Java theory and practice: Characterizing thread safety

Thread safety is not an all-or-nothing proposition

Level: Intermediate

Brian Goetz (mailto:brian@quiotix.com?cc=&subject=Characterizing thread safety)
Principal Consultant, Quiotix Corp
23 September 2003

Contents:
Write it down before you forget it
Defining thread safety
Degrees of thread safety
Conclusion
Resources
About the author
Rate this article

Related content:
Java theory and practice series
Concurrent collections classes
Writing efficient thread-safe classes

Subscriptions:
dW newsletters
dW Subscription (CDs and downloads)

In July our concurrency expert Brian Goetz described the `Hashtable` and `Vector` classes as being "conditionally thread-safe." Shouldn't a class either be thread-safe or not? Unfortunately, thread safety is not an all-or-nothing proposition, and it is surprisingly difficult to define. But, as Brian explains in this month's *Java theory and practice*, it is critically important that you make an effort to classify the thread safety of your classes in their Javadoc. Share your thoughts on this article with the author and other readers in the accompanying discussion forum. (You can also click **Discuss** at the top or bottom of the article to access the forum.)

In Joshua Bloch's excellent book, *Effective Java Programming Language Guide* (see Resources), Item 52 is entitled "Document Thread Safety," in which developers are entreated to document in prose exactly what thread safety guarantees are made by the class. This, like most of the advice in Bloch's book, is excellent advice, often repeated, but less often implemented. (Like Bloch says in his *Programming Puzzlers* talks, "Don't code like my brother.")

How many times have you looked at the Javadoc for a class, and wondered, "Is this class thread-safe?" In the absence of clear documentation, readers may make bad assumptions about a class's thread safety. Perhaps they'll just assume it is thread-safe when it's not (that's really bad!), or maybe they'll assume that it can be made thread-safe by synchronizing on the object before calling one of its methods (which may be correct, or may simply be inefficient, or in the worst case, could provide only the illusion of thread safety). In any case, it's better to be clear in the documentation about how a class behaves when an instance is shared across threads.

As an example of this pitfall, the class `java.text.SimpleDateFormat` is not thread-safe, but it wasn't until the 1.4 JDK that this was documented in the Javadoc. How many developers mistakenly created a static instance of `SimpleDateFormat` and used it from multiple threads, unaware that their programs were not going to behave correctly under heavy load? Don't do this to your customers or colleagues!

Write it down before you forget it (or leave the company)

The time to document thread safety is definitely when the class is first written -- it is much easier to assess the thread safety requirements and behavior of a class when you are writing it than when you (or someone else) come back to it months later. You'll never have as clear a picture of what's going on in an implementation than you do when you're writing it. Additionally, documenting thread safety guarantees at the time the class is written will increase the chance that future modifications will preserve your initial thread-safety intentions, as maintainers will hopefully see the documentation as part of the class's specification.

It would be nice if thread safety were a binary attribute of a class, and you could just document whether the class is thread-safe or not. But unfortunately, it's not that simple. If a class is not thread-safe, can it be made thread-safe by synchronizing every access to objects of that class? Are there sequences of operations that cannot tolerate interference from other threads, and therefore would require synchronization not only on the basic operation but around a compound

operation? Are there state dependencies between methods that require certain groups of operations to be performed atomically? This is the sort of information that developers need to have when preparing to use a class in a concurrent application.

Defining thread safety

Defining thread safety clearly is surprisingly hard, and most definitions seem downright circular. A quick Google search reveals the following examples of typical, but unhelpful definitions (or rather descriptions) of thread-safe code:

- ...can be called from multiple programming threads without unwanted interaction between the threads.
- ...may be called by more than on thread at a time without requiring any other action on the caller's part.

With definitions like these, it's no wonder we're so confused by thread safety. These definitions are no better than saying "a class is thread-safe if it can be called safely from multiple threads," which is, of course, what it means, but that doesn't help us tell a thread-safe class from an unsafe one. What do we mean by safe?

In reality, any definition of thread safety is going to have a certain degree of circularity, as it must appeal to the class's specification -- which is an informal, prose description of what the class does, its side effects, which states are valid or invalid, invariants, preconditions, postconditions, and so on. (Constraints on an object's state imposed by the specification apply only to the externally visible state -- that which can be observed by calling its public methods and accessing its public fields -- rather than its internal state, which is what is actually represented in its private fields.)

Thread safety

For a class to be thread-safe, it first must behave correctly in a single-threaded environment. If a class is correctly implemented, which is another way of saying that it conforms to its specification, no sequence of operations (reads or writes of public fields and calls to public methods) on objects of that class should be able to put the object into an invalid state, observe the object to be in an invalid state, or violate any of the class's invariants, preconditions, or postconditions.

Furthermore, for a class to be thread-safe, it must continue to behave correctly, in the sense described above, when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, without any additional synchronization on the part of the calling code. The effect is that operations on a thread-safe object will appear to all threads to occur in a fixed, globally consistent order.

The relationship between correctness and thread safety is very similar to the relationship between consistency and isolation used when describing ACID (atomicity, consistency, isolation, and durability) transactions: from the perspective of a given thread, it appears that operations on the object performed by different threads execute sequentially (albeit in a nondeterministic order) rather than in parallel.

State dependencies between methods

Consider the code fragment below, which iterates the elements of a `Vector`. Even though all the methods of `Vector` are synchronized, this code is still not safe to use in a multithreaded environment without additional synchronization, because if another thread deletes an element at exactly the wrong time, `get()` could throw an `ArrayIndexOutOfBoundsException`.

```
Vector v = new Vector();

// contains race conditions -- may require external synchronization
for (int i=0; i<v.size(); i++) {
  doSomething(v.get(i));
}
```

What is going on here is that there is a precondition that is part of the specification of `get(index)` that says `index` must be nonnegative and less than `size()`. But, in a multithreaded environment, there's no way you can know that the last observed value of `size()` is still valid, and therefore you can't know that `i<size()`, unless you have been holding an exclusive lock on the `Vector` since before you last called `size()`.

More specifically, the problem stems from the fact that the precondition of `get()` is defined in terms of the result of `size()`. Whenever you see a pattern like this, where you must use the results of one method to condition the inputs of

another, you have a *state dependency*, and you must ensure that at least that element of the state does not change between invoking the two methods. Generally, the only way to do so is to hold an exclusive lock on the object from before you call the first method until after you call the last one; in the example above, where you iterate through the elements of a `Vector`, you would want to synchronize on the `Vector` object for the duration of the iteration.

Degrees of thread safety

As the above example shows, thread safety is not all-or-nothing. The methods of `Vector` are all synchronized, and `Vector` is clearly designed to function in multithreaded environments. But there are limitations to its thread safety, namely that there are state dependencies between certain pairs of methods. (Similarly, the iterators returned by `Vector.iterator()` will throw a `ConcurrentModificationException` if the `Vector` is modified by another thread during iteration.)

No widely accepted set of terms is available for the various levels of thread safety that commonly occur in Java classes, but it is important that you make some attempt to document the thread-safety behavior of your classes while they are being written.

Bloch outlines a taxonomy that describes five categories of thread safety: immutable, thread-safe, conditionally thread-safe, thread-compatible, and thread-hostile. It doesn't matter whether you use this system or not, as long as you document thread-safety characteristics clearly. This system has limitations -- the boundaries between the categories are not 100-percent clear, and there are cases that it doesn't address -- but this system is a pretty good starting point. Central to this classification system is whether or not a caller can or must surround operations -- or sequences of operations -- with external synchronization. Each of these five categories of thread safety are described in the following sections.

Immutable

Regular readers of this column will not be surprised to hear me extolling the virtues of immutability. Immutable objects are guaranteed to be thread-safe, and never require additional synchronization. Because an immutable object's externally visible state never changes, as long as it is constructed correctly, it can never be observed to be in an inconsistent state. Most of the basic value classes in the Java class libraries, such as `Integer`, `String`, and `BigInteger`, are immutable.

Thread-safe

Thread-safe objects have the properties described above in the section "Thread Safety" -- that constraints imposed by the class's specification continue to hold when the object is accessed by multiple threads, regardless of how the threads are scheduled by the runtime environment, without any additional synchronization. This thread-safety guarantee is a strong one -- many classes, like `Hashtable` or `Vector`, will fail to meet this stringent definition.

Conditionally thread-safe

We discussed conditional thread-safety in July's article, "Concurrent collections classes." Conditionally thread-safe classes are those for which each individual operation may be thread-safe, but certain sequences of operations may require external synchronization. The most common example of conditional thread safety is traversing an iterator returned from `Hashtable` or `Vector` -- the fail-fast iterators returned by these classes assume that the underlying collection will not be mutated while the iterator traversal is in progress. To ensure that other threads will not mutate the collection during traversal, the iterating thread should be sure that it has exclusive access to the collection for the entirety of the traversal. Typically, exclusive access is ensured by synchronizing on a lock -- and the class's documentation should specify which lock that is (typically the object's intrinsic monitor).

If you are documenting a conditionally thread-safe class, you should not only document that it is conditionally thread-safe, but also which sequences of operations must be protected from concurrent access. Users may reasonably assume other sequences of operations do not require any additional synchronization.

Thread-compatible

Thread-compatible classes are not thread-safe, but can be used safely in concurrent environments by using synchronization appropriately. This might mean surrounding every method call with a `synchronized` block or creating a wrapper object where every method is synchronized (like `Collections.synchronizedList()`). Or it might mean surrounding certain sequences of operations with a `synchronized` block. To maximize the usefulness of thread-compatible classes, they should not require that callers synchronize on a specific lock, just that the same lock is used in all invocations. Doing so will enable thread-compatible objects held as instance variables in other thread-safe objects to piggyback on the synchronization of the owning object.

Many common classes are thread-compatible, such as the collection classes `ArrayList` and `HashMap`, `java.text.SimpleDateFormat`, or the JDBC classes `Connection` and `ResultSet`.

Thread-hostile

Thread-hostile classes are those that cannot be rendered safe to use concurrently, regardless of what external synchronization is invoked. Thread hostility is rare, and typically arises when a class modifies static data that can affect the behavior of other classes that may execute in other threads. An example of a thread-hostile class would be one that calls `System.setOut()`.

Other thread-safety documentation considerations

Thread-safe classes (and classes with lesser degrees of thread safety) may or may not allow callers to lock the object for exclusive access. The `Hashtable` class uses the object's intrinsic monitor for all synchronization, but the `ConcurrentHashMap` class does not, and in fact there is no way to lock a `ConcurrentHashMap` object for exclusive access. In addition to documenting the degree of thread safety, you should also document whether any specific locks -- such as the object's intrinsic lock -- have special significance in the class's behavior.

By documenting that a class is thread-safe (assuming it actually *is* thread-safe), you perform two valuable services: you inform maintainers of the class that they should not make modifications or extensions that compromise its thread safety, and you inform users of the class that it can be used without external synchronization. By documenting that a class is thread-compatible or conditionally thread-safe, you inform users that the class can be used safely by multiple threads through the appropriate use of synchronization. By documenting that a class is thread-hostile, you inform users that they cannot use the class safely from multiple threads, even with external synchronization. In each case, you are preventing potentially serious bugs, which would be expensive to find and fix, *before* they happen.

Conclusion

A class's thread-safety behavior is an intrinsic part of its specification, and should be part of its documentation. Because there is no declarative way of describing a class's thread-safety behavior (yet), it must be described textually. While Bloch's five-tier system for describing a class's degree of thread safety does not cover all possible cases, it's a very good start. Certainly we'd all be better off if every class included this degree of threading behavior in its Javadoc.

Resources

- Participate in the discussion forum on this article. (You can also click **Discuss** at the top or bottom of the article to access the forum.)
- Read the complete *Java theory and practice* series by Brian Goetz. Of particular relevance to this column are the following installments:
  - "I have to document THAT?" (August 2002), which provides examples of Javadoc
  - "To mutate or not to mutate?" (February 2003), which discusses thread-safety benefits of immutability
  - "Concurrent collections classes" (July 2003), which examines scalability bottlenecks and how to achieve higher concurrency and throughput in shared data structures
- Compare the characterization of thread safety to the definition of isolation in "Understanding JTS, part 1" (*developerWorks*, March 2002).
- Doug Lea's *Concurrent Programming in Java, Second Edition* (Addison-Wesley, 1999) is a masterful book on the subtle issues surrounding multithreaded programming in Java applications.
- Item 52, "Document Thread Safety" of Joshua Bloch's *Effective Java Programming Language Guide* (Addison-Wesley, 2001) details the five-level taxonomy described here.
- "Writing efficient thread-safe classes" (*developerWorks*, April 2000), by Neel Kumar, tells you how you can have both "efficient" and "thread-safe."
- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks* Java technology zone.

About the author

Brian Goetz has been a professional software developer for the past 15 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California, and he serves on several JCP Expert Groups.

See Brian's published and upcoming articles in popular industry publications. Contact Brian at brian@quiotix.com.