developerWorks®

Double-checked locking and the Singleton pattern

A comprehensive look at this broken programming idiom

Peter Haggar

May 01, 2002

All programming languages have their share of idioms. Many are useful to know and use, and programmers spend valuable time creating, learning, and implementing them. The problem is that some idioms are later proven not to be all that they were purported, or to simply not work as described. The Java programming language contains several useful programming idioms. It also contains some that further study has shown should not be used. Double-checked locking is one such idiom that should never be used. In this article, Peter Haggar examines the roots of the double-checked locking idiom, why it was developed, and why it doesn't work.

Editor's note*: This article refers to the Java Memory Model before it was revised for Java 5.0; statements about memory ordering may no longer be correct. However, the double-checked locking idiom is still broken under the new memory model. For more information on the memory model in Java 5.0, see "Java theory and practice: Fixing the Java Memory Model" Part 1 and Part 2.*

The Singleton creation pattern is a common programming idiom. When used with multiple threads, you must use some type of synchronization. In an effort to create more efficient code, Java programmers created the double-checked locking idiom to be used with the Singleton creation pattern to limit how much code is synchronized. However, due to some little-known details of the Java memory model, this double-checked locking idiom is not guaranteed to work. Instead of failing consistently, it will fail sporadically. In addition, the reasons for its failure are not obvious and involve intimate details of the Java memory model. These facts make a code failure due to double-checked locking very difficult to track down. In the remainder of this article, we'll examine the double-checked locking idiom in detail to understand just where it breaks down.

Singleton creation idiom

To understand where the double-checked locking idiom originated, you must understand the common singleton creation idiom, which is illustrated in Listing 1:

Listing 1. Singleton creation idiom

```
import java.util.*; class Singleton {
             private static Singleton instance; private Vector v; private boolean inUse; private
             Singleton() { v = new Vector(); v.addElement(new Object()); inUse = true; } public
             static Singleton getInstance() { if (instance == null) //1 instance = new
             Singleton(); //2 return instance; //3 } }
```

The design of this class ensures that only one `Singleton` object is ever created. The constructor is declared `private` and the `getInstance()` method creates only one object. This implementation is fine for a single-threaded program. However, when multiple threads are introduced, you must protect the `getInstance()` method through synchronization. If the `getInstance()` method is not protected, it is possible to return two different instances of the `Singleton` object. Consider two threads calling the `getInstance()` method concurrently and the following sequence of events:

1. Thread 1 calls the `getInstance()` method and determines that `instance` is `null` at //1.
2. Thread 1 enters the `if` block, but is preempted by thread 2 before executing the line at //2.
3. Thread 2 calls the `getInstance()` method and determines that `instance` is `null` at //1.
4. Thread 2 enters the `if` block and creates a new `Singleton` object and assigns the variable `instance` to this new object at //2.
5. Thread 2 returns the `Singleton` object reference at //3.
6. Thread 2 is preempted by thread 1.
7. Thread 1 starts where it left off and executes line //2 which results in another `Singleton` object being created.
8. Thread 1 returns this object at //3.

The result is that the `getInstance()` method created two `Singleton` objects when it was supposed to create only one. This problem is corrected by synchronizing the `getInstance()` method to allow only one thread to execute the code at a time, as shown in Listing 2:

Listing 2. Thread-safe getInstance() method

```
public static synchronized
             Singleton getInstance() { if (instance == null) //1 instance = new Singleton(); //2
             return instance; //3 }
```

The code in Listing 2 works fine for multithreaded access to the `getInstance()` method. However, when you analyze it you realize that synchronization is required only for the first invocation of the method. Subsequent invocations do not require synchronization because the first invocation is the only invocation that executes the code at //2, which is the only line that requires synchronization. All other invocations determine that `instance` is non-`null` and return it. Multiple threads can safely execute concurrently on all invocations except the first. However, because the method is `synchronized`, you pay the cost of synchronization for every invocation of the method, even though it is only required on the first invocation.

In an effort to make this method more efficient, an idiom called double-checked locking was created. The idea is to avoid the costly synchronization for all invocations of the method except the first. The cost of synchronization differs from JVM to JVM. In the early days, the cost could be quite high. As more advanced JVMs have emerged, the cost of synchronization has decreased, but there is still a performance penalty for entering and leaving a `synchronized` method or block. Regardless of the advancements in JVM technology, programmers never want to waste processing time unnecessarily.

Because only line //2 in Listing 2 requires synchronization, we could just wrap it in a synchronized block, as shown in Listing 3:

Listing 3. The getInstance() method

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

The code in Listing 3 exhibits the same problem as demonstrated with multiple threads and Listing 1. Two threads can get inside of the `if` statement concurrently when `instance` is `null`. Then, one thread enters the `synchronized` block to initialize `instance`, while the other is blocked. When the first thread exits the `synchronized` block, the waiting thread enters and creates another `Singleton` object. Note that when the second thread enters the `synchronized` block, it does not check to see if `instance` is non-`null`.

Double-checked locking

To fix the problem in Listing 3, we need a second check of `instance`. Thus, the name "double-checked locking." Applying the double-checked locking idiom to Listing 3 results in Listing 4.

Listing 4. Double-checked locking example

```
public static Singleton
                getInstance() { if (instance == null) { synchronized(Singleton.class) { //1 if
                (instance == null) //2 instance = new Singleton(); //3 } } return instance; }
```

The theory behind double-checked locking is that the second check at //2 makes it impossible for two different `Singleton` objects to be created as occurred in Listing 3. Consider the following sequence of events:

1. Thread 1 enters the `getInstance()` method.
2. Thread 1 enters the `synchronized` block at //1 because `instance` is `null`.
3. Thread 1 is preempted by thread 2.
4. Thread 2 enters the `getInstance()` method.
5. Thread 2 attempts to acquire the lock at //1 because `instance` is still `null`. However, because thread 1 holds the lock, thread 2 blocks at //1.
6. Thread 2 is preempted by thread 1.
7. Thread 1 executes and because instance is still `null` at //2, creates a `Singleton` object and assigns its reference to `instance`.
8. Thread 1 exits the `synchronized` block and returns instance from the `getInstance()` method.
9. Thread 1 is preempted by thread 2.
10. Thread 2 acquires the lock at //1 and checks to see if `instance` is `null`.
11. Because `instance` is non-`null`, a second `Singleton` object is not created and the one created by thread 1 is returned.

The theory behind double-checked locking is perfect. Unfortunately, reality is entirely different. The problem with double-checked locking is that there is no guarantee it will work on single or multi-processor machines.

The issue of the failure of double-checked locking is not due to implementation bugs in JVMs but to the current Java platform memory model. The memory model allows what is known as "out-of-order writes" and is a prime reason why this idiom fails.

Out-of-order writes

To illustrate the problem, you need to re-examine line //3 from Listing 4 above. This line of code creates a `Singleton` object and initializes the variable `instance` to refer to this object. The problem with this line of code is that the variable `instance` can become non-`null` before the body of the `Singleton` constructor executes.

Huh? That statement might be contradictory to everything you thought possible, but it is, in fact, the case. Before explaining how this happens, accept this fact for a moment while examining how this breaks the double-checked locking idiom. Consider the following sequence of events with the code in Listing 4:

1. Thread 1 enters the `getInstance()` method.
2. Thread 1 enters the `synchronized` block at //1 because `instance` is `null`.
3. Thread 1 proceeds to //3 and makes instance non-`null`, but *before* the constructor executes.
4. Thread 1 is preempted by thread 2.
5. Thread 2 checks to see if instance is `null`. Because it is not, thread 2 returns the `instance` reference to a fully constructed, but partially initialized, `Singleton` object.
6. Thread 2 is preempted by thread 1.
7. Thread 1 completes the initialization of the `Singleton` object by running its constructor and returns a reference to it.

This sequence of events results in a period of time where thread 2 returned an object whose constructor had not executed.

To show how this occurs, consider the following pseudo code for the line: `instance =new Singleton();`

```
mem = allocate(); //Allocate memory for Singleton
               object. instance = mem; //Note that instance is now non-null, but //has not been
               initialized. ctorSingleton(instance); //Invoke constructor for Singleton passing
               //instance.
```

This pseudo code is not only possible, but is exactly what happens on some JIT compilers. The order of execution is perceived to be out of order, but is allowed to happen given the current memory model. The fact that JIT compilers do just this makes the issues of double-checked locking more than simply an academic exercise.

To demonstrate this, consider the code in Listing 5. It contains a stripped-down version of the `getInstance()` method. I've removed the "double-checkedness" to ease our review of the assembly code produced (Listing 6). We are interested only in seeing how the line `instance=new Singleton();` is compiled by the JIT compiler. In addition, I've provided a simple constructor to make it clear when the constructor is run in the assembly code.

Listing 5. Singleton class to demonstrate out-of-order writes

```
class Singleton
               { private static Singleton instance; private boolean inUse; private int val; private
               Singleton() { inUse = true; val = 5; } public static Singleton getInstance() { if
               (instance == null) instance = new Singleton(); return instance; } }
```

Listing 6 contains the assembly code produced by the Sun JDK 1.2.1 JIT compiler for the body of the `getInstance()` method from Listing 5.

Listing 6. Assembly code produced from code in Listing 5

```
;asm code generated
               for getInstance 054D20B0 mov eax,[049388C8] ;load instance ref 054D20B5 test eax,eax
               ;test for null 054D20B7 jne 054D20D7 054D20B9 mov eax,14C0988h 054D20BE call
               503EF8F0 ;allocate memory 054D20C3 mov [049388C8],eax ;store pointer in ;instance
               ref. instance ;non-null and ctor ;has not run 054D20C8 mov ecx,dword ptr [eax]
               054D20CA mov dword ptr [ecx],1 ;inline ctor - inUse=true; 054D20D0 mov dword ptr
               [ecx+4],5 ;inline ctor - val=5; 054D20D7 mov ebx,dword ptr ds:[49388C8h] 054D20DD
               jmp 054D20B0
```

Note: To reference the lines of assembly code in the following explanation, I refer to the last two values of the instruction address because they all begin with `054D20`. For example, `B5` refers to `test eax,eax`.

The assembly code is produced by running a test program that calls the `getInstance()` method in an infinite loop. While the program runs, run the Microsoft Visual C++ debugger and attach it to the Java process representing the test program. Then, break the execution and find the assembly code representing the infinite loop.

The first two lines of assembly code at `B0` and `B5` load the `instance` reference from memory location `049388C8` into `eax` and test for `null`. This corresponds to the first line of the `getInstance()` method in Listing 5. The first time this method is called, `instance` is `null` and the code proceeds to `B9` . The code at `BE` allocates the memory from the heap for the `Singleton` object and stores a pointer to that memory in `eax`. The next line, `C3`, takes the pointer in `eax` and stores it back into the instance reference at memory location `049388C8`. As a result, `instance` is now non-`null` and refers to a valid `Singleton` object. However, the constructor for this object has not run yet, which is precisely the situation that breaks double-checked locking. Then at line `C8`, the `instance` pointer is dereferenced and stored in `ecx`. Lines `CA` and `D0` represent the inline constructor storing the values `true` and `5` into the `Singleton` object. If this code is interrupted by another thread after executing line `C3` but before completing the constructor, double-checked locking fails.

Not all JIT compilers generate the code as above. Some generate code such that `instance` becomes non-`null` only after the constructor executes. Both the IBM SDK for Java technology, version 1.3 and the Sun JDK 1.3 produce code such as this. However, this does not mean you should use double-checked locking in these instances. There are other reasons it could fail. In addition, you do not always know which JVMs your code will run on, and the JIT compiler could always change to generate code that breaks this idiom.

Double-checked locking: Take two

Given that the current double-checked locking code does not work, I've put together another version of the code, shown in Listing 7, to try to prevent the out-of-order write problem you just saw.

Listing 7. Attempting to solve the out-of-order write problem

```
public static
               Singleton getInstance() { if (instance == null) { synchronized(Singleton.class) {
               //1 Singleton inst = instance; //2 if (inst == null) { synchronized(Singleton.class)
               { //3 inst = new Singleton(); //4 } instance = inst; //5 } } } return instance; }
```

Looking at the code in Listing 7 you should realize that things are getting a little ridiculous. Remember, double-checked locking was created as a way to avoid synchronizing the simple three-line `getInstance()` method. The code in Listing 7 has gotten out of hand. In addition, the code does not fix the problem. Careful examination reveals why.

This code is trying to avoid the out-of-order write problem. It tries to do this by introducing the local variable `inst` and a second `synchronized` block. The theory works as follows:

1. Thread 1 enters the `getInstance()` method.
2. Because `instance` is `null`, thread 1 enters the first `synchronized` block at //1.
3. The local variable `inst` gets the value of `instance`, which is `null` at //2.
4. Because `inst` is `null`, thread 1 enters the second `synchronized` block at //3.
5. Thread 1 then begins to execute the code at //4, making `inst` non-`null` but before the constructor for `Singleton` executes. (This is the out-of-order write problem we just saw.)
6. Thread 1 is preempted by Thread 2.
7. Thread 2 enters the `getInstance()` method.
8. Because `instance` is `null`, thread 2 attempts to enter the first `synchronized` block at //1. Because thread 1 currently holds this lock, thread 2 blocks.
9. Thread 1 then completes its execution of //4.
10. Thread 1 then assigns a fully constructed `Singleton` object to the variable `instance` at //5 and exits both `synchronized` blocks.
11. Thread 1 returns `instance`.
12. Thread 2 then executes and assigns `instance` to `inst` at //2.
13. Thread 2 sees that `instance` is non-`null`, and returns it.

The key line here is //5. This line is supposed to ensure that `instance` will only ever be `null` or refer to a fully constructed `Singleton` object. The problem occurs where theory and reality run orthogonal to one another.

The code in Listing 7 doesn't work because of the current definition of the memory model. The Java Language Specification (JLS) demands that code within a `synchronized` block not be moved out of a `synchronized` block. However, it does not say that code not in a `synchronized` block cannot be moved *into* a `synchronized` block.

A JIT compiler would see an optimization opportunity here. This optimization would remove the code at //4 and the code at //5, combine it and generate the code shown in Listing 8:

Listing 8. Optimized code from Listing 7

```
public static Singleton getInstance()
               { if (instance == null) { synchronized(Singleton.class) { //1 Singleton inst =
               instance; //2 if (inst == null) { synchronized(Singleton.class) { //3 //inst = new
               Singleton(); //4 instance = new Singleton(); } //instance = inst; //5 } } } return
               instance; }
```

If this optimization takes place, you have the same out-of-order write problem we discussed earlier.

volatile anyone?

Another idea is to use the keyword `volatile` for the variables `inst` and `instance`. According to the JLS (see Related topics), variables declared `volatile` are supposed to be sequentially consistent, and therefore, not reordered. But two problems occur with trying to use `volatile` to fix the problem with double-checked locking:

- The problem here is not with sequential consistency. Code is being moved, not reordered.
- Many JVMs do not implement `volatile` correctly regarding sequential consistency anyway.

The second point is worth expanding upon. Consider the code in Listing 9:

Listing 9. Sequential consistency with volatile

```
class test { private volatile
               boolean stop = false; private volatile int num = 0; public void foo() { num = 100;
               //This can happen second stop = true; //This can happen first //... } public void
               bar() { if (stop) num += num; //num can == 0! } //... }
```

According to the JLS, because `stop` and `num` are declared `volatile`, they should be sequentially consistent. This means that if `stop` is ever `true`, `num` must have been set to `100`. However, because many JVMs do not implement the sequential consistency feature of `volatile`, you cannot count on this behavior. Therefore, if thread 1 called `foo` and thread 2 called `bar` concurrently, thread 1 might set `stop` to `true` before `num` is set to `100`. This could lead thread 2 to see `stop` as `true`, but `num` still set to `0`. There are additional problems with `volatile` and the atomicity of 64-bit variables, but this is beyond the scope of this article. See Related topics for more information on this topic.

The solution

The bottom line is that double-checked locking, in whatever form, should not be used because you cannot guarantee that it will work on any JVM implementation. JSR-133 is addressing issues regarding the memory model, however, double-checked locking will not be supported by the new memory model. Therefore, you have two options:

- Accept the synchronization of a `getInstance()` method as shown in Listing 2.
- Forgo synchronization and use a `static` field.

Option 2 is shown in Listing 10:

Listing 10. Singleton implementation with static field

```
class Singleton {
               private Vector v; private boolean inUse; private static Singleton instance = new
               Singleton(); private Singleton() { v = new Vector(); inUse = true; //... } public
               static Singleton getInstance() { return instance; } }
```

The code in Listing 10 does not use synchronization and ensures that the `Singleton` object is not created until a call is made to the `static getInstance()` method. This is a good alternative if your objective is to eliminate synchronization.

String is not immutable

You might wonder about the `String` class given the issue of out-of-order writes and a reference becoming non-`null` prior to the constructor executing. Consider the following code:

```
private String str; //... str = new String("hello");
```

The `String` class is supposed to be immutable. However, given the out-of-order write problem we discussed previously, could that cause a problem here? The answer is it could. Consider two threads with access to the `String str`. One thread could see the `str` reference refer to a `String` object in which the constructor has not run. In fact, Listing 11 contains code that shows this occurring. Note that this code breaks only with older JVMs that I tested. Both the IBM 1.3 and Sun 1.3 JVMs produce immutable `String`s as expected.

Listing 11. Example of a Mutable String

```
class StringCreator extends Thread {
               MutableString ms; public StringCreator(MutableString muts) { ms = muts; } public
               void run() { while(true) ms.str = new String("hello"); //1 } } class StringReader
               extends Thread { MutableString ms; public StringReader(MutableString muts) { ms =
               muts; } public void run() { while(true) { if (!(ms.str.equals("hello"))) //2 {
               System.out.println("String is not immutable!"); break; } } } } class MutableString {
               public String str; //3 public static void main(String args[]) { MutableString ms =
               new MutableString(); //4 new StringCreator(ms).start(); //5 new
               StringReader(ms).start(); //6 } }
```

This code creates a `MutableString` class at //4 that contains a `String` reference shared by two threads at //3. Two objects are created, `StringCreator` and `StringReader`, on two separate threads at lines //5 and //6, passing a reference to the `MutableString` object. The `StringCreator` class enters an infinite loop and creates `String` objects with the value "hello" at //1. The `StringReader` also enters an infinite loop and checks to see if the current `String` object has the value "hello" at //2. If it doesn't, the `StringReader` thread prints out a message and stops. If the `String` class is immutable, you should never see any output from this program. The only way for `StringReader` to see the `str` reference to be anything other than a `String` object with "hello" as its value is if the problem of out-of-order writes occurs.

Running this code on old JVMs like Sun JDK 1.2.1 results in the out-of-order write problem, and thus, a non-immutable `String`.

Summary

In an effort to avoid costly synchronization in singletons, programmers, quite ingeniously, invented the double-checked locking idiom. Unfortunately, it was not until this idiom was in fairly wide use that it became apparent that it is not a safe programming construct due to the current memory model. Work is underway to redefine areas of the memory model that are weak. However, even under the newly proposed memory model, double-checked locking will not work. The best solution to this problem is to accept synchronization or use a `static field`.

Related topics

- Visit Bill Pugh's Java Memory Model Web site for a wealth of information on this important topic.
- For more information on `volatile` and 64-bit variables, see Peter Haggar's article "Does Java Guarantee Thread Safety?" in the June 2002 issue of *Dr. Dobb's Journal*.
- JSR-133 deals with the revision to Java platform's memory model and thread specification.
- Java software consultant Brian Goetz examines when to use synchronization in "Threading lightly: Synchronization is not the enemy" (developerWorks, July 2001).
- In "Threading lightly: Sometimes it's best not to share" (developerWorks, October 2001), Brian Goetz examines `ThreadLocal` and offers tips for exploiting its power.
- In "Writing multithreaded Java applications" (developerWorks, February 2001), Alex Roetter introduces the Java Thread API, outlines issues involved in multithreading, and offers solutions to common problems.
- Find other Java technology resources on the *developerWorks*Java technology zone.