

The evolution of the Java Memory Model

Lee Provoost

Software Technology Group

Department of Information and Computing Sciences

University of Utrecht, P.O. Box 80089 3508 TB, Utrecht, The Netherlands

{l.provoost} @students.uu.nl

Abstract

The Java Memory Model specifies the legal behaviors for a multithreaded Java program. Although it influences every Java programmer, unfortunately for most people it is an unknown concept. Even the targeted audience (virtual machine, compiler and processor implementers) does not have a uniform understanding of the Java Memory Model, because the original model is hard to interpret. This leads for instance to virtual machines that violate the constraints of the model, resulting in unexpected behavior of Java code. The big problem when you start reading papers and articles related to the Java Memory Model is that you should really pay attention to the timeframe when they were written. A good example is the confusion about the semantics of the volatile keyword. This paper tries to give an evolution of the Java Memory Model, starting from the original Chapter 17 from the Java Language Specification of Java 1.0 to the implementation of the JSR-133 specification in the latest Java 5. To clearly illustrate the problems and give a better understanding of the Java Memory Model, a big part of this paper will deal with the famous lazy initialization / double-checked locking programming idiom.

Keywords

Java Memory Model, double-checked locking, JSR-133

Meta-info

Written for the Software Technology Colloquium, Utrecht University.

Date: May 1st, 2006

1. Introduction

This paper tries to provide a good understanding of the Java Memory Model and memory models in general. We'll take a look at the specifications of the original version of the memory model and try to discuss the issues that developers have been faced with during the years. To give a better understanding of the problem, we'll take a look at the famous double-checked locking (bug) pattern and also show how we can make it work.

2. The Java Memory Model

2.1 Definition

Writing multithreaded applications is a very difficult task. You are faced with several hardware aspects that can be specific for each architecture. You are faced with the interaction between the processor cache and the main memory. You can have a multiple processor system with shared memory. Basically, it's very hard to be aware of all these details.

Not only at the hardware level, but also at the lower software level, certain operations can influence the execution of your multithreaded code. Let's think about compiler optimizations or the inner workings of virtual machines (Just In Time (JIT) works differently than HotSpot).

So the whole idea of having a memory model is to shield developers from the underlying system's architecture. It describes how we can make correctly synchronized code, but it also defines what behavior we can expect with incorrect synchronization. It describes how variables are mapped

between programs and memory and how the low-level actions like storing and retrieving are happening.

The Java Memory Model adds extra specific language constructs like the `volatile`, `final` and `synchronized` keywords, how to use them and what kind of behavior we can expect from them.

Thanks to the memory model we can reason about complex multithreaded algorithms without being concerned whether the code will be executed the way we intended to do on each platform.

2.2 The (in)famous chapter 17 of the Java Language Specification (JLS)

Once you start reading more about the Java Memory Model, you will see a lot of references to the Java Language specification (JLS). The JLS on its own is, as its name implies, a specification of the semantics and syntax of the Java language. This JLS is defined by Sun Microsystems, but is open for suggestions from third parties. They formalized the process that allows third parties to be involved in the definition of future versions and features in the Java Community Process or JCP [JCP1]. Third parties can propose new specifications or technologies in the form of a Java Specification Request and then a team is appointed to make some formal specification. These formal specifications undergo a long process of public reviewing and voting by the members of the JCP Executive Committee before it is added to the Java language.

The first edition of the JLS, written by James Gosling, Bill Joy and Guy Steele, appeared in 1996 for the Java 1.0 language. The revision for the Java 1.1 language was added as an appendix [ARNO97] in the fourth revision of the book *The Java Programming Language* by Ken Arnold and James Gosling. In 2000 they published a second edition that added some concepts like inner classes and the third edition appeared just recently in June 2005. The third edition incorporates major changes that have been applied in the Java 5 language like generics, annotations, enumerations, auto-boxing and other neat features. All books are freely downloadable from Sun's Java website [JLSPDF].

The Java Memory Model is specified in the JLS in chapter 17 about threads and locks. In this paper I'll discuss briefly the contents of this chapter in the first edition and also the changed chapter in the latest third edition. Interesting to note is that in the first edition they only needed 18 pages to describe the threading and locking behavior of the Java language, in the third edition however they needed 30 pages. The changes will be described later in the paper in the context of the JSR-133 specification.

When I read the chapter of the first edition I could not believe that this is a (though not formal) language specification. Perhaps I am influenced by the knowledge that I have know about the underlying problems, but it seems like Sun defined certain aspects of the multi-threaded behavior very loosely and therefore you do feel in a way that things might get interpreted differently amongst different people.

When you read through the chapter, the JLS points out in certain situations that it actually leaves some details open for the implementers of for instance virtual machines. A nice example is the non-atomic behavior of double and long variables. Apparently longs and doubles (which are 64-bit) are treated as two 32-bit variables by the specification. However they encourage implementers to implement this as a 64-bit variable instead of the proposed two 32-bit variables. This might seem strange at first, but they took in account that some processor architectures have problems with efficiently executing atomic memory behaviors on 64-bit quantities. The JLS strongly advises developers to make longs and doubles volatile, otherwise things might go wrong when two threads want to access non-volatile longs or doubles. Due to the non-atomic behavior in the latter case, one thread may be preempted by another one while reading the two 32-bit variables.

An interesting paragraph is 17.11 where they give an example of out-of-order writes. With the background knowledge I previously acquired I was surprised to read something about out-of-order writes. After reading through the paragraph I noticed that it was not the out-of-order writes caused by processor optimization, but the ordinary situation caused by use of unsynchronized methods. Example

17.11 basically shows what can happen when you do not synchronize methods in a concurrent environment.

2.3 Things can get ugly

2.3.1 Introduction

The whole problem with the earlier described chapter 17 of the JLS is that this part is hard to interpret and poorly understood. Certain optimizations done by compiler implementers are actually illegal in respect to the original JLS. The problem is that if the designers of the Java Virtual Machine had strictly followed the constraints imposed by the JLS, then this would have resulted in very poor performing virtual machines. The confusion caused by the specification also influences developers. Certain programming idioms, which are widely accepted as correct, seems to be broken afterwards (as with the double-checked locking design pattern).

The authors of the JLS were pioneering with this impressive piece of work. No one defined a uniform and platform/architecture independent memory model for a language before. It is understandable that certain aspects were not anticipated at that time; let's not forget that this was written ten years ago when Java was still in its infancy. Guy Steele, one of the authors of the JLS, admitted to William Pugh at OOPSLA98 [Pugh1] that he was unaware that the memory model prohibited common compiler optimizations. William Pugh, who was the specifications lead of the JSR-133, determined that one of the confusing points of the JLS was the usage of certain terms like “a variable”, “use”, “assign”, “lock” and “unlock”. These are general terms and not specifically related to the JMM or even Java programming. Therefore, these were not strictly defined, resulting in different understanding by people.

William Pugh stated in a paper for the Java Grande Conference of 1999 [Pugh2] that the Java Memory Model has two basic problems: it is too weak and it is too strong. It is too strong in the sense that it prohibits compiler optimizations and it is too weak in the sense that much of the code that has been written for Java is not guaranteed to be valid.

2.3.2 Original Java Memory Model too strong

The original JMM has a strong ordering restriction, i.e. it requires a total order for memory operations on each individual variable. This concept is often called coherence or with the help of Wikipedia easily explained as consistency. What it basically means is that each variable in isolation is sequentially consistent. So we are limited in reordering statements within a thread, even if there are no dependencies between them. This ordering restriction prohibits important compiler optimizations like fetch elimination (this is the elimination of a memory read operation if it is preceded by a read/write operation on the same variable [TAS1]).

```
int i = p.x
int j = q.x
int k = p.x
```

Example 1: Problems with compiler optimizations

A standard compiler optimization in example 1 would be the replacement of the assignment of `int k = p.x` with `int k = int i`. If `p` and `q` pointed to the same object (so if they were aliases) then we could also replace `q.x` with `int i`. Problems might occur here when a thread assigns `p.x` to `int i` and another thread assigns a new variable to `p.x` right after that. When the compiler applies a fetch elimination and replaces `int k = p.x` with `int k = int i`, then the first thread will still see the original value, although this has been changed by thread two. Languages that require coherence do not allow such behavior.

2.3.3 ... but also too weak

The JMM can sometimes be also too weak. Consider example 2:

```
Initially: p = new Point(1,2)
Thread 1: p = new Point(3,4)
Thread 2: a = p.x
Result: a = ???
```

Example 2: JMM too weak

We initialize the Point object with arguments 1 and 2. Thread 1 kicks in and initializes a new Point with arguments 3 and 4. However the original JMM does not require flushing the written values to main memory before the write of the reference to the newly created Point into p. This can have the result that Thread 2 sees the value 0 instead of 3. Why 0? If we only write the new values for the Point p to main memory after writing the reference, then there is a short period that we have garbage or default values for x and y. In the case of integers, we probably end up with 0. However when we are dealing with references to objects, we might receive references to a garbage value resulting in possible type violations. In such environment it is impossible to make any kind of security/safety guarantee.

2.3.4 Implication on immutable objects and final fields

This has also an implication on final fields. If there were final fields in the Point object that take the value passed by the arguments of the default constructor, then in the above described situation, Thread 2 would wrongly assume that it reads the immutable values of the final fields but is in fact seeing some default value.

A very famous example is the immutable String object, which does not need synchronization (well that's what people thought). It is implemented as an object with three final fields: a character array, an offset into that array and a length. For optimization reasons, when we take for instance a substring of a string, the new string shares the character array, but has a different offset and length.

```
String s1 = "testing123";
String s2 = s1.substring(3);
```

Example 3: immutable string object?

Printing out the value of s2 in example 3 will return ing123, because we supplied an offset 3. It is however possible, that a different thread might read testing123 for the variable s2 because it sees offset 0. This is for the same reason as earlier described. It is possible that s2 is not fully initialized and the values not yet flushed to main memory.

2.3.5 What about volatile?

When you declare a field as volatile, an accessing thread is guaranteed to see the most up-to-date value of the field. This is done by immediately flushing the cache to main memory after a write to it. Before a volatile field is read, the cache must be invalidated to be ensured that we are accessing the latest value. However, under the current JMM, we have a small problem. Although volatile variables can't be reordered with respect to each other, they can be reordered with non-volatile variables, therefore losing its "guard" property. Acting as a guard can be useful to signal a condition from one thread to another, for instance a volatile boolean field that denotes if something is read or not.

2.4 JSR-133

2.4.1 Introduction

There were more and more voices from people from the industry and academic world that there were serious problems with the original Java Memory Model. Sun started within the Java Community Process a "Java Memory Model and Thread Specification Revision" also known as JSR-133. Tim Lindholm from Sun Microsystems led this specification revision together with William Pugh from

Maryland University. Doug Lea from the State University of New York at Oswego was assigned as one of the members of the expert group. The JSR-133 group officially started in June 2001, but several people like William Pugh had already done a lot of work before. The JSR-133 has been incorporated in the Java 5 release and in the third version of the Java Language Specification.

2.4.2 Goal

The goals of the specification revision are as following [JCP3]:

- There should be both a formal and intuitive definition of “correctly synchronized” programs.
- Incompletely or incorrectly synchronized programs should not compromise the security of a system. (For instance with immutable objects like String instances)
- Programmers should be able to create multithreaded programs with confidence that they will be reliable and correct.
- The JMM has to be weak enough to allow the implementation of high performing virtual machines.
- Breaking existing code should be avoided.

Proposed areas of specification revision or clarification [JCP3]:

- Volatile variables
- Final variables
- Immutable Objects (objects whose fields are only set in their constructors)
- Thread- and memory related JVM functionality and API's

2.4.2 Data-race-free memory model

The original Java Memory Model required coherence or sequential consistency. In a way it's the easiest model to understand and makes it easier to write complex concurrent algorithms that do not use explicit synchronization. However a lot of these complex synchronization-free algorithms are wrong designed or implemented. As in the previous chapter described, it also prohibits common compiler optimizations.

To satisfy both developers (with simplicity) and implementers of compilers and virtual machines (flexibility), the new Java Memory Model is based on a more relaxed memory model called data-race-free model. This means that correctly synchronized (or data-race-free) programs are guaranteed to be sequentially consistent. This might sound contradictory with the fact that the new JMM moved from a sequential consistency model to a data-race-free model, but let me explain. The whole point here is to let programmers reason about multithreaded programs and guarantee them that their (data-race-free) applications will be reliable and correct. So that means that in a single-threaded program, the program should not be able to observe the effect of reordering. This concept is called as-if-serial. So the compiler and virtual machine can execute optimizations with reordering, but we guarantee that these reordering will work 100 % correctly and hereby the developer can reason about his application as if there were no reordering.

This is a very nice concept but this sequential consistency is only guaranteed for data-race-free programs. However we have to specify the behavior of the code for incorrectly synchronized applications, otherwise it allows violation of security properties that are necessary for Java programs. The problem is that we have to prohibit some executions that contain data races. Several questions pop up, like how do you characterize such executions? How can we prohibit them without limiting compiler and hardware optimizations?

2.4.3 Happens-before memory model

We have described earlier the concept of “as-if-serial” where we guarantee some kind of sequential consistency for data-race free programs. When we consider multiple threads, under the old JMM it's completely unpredictable how certain actions will be executed in one thread in relation with the other actions in the other threads. The new JMM introduces the concept of partial ordering of all actions within a program, called happens-before. Brian Goetz nicely summarized the rules of partial ordering in the happens-before model in one of his articles for the IBM developerWorks [Goe04]:

- 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's quite hard to understand the whole happens-before model, because there are some subtle problems with it. It allows "out-of-thin-air" results for incorrectly synchronized programs. That means that a thread can see a value X for a variable, which has not yet been written by another thread, so it's not actually there yet. This is caused by early writes. For the inner details of this, I'd like to refer to the in-depth paper on this topic by Jeremy Manson, William Pugh and Sarita Adve [MPA05].

2.4.4 Volatile

The new memory model restricts the reordering of volatile fields. The problem with the old memory model was that volatile fields could be reordered with non-volatile fields, but this is now very limited. Therefore we can again use the volatile variables for instance as a guard. This guard principle is often used by Java developers to indicate whether a set of operations has finished or not. Example 4 is only guaranteed to work with the latest version of the Java language.

```
volatile boolean guard = false;
...
// thread 1
... (do stuff)
guard = true;
...
// thread 2
while (!guard) {
    sleep();
    ... (do stuff)
}
```

Example 4: immutable string object?

3. Memory models for other languages

3.1 C/C++

Apparently C and C++ were not designed with direct support for multithreading. Therefore they don't have a uniform memory model. They basically rely on the guarantees provided by the threading libraries like the POSIX threads on *nix or Win32 threads on Windows, the compiler and the platform.

I have only a basic knowledge of C and C++ (using Microsoft Visual Studio 6) so I can't tell you much from my own experience. But I looked around on the Internet and found out that you can actually tell the compiler what memory model you want to use by providing some specific command line arguments. Examples can be found in the Intel C++ Compiler for Linux [INTEL1], Digital Mars C++ for 8086 [MARS1] and GCC [GCC1].

The double-checked locking problem that I will discuss later on can actually work in C++ when you use explicit memory barriers. Although Wikipedia receives lots of critic for the quality of their content, sometimes you find perfect explanations, as with the definition of memory barrier: "*A memory barrier is a general term used to refer to instructions which cause the CPU to enforce an ordering constraint on memory operations issued before and after the barrier instruction. The exact nature of the ordering constraint is hardware dependent, and is defined by the architecture's memory*

model.” An example of a C++ version of the double-checked locking pattern with explicit memory barriers can be found here [DECLA].

According to a blog entry on MSDN [OLD04] processors like PPC, MIPS, Alpha and SPARC do support explicit memory barriers, whereas x86 does not. The explanation is that x86 has a strong memory model, where the others have a more weak memory model. So they need explicit memory barriers to ensure that issues to the bus are made and completed in a specific order. In an x86 processor, the strict memory model guarantees that external memory access matches the order in which the code stream issues memory accesses.

If you want to know more about memory-barrier primitives in the Linux kernel, then there is a great article on LinuxJournal.com [MCKE05] that takes a closer look at how it is implemented in the kernel and a comparison of different processor architectures.

Some of the driving forces behind the revision of the Java Memory Model, i.e. William Pugh and Doug Lea, are trying to develop a uniform memory model for C++ based on their work and experiences with the JMM. A paper about their efforts can be found here [MEM04].

3.2 .NET

C# is so close to Java that it is interesting to know if it also suffers from the same problems, i.e. an inconsistent memory model. An interesting article in the MSDN magazine by Vance Morrisson [MORR05] (compiler architect for the .NET framework) talks about the memory model approach in the .NET framework.

The original .NET framework was based on a quite relaxed model, described in the .NET framework ECMA standard [ECMA1]. It's called relaxed because it has few constraints, thus giving compilers and processors a lot of freedom to do optimizations. Basically it has the following rules:

- no memory read (volatile or regular to any memory location), can move before a lock acquire
- no memory write (volatile or regular to any memory location), can move after a lock release
- all other reads and writes can reorder arbitrarily

When you look at the architecture described in the ECMA CLI Architecture Document than you can see that the JLS chapter 17 has its counterpart in .NET: ECMA-335 chapter 12 [ECMA2]. In the interview with Pratap Lakshman (Lead Program manager of the Visual J# team) you can read that for instance Visual J# .NET 2005 is based on the .NET framework ECMA standard.

With the growing popularity of 64 bit processors, it was clear that the original ECMA standard was too relaxed or in terms of memory model: too weak. It was likely that code written according to the ECMA standards would break on for instance the Intel IA-64 processor. Therefore Microsoft introduced in the 2.0 version of the .NET framework a new revised memory model with more constraints.

Because this paper talks explicitly about the Java Memory Model, I'd like you to refer to an article on MSDN magazine for further details about the memory models in .NET [MORR05] and a great article about multi-threading in .NET from Jon, a software engineer at ClearSwift [JON1]. For people who are interested in the use of the volatile keyword in C#, they can read this blog entry from Jerome Laban [LABA04].

4. Classic example: double-checked locking

4.1 The lazy initialization problem

The double-checked locking pattern started as an optimization of the well-known singleton pattern [WIKI1]. It was designed to reduce the locking overhead when you want to defer initialization of a value until the first time it is accessed (also known as lazy initialization) [WIKI2]. One of the reasons why you would want to delay the initialization is for startup speed of your application.

Let's first start with the singleton pattern to explain the history behind double-checked locking. Sometimes it is desirable in applications that you have just one instantiation of a class. This can be done with the singleton pattern. An example could be an object that contains configuration data that has been loaded from a property file. Basically a singleton is implemented by creating a class with a method that creates a new instance of the object if it doesn't exist yet. If it does, then it just returns the reference to that already instantiated object. An important thing is to avoid that programmers are calling the constructor to create a new instance. Making the constructor private or protected can solve this.

```
public class Singleton {
    private static Singleton instance;
    // omitted code

    private Singleton() {
        // omitted code
    }

    public static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

Example 5: Basic Singleton

Example 5 [HAG02] works perfectly for single threaded applications, however we are looking for solutions in a multithreaded environment. In the latter case, things can go wrong when we call the `getInstance` method. With the current implementation a race condition can occur: two simultaneously running threads can each initialize an instance of the Singleton object. This is easily explained as follows:

1. Thread 1 enters the `(instance == null)` check and notices that there is no object created yet.
2. Thread 2 calls the `getInstance` method and sees also that there is no object created yet. This can be in the situation that thread 1 hasn't created a new instance of Singleton yet.
3. In the meanwhile thread 1 creates an instance and returns.
4. Thread 2 is now already in the `if` branch and creates also an instance.
5. Result: we have two instances of Singleton.

We can easily find a solution for this: just make the `getInstance` method synchronized as shown in example 6.

```
public static synchronized Singleton getInstance() {
    if (instance == null)
        instance = new Singleton();
    return instance;
}
```

Example 6: Singleton with synchronized getInstance

This solution works perfect, but is not optimal from a performance point of view. In order to get faster startup times, apparently the synchronization penalizes us with slower execution time once the application is running. The reason why we added the `synchronized` keyword to our method is to avoid that two threads can both create an instance. When we have an instantiation of the Singleton class, then the check `(instance==null)` will always return false and thus return a reference to the object. This means that once the object is created we unnecessarily make use of the `synchronized` keyword to

protect the `getInstance` method. Although the Java people have put a lot of work in optimizing the synchronization process, it still results in a performance penalty.

A solution could be moving the synchronization from the method level to the point where we create an instance of Singleton as shown in example 7.

```
public static Singleton getInstance() {
    if (instance == null) {
        synchronized(Singleton.class) {
            instance = new Singleton();
        }
    }
    return instance;
}
```

Example 7: Singleton with (pseudo) optimized getInstance

It is very clear that this solution suffers from the same problem as the initial Singleton implementation. The only thing is that once thread A and B are both in the if branch, the B thread has to wait till the A thread releases the lock when exiting the synchronized block. After finishing, thread B can enter the synchronized block and create a second instance.

Now we come to the point where the double-checked locking (I'll abbreviate this to DCL) kicks in. The problem with the previous example was that once thread A and B are in the if branch, they can both make an instance of Singleton. Basically the synchronized block doesn't prevent it at all. What if we add a second check on `(instance==null)` within the synchronized block? This could solve the problem when both thread A and B entered the first if branch, that thread B is stopped by the second if branch in the synchronized block.

```
public static Singleton getInstance() {
    if (instance == null) {
        synchronized(Singleton.class) {
            if (instance == null)
                instance = new Singleton();
        }
    }
    return instance;
}
```

Example 8: Broken Double-Checked Locking

Of course I would not talk about the DCL if it has no relation to the Java Memory Model (JMM). Although the DCL pattern nicely addresses the shortcomings in our Singleton pattern at first sight, there is a caveat. The JMM allows the phenomenon of "out-of-order writes". The JMM allows compilers and virtual machines to do reordering for performance reasons. As long as the reordering of execution within a single thread would lead to the same results as without the reordering, it is allowed. This is also known as the "as-if-serial semantics" [GOET1]. To understand this rather illogical issue, let's look back at the previous code fragment.

Thread 1 enters the second if branch (the one within the synchronization block) and because there is no instance yet, it calls the constructor. BUT! The compiler or the cache can reorder the writes that initialize the instance field and the write to the instance field. So we get an object that is apparently not completely initialized yet. This wouldn't be a problem if the creating thread could just finish his thing, but the creating thread could be preempted by another thread before it completely writes everything to main memory. When thread 2 checks if the instance is null, it gets "not null" because the initialization of the object has started, but because it is not yet finished, thread 2 gets an incomplete object.

The problem with this behavior is that not all JIT compilers allow that the instance becomes non-null before the constructor executes. Sun's JIT implementation in JDK version 1.3 and above is known to disallow such behavior and work as expected. However the more recent HotSpot technology knows the concept of code in lining for performance optimizations [HOTSP]. I thought that the current virtual machines and optimizers would think about the JMM related problems in the JIT compiler and learn from it, but according to the "double-checked locking is broken declaration group" [DECLA] it's not. They state that if the compiler in-lines the call to the constructor, then the writes that initialize the object and the writes to the instance field can be freely reordered if the compiler can prove that the constructor cannot throw an exception or perform synchronization.

4.2 How to fix double-checked locking?

4.2.1 The use of the volatile keyword

This works only from Java 5 on. We can make the instance volatile, hereby creating a happens-before relationship. However this can impose performance overhead because the semantic of volatile in Java 5 makes it more expensive in terms of performance than in earlier Java versions. Each read of a volatile field has the same memory semantics as a monitor acquire and each write of a volatile field has the same memory semantics as a monitor release.

4.2.2 Get rid of lazy initialization and use eager initialization

A possible solution is just to get rid of DCL. The easiest way to do this is using the synchronization keyword. Because the DCL was developed to support lazy initialization, we can also look at an alternative called eager initialization. You force the construction of the object at startup time. With this solution, you do have a performance penalty at startup time, but at run time you can get rid of the synchronization overhead. This can be done with static initializers.

4.2.3 Initialization-On-Demand Holder Class Idiom

```
private static class LazySingletonHolder {
    public static Singleton instance = new Singleton();
}

...

public static Singleton getInstance() {
    return LazySingletonHolder.instance;
}
```

Example 9: Initialization-On-Demand Holder Class Idiom

So when we want an instance of the Singleton class, then we let an inner "holder" class create a new object during the first invocation and on all further invocations return the instance. The thread safety of this unsynchronized code is based on the fact that operations that are part of class initialization are guaranteed to be visible to all threads that use that class. Because an inner class is not loaded until some thread references one of its fields or methods, we get a lazy initialization.

5. Conclusion

The JSR-133 group had a tough job with this rethinking of the Java Memory Model, but now we have finally a consistent model that both developers and compiler/VM builders can rely and reason upon. The hard tasks was that developers wanted to be able to develop multi-threaded applications that can contain data-races without having to be worried about compiler and VM optimizations, but the builders of compilers and VMs wanted to have enough freedom to implement optimization techniques without being worried that applications would be executed in a wrong way. Both groups' needs have been addressed in the current 1.5 release of the Java language.

6. Acknowledgements

I'd like to thank my supervisor Piet van Oostrum from the University of Utrecht and Mathieu Cardinael from the Vrije Universiteit Brussel for reviewing this paper.

I'm also grateful to William Pugh, Doug Lea, Brian Goetz and all the other driving forces behind the JSR-133 for doing an excellent job on delivering a reworked and consistent Java Memory Model.

7. References

- [INTEL1] Intel Corporation, Intel C++ Compiler 8.1 Extended Memory 64 Technology Edition for Linux Release Notes, 2005, Available from <http://www.scripps.edu/rc/supercomputing/docs/C++ReleaseNotes-E.htm>
- [MARS1] Digital Mars, Choosing a memory model, 22 August 2005, Available from <http://www.digitalmars.com/ctg/ctgMemoryModel.html>
- [MEM04] Andrei Alexandrescu and Hans Boehm and Kevlin Henney and Doug Lea and Bill Pugh, Memory Model for multi-threaded C++, 10 September 2004, Available from <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1680.pdf>
- [GCC1] GCC Manual, 2005, Available from <http://gcc.gnu.org/onlinedocs/>
- [MCKE05] Paul McKenney, Memory ordering in modern microprocessors, Linux Journal, 30 June 2005, Available from <http://www.linuxjournal.com/article/8211>
- [OLD04] "oldnewthing", The x86 architecture is the weirdo, 14 September 2004, Available from <http://blogs.msdn.com/oldnewthing/archive/2004/09/14/229387.aspx>
- [WIKI3] Wikipedia, Memory Barrier, 10 July 2005, Available from http://en.wikipedia.org/wiki/Memory_model
- [MORR05] Vance Morrison (Compiler Architect for the .NET runtime, .NET intermediate language Lead and Just In Time compiler Lead at Microsoft Corporation), Understand the low-lock techniques in multithreaded apps, MSDN Magazine, October 2005, Available from <http://msdn.microsoft.com/msdnmag/issues/05/10/MemoryModels/default.aspx>
- [ECMA1] ECMA and ISO/IEC C# and Common Language Infrastructure standards, MSDN .NET framework developer center, Available from <http://msdn.microsoft.com/netframework/ecma/>
- [ECMA2] ECMA TC39/TG3, Common Language Infrastructure. Partition 1: Concepts and architecture, October 2002, Available from http://download.microsoft.com/download/f/9/a/f9a26c29-1640-4f85-8d07-98c3c0683396/Partition_I_Architecture.zip
- [LAKS05] Pratap Lakshman and Varun Gupta, Visual J# Design Choices: a conversation with Pratap Lakshman, Microsoft Visual J# .NET developer center, 26 September 2005, Available from <http://msdn.microsoft.com/vjsharp/productinfo/news/transcripts/default.aspx>
- [JON1] Jon, Multi-threading in .NET, Available from <http://www.yoda.arachsys.com/csharp/threads/>
- [LABA04] Jerome Laban, Don't get C# volatile the wrong way, 24 August 2004, Available from <http://blogs.labtech.epitech.net/blogs/jaylee/archive/2004/08/05/704.aspx>
- [HAG02] Peter Haggart, Double-checked locking and the singleton pattern: a comprehensive look at this broken programming idiom, IBM Developerworks, 01 May 2002, Available from <http://www-128.ibm.com/developerworks/java/library/j-dcl.html>
- [WIKI1] Wikipedia, The singleton pattern, 11 October 2005, Available from http://en.wikipedia.org/wiki/Singleton_pattern
- [WIKI2] Wikipedia, Double-checked locking, 6 August 2005, Available from http://en.wikipedia.org/wiki/Double-checked_locking
- [DECLA] David Bacon (IBM Research) and Joshua Bloch (Javasoft) and Jeff Bogda, and Cliff Click (Hotspot JVM project) and Paul Haahr and Doug Lea and Tom May and Jan-Willem Maessen and John D. Mitchell (jGuru) and Kelvin Nilsen and Bill Pugh and Emin Gun Sirer, The "double-checked locking is broken" declaration, Available from <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>
- [HOTSP] The Java HotSpot performance architecture, Sun Microsystems, April 1999, Available from <http://java.sun.com/products/hotspot/whitepaper.html>
- [GOET1] Brian Goetz, Double-checked locking: clever, but broken: do you know what synchronized really means?, February 2001, Available from <http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-double.html>
- [JCP1] Java Community Process, Introduction Overview, Available from <http://jcp.org/en/introduction/overview>
- [JCP2] Tim Lindholm and William Pugh, Java Community Process, JSR-133: Java Memory Model and Thread Specification Revision (Final Release), Available from <http://jcp.org/aboutJava/communityprocess/final/jsr133/index.html>
- [WIKI4] Wikipedia, Java Language Specification, 31 August 2005, Available from http://de.wikipedia.org/wiki/Java_Language_Specification
- [ARNO97] Ken Arnold and James Gosling, The Java Programming Language, Addison-Wesley, 1996, ISBN 0-201-63455-4, Appendix D
- [JLSPDF] The Java Language Specification Books, <http://java.sun.com/docs/books/jls/>
- [Pugh1] William Pugh, The Java Memory Model is Fatally Flawed, Concurrency: Practice and Experience
- [Pugh2] William Pugh, Fixing the Java Memory Model, ACM, Java Grande Conference 1999

- [Wiki4] Cache Coherency, Available from http://en.wikipedia.org/wiki/Cache_coherence
- [TAS1] Tulika Mitra and Abhik Roychoudhury and Qinghua Shen, School of Computing, National University of Singapore, Impact of Java Memory Model on Out-of-Order Multiprocessors
- [JCP3] JSR-133: Java Memory Model and Thread Specification Revision, Available from <http://jcp.org/en/jsr/detail?id=133>
- [MPA05] Jeremy Manson and William Pugh and Sarita V. Adve, The Java Memory Model, POPL '05
- [Goe04] Brian Goetz, Java Theory and practice: Fixing the Java Memory Model, Part 1 & Part 2, IBM developerWorks