Java theory and practice: Fixing the Java Memory Model, Part 2

How will the JMM change under JSR 133?

Brian Goetz March 30, 2004

JSR 133, which has been active for nearly three years, has recently issued its public recommendation on what to do about the Java Memory Model (JMM). In Part 1 of this series, columnist Brian Goetz focused on some of the serious flaws that were found in the original JMM, which resulted in some surprisingly difficult semantics for concepts that were supposed to be simple. This month, he reveals how the semantics of `volatile` and `final` will change under the new JMM, changes that will bring their semantics in line with most developers' intuition. Some of these changes are already present in JDK 1.4; others will have to wait until JDK 1.5.

View more content in this series

Writing concurrent code is hard to begin with; the language should not make it any harder. While the Java platform included support for threading from the outset, including a cross-platform memory model that was intended to provide "Write Once, Run Anywhere" guarantees for properly synchronized programs, the original memory model had some holes. And while many Java platforms provided stronger guarantees than were required by the JMM, the holes in the JMM undermined the ability to easily write concurrent Java programs that could run on any platform. So in May of 2001, JSR 133 was formed, charged with fixing the Java Memory Model. Last month, I talked about some of those holes; this month, I'll talk about how they've been plugged.

Visibility, revisited

One of the key concepts needed to understand the JMM is that of *visibility* -- how do you know that if thread A executes `someVariable = 3`, other threads will see the value 3 written there by thread A? A number of reasons exist for why another thread might not immediately see the value 3 for `someVariable`: it could be because the compiler has reordered instructions in order to execute more efficiently, or that `someVariable` was cached in a register, or that its value was written to the cache on the writing processor but not yet flushed to main memory, or that there is an old (or stale) value in the reading processor's cache. It is the memory model that determines when a thread can reliably "see" writes to variables made by other threads. In particular, the memory model defines

semantics for `volatile`, `synchronized`, and `final` that make guarantees of visibility of memory operations across threads.

Don't miss the rest of this series

Part 1, "What is the Java Memory Model, and how was it broken in the first place?" (February 2004)

When a thread exits a synchronized block as part of releasing the associated monitor, the JMM requires that the local processor cache be flushed to main memory. (Actually, the memory model does not talk about caches -- it talks about an abstraction, *local memory*, which encompasses caches, registers, and other hardware and compiler optimizations.) Similarly, as part of acquiring the monitor when entering a synchronized block, local caches are invalidated so that subsequent reads will go directly to main memory and not the local cache. This process guarantees that when a variable is written by one thread during a synchronized block protected by a given monitor and read by another thread during a synchronized block protected by the same monitor, the write to the variable will be visible by the reading thread. The JMM does not make this guarantee in the absence of synchronization -- which is why synchronization (or its younger sibling, `volatile`) must be used whenever multiple threads are accessing the same variables.

New guarantees for volatile

The original semantics for `volatile` guaranteed only that reads and writes of `volatile` fields would be made directly to main memory, instead of to registers or the local processor cache, and that actions on volatile variables on behalf of a thread are performed in the order that the thread requested. In other words, this means that the old memory model made promises only about the visibility of the variable being read or written, and no promises about the visibility of writes to other variables. While this was easier to implement efficiently, it turned out to be less useful than initially thought.

While reads and writes of volatile variables could not be reordered with reads and writes of other volatile variables, they could still be reordered with reads and writes of nonvolatile variables. In Part 1, you learned how the code in Listing 1 was not sufficient (under the old memory model) to guarantee that the correct value for `configOptions` and all the variables reachable indirectly through `configOptions` (such as the elements of the `Map`) would be visible to thread B, because the initialization of `configOptions` could have been reordered with the initialization of the volatile `initialized` variable.

Listing 1. Using a volatile variable as a "guard"

```
Map configOptions;
char[] configText;
volatile boolean initialized = false;

// In Thread A
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText, configOptions);
initialized = true;

// In Thread B
while (!initialized)
  sleep();
// use configOptions
```

Unfortunately, this situation is a common use case for volatile -- using a volatile field as a "guard" to indicate that a set of shared variables had been initialized. The JSR 133 Expert Group decided that it would be more sensible for volatile reads and writes not to be reorderable with any other memory operations -- to support precisely this and other similar use cases. Under the new memory model, when thread A writes to a volatile variable V, and thread B reads from V, any variable values that were visible to A at the time that V was written are guaranteed now to be visible to B. The result is a more useful semantics of `volatile`, at the cost of a somewhat higher performance penalty for accessing volatile fields.

What happens before what?

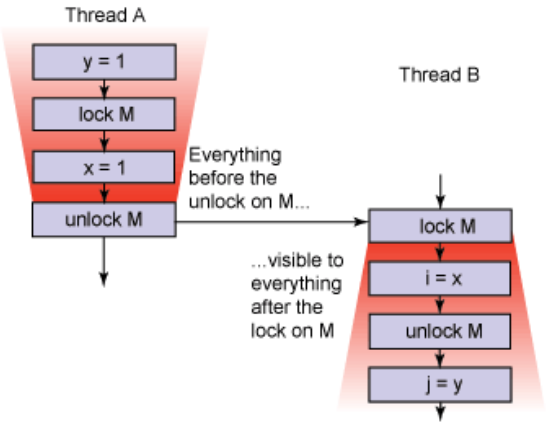
Actions, such as reads and writes of variables, are ordered within a thread according to what is called the "program order" -- the order in which the semantics of the program say they should occur. (The compiler is actually free to take some liberties with the program order within a thread as long as as-if-serial semantics are preserved.) Actions in different threads are not necessarily ordered with respect to each other at all -- if you start two threads and they each execute without synchronizing on any common monitors or touching any common volatile variables, you can predict exactly *nothing* about the relative order in which actions in one thread will execute (or become visible to a third thread) with respect to actions in the other thread.

Additional ordering guarantees are created when a thread is started, a thread is joined with another thread, a thread acquires or releases a monitor (enters or exits a synchronized block), or a thread accesses a volatile variable. The JMM describes the ordering guarantees that are made when a program uses synchronization or volatile variables to coordinate activities in multiple threads. The new JMM, informally, defines an ordering called *happens-before*, which is a partial ordering of all actions within a program, as follows:

- Each action in a thread *happens-before* every action in that thread that comes later in the program order
- An unlock on a monitor *happens-before* every subsequent lock on that same monitor
- A write to a volatile field *happens-before* every subsequent read of that same volatile
- A call to `Thread.start()` on a thread *happens-before* any actions in the started thread
- All actions in a thread *happen-before* any other thread successfully returns from a `Thread.join()` on that thread

It is the third of these rules, the one governing reads and writes of volatile variables, that is new and fixes the problem with the example in Listing 1. Because the write of the volatile variable `initialized` happens after the initialization of `configOptions`, the use of `configOptions` happens after the read of `initialized`, and the read of `initialized` happens after the write to `initialized`, you can conclude that the initialization of `configOptions` by thread A happens before the use of `configOptions` by thread B. Therefore, `configOptions` and the variables reachable through it will be visible to thread B.

Figure1. Using synchronization to guarantee visibility of memory writes across threads



Data races

A program is said to have a *data race*, and therefore not be a "properly synchronized" program, when there is a variable that is read by more than one thread, written by at least one thread, and the write and the reads are not ordered by a happens-before relationship.

Does this fix the double-checked locking problem?

One of the proposed fixes to the double-checked locking problem was to make the field that holds the lazily initialized instance a volatile field. (See Related topics for a description of the double-checked locking problem and an an explanation of why the proposed algorithmic fixes don't work.) Under the old memory model, this did not render double-checked locking thread-safe, because writes to a volatile field could still be reordered with writes to other nonvolatile fields (such as the fields of the newly constructed object), and therefore the volatile instance reference could still hold a reference to an incompletely constructed object.

Under the new memory model, this "fix" to double-checked locking renders the idiom thread-safe. But that still doesn't mean that you should use this idiom! The whole point of double-checked locking was that it was supposed to be a performance optimization, designed to eliminate synchronization on the common code path, largely because synchronization was relatively expensive in very early JDKs. Not only has uncontended synchronization gotten *a lot* cheaper since then, but the new changes to the semantics of `volatile` make it relatively more expensive than the old semantics on some platforms. (Effectively, each read or write to a volatile field is

like "half" a synchronization -- a read of a volatile has the same memory semantics as a monitor acquire, and a write of a volatile has the same semantics as a monitor release.) So if the goal of double-checked locking is supposed to offer improved performance over a more straightforward synchronized approach, this "fixed" version doesn't help very much either.

Instead of double-checked locking, use the Initialize-on-demand Holder Class idiom, which provides lazy initialization, is thread-safe, and is faster and less confusing than double-checked locking:

Listing 2. The Initialize-On-Demand Holder Class idiom

```
private static class LazySomethingHolder {
  public static Something something = new Something();
}

...

public static Something getInstance() {
  return LazySomethingHolder.something;
}
```

This idiom derives its thread safety from the fact that operations that are part of class initialization, such as static initializers, are guaranteed to be visible to all threads that use that class, and its lazy initialization from the fact that the inner class is not loaded until some thread references one of its fields or methods.

Initialization safety

The new JMM also seeks to provide a new guarantee of *initialization safety* -- that as long as an object is properly constructed (meaning that a reference to the object is not published before the constructor has completed), then all threads will see the values for its final fields that were set in its constructor, regardless of whether or not synchronization is used to pass the reference from one thread to another. Further, any variables that can be reached through a final field of a properly constructed object, such as fields of an object referenced by a final field, are also guaranteed to be visible to other threads as well. This means that if a final field contains a reference to, say, a `LinkedList`, in addition to the correct value of the reference being visible to other threads, also the contents of that `LinkedList` at construction time would be visible to other threads without synchronization. The result is a significant strengthening of the meaning of `final` -- that final fields can be safely accessed without synchronization, and that compilers can assume that final fields will not change and can therefore optimize away multiple fetches.

Final means final

The mechanism by which final fields could appear to change their value under the old memory model was outlined in Part 1 -- in the absence of synchronization, another thread could first see the default value for a final field and then later see the correct value.

Under the new memory model, there is something similar to a happens-before relationship between the write of a final field in a constructor and the initial load of a shared reference to that

object in another thread. When the constructor completes, all of the writes to final fields (and to variables reachable indirectly through those final fields) become "frozen," and any thread that obtains a reference to that object after the freeze is guaranteed to see the frozen values for all frozen fields. Writes that initialize final fields will not be reordered with operations following the freeze associated with the constructor.

Summary

JSR 133 significantly strengthens the semantics of `volatile`, so that volatile flags can be used reliably as indicators that the program state has been changed by another thread. As a result of making volatile more "heavyweight," the performance cost of using volatile has been brought closer to the performance cost of synchronization in some cases, but the performance cost is still quite low on most platforms. JSR 133 also significantly strengthens the semantics of `final`. If an object's reference is not allowed to escape during construction, then once a constructor has completed and a thread publishes a reference to an object, that object's final fields are guaranteed to be visible, correct, and constant to all other threads without synchronization.

These changes greatly strengthen the utility of immutable objects in concurrent programs; immutable objects finally become inherently thread-safe (as they were intended to be all along), even if a data race is used to pass references to the immutable object between threads.

The one caveat with initialization safety is that the object's reference must not "escape" its constructor -- the constructor should not publish, directly or indirectly, a reference to the object being constructed. This includes publishing references to nonstatic inner classes, and generally precludes starting threads from within a constructor. For a more detailed explanation of safe construction, see Related topics.

Related topics

- Bill Pugh, who originally discovered many of the problems with the Java Memory Model, maintains a Java Memory Model page.
- The issues with the old memory model and a summary of the new memory model semantics can be found in the JSR 133 FAQ.
- Read more about the double-checked locking problem, and why the obvious attempts to fix it don't work.
- Read more about why you don't want to let a reference to an object escape during construction.
- JSR 133, charged with revising the JMM, was convened under the Java Community Process. JSR 133 has recently released its public review specification.
- If you want to see how specifications like this are made, browse the JMM mailing list archive.
- *Concurrent Programming in Java* by Doug Lea (Addison-Wesley, 1999) is a masterful book on the subtle issues surrounding multithreaded Java programming.
- Synchronization and the Java Memory Model summarizes the actual meaning of synchronization.
- Chapter 17 of *The Java Language Specification* by James Gosling, Bill Joy, Guy Steele, and Gilad Bracha (Addison-Wesley, 1996) covers the gory details of the original Java Memory Model.