Coping with Java Threads

Java works for many kinds of concurrent software, but it was not designed for safety-critical real-time applications and does not protect the programmer from the pitfalls associated with multithreading.

Bo Sandén
Colorado Technical University

A thread is a basic unit of program execution that can share a single address space with other threads—that is, they can read and write the same variables and data structures. Originally, only assembly programmers used threads. A few older programming languages such as PL/I supported thread concurrency, but newer languages such as C and C++ use libraries instead. Only recently have programming languages again begun to build in direct support for threads. Java and Ada are examples of industry-strength languages for multithreading.[1-3]

The Java thread model has its roots in traditional concurrent programming.[4,5] It nevertheless has serious, well-known inherent limitations, and it includes constructs that are easily abused. As the "Real-Time Specification for Java" sidebar describes, RTSJ attempts to remove some of the limitations relative to real-time applications—primarily by circumventing garbage collection. But RTSJ does not make the language safer.[6] It retains standard Java's threading pitfalls and is a risky candidate for critical concurrent applications.

Notably, the pitfalls are almost nonexistent in Ada,[1,3] which, unlike Java, was designed for safety-critical real-time applications from the start and has support mechanisms built into the syntax. While an Ada programmer can rely on the compiler to guard against major mistakes, a Java programmer must understand the basics of threading and synchronization as well as the Java thread model and strategies for avoiding its pitfalls.

THREADING AND SYNCHRONIZATION

The Java concurrency model relies on two entities: threads and synchronized objects. Each thread has its own *context* including a *program counter* that indicates what instruction the thread must execute next and a *stack* that reflects what method it's currently processing and what methods it must return to. With its own program counter and stack, each thread proceeds independently through the program code.

Each thread also has a priority governing its access to a processor. A processor executes at most one thread at a time—normally the highest-priority thread that is ready to run. If a higher-priority thread becomes ready to run while a lower-priority thread is using the processor, the higher-priority thread preempts the lower-priority thread and starts processing. An operation where the processor starts running a new thread and using its context is called a *context switch*. A thread can force a context switch by yielding the processor—for example, by calling *sleep*.

Threads can speed up a program by taking advantage of symmetric multiprocessors, but they are also useful on a single processor where one thread can compute while others are waiting for external input.

A shared object is one that multiple threads can access. Those accesses by different threads must be synchronized to guard against data inconsistencies. Java provides *exclusion synchronization* through the keyword *synchronized*. Exclusion synchronization, or mutual exclusion, makes a synchronized object *thread safe*—that is, only one thread at a time can call its synchronized methods. A *critical section* is a code sequence that is executed under exclusion synchronization.

Java also provides *condition synchronization* for threads that must wait for a certain condition before proceeding in a synchronized method.

Distinguishing between these two forms of synchronization is crucial to correct programming, and

 Published by the IEEE Computer Society 0018-9162/04/$20.00 © 2004 IEEE

many of the Java threading pitfalls result from confusing them.

Exclusion synchronization

This mechanism stops different threads from calling methods on the same object at the same time and thereby jeopardizing data integrity. In a well-programmed system, each thread maintains exclusive access for a very short time. A thread rarely finds an object locked and, if it does, the wait is brief.

In such a well-programmed system, it's highly unlikely that two or more threads will attempt access to the same object while it is locked; so it's not necessary to maintain an orderly queue of threads pending an object lock. Instead, when a thread encounters a locked object, it yields the processor. If the object is still locked when the thread becomes eligible for processing, the thread again yields the processor. If the thread has its own processor, it can instead enter a loop where it repeatedly attempts access ("spins") until successful. I'll use the term *spin lock* for both the single processor and multiprocessor cases.

A *priority inversion* occurs when a higher-priority thread is waiting for a lower-priority thread. While a thread, *low*, is operating on a shared object, *o*, under exclusion synchronization, a higher-priority thread, *high*, which also needs exclusive access to *o,* can preempt *low*. Unavoidably, *high* must wait for *low* to exit a critical section.

If *low* continues executing at its normal priority after *high* has begun waiting, a third thread, *intermediate*, whose priority is between *high* and *low*, can preempt *low*. If *intermediate* does not need access to *o*, it can take over the processor while *low* is still locking *o*. To avoid this situation, where *high* is waiting for more than one lower-priority thread, the synchronization mechanism can give *low* a priority boost in one of two ways:

- let *low* inherit *high*'s priority once *high* tries to access *o*; or
- define a *ceiling priority* for *o*, which means that any thread has this ceiling priority while executing a synchronized method on *o*.

The ceiling priority must be that of the highest-priority thread that ever operates on *o*, also called the "highest locker."

Condition synchronization

In condition synchronization, a thread waits for some resource other than a synchronized object. Condition synchronization includes no assumption that the wait will be brief; threads can wait indefinitely. A classic example of condition synchronization is a Buffer class with the methods *put* and *get*. Producer threads call *put* and consumer threads call *get*. Producers must wait if the buffer is full, and consumers must wait if it's empty.

Condition synchronization complements exclusion synchronization. A producer thread, *t*, must first lock the buffer to see that it's full. But while *t* is waiting for the condition to change, the buffer must remain unlocked so consumers can call *get*. Java provides the operation *wait* for suspending threads that are waiting for a condition to change. When *t* finds the buffer full, it calls *wait*, which releases the lock and suspends the thread. After some other thread notifies *t* that the buffer may no longer be full, *t* regains the lock on the buffer and retests the condition.

Condition synchronization is used when threads control shared resources in a problem domain. For example, in an automated factory application,[7,8] jobs may need exclusive access to an automated forklift truck. A Job thread represents each job. Because a forklift operation can continue for several minutes, it requires condition synchronization so that waiting Job threads won't spin.

Real-Time Specification for Java

RTSJ is based on the premise that a real-time program must be predictable so that the programmer can determine a priori when certain events will occur. Standard Java does not meet this premise for various reasons.

For example, the garbage collector can interrupt any processing at any time, which introduces an element of randomness that makes it impossible to predict how long a certain operation will take. RTSJ addresses this problem by introducing several new classes, one of which, NoHeapRealtimeThread, is a descendent of Thread, but its instances have higher priority than the garbage collector. This lets NHRTs execute predictably, but it places restrictions on the programmer, who must explicitly allocate any new objects in special memory areas.

Another example of particular interest to this article: A *notify* call in standard Java reactivates threads in a wait set in arbitrary order, no matter how long they have waited. In RTSJ, the wait set is a first-in, first-out (FIFO) queue within priorities, and *notify* reactivates the thread that has the highest priority and has waited the longest. RTSJ uses priority inheritance as the default policy to control priority inversion. It also specifies a priority ceiling protocol.

To further support real-time programming, RTSJ allows the programmer to specify interrupt handlers.

THE JAVA THREAD MODEL

Programming threads in Java is much simpler than programming with the thread packages that come with C and C++. Exclusion synchronization is built into the Java language so programmers need not manipulate semaphore objects to synchronize threads. Further, programmers can use Java's object

All Java objects have the potential for exclusion synchronization.

model to fit thread classes and synchronized classes into inheritance hierarchies.

Defining and starting threads

Java provides the abstract class Thread, whose method *run* is intended to contain a thread's main logic. A standard way of creating threads is to declare a new class—say, *T*—that extends Thread and overrides *run* with appropriate processing.

Each instance of *T*—say, *to*—has its own thread, which is started by the call *to.start*. Once started, the thread executes *T*'s *run* method and has access to the instance *to*'s data.

Java does not allow multiple inheritance, so another mechanism is necessary if a class—say, *R*—that needs a thread already extends another class—say, Applet. For this situation, Java provides the interface Runnable. So, *R* extends Applet and implements Runnable. Instantiating *R* creates a *runnable object*. To associate a thread with a runnable object, the programmer submits the object as a parameter to one of Thread's constructors. This results in a thread instance, which is then started by a *start* call.

Synchronizing objects

All Java objects have the potential for exclusion synchronization. Every object has a lock variable, which is hidden from the programmer. A method, *m*, is synchronized as follows:

```
void synchronized m( ) { ...  }
```

Java brackets a synchronized method's code with statements that acquire and release the lock on the object, *o*, on which *m* is invoked. In other words, a thread calling *o.m* locks the object as a whole, and no other thread can perform any synchronized method on it. The thread always releases the lock when it leaves the synchronized method, even if it leaves through Java's exception-handling mechanism. Any class instance that has at least one synchronized method or block is a *synchronized object*.

A Java programmer can choose to specify only some methods of a class as synchronized. This technique has useful applications. For example, there is no need to synchronize a read-only method that returns a single attribute value. Different threads can execute nonsynchronized methods simultaneously while yet another thread is executing a synchronized method. If the methods are lengthy, this option can increase concurrency, especially on multiprocessors, where different threads can really execute at the same time.

Synchronized blocks. In addition to synchronized methods, Java provides synchronized blocks. The following syntax can synchronize a block in any method with respect to an object—in this case, block *B*:

```
synchronized (Expression)
  {  /* Block B */  }
```

The value of *Expression* must be a reference to some object—say, *vo* of class *V*.

Like a synchronized method, a synchronized block is a critical section that Java brackets by statements to acquire and release the lock on the specified object. Consider first the case where block *B* is part of some method, *m*, of class *V* and is synchronized with respect to the current object as follows:

```
class V ... {
 void m( ) {
   synchronized (this) {
     /* Block B*/
   }
 }
}
```

In this excerpt, *this* is a reference to the current object.

Synchronized blocks offer a way to synchronize only part of *m* when all of *m* does not require exclusive access.[9] If only *B* is synchronized, two or more threads can simultaneously execute the rest of *m*. Another option is to make *B* a separate, synchronized method of *V* and call it from within *m*.

The programmer can synchronize *B* with respect to any object. The following construct synchronizes *B* with respect to object *wo* of class *W* (though *wo* could also be another instance of class *V*):

```
class V ...{
 void m( ) {
   synchronized (wo) {
     /* Block B*/
   }
 }
}
```

Arguably, this is bad programming style: *B* is an operation on *wo* and so should be a method defined in *W*'s declaration. But synchronized blocks

can prove handy when different threads must perform their own operations on a shared object under exclusive access. For example, many threads can write tailored outputs to a printer object as follows:

```
synchronized (myPrinter) {
  // Block of statements that
  produce output to myPrinter
}
```

In this case, the synchronized block obviates the need to make every possible combination of output statements into a printer class method.

Nested critical sections. A programmer can nest critical sections in various ways. If *m* is itself synchronized, the example for *wo* becomes

```
class V ... {
 synchronized void m( ) {
   synchronized (wo) {
     /* Block B */
   }
 }
}
```

Here, *B* executes with exclusive access to both *wo* and the current instance of *V*. Another way of nesting is to call a synchronized method from within a synchronized block or method.

Nesting can be necessary for coordinating updates. Say a synchronized object *z* controls two synchronized objects, *x* and *y*. As part of a synchronized method *m* on *z*, a thread must also update *x* and *y* to maintain consistency. For this, *m* can contain nested blocks that are synchronized with respect to *x* and *y*. The innermost block provides exclusive access to *x*, *y*, and *z*.

Multiple locks on the same object. A programmer can also nest methods and blocks that are synchronized with respect to the same object. Assume that thread *t* calls *o.m* and that *m* is synchronized. If *t* calls another synchronized method on *o* from within *m*, *t* gets an additional lock on *o*. Each time *t* exits a critical section, it releases one lock. That way, *t* keeps *o* locked until it exits its outermost critical section with respect to *o*.

Unfortunately, excessive multiple locking of single objects can cause performance problems.

Syntax complications

The standard Java idiom for condition synchronization is the statement

```
while (cond) wait( );
```

Such a *wait loop* statement must be inside a synchronized method or block. The wait loop stops any calling thread, *t*, for as long as the condition *cond* holds. If *cond* is true, *t* calls *wait*, thereby placing itself in the current object's wait set and releasing all its locks on that object.

The need to handle an InterruptedException can complicate the wait loop syntax. By means of this exception, one thread can interrupt another. If the second thread is in a wait set, it is activated and proceeds to an exception handler. For this reason, the following construct is often necessary:

```
while (cond) try {wait( );}
catch(InterruptedException e)
  {/* Handle exception */}
```

Java syntax allows a wait loop to appear anywhere within a synchronized method or block.

Wait loop placement. The wait loop usually appears at the very beginning of a critical section, and a thread reaches the loop immediately after it locks the object. But the Java syntax allows a wait loop to appear anywhere within a synchronized method or block.

For example, here's a way to count the calls to a method in an instance variable:

```
synchronized void m( ) {
  callCounter ++;
  while (cond) wait( );
  . . . .
}
```

The textbook case for placing the wait loop deep inside a method is when the method allocates resources to calling threads. If the method cannot satisfy a request from the calling thread until additional resources become available, the calling thread, *t*, can place itself in the wait set until some other thread notifies it of released resources; *t* then reacquires its locks on the object and continues processing immediately after the wait call.

Notification of waiting threads. A thread that executes a synchronized method on an object, *o*, and changes a condition that can affect one or more threads in *o*'s wait set must notify those threads. In standard Java, the call *o.notify* reactivates one arbitrarily chosen thread, *t*, in *o*'s wait set. If *t* called *wait* from within a correct wait loop, it then reevaluates the condition and either proceeds in the synchronized method or reenters the wait set. In RTSJ, the most eligible thread is reactivated.

The first line of defense for managing pitfalls is to establish a style guide for safe programming.

The call *o.notifyAll* releases all threads in *o*'s wait set. This is useful after a thread changes the condition in such a way that multiple other threads can proceed. But sometimes a programmer must use *notifyAll* instead of *notify* to let a single thread proceed. In fact, in standard Java, this is the only way to be sure to activate the highest priority thread. It is inefficient if the wait set includes many threads, since all the threads must attempt access while only one will succeed.[9]

Because each object has only one wait set, the programmer must also use *notifyAll* instead of *notify* if the wait set might include threads pending on different conditions. If a thread changes one of the conditions, it must activate all waiting threads to be sure of activating one pending on that condition. This holds in RTSJ as well as in standard Java.

Calls to *o.wait*, *o.notify*, or *o.notifyAll* can only occur inside a method or block that is synchronized with respect to *o*. The wait set is itself a shared data structure but does not have its own lock. The object lock protects it, and a thread can only operate on the wait set if it holds the object lock.

Shared-domain resource access

A programmer must use condition synchronization to control shared resources in the problem domain, such as the forklift in the factory application. Typically, the object controlling access to the forklift—say, instance *f* of class Forklift—has synchronized operations, such as *acquire* and *release*, and a Boolean attribute, *busy*, indicating the forklift's availability. The method *acquire* can be as follows:

```
synchronized void acquire( ) {
  while (busy) wait( );
  busy = true;
}
```

The method *release* sets *busy* to false and notifies any waiting threads. Calls to *acquire* and *release* bracket each statement sequence for operating the forklift. While one job is using the forklift, other Job threads can call *f.acquire* and place themselves in *f*'s wait set. The variable *busy* locks the physical forklift while *f*'s hidden lock protects only the variable *busy* and the wait set data structure.

In the earlier example of a synchronized block enclosing the operations on a printer, threads spin while waiting for the printer. In an alternative solution, a class Printer has exactly the same methods *acquire* and *release* we used for the forklift. If *p* is a Printer instance, the printer operations appear as follows:

```
p.acquire( );
// Series of statements producing
  output
p.release( );
```

With this solution, threads waiting for the printer are in *p*'s wait set.

PITFALLS AND STRATEGIES

Java gives the virtuoso thread programmer considerable freedom, but it also presents many pitfalls for less experienced programmers who can create complex programs that fail in baffling ways. The first line of defense for managing those pitfalls is to establish a style guide for safe programming. A precompilation tool or compiler can enforce certain rules, but in other cases, inspection is the best way to ensure compliance.

The pitfalls discussed here are inherent and not easily removed. They exist in RTSJ as well as standard Java.

Multiple threads, one object

Given a class *R* that implements Runnable, Java gives programmers two ways to create multiple threads that execute *R*'s *run* method:

- Instantiate *R* *n* times, and submit each instance once as a parameter to one of Thread's constructors. Each instance of *R* now has a thread with its own set of instance variables.
- Submit one *R* instance repeatedly to Thread's constructor. This generates multiple threads tied to one object. All these threads can freely manipulate the object's data and potentially introduce inconsistencies.

Strategies. For most applications, each runnable object needs no more than one thread. In that case, the programmer submits each runnable object to a Thread constructor only once. Someone reading the program can easily verify compliance of such a policy if programmers always include thread creation and start in the constructor as follows:

```
new Thread(this).start( );
```

Some applications might require a more liberal policy that allows a given runnable object to have

at most one thread at a time but to acquire a new thread after the earlier one has expired.

Omitting the *synchronized* keyword

The freedom to synchronize selected methods of a class opens the door for mistakes. In the buffer example, the programmer can declare *get* synchronized and not *put*. This allows different threads to call *put* simultaneously. The *put* calls can also overlap with a call to *get*, jeopardizing the buffer data structure's integrity. The program may still work much of the time, but it will produce occasional errors, especially when run on multiprocessors. Finding such errors by testing tends to be difficult; an experienced thread programmer can find them more easily by inspection. Omitting the keyword *synchronized* altogether, for *put* as well as *get*, exacerbates the situation.

To enforce exclusion synchronization, the programmer must also ensure that synchronized methods use only private instance variables. If the variables are public, a different method, perhaps belonging to another class, can change them without first acquiring the object lock.

However, a static method defined for a class—say, *static void m*(*C o*) in the class *C*—can change *o*'s instance variables even if they are private. It may be tempting to solve this potential conflict by synchronizing *m*. Unfortunately, this does not synchronize *m* with respect to *o*. Instead, a thread calling *m* gets the static lock intended to protect *C*'s static data.

Strategies. A tool or a compiler can identify classes in which some methods are synchronized and others not, and it can easily flag the unsynchronized ones. It is clearly more difficult to spot classes that should have synchronized methods but don't.

Maintaining synchronized blocks

A synchronized block is essentially a synchronized method declared outside the class. Someone maintaining the software may interpret a class without synchronized methods as nonsynchronized, even if some blocks are synchronized with respect to its instances. The maintainer may add a method to the class without realizing that synchronization is required.

Strategies. In the interest of program readability, synchronized methods are preferable to synchronized blocks. One exception is in coordinating operations on different objects. If blocks are synchronized with respect to a certain class's instances, the class definition should contain a comment to that effect.

Wait loop placement

The wait loop is like an incantation that hardly ever changes. It should always make a thread retest the condition after being reactivated from the wait set.

Many erroneous variations are possible that compile and can be difficult to debug.[10] For example, *yield* can replace *wait*:

```
while (cond) yield( );
```

This wait loop variation stops the calling thread, *t*, from proceeding against *cond* but fails to release *t*'s locks on the current object. Thus, other threads that are supposed to change *cond* by calling synchronized methods on the object cannot do so, and *cond* can remain true forever.

More insidious mistakes allow threads to proceed against *cond*. The statement

```
if (cond) wait( );
```

substitutes *if* for *while,* which makes threads call *wait* if *cond* holds,[1] but only once**.** Once reactivated from the wait set, each thread proceeds in the synchronized method even if *cond* is still true.

The freedom to place the wait loop anywhere within a critical section opens the way to certain errors. Even if the loop is initially at the beginning of the section, someone maintaining the program can unintentionally insert statements between the beginning and the loop. Every thread entering the critical section executes these statements once. This can be more treacherous if statements already exist between the beginning of the critical section and the wait loop, as in the callCounter example:

```
void synchronized m( ) {
   callCounter ++;
   while (cond) wait( );
   . . . .
}
```

In this case, a maintainer may not understand the difference in status between the statements before and after the wait loop. If a thread, *t*, executing *m* finds *cond* true and calls *wait*, it releases the object lock and then reacquires it upon reactivation. In this case, the statements before and after the call are in different critical sections. If *t* accesses some of the synchronized object's data before the wait call and some after, the values can be inconsistent if another thread has changed them in the interim.

The freedom to synchronize selected methods of a class opens the door to mistakes.

Including a time-out parameter in every wait call mitigates the effect of missed notifications in standard Java.

Strategies. A tool or a compiler can flag any call to wait that is not in a correct wait loop. It can also flag any wait loop that is not the first statement in a synchronized method or block. These situations are not necessarily erroneous, but they should be rare and they warrant manual inspection.

Missing notification of waiting threads

Unlike exclusion synchronization, condition synchronization is not automatic; the programmer must include calls to *notify* or *notifyAll* to explicitly reactivate waiting threads. It's easy to forget inserting *notify* calls at all the necessary places.

A related mistake is to call *notify* instead of *notifyAll* when threads in the wait set may be pending on different conditions. In this case, the notify call is insufficient; it may activate a thread whose condition has not changed.

Strategies. Short of a tool that identifies all the paths through a method, inspection is the only way to ensure that *notify* or *notifyAll* are called in all necessary circumstances including those where a method has unusual exits, for example, via exception handlers. Keeping the logic in all synchronized methods simple makes this easier.

A way to mitigate the effect of missed notifications in standard Java is to include a time-out parameter in every wait call. After the given time, the thread is automatically activated. If the call is part of a correct wait loop, the thread reevaluates the condition and then either proceeds or reenters the wait set.

This technique does not work in RTSJ, however, because it defeats the FIFO queuing discipline.

Confusing long and short waits

Every critical section should be programmed to minimize the time an object is held locked, but nothing prevents a thread from keeping a lock for a long time, while other threads spin. A trivial way to do this is to call *sleep* inside a critical section. A less obvious way is to use exclusion synchronization where condition synchronization is called for.

A programmer can make this mistake in a real-time application that controls problem-domain resources. In the factory example, the programmer should use condition synchronization to share the forklift among different jobs, but can erroneously try to do it by means of the following synchronized block within the Job class *run* method:

```
synchronized (f) {  // f is a
  Forklift instance
    .....
}
```

This ensures mutual exclusion of jobs using the forklift. But a Job thread that needs the forklift isn't put in a wait set (and FIFO-queued within priorities in RTSJ). Instead it spins, perhaps for minutes, until it finds *f* unlocked. Which Job thread gets to the forklift next is rather arbitrary, even though higher-priority threads stand a better chance.

With RTSJ, using exclusion synchronization also invokes the priority-inversion control policy. Assume first that the default policy, priority inheritance, is in effect. If a job *low* is currently operating the forklift, and a higher-priority job, *high*, attempts to get the lock, *low*'s remaining forklift operations will execute at *high*'s priority. This can affect other jobs with priorities between *low* and *high*. The ceiling-priority protocol has the even more fundamental effect of giving all forklift operations the highest priority of any job.

A beginner might try to achieve condition synchronization by inserting a wait loop in the block where the physical forklift is operated:

```
synchronized (f) {
   while (busy) wait( );
   busy = true;

   // Operate the forklift

   busy = false;
   notify( );
}
```

This wait loop has no effect. The thread that sets *busy* to true also keeps object *f* locked so no other job that needs the forklift can reach the wait loop.

The only appropriate solution is to let the Job thread call *f.acquire* before operating the forklift and call *f.release* after it's done. This essentially changes the Forklift instances into semaphores. Unfortunately, it looks less elegant than the synchronized block solution, which abstracts away from the lock operations. But controlling access to shared problem-domain resources forces the programmer to invert the abstraction by using a synchronized object to implement a semaphore.[7]

The correct solution has its own pitfall: To avoid resource leakage, where—in this case—a thread

would keep the forklift from the other threads forever, the programmer must ensure that each thread always calls *f.release* even if an exception is thrown during the forklift operation.

Nested critical sections are another way to inadvertently mix long and short waits. A programmer can insert a wait loop in nested synchronized blocks as follows:

```
class V ... {
  synchronized void m( ) {
    synchronized (wo) {
      while (cond) wo.wait( );
    }
  }
}
```

If *cond* is true, the calling thread enters *wo*'s wait set and releases its lock on *wo*; but it keeps the current *V* instance, *vo*, locked and lets other threads that need access to *vo* spin rather than wait in a wait set.

The following is also legal:

```
class V ... {
  synchronized void m( ) {
    synchronized (wo) {
    while (cond) wait( );
    }
  }
}
```

If *vo* is the current element of *V* and *cond* is true, the calling thread enters *vo*'s wait set and releases its locks on *vo* while keeping *wo* locked.

Strategies. There seems to be no reasonable way to stop a programmer from using an object lock for exclusive access to some problem-domain resource and holding the lock for seconds or minutes. Avoiding this pitfall requires a clear understanding of the difference between exclusion and condition synchronization. On the other hand, a tool can flag any wait calls in statically nested critical sections, thereby reducing the risk of inadvertent spin locking.

To find out which kind of synchronization to use, imagine an implementation without threads. With no threads, the implementation requires no exclusion synchronization, and any remaining constraint must use condition synchronization. For example, a sequential implementation of the automated factory must still allocate a forklift to one job at a time. Consequently, this task requires condition synchronization.

Java's popularity and the many technologies developed for it have prompted its use for ever-wider application sets. Java is adequate for many kinds of concurrent software, but for critical real-time applications it remains a considerably riskier choice than a language with concurrency features built in, such as Ada. RTSJ removes some of the obstacles associated with garbage collection but retains most pitfalls. Programmers who choose to implement in Java must understand and address the potential consequences of the programming mistakes that the language readily allows. ■

Acknowledgment

Roger Alexander made valuable suggestions regarding the structure of this article.

References

1. B. Brosgol, "A Comparison of the Concurrency Features of Ada 95 and Java," *Proc. ACM SIGAda Ann. Int'l Conf.* (SIGAda 98), *Ada Letters XVIII*, ACM Press, 1998, pp. 175-192.
2. D. Lea, *Concurrent Programming in Java*, 2nd ed., Addison-Wesley, 2000.
3. B. Sandén, "Real-Time Programming Safety in Java and Ada," *Ada User J.*, June 2002, pp. 105-113.
4. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.
5. C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," *Comm. ACM*, Oct. 1974, pp. 549-557.
6. G. Bollella and J. Gosling, "The Real-Time Specification for Java," *Computer*, June 2000, pp. 47-54.
7. J. Carter and B. Sandén, "Practical Uses of Ada-95 Concurrency Features," *IEEE Concurrency*, Oct.-Dec. 1998, pp. 47-56.
8. B. Sandén, "Modeling Concurrent Software," *IEEE Software*, Sept. 1997, pp. 93-100.
9. A. Vermeulen et al., *The Elements of Java Style*, Cambridge Univ. Press, 2000.
10. J. Carter, "Java Questions," *Computer*, Oct. 2002, p. 9.

Bo Sandén *is a professor of computer science at the Colorado Technical University in Colorado Springs. His interests include software design, especially for concurrent real-time and simulation systems; object-oriented analysis; and concurrent object-oriented systems. Sandén received a PhD from the Royal Institute of Technology, Stockholm. He is a member of the IEEE Computer Society and the ACM. Contact him at bsanden@acm.org.*