Java theory and practice: Fixing the Java Memory Model, Part 1

What is the Java Memory Model, and how was it broken in the first place?

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix Corp

24 February 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). Several serious flaws were found in the original JMM, resulting in some surprisingly difficult semantics for concepts that were supposed to be simple, like `volatile`, `final`, and `synchronized`. In this installment of *Java theory and practice*, Brian Goetz shows how the semantics of `volatile` and `final` will be strengthened in order to fix the JMM. Some of these changes have already been integrated in JDK 1.4; others are slated for inclusion in JDK 1.5.

View more content in this series

The Java platform has integrated threading and multiprocessing into the language to a much greater degree than most previous programming languages. The language's support for platform-independent concurrency and multithreading was ambitious and pioneering, and, perhaps not surprisingly, the problem was a bit harder than the Java architects originally thought. Underlying much of the confusion surrounding synchronization and thread safety are the non-intuitive subtleties of the Java Memory Model (JMM), originally specified in Chapter 17 of the Java Language Specification, and re-specified by JSR 133.

For example, not all multiprocessor systems exhibit *cache coherency*; if one processor has an updated value of a variable in its cache, but one which has not yet been flushed to main memory, other processors may not see that update. In the absence of cache coherency, two different processors may see two different values for the same location in memory. This may sound scary, but it is by design -- it is a means of obtaining higher performance and scalability -- but it places a burden on developers and compilers to create code that accommodates these issues.

What is a memory model, and why do I need one?

A memory model describes the relationship between variables in a program (instance fields, static fields, and array elements) and the low-level details of storing them to and retrieving them from memory in a real computer system. Objects are ultimately stored in memory, but the compiler, runtime, processor, or cache may take some liberties with the timing of moving values to or from a variable's assigned memory location. For example, a compiler may choose to optimize a loop index variable by storing it in a register, or the cache may delay flushing a new value of a variable to main memory until a more opportune time. All of these optimizations are to aid in higher performance, and are generally transparent to the user, but on multiprocessor systems, these complexities may sometimes show through.

Don't miss the rest of this series

Part 2, "How will the JMM change under JSR 133" (March 2004)

The JMM allows the compiler and cache to take significant liberties with the order in which data is moved between a processor-specific cache (or register) and main memory, unless the programmer has explicitly asked for certain visibility guarantees using `synchronized` or `volatile`. This means that in the absence of synchronization, memory operations can appear to happen in different orders from the perspective of different threads.

By contrast, languages like C and C++ do not have explicit memory models -- C programs instead inherit the memory model of the processor executing the program (although the compiler for a given architecture probably does know something about the memory model of the underlying processor, and some of the responsibility for compliance falls to the compiler). This means that concurrent C programs may run correctly on one processor architecture, but not another. While the JMM may be confusing at first, there is a significant benefit to it -- a program that is correctly synchronized according to the JMM should run correctly on any Java-enabled platform.

Shortcomings of the original JMM

While the JMM specified in chapter 17 of the *Java Language Specification* was an ambitious attempt to define a consistent, cross-platform memory model, it had some subtle but significant failings. The semantics of `synchronized` and `volatile` were quite confusing, so much so that many knowledgeable developers chose to sometimes ignore the rules because writing properly synchronized code under the old memory model was so difficult.

The old JMM allowed some surprising and confusing things to happen, such as final fields appearing not to have the value that was set in the constructor (thus making supposedly immutable objects not immutable) and unexpected results with memory operation reordering. It also prevented some otherwise effective compiler optimizations. If you've read any of the articles on the double-checked locking problem (see Resources), you'll recall how confusing memory operation reordering can be, and how subtle but serious problems can sneak into your code when you don't properly synchronize (or actively try to avoid synchronizing). Worse, many incorrectly synchronized programs appear to work correctly in some situations, such as under light load, on uniprocessor systems, or on processors with stronger memory models than required by the JMM.

The term reordering is used to describe several classes of actual and apparent reorderings of memory operations:

- The compiler is free to reorder certain instructions as an optimization when it would not change the semantics of the program.
- The processor is allowed to execute operations out of order under some circumstances.
- The cache is generally allowed to write variables back to main memory in a different order than they were written by the program.

Any of these conditions could cause operations to appear, from the perspective of another thread, to occur in a different order than specified by the program -- and regardless of the source of the reordering, all are considered equivalent by the memory model.

Goals of JSR 133

JSR 133, chartered to fix the JMM, has several goals:

- Preserve existing safety guarantees, including type-safety.
- Provide *out-of-thin-air safety*. This means that variable values are not created "out of thin air" -- so for a thread to observe a variable to have value X, some thread must have actually written the value X to that variable in the past.
- The semantics of "correctly synchronized" programs should be as simple and intuitive as feasible. Thus, "correctly synchronized" should be defined both formally and intuitively (and the two definitions should be consistent with each other!).
- Programmers should be able to create multithreaded programs with confidence that they will be reliable and correct. Of course, there is no magic that makes writing concurrent applications easy, but the goal is to relieve application writers of the burden of understanding all the subtleties of the memory model.
- High performance JVM implementations across a wide range of popular hardware architectures should be possible. Modern processors differ substantially in their memory models; the JMM should accommodate as many possible architectures as practical, without sacrificing performance.
- Provide a synchronization idiom that allows us to publish an object and have it be visible without synchronization. This is a new safety guarantee called *initialization safety*.
- There should be minimal impact on existing code.

It is worth noting that broken techniques like double-checked locking are still broken under the new memory model, and that "fixing" double-checked locking was not one of the goals of the new memory model effort. (However, the new semantics of `volatile` allow one of the commonly proposed alternatives to double-checked locking to work correctly, although the technique is still discouraged.)

In the three years since the JSR 133 process has been active, it has been discovered that these issues are far more subtle than anyone gave them credit for. Such is the price of being a pioneer! The final formal semantics are more complicated than originally expected, and in fact took quite a different form than initially envisioned, but the informal semantics are clear and intuitive and will be outlined in Part 2 of this article.

Synchronization and visibility

Most programmers know that the `synchronized` keyword enforces a mutex (mutual exclusion) that prevents more than one thread at a time from entering a synchronized block protected by a given monitor. But synchronization also has another aspect: It enforces certain memory visibility rules as specified by the JMM. It ensures that caches are flushed when exiting a synchronized block and invalidated when entering one, so that a value written by one thread during a synchronized block protected by a given monitor is visible to any other thread executing a synchronized block protected by that same monitor. It also ensures that the compiler does not move instructions from inside a synchronized block to outside (although it can in some cases move instructions from outside a synchronized block inside). 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.

Problem #1: Immutable objects weren't

One of the most surprising failings of the JMM is that immutable objects, whose immutability was intended to be guaranteed through use of the `final` keyword, could appear to change their value. (Public Service Reminder: Making all fields of an object `final` does not necessarily make the object immutable -- all fields must *also* be primitive types or references to immutable objects.) Immutable objects, like `String`, are supposed to require no synchronization. However, because of potential delays in propagating changes in memory writes from one thread to another, there is a possible race condition that would allow a thread to first see one value for an immutable object, and then at some later time see a different value.

How can this happen? Consider the implementation of `String` in the Sun 1.4 JDK, where there are basically three important final fields: a reference to a character array, a length, and an offset into the character array that describes the start of the string being represented. `String` was implemented this way, instead of having only the character array, so character arrays could be shared among multiple `String` and `StringBuffer` objects without having to copy the text into a new array every time a `String` is created. For example, `String.substring()` creates a new string that shares the same character array with the original `String` and merely differs in the length and offset fields.

Suppose you execute the following code:

```
String s1 = "/usr/tmp";
String s2 = s1.substring(4);   // contains "/tmp"
```

The string `s2` will have an offset of 4 and a length of 4, but will share the same character array, the one containing `"/usr/tmp"`, with `s1`. Before the `String` constructor runs, the constructor for `Object` will initialize all fields, including the final length and offset fields, with their default values. When the `String` constructor runs, the length and offset are then set to their desired values. But under the old memory model, in the absence of synchronization, it is possible for another thread to temporarily see the offset field as having the default value of 0, and then later see the correct value of 4. The effect is that the value of `s2` changes from `"/usr"` to `"/tmp"`. This is not what was

intended, and it might not be possible on all JVMs or platforms, but it *was* allowed by the old memory model specification.

Problem #2: Reordering volatile and nonvolatile stores

The other major area where the existing JMM caused some very confusing results was with memory operation reordering on `volatile` fields. The existing JMM says that volatile reads and writes are to go directly to main memory, prohibiting caching values in registers and bypassing processor-specific caches. This allows multiple threads to always see the most up-to-date value for a given variable. However, it turns out that this definition of `volatile` was not as useful as first intended, and resulted in significant confusion on the actual meaning of `volatile`.

In order to provide good performance in the absence of synchronization, the compiler, runtime, and cache are generally allowed to reorder ordinary memory operations as long as the currently executing thread cannot tell the difference. (This is referred to as *within-thread as-if-serial semantics*.) Volatile reads and writes, however, are totally ordered across threads; the compiler or cache cannot reorder volatile reads and writes with each other. Unfortunately, the JMM did allow volatile reads and writes to be reordered with respect to ordinary variable reads and writes, meaning that we cannot use volatile flags as an indication of what operations have been completed. Consider the following code, where the intention is that the volatile field `initialized` is supposed to indicate that initialization has been completed:

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

```
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
```

The idea here is that the volatile variable `initialized` acts as a guard to indicate that a set of other operations have completed. It's a good idea, but under the old JMM it didn't work, because the old JMM allowed nonvolatile writes (such as the write to the `configOptions` field, as well as the writes to the fields of the `Map` referenced by `configOptions`) to be reordered with volatile writes. So another thread might see `initialized` as true, but not yet have a consistent or current view of the field `configOptions` or the objects it references. The old semantics of `volatile` only made promises about the visibility of the variable being read or written, and no promises about other

variables. While this approach was easier to implement efficiently, it turned out to be less useful than initially thought.

Summary

As specified in Chapter 17 of the *Java Language Specification*, the JMM has some serious flaws that allow some unintuitive and undesirable things to happen to reasonable-looking programs. If it is too difficult to write concurrent classes properly, then we are guaranteed that many concurrent classes will not work as expected, and that this is a flaw in the platform. Fortunately, it was possible to create a memory model that was more consistent with most developer's intuition while not breaking any code that was properly synchronized under the old memory model, and the JSR 133 process has done just that. Next month, we'll look at the details of the new memory model (much of which is already built into the 1.4 JDK).

Resources

- 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 is a masterful book on the subtle issues surrounding multithreaded programming in Java.
- Synchronization and the Java Memory Model summarizes the actual meaning of synchronization.
- The Java Language Specification, Chapter 17, covers the gory details of the original Java Memory Model.
- Find hundreds more Java technology resources on the *developerWorks* Java technology zone.
- Browse for books on these and other technical topics.

About the author

Brian Goetz

Brian Goetz has been a professional software developer for over 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.