otherwise works identically to `private private`. The `$property` attribute should be applicable to both fields and methods so that existing JavaBean getter/setter methods could be easily defined as properties.

Immutability

The notion of immutability (an object whose value cannot change once it's created) is invaluable in multithreaded situations since read-only access to immutable objects doesn't have to be synchronized. Java's implementation of immutability isn't tight enough for two reasons:

1. It is possible for an immutable object be accessed before its fully created, and this access might yield an incorrect value for some field.

2. The definition of immutable (a class, all of whose fields are `final`) is too loose: Objects addressed by `final` references can indeed change state, even though the reference itself cannot change state.

The first problem is related to the access-to-partially-constructed-objects problem discussed above.

The second problem can be solved by requiring that `final` references point to immutable objects. That is, an object is really immutable only if all of its fields are final and all of the fields of any referenced objects are final as well. In order not to break existing code, this definition could be enforced by the compiler only when a class is explicitly tagged as immutable as follows:

```
$immutable public class Fred
{
    // all fields in this class must be final, and if the
    // field is a reference, all fields in the referenced
    // class must be final as well (recursively).

    static int x constant = 0;  // use of `final` is optional when $immutable
                                // is present.
}
```

Given the `$immutable` tag, the use of `final` in the field definitions could be optional.

Finally, a bug in the Java compiler makes it impossible to reliably create immutable objects when inner classes are on the scene. When a class has nontrivial inner classes (as most of mine do), the compiler often incorrectly prints the error message: "Blank final variable '*name*' may not have been initialized. It must be assigned a value in an initializer, or in every constructor," even though the blank final is indeed initialized in every constructor. This bug has been in the compiler since inner classes were first introduced in version 1.1, and at this writing (three years later—May 2000) the bug is still there. It's about time that this bug was fixed.

Instance-Level Access of Class-Level Fields

In addition to access privileges, there is also the problem that both class-level (`static`) methods and instance (non-`static`) methods can directly access class-level (`static`) fields. This access is dangerous because synchronizing the instance method doesn't grab the class-level lock, so a `synchronized static` method can access the class field at the same time as a `synchronized` instance method. The obvious solution to this problem is to require that non-immutable `static` fields be accessed from instance methods via `static` accessor methods. This requirement would mandate both compile and run-time checks, of course. Under these guidelines, the following code would be illegal:

```
class Broken
{
    static long x;

    synchronized static void f()
    {   x = 0;
    }

    synchronized void g()
    {   x = -1;
    }
};
```

because `f()` and `g()` can run in parallel and modify `x` simultaneously (with undefined results). Remember, there are two locks here: the `static` method acquires the lock associated with the `Class` object and the non-`static` method acquires the lock associated with the instance. The compiler should either require the following structure when accessing non-immutable `static` fields from instance methods:

```
class Broken
{
    static long x;

    synchronized private static accessor( long value )
    {   x = value;
    }
```

```
        synchronized static void f()
        {   x = 0;
        }

        synchronized void g()
        {   accessor( -1 );
        }
    }
```

or the compiler should require the use of a reader/writer lock:

```
    class Broken
    {
        static long x;

        synchronized static void f()
        {   $writing(x){ x = 0 };
        }

        synchronized void g()
        {   $writing(x){ x = -1 };
        }
    }
```

Alternatively—and this is the ideal solution—the compiler should *automatically* synchronize access to non-immutable `static` fields with a reader/writer lock so that the programmer wouldn't have to worry about it.

Singleton Destruction

The singleton-destruction problem discussed in Chapter 7 is a serious one. The best solution is to introduce the `$singleton` tag to the class definition. The Singleton creation would then be handled automatically by the system. For example, if you defined a class as follows:

```
    $singleton class Fred
    {   //...
    }
```

then *all* calls to `new Fred()` would return a reference to the *same* object, and that object would not be created until the first call to `new Fred()`. Moreover, the language should guarantee that the finalizer method of a `$singleton` object will always be called as part of the JVM shut-down process.

Abrupt Shut Down of Daemon Threads

Daemon threads are shut down abruptly when all the non-daemon threads terminate. This is a problem when the daemon has created some sort of global resource (such as a database connection or a temporary file), but hasn't closed or destroyed that resource when it is terminated.

I'd solve that problem by making the rule that the JVM will not shut down the application if any:

1. non-daemon threads are running, or

2. daemon threads are executing a `synchronized` method or `synchronized` block of code.

The daemon thread would be subject to abrupt termination as soon as it left the `synchronized` block or `synchronized` method.

Bring Back the **stop()**, **suspend()**, and **resume()** Methods

This one may not be possible for practical reasons, but I'd like `stop()` not to be deprecated [in both `Thread` and `ThreadGroup`]. I would change the semantics so that calling `stop()` wouldn't break your code, however: The problem with `stop()`, you'll remember, is that `stop()` gives up any locks when the thread terminates, thereby potentially leaving the objects that the thread is working on in an unstable (partially modified) state. These objects can nonetheless be accessed since the stopped thread has released its lock on the object. This problem can be solved by redefining the behavior of `stop()` such that the thread would terminate immediately only if it is holding no locks. If it is holding locks, I'd like the thread to be terminated immediately after releasing the last one. You could implement this behavior with a mechanism similar to an exception toss. The stopped thread would set a flag that would be tested immediately after exiting all synchronized blocks. If the flag was set, an implicit exception would be thrown, but the exception would not be catchable and would not cause any output to be generated when the thread terminated. Note that Microsoft's NT operating system doesn't handle an abrupt externally implied stop very well. (It doesn't notify dynamic-link libraries of the stop, so systemwide resource leaks can develop.) That's why I'm recommending an exception-like approach that simply causes `run()` to return.

The practical problem with this exception-style approach is that you'd have insert code to test the "stopped" flag at the end of every `synchronized` block, and this extra code would both slow down the program and make it larger. Another approach that comes to mind is to make `stop()` implement a "lazy" stop in which the thread terminates the next time it calls `wait()` or `yield()`. I'd also add `isStopped()` and `stopped()` methods to `Thread` [which would work much like `isInterrupted()`

and interrupted(), but would detect the "stop-requested" state.] This solution isn't as universal as the first, but is probably workable and wouldn't have the overhead.

The suspend() and resume() methods should just be put back into Java. They're useful, and I don't like being treated like a kindergartener: Removing them simply because they are potentially dangerous—a thread can be holding a lock when suspended—is insulting. Let me decide whether or not I want to use them. Sun could always make it a run-time exception for suspend() to be called if the receiving thread is holding any locks, or better yet, defer the actual suspension until the thread gives up its locks.

Blocking I/O Should Work Correctly

You should be able to interrupt any blocking operation, not just just wait() and sleep(). I discussed this issue in the context of sockets in Chapter 2, but right now, the only way to interrupt a blocking I/O operation on a socket is to close the socket, and there's no way to interrupt a blocking I/O operation on a file. Once you initiate a read request, for example, the thread is blocked until it actually reads something. Even closing the file handle does not break you out of the read operation.

Moreover, your program must be able to time out of a blocking I/O operation. All objects on which blocking operations can occur (such as InputStream objects) should support a method like this:

```
InputStream s = ...;
s.set_timeout( 1000 );
```

(this is the equivalent to the Socket class's setSoTimeout(time) method. Similarly, you should be able to pass a timeout into the blocking call as an argument.

The **ThreadGroup** Class

ThreadGroup should implement all the methods of Thread that can change a thread's state. I particularly want it to implement join() so that I can wait for all threads in the group to terminate.

Wrapping Up

So that's my list. As I said in the title of this chapter: if I were king... (sigh). I'm hoping that some of these changes (or their equivalent) will eventually migrate into the language. I really think that Java is a great programming language; but I also think that Java's threading model simply wasn't thought out well enough, and that's a pity. Java is evolving, however, so there is a path to follow to improve the situation.

This section also wraps up the book. Programming threads properly is difficult but rewarding, simultaneously frustrating and fun. I hope that the code I've offered here will take some of the frustration out of the process. As I said in the Preface, the code is not in the public domain, but I'm happy to grant permission to anyone to use it in exchange for a mention (of both my name and URL) in your "about box," your "startup" screen, or if you don't have either in your program, your documentation. I'm also *really* interested in any bugs you find (please send me email at *bugs@holub.com*). The most recent version of the code is available on my Web site *http://www.holub.com*.

Have fun.