JSR 133 (Java Memory Model) FAQ

Jeremy Manson and Brian Goetz, February 2004

Table of Contents

What is a memory model, anyway?

In multiprocessor systems, processors generally have one or more layers of memory cache, which improves performance both by speeding access to data (because the data is closer to the processor) and reducing traffic on the shared memory bus (because many memory operations can be satisfied by local caches.) Memory caches can improve performance tremendously, but they present a host of new challenges. What, for example, happens when two processors examine the same memory location at the same time? Under what conditions will they see the same value?

At the processor level, a memory model defines necessary and sufficient conditions for knowing that writes to memory by other processors are visible to the current processor, and writes by the current processor are visible to other processors. Some processors exhibit a strong memory model, where all processors see exactly the same value for any given memory location at all times. Other processors exhibit a weaker memory model, where special instructions, called memory barriers, are required to flush or invalidate the local processor cache in order to see writes made by other processors or make writes by this processor visible to others. These memory barriers are usually performed when lock and unlock actions are taken; they are invisible to programmers in a high level language.

It can sometimes be easier to write programs for strong memory models, because of the reduced need for memory barriers. However, even on some of the strongest memory models, memory barriers are often necessary; quite frequently their placement is counterintuitive. Recent trends in processor design have encouraged weaker memory models, because the relaxations they make for cache consistency allow for greater scalability across multiple processors and larger amounts of memory.

The issue of when a write becomes visible to another thread is compounded by the compiler's reordering of code. For example, the compiler might decide that it is more efficient to move a write operation later in the program; as long as this code motion does

not change the program's semantics, it is free to do so. If a compiler defers an operation, another thread will not see it until it is performed; this mirrors the effect of caching.

Moreover, writes to memory can be moved earlier in a program; in this case, other threads might see a write before it actually "occurs" in the program. All of this flexibility is by design -- by giving the compiler, runtime, or hardware the flexibility to execute operations in the optimal order, within the bounds of the memory model, we can achieve higher performance.

A simple example of this can be seen in the following code:

```
Class Reordering {
  int x = 0, y = 0;
  public void writer() {
    x = 1;
    y = 2;
  }

  public void reader() {
    int r1 = y;
    int r2 = x;
  }
}
```

Let's say that this code is executed in two threads concurrently, and the read of y sees the value 2. Because this write came after the write to x, the programmer might assume that the read of x must see the value 1. However, the writes may have been reordered. If this takes place, then the write to y could happen, the reads of both variables could follow, and then the write to x could take place. The result would be that r1 has the value 2, but r2 has the value 0.

The Java Memory Model describes what behaviors are legal in multithreaded code, and how threads may interact through memory. It describes the relationship between variables in a program and the low-level details of storing and retrieving them to and from memory or registers in a real computer system. It does this in a way that can be implemented correctly using a wide variety of hardware and a wide variety of compiler optimizations.

Java includes several language constructs, including volatile, final, and synchronized, which are intended to help the programmer describe a program's concurrency requirements to the compiler. The Java Memory Model defines the behavior of volatile and synchronized, and, more importantly, ensures that a correctly synchronized Java program runs correctly on all processor architectures.

Do other languages, like C++, have a memory model?

Most other programming languages, such as C and C++, were not designed with direct support for multithreading. The protections that these languages offer against the kinds of reorderings that take place in compilers and architectures are heavily dependent on the guarantees provided by the threading libraries used (such as pthreads), the compiler used, and the platform on which the code is run.

What is JSR 133 about?

Since 1997, several serious flaws have been discovered in the Java Memory Model as defined in Chapter 17 of the Java Language Specification. These flaws allowed for confusing behaviors (such as final fields being observed to change their value) and undermined the compiler's ability to perform common optimizations.

The Java Memory Model was an ambitious undertaking; it was the first time that a programming language specification attempted to incorporate a memory model which could provide consistent semantics for concurrency across a variety of architectures. Unfortunately, defining a memory model which is both consistent and intuitive proved far more difficult than expected. JSR 133 defines a new memory model for the Java language which fixes the flaws of the earlier memory model. In order to do this, the semantics of final and volatile needed to change.

The full semantics are available at http://www.cs.umd.edu/users/pugh/java/memoryModel, but the formal semantics are not for the timid. It is surprising, and sobering, to discover how complicated seemingly simple concepts like synchronization really are. Fortunately, you need not understand the details of the formal semantics -- the goal of JSR 133 was to create a set of formal semantics that provides an intuitive framework for how volatile, synchronized, and final work.

The goals of JSR 133 include:

- Preserving existing safety guarantees, like type-safety, and strengthening others. For example, variable values may not be created "out of thin air": each value for a variable observed by some thread must be a value that can reasonably be placed there by some thread.
- The semantics of correctly synchronized programs should be as simple and intuitive as possible.
- The semantics of incompletely or incorrectly synchronized programs should be defined so that potential security hazards are minimized.
- Programmers should be able to reason confidently about how multithreaded programs interact with memory.
- It should be possible to design correct, high performance JVM implementations across a wide range of popular hardware architectures.
- A new guarantee of initialization safety should be provided. If an object is properly constructed (which means that references to it do not escape during construction), then all threads which see a reference to that object will also see the values for its final fields that were set in the constructor, without the need for synchronization.
- There should be minimal impact on existing code.

What is meant by reordering?

There are a number of cases in which accesses to program variables (object instance fields, class static fields, and array elements) may appear to execute in a different order than was specified by the program. The compiler is free to take liberties with the ordering of instructions in the name of optimization. Processors may execute instructions out of order under certain circumstances. Data may be moved between registers, processor caches, and main memory in different order than specified by the program.

For example, if a thread writes to field `a` and then to field `b`, and the value of `b` does not depend on the value of `a`, then the compiler is free to reorder these operations, and the

cache is free to flush `b` to main memory before `a`. There are a number of potential sources of reordering, such as the compiler, the JIT, and the cache.

The compiler, runtime, and hardware are supposed to conspire to create the illusion of as-if-serial semantics, which means that in a single-threaded program, the program should not be able to observe the effects of reorderings. However, reorderings can come into play in incorrectly synchronized multithreaded programs, where one thread is able to observe the effects of other threads, and may be able to detect that variable accesses become visible to other threads in a different order than executed or specified in the program.

Most of the time, one thread doesn't care what the other is doing. But when it does, that's what synchronization is for.

What was wrong with the old memory model?

There were several serious problems with the old memory model. It was difficult to understand, and therefore widely violated. For example, the old model did not, in many cases, allow the kinds of reorderings that took place in every JVM. This confusion about the implications of the old model was what compelled the formation of JSR-133.

One widely held belief, for example, was that if final fields were used, then synchronization between threads was unnecessary to guarantee another thread would see the value of the field. While this is a reasonable assumption and a sensible behavior, and indeed how we would want things to work, under the old memory model, it was simply not true. Nothing in the old memory model treated final fields differently from any other field -- meaning synchronization was the only way to ensure that all threads see the value of a final field that was written by the constructor. As a result, it was possible for a thread to see the default value of the field, and then at some later time see its constructed value. This means, for example, that immutable objects like String can appear to change their value -- a disturbing prospect indeed.

The old memory model allowed for volatile writes to be reordered with nonvolatile reads and writes, which was not consistent with most developers intuitions about volatile and therefore caused confusion.

Finally, as we shall see, programmers' intuitions about what can occur when their programs are incorrectly synchronized are often mistaken. One of the goals of JSR-133 is to call attention to this fact.

What do you mean by “incorrectly synchronized”?

Incorrectly synchronized code can mean different things to different people. When we talk about incorrectly synchronized code in the context of the Java Memory Model, we mean any code where

1. there is a write of a variable by one thread,
2. there is a read of the same variable by another thread and
3. the write and read are not ordered by synchronization

When these rules are violated, we say we have a data race on that variable. A program with a data race is an incorrectly synchronized program.

What does synchronization do?

Synchronization has several aspects. The most well-understood is mutual exclusion -- only one thread can hold a monitor at once, so synchronizing on a monitor means that once one thread enters a synchronized block protected by a monitor, no other thread can enter a block protected by that monitor until the first thread exits the synchronized block.

But there is more to synchronization than mutual exclusion. Synchronization ensures that memory writes by a thread before or during a synchronized block are made visible in a predictable manner to other threads which synchronize on the same monitor. After we exit a synchronized block, we **release** the monitor, which has the effect of flushing the cache to main memory, so that writes made by this thread can be visible to other threads. Before we can enter a synchronized block, we **acquire** the monitor, which has the effect of invalidating the local processor cache so that variables will be reloaded from main memory. We will then be able to see all of the writes made visible by the previous release.

Discussing this in terms of caches, it may sound as if these issues only affect multiprocessor machines. However, the reordering effects can be easily seen on a single processor. It is not possible, for example, for the compiler to move your code before an acquire or after a release. When we say that acquires and releases act on caches, we are using shorthand for a number of possible effects.

The new memory model semantics create a partial ordering on memory operations (read field, write field, lock, unlock) and other thread operations (start and join), where some actions are said to happen before other operations. When one action happens before another, the first is guaranteed to be ordered before and visible to the second. The rules of this ordering are as follows:

- Each action in a thread happens before every action in that thread that comes later in the program's 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 `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 `join()` on that thread.

This means that any memory operations which were visible to a thread before exiting a synchronized block are visible to any thread after it enters a synchronized block protected by the same monitor, since all the memory operations happen before the release, and the release happens before the acquire.

Another implication is that the following pattern, which some people use to force a memory barrier, doesn't work:

```
synchronized (new Object()) {}
```

This is actually a no-op, and your compiler can remove it entirely, because the compiler knows that no other thread will synchronize on the same monitor. You have to set up a happens-before relationship for one thread to see the results of another.

Important Note: Note that it is important for both threads to synchronize on the same monitor in order to set up the happens-before relationship properly. It is not the case that everything visible to thread A when it synchronizes on object X becomes visible to thread B after it synchronizes on object Y. The release and acquire have to "match" (i.e., be performed on the same monitor) to have the right semantics. Otherwise, the code has a data race.

How can final fields appear to change their values?

One of the best examples of how final fields' values can be seen to change involves one particular implementation of the `String` class.

A `String` can be implemented as an object with three fields -- a character array, an offset into that array, and a length. The rationale for implementing `String` this way, instead of having only the character array, is that it lets multiple `String` and `StringBuffer` objects share the same character array and avoid additional object allocation and copying. So, for example, the method `String.substring()` can be implemented by creating a new string which shares the same character array with the original `String` and merely differs in the length and offset fields. For a `String`, these fields are all final fields.

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

The string `s2` will have an offset of 4 and a length of 4. But, under the old model, it was possible for another thread to see the offset as having the default value of 0, and then later see the correct value of 4, it will appear as if the string "/usr" changes to "/tmp".

The original Java Memory Model allowed this behavior; several JVMs have exhibited this behavior. The new Java Memory Model makes this illegal.

How do final fields work under the new JMM?

The values for an object's final fields are set in its constructor. Assuming the object is constructed "correctly", once an object is constructed, the values assigned to the final fields in the constructor will be visible to all other threads without synchronization. In addition, the visible values for any other object or array referenced by those final fields will be at least as up-to-date as the final fields.

What does it mean for an object to be properly constructed? It simply means that no reference to the object being constructed is allowed to "escape" during construction. (See Safe Construction Techniques for examples.) In other words, do not place a reference to the object being constructed anywhere where another thread might be able to see it; do not assign it to a static field, do not register it as a listener with any other object, and so on. These tasks should be done after the constructor completes, not in the constructor.

```
class FinalFieldExample {
  final int x;
  int y;
  static FinalFieldExample f;
  public FinalFieldExample() {
    x = 3;
    y = 4;
  }
```

```
    static void writer() {
      f = new FinalFieldExample();
    }

    static void reader() {
      if (f != null) {
        int i = f.x;
        int j = f.y;
      }
    }
  }
```

The class above is an example of how final fields should be used. A thread executing `reader` is guaranteed to see the value 3 for `f.x`, because it is final. It is not guaranteed to see the value 4 for `y`, because it is not final. If `FinalFieldExample`'s constructor looked like this:

```
  public FinalFieldExample() { // bad!
    x = 3;
    y = 4;
    // bad construction - allowing this to escape
    global.obj = this;
  }
```

then threads that read the reference to `this` from `global.obj` are **not** guaranteed to see 3 for `x`.

The ability to see the correctly constructed value for the field is nice, but if the field itself is a reference, then you also want your code to see the up to date values for the object (or array) to which it points. If your field is a final field, this is also guaranteed. So, you can have a final pointer to an array and not have to worry about other threads seeing the correct values for the array reference, but incorrect values for the contents of the array. Again, by "correct" here, we mean "up to date as of the end of the object's constructor", not "the latest value available".

Now, having said all of this, if, after a thread constructs an immutable object (that is, an object that only contains final fields), you want to ensure that it is seen correctly by all of the other thread, you **still** typically need to use synchronization. There is no other way to ensure, for example, that the reference to the immutable object will be seen by the second thread. The guarantees the program gets from final fields should be carefully tempered with a deep and careful understanding of how concurrency is managed in your code.

There is no defined behavior if you want to use JNI to change final fields.

What does volatile do?

Volatile fields are special fields which are used for communicating state between threads. Each read of a volatile will see the last write to that volatile by any thread; in effect, they are designated by the programmer as fields for which it is never acceptable to see a "stale" value as a result of caching or reordering. The compiler and runtime are prohibited from allocating them in registers. They must also ensure that after they are written, they are flushed out of the cache to main memory, so they can immediately become visible to other threads. Similarly, before a volatile field is read, the cache must be invalidated so that the value in main memory, not the local processor cache, is the one seen. There are also additional restrictions on reordering accesses to volatile variables.

Under the old memory model, accesses to volatile variables could not be reordered with each other, but they could be reordered with nonvolatile variable accesses. This undermined the usefulness of volatile fields as a means of signaling conditions from one thread to another.

Under the new memory model, it is still true that volatile variables cannot be reordered with each other. The difference is that it is now no longer so easy to reorder normal field accesses around them. Writing to a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire. In effect, because the new memory model places stricter constraints on reordering of volatile field accesses with other field accesses, volatile or not, anything that was visible to thread A when it writes to volatile field `f` becomes visible to thread B when it reads `f`.

Here is a simple example of how volatile fields can be used:

```
class VolatileExample {
  int x = 0;
  volatile boolean v = false;
  public void writer() {
    x = 42;
    v = true;
  }

  public void reader() {
    if (v == true) {
      //uses x - guaranteed to see 42.
    }
  }
}
```

Assume that one thread is calling `writer`, and another is calling `reader`. The write to `v` in `writer` releases the write to `x` to memory, and the read of `v` acquires that value from memory. Thus, if the reader sees the value `true` for v, it is also guaranteed to see the write to 42 that happened before it. This would not have been true under the old memory model. If `v` were not volatile, then the compiler could reorder the writes in `writer`, and `reader`'s read of `x` might see 0.

Effectively, the semantics of volatile have been strengthened substantially, almost to the level of synchronization. Each read or write of a volatile field acts like "half" a synchronization, for purposes of visibility.

Important Note: Note that it is important for both threads to access the same volatile variable in order to properly set up the happens-before relationship. It is not the case that everything visible to thread A when it writes volatile field `f` becomes visible to thread B after it reads volatile field `g`. The release and acquire have to "match" (i.e., be performed on the same volatile field) to have the right semantics.

Does the new memory model fix the "double-checked locking" problem?

The (infamous) double-checked locking idiom (also called the multithreaded singleton pattern) is a trick designed to support lazy initialization while avoiding the overhead of synchronization. In very early JVMs, synchronization was slow, and developers were eager to remove it -- perhaps too eager. The double-checked locking idiom looks like this:

```
// double-checked-locking - don't do this!
```

```
private static Something instance = null;

public Something getInstance() {
  if (instance == null) {
    synchronized (this) {
      if (instance == null)
        instance = new Something();
    }
  }
  return instance;
}
```

This looks awfully clever -- the synchronization is avoided on the common code path. There's only one problem with it -- **it doesn't work**. Why not? The most obvious reason is that the writes which initialize `instance` and the write to the `instance` field can be reordered by the compiler or the cache, which would have the effect of returning what appears to be a partially constructed `Something`. The result would be that we read an uninitialized object. There are lots of other reasons why this is wrong, and why algorithmic corrections to it are wrong. There is no way to fix it using the old Java memory model. More in-depth information can be found at Double-checked locking: Clever, but broken and The "Double Checked Locking is broken" declaration

Many people assumed that the use of the `volatile` keyword would eliminate the problems that arise when trying to use the double-checked-locking pattern. In JVMs prior to 1.5, `volatile` would not ensure that it worked (your mileage may vary). Under the new memory model, making the `instance` field volatile will "fix" the problems with double-checked locking, because then there will be a happens-before relationship between the initialization of the `Something` by the constructing thread and the return of its value by the thread that reads it.

~~However, for fans of double-checked locking (and we really hope there are none left), the news is still not good. The whole point of double-checked locking was to avoid the performance overhead of synchronization. Not only has brief synchronization gotten a LOT less expensive since the Java 1.0 days, but under the new memory model, the performance cost of using volatile goes up, almost to the level of the cost of synchronization. So there's still no good reason to use double-checked-locking.~~ Redacted -- volatiles are cheap on most platforms.

Instead, use the Initialization On Demand Holder idiom, which is thread-safe and a lot easier to understand:

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

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

This code is guaranteed to be correct because of the initialization guarantees for static fields; if a field is set in a static initializer, it is guaranteed to be made visible, correctly, to any thread that accesses that class.

What if I'm writing a VM?

You should look at http://gee.cs.oswego.edu/dl/jmm/cookbook.html .

Why should I care?

Why should you care? Concurrency bugs are very difficult to debug. They often don't appear in testing, waiting instead until your program is run under heavy load, and are hard to reproduce and trap. You are much better off spending the extra effort ahead of time to ensure that your program is properly synchronized; while this is not easy, it's a lot easier than trying to debug a badly synchronized application.