**How to Test If a Class Is Thread-Safe in Java**

Learn how to test if a class is thread-safe in Java.

[**[](https://dzone.com/users/1171173/thomas.krieger.html)**](https://dzone.com/users/1171173/thomas.krieger.html)**by**

[**Thomas Krieger**](https://dzone.com/users/1171173/thomas.krieger.html)

**·**

**Dec. 16, 19 ·**[**Java Zone**](https://dzone.com/java-jdk-development-tutorials-tools-news)**· Tutorial**

**Like** [**(6)**](https://dzone.com/articles/how-to-test-if-a-class-is-thread-safe-in-java)

**Comment (0)**

**Save**

[**Tweet**](https://dzone.com/articles/how-to-test-if-a-class-is-thread-safe-in-java)

**17.05k Views**

Join the DZone community and get the full member experience.

[**JOIN FOR FREE**](https://dzone.com/articles/how-to-test-if-a-class-is-thread-safe-in-java)

*Learn how to test if a class is thread-safe in Java.*

Tests for thread safety differ from typical single-threaded tests. To test if a method is thread-safe we need to call the method in parallel from multiple threads. We need to do this for all potential thread interleavings. And afterward, we need to check if the result is correct.

Those three requirements for our test lead to a special type of tests for thread safety which differ from typical single-threaded tests. Since we want to test all thread interleavings our test must be repeatable and run automatically. And since the methods run in parallel the potential result is a combination of different outcomes.

**You may also like:**[**What Does Thread-Safety Mean in Java?**](https://dzone.com/articles/what-does-thread-safety-mean-in-java)

Let us look at an example to see how this looks in practice.

**Testing for Thread Safety**

Suppose we want to test if the following class representing an Address is thread-safe. It offers one method to update the street and city, the method update and one method to read the complete Address, the method  toString:

Java



1

public class MutableAddress {

2

   private volatile String street;

3

   private volatile String city;

4

   private volatile String phoneNumber;

5

   public MutableAddress(String street, String city,

6

       String phoneNumber) {

7

       this.street = street;

8

       this.city = city;

9

       this.phoneNumber = phoneNumber;

10

  }

11

   public void update(String street ,String city ) {

12

       this.street = street;

13

       this.city = city;

14

  }

15

   public String toString() {

16

       return "street=" + street + ",city=" + city + ",

17

       phoneNumber=" + phoneNumber;

18

  }

19

}

I use volatile fields, line 2 through 4, to make sure that the threads always see the current values, as explained in [greater detail here](https://vmlens.com/articles/cp/why_volatile/). You can download the source code of all examples from [GitHub here](https://github.com/vmlens/tutorial-copy-on-write).

Now, let us first see if the combination of  toString and update is thread-safe. Here is the test:

Java



1

import com.vmlens.api.AllInterleavings;

2

public class TestToStringAndUpdate {

3

   @Test

4

   public void testMutableAddress() throws InterruptedException {

5

       try (AllInterleavings allInterleavings =

6

           new AllInterleavings("TestToStringAndUpdate\_Not\_Thread\_Safe");) {

7

           while (allInterleavings.hasNext()) {

8

               MutableAddress address = new MutableAddress("E. Bonanza St.",

9

                    "South Park", "456 77 99");

10

               String readAddress = null;

11

               Thread first = new Thread(() -> {

12

                   address.update("Evergreen Terrace", "Springfield");

13

              });

14

               first.start();

15

               readAddress = address.toString();

16

               first.join();

17

               assertTrue("readAddress:" + readAddress,readAddress.equals(

18

       "street=E. Bonanza St.,city=South Park,phoneNumber=456 77 99")

19

                   || readAddress.equals(

20

       "street=Evergreen Terrace,city=Springfield,phoneNumber=456 77 99"));

21

          }

22

      }

23

  }

24

}

The test executes the two methods in parallel from two threads. To test all thread interleavings, we put the complete test in a while loop iterating over all thread interleavings using the class AllInterleavings from [vmlens](https://vmlens.com/), line 7. To see if the class is thread-safe, we compare the result against the to potential outcomes, the value before the update and after the update, lines 17 through 20.

Running the test leads to the following error:

Java



1

java.lang.AssertionError: readAddress:street=Evergreen Terrace

2

      ,city=South Park,phoneNumber=456 77 99

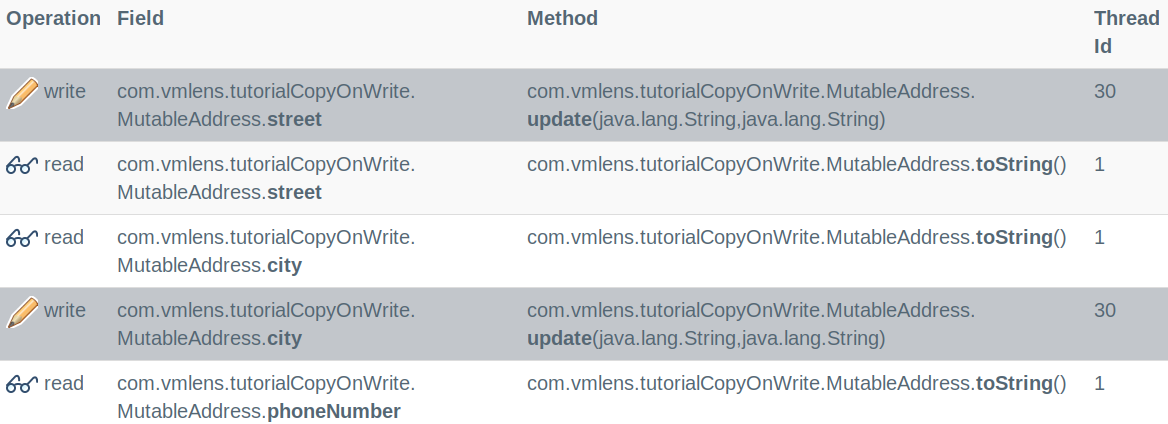
3

   at com.vmlens.tutorialCopyOnWrite.TestToStringAndUpdate.

4

       testMutableAddress(TestToStringAndUpdate.java:22)

To see what went wrong, we look at the report vmlens generated:



The problem is that for one thread interleaving the thread with Thread id 30 first updates the street name and then the main thread, thread id 1, reads the street and city name. So, the main thread reads a partial updated address which leads to the error.

To make the address class thread-safe, we copy the address value every time we update the address. Here is a thread-safe implementation using this technique. It consists of two classes, an immutable value, and a mutable container.

First, the immutable value class:

Java



1

public class AddressValue {

2

   private final String street;

3

   private final String city;

4

   private final String phoneNumber;

5

   public AddressValue(String street, String city,

6

               String phoneNumber) {

7

       super();

8

       this.street = street;

9

       this.city = city;

10

       this.phoneNumber = phoneNumber;

11

  }

12

   public String getStreet() {

13

       return street;

14

  }

15

   public String getCity() {

16

       return city;

17

  }

18

   public String getPhoneNumber() {

19

       return phoneNumber;

20

  }

21

}

Second is the mutable container class:

Java



1

public class AddressUsingCopyOnWrite {

2

   private volatile AddressValue addressValue;

3

   private final Object LOCK = new Object();

4

   public AddressUsingCopyOnWrite(String street,

5

           String city, String phone) {

6

       this.addressValue = new AddressValue( street,

7

               city,  phone);

8

  }

9

   public void update(String street ,String city ) {

10

       synchronized(LOCK){

11

           addressValue = new AddressValue(  street,  city,

12

                   addressValue.getPhoneNumber() );

13

      }

14

  }

15

   public String toString() {

16

       AddressValue local = addressValue;

17

       return "street=" + local.getStreet()

18

       + ",city=" +    local.getCity() +

19

       ",phoneNumber=" + local.getPhoneNumber();

20

  }

21

}

The class  AddressUsingCopyOnWrite creates a new address value every time it updates the variable  addressValue. This makes sure that we always read a consistent address, either the value before or after the update.

If we run the test with those two classes, the test succeeds.

**What Do We Need to Test?**

So far, we tested the combination of toString and  update for thread safety. To test if a class is thread-safe, we need to test all combinations of modifying methods and all combinations of read-only methods together with modifying methods. So, for our example class, we need to test the following two combinations:

1. update and update
2. toString and update

Since the combinations of read-only methods are automatically thread-safe, we do not need to test the combination of the method toString with itself.

**Data Races**

So far, we used volatile fields to avoid data races. Let us see what happens when we use normal fields instead. So, in our thread-safe class  AddressUsingCopyOnWrite, we remove the volatile modifier and re-run our test. Now, vmlens reports a data race in the file target/interleave/issues.html

A data race is an access to a field where a thread might read a stale value. If the thread, indeed, reads a stale value depends on external factors like which optimizations the compiler is using or on which hardware architecture the JVM is running and on which cores the threads are running. To make it possible to always detect such a data race independent of those external factors, vmlens searches for data races in the execution trace of the test run. And if vmlens have found one as in the example, it reports them in the issue report.

**Summary**

Tests for thread safety differ from typical single-threaded tests. To test if the combination of two methods, a and b, is thread-safe, call them from two different threads. Put the complete test in a while loop iterating over all thread interleavings with the help from the class AllInterleavings from [vmlens](https://vmlens.com/). Test if the result is either an after b or b after a. And to test if a class is a thread-safe, test all combinations of modifying methods and all combinations of read-only methods together with modifying methods.

**Further Reading**

[7 Techniques for Thread-Safe Classes](https://dzone.com/articles/7-techniques-for-thread-safe-classes)

[What Does Thread-Safety Mean in Java?](https://dzone.com/articles/what-does-thread-safety-mean-in-java)

[5 Tips to Make Your Classes Thread-Safe](https://dzone.com/articles/5-tips-to-make-your-classes-thread-safe)

# 7 Techniques for Thread-Safe Classes

### There are plenty of ways to create thread-safe classes in Java. Here, we dive into state handling, message passing, volatile fields, and more.

[**[](https://dzone.com/users/1171173/thomas.krieger.html)**](https://dzone.com/users/1171173/thomas.krieger.html)**by**

[**Thomas Krieger**](https://dzone.com/users/1171173/thomas.krieger.html)

**·**

**Jan. 12, 18 ·**[**Java Zone**](https://dzone.com/java-jdk-development-tutorials-tools-news)**· Tutorial**

**Like** [**(24)**](https://dzone.com/articles/7-techniques-for-thread-safe-classes)

**Comment (1)**

**Save**

[**Tweet**](https://dzone.com/articles/7-techniques-for-thread-safe-classes)

**95.83k Views**

Almost every Java application uses threads. A web server like Tomcat process each request in a separate worker thread, fat clients process long-running requests in dedicated worker threads, and even batch processes use the java.util.concurrent.ForkJoinPool to improve performance.

It is, therefore, necessary to write classes in a thread-safe way, which can be achieved by one of the following techniques.

## No State

When multiple threads access the same instance or static variable, you must somehow coordinate the access to this variable. The easiest way to do this is simply by avoiding instance or static variables. Methods in classes without instance variables do only use local variables and method arguments. The following example shows such a method which is part of the class java.lang.Math:

public static int subtractExact(int x, int y) {

int r = x - y;

if (((x ^ y) & (x ^ r)) < 0) {

throw new ArithmeticException("integer overflow");

}

return r;

}

## No Shared State

If you cannot avoid state, do not share the state. The state should only be owned by a single thread. [An example of this technique is the event processing thread of the SWT or Swing graphical user interface frameworks](http://flylib.com/books/en/2.558.1/why_are_guis_single_threaded_.html).

You can achieve thread-local instance variables by extending the thread class and adding an instance variable. In the following example, the field pool and workQueue are local to a single worker thread.

package java.util.concurrent;

public class ForkJoinWorkerThread extends Thread {

final ForkJoinPool pool;

final ForkJoinPool.WorkQueue workQueue;

}

The other way to achieve thread-local variables is to use the class java.lang.ThreadLocal for the fields you want to make thread-local. Here is an example of an instance variable using java.lang.ThreadLocal:

public class CallbackState {

public static final ThreadLocal<CallbackStatePerThread> callbackStatePerThread =

new ThreadLocal<CallbackStatePerThread>()

{

@Override

protected CallbackStatePerThread initialValue()

{

return getOrCreateCallbackStatePerThread();

}

};

}

You wrap the type of your instance variable inside the java.lang.ThreadLocal. You can provide an initial value for your java.lang.ThreadLocal through the method initialValue().

The following shows how to use the instance variable:

CallbackStatePerThread callbackStatePerThread = CallbackState.callbackStatePerThread.get();

Through calling the method get(), you receive the object associated with the current thread.

Since, in application servers, a pool of many threads is used to process requests, java.lang.ThreadLocal leads to a high memory consumption in this environment. java.lang.ThreadLocal is therefore not recommended for classes executed by the request processing threads of an application server.

## Message Passing

If you do not share state using the above techniques, you need a way for the threads to communicate. A technique to do this is by passing messages between threads. You can implement message passing using a concurrent queue from the package java.util.concurrent. Or, better yet, use a framework like [Akka](https://akka.io/), a framework for actor style concurrency. The following example shows how to send a message with Akka:

target.tell(message, getSelf());

And receive a message:

@Override

public Receive createReceive() {

return receiveBuilder()

.match(String.class, s -> System.out.println(s.toLowerCase()))

.build();

}

## Immutable State

To avoid the problem where a sending thread changes the message when the message is read by another thread, messages should be immutable. [The Akka framework, therefore, has the convention that all messages have to be immutable](https://doc.akka.io/docs/akka/2.5.5/java/actors.html#messages-and-immutability)

When you implement an immutable class, you should declare its fields as final. This not only makes sure that the compiler can check that the fields are in fact immutable [but also makes them correctly initialized even when they are incorrectly published.](https://shipilev.net/blog/2014/jmm-pragmatics/#_part_v_finals) Here is an example of a final instance variable:

public class ExampleFinalField

{

private final int finalField;

public ExampleFinalField(int value)

{

this.finalField = value;

}

}

## Use the Data Structures From java.util.concurrent

Message passing uses concurrent queues for communication between threads. Concurrent queues are one of the data structures provided in the package java.util.concurrent. This package provides classes for concurrent maps, queues, dequeues, sets, and lists. Those data structures are highly optimized and tested for thread safety.

## Synchronized Blocks

If you cannot use one of the above techniques, use synchronized locks. By putting a lock inside a synchronized block, you make sure that only one thread at a time can execute this section.

synchronized(lock)

{

i++;

}

Beware that when you use multiple nested synchronize blocks, you risk deadlocks. [A deadlock happens when two threads are trying to acquire a lock held by the other thread.](http://vmlens.com/articles/detect_deadlocks/)

## Volatile Fields

Normal, nonvolatile fields can be cached in registers or caches. Through the declaration of a variable as volatile, you tell the JVM and the compiler to always return the latest written value. This not only applies to the variable itself, but to all values written by the thread that has written to the volatile field. The following shows an example of a volatile instance variable:

public class ExampleVolatileField

{

private volatile int volatileField;

}

[You can use volatile fields if the writes do not depend on the current value. Or if you can make sure that only one thread at a time can update the field.](http://vmlens.com/articles/3_tips_volatile_fields/)

## Even More Techniques

I excluded the following more advanced techniques from this list:

* Atomic updates: A technique in which you call atomic instructions like compare and set provided by the CPU
* java.util.concurrent.locks.ReentrantLock: A lock implementation that provides more flexibility than synchronized blocks
* java.util.concurrent.locks.ReentrantReadWriteLock: A lock implementation in which reads do not block reads
* java.util.concurrent.locks.StampedLock a nonreeantrant Read-Write lock with the possibility of optimistically reading values.

## Conclusion

The best way to achieve thread safety is to avoid shared state. For the state, you need to share you can either use message parsing together with immutable classes or the concurrent data structures together with synchronized blocks and volatile fields. And if you want to test whether your application is thread-safe, try [vmlens](http://vmlens.com/) for free.

I would be glad to hear from you about the techniques you use to achieve thread-safe classes.

### Like This Article? Read More From DZone

**[](https://dzone.com/articles/5-tips-for-high-performance-thread-safe-java-from?fromrel=true)**[**5 Tips for Performant, Thread-Safe Java From ConcurrentHashMap**](https://dzone.com/articles/5-tips-for-high-performance-thread-safe-java-from?fromrel=true)

**[](https://dzone.com/articles/how-to-write-thread-safe-yet-scalable-classes?fromrel=true)**[**How to Write Thread-Safe-Yet-Scalable Classes**](https://dzone.com/articles/how-to-write-thread-safe-yet-scalable-classes?fromrel=true)

**[](https://dzone.com/articles/5-ways-to-thread-safe-update-a-field-in-java?fromrel=true)**[**Five Ways to Update Fields in a Thread Safe Way: Part 1**](https://dzone.com/articles/5-ways-to-thread-safe-update-a-field-in-java?fromrel=true)

**[](https://dzone.com/refcardz/getting-started-with-headless-cms?fromrel=true)**

**[Free DZone Refcard](https://dzone.com/refcardz/getting-started-with-headless-cms?fromrel=true)**

**[Getting Started With Headless CMS](https://dzone.com/refcardz/getting-started-with-headless-cms?fromrel=true)**

Topics:

JAVA ,THREAD SAFE ,CONCURRENT PROGRAMMING ,STATE ,TUTORIAL

**Like** [**(24)**](https://dzone.com/articles/7-techniques-for-thread-safe-classes)

**Comment (1)**

**Save**

[**Tweet**](https://dzone.com/articles/7-techniques-for-thread-safe-classes)

# Tips to Make Your Classes Thread Safe

### Remember folks, safety first! Here are five ways to ensure your classes are safe.

[**[](https://dzone.com/users/1171173/thomas.krieger.html)**](https://dzone.com/users/1171173/thomas.krieger.html)**by**

[**Thomas Krieger**](https://dzone.com/users/1171173/thomas.krieger.html)

**·**

**Oct. 01, 15 ·**[**Java Zone**](https://dzone.com/java-jdk-development-tutorials-tools-news)**· Tutorial**

**Like** [**(8)**](https://dzone.com/articles/5-tips-to-make-your-classes-thread-safe)

**Comment (2)**

**Save**

[**Tweet**](https://dzone.com/articles/5-tips-to-make-your-classes-thread-safe)

**30.21k Views**

While testing [vmlens](http://vmlens.com/), a tool to find data races in java applications, on open source projects, I found the following 5 tricks how to make classes thread safe.

## 1) Declare Immutable Member Variables as Final

Always declare immutable member variables as final. This makes sure that your class behaves correctly independent on how it is used. Take for example the field fieldAccessor in the class java.lang.reflect.Field.

private FieldAccessor fieldAccessor;

private FieldAccessor getFieldAccessor(Object obj)

throws IllegalAccessException

{

boolean ov = override;

FieldAccessor a = (ov) ? overrideFieldAccessor : fieldAccessor;

return (a != null) ? a : acquireFieldAccessor(ov);

}

Since it is not synchronized and not declared volatile, a thread reading this field might not see a completely initialized object as described in [DoubleCheckedLocking](http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html) But since the created object type sun.reflect.UnsafeQualifiedIntegerFieldAccessorImpl only uses final problem, there is no problem. Threads reading this field will always see a fully initialized object or null.

## 2) Create Objects Eagerly

Using final fields forces you to initialize your objects in the constructor. Lazy initialization of your objects on the other side is almost never a good idea in concurrent programs.

Take for example the old version from org.apache.commons.lang.StringEscapeUtils. It uses the lazy initialized class org.apache.commons.lang.Entities$LookupEntityMap:

private String[] lookupTable() {

if (lookupTable == null) {

createLookupTable();

}

return lookupTable;

}

This only works with locks or synchronization. Much better is the new version org.apache.commons.lang3.StringEscapeUtils witch eagerly creates the lookup tables and also uses a final field.

public static final CharSequenceTranslator ESCAPE\_XML10 =

new AggregateTranslator(

...

## 3) Use Volatile for Mutable Boolean Variables

Mutable boolean fields are often used for controlling the flow of your application. For example to control the life cycle of a thread the following pattern can be used:

private volatile boolean isWorking;

while(isWorking)

{

// do something

}

Use a volatile field to make the changes done in one thread visible in other threads.

## 4) Check 3rd Party Classes

A typical example for not doing so, is the use of the non thread safe java.util.date as member variable without synchronization. Therefore always check if the class is documented as thread safe. If not chances are high that it is not.

## 5) Test

Like all other features of your application, concurrency must be tested. In my next Blog post I will write how to test concurrency. In the meanwhile you can give [vmlens](http://vmlens.com/) a trial, which helps you to detect data races during testing.

# The "Double-Checked Locking is Broken" Declaration

Signed by: [David Bacon](http://www.research.ibm.com/people/d/dfb) (IBM Research) Joshua Bloch (Javasoft), [Jeff Bogda](http://www.cs.ucsb.edu/~bogda/), Cliff Click (Hotspot JVM project), [Paul Haahr](http://www.webcom.com/~haahr/), [Doug Lea](http://www.cs.oswego.edu/~dl), [Tom May](mailto:tom@go2net.com), [Jan-Willem Maessen](http://www.csg.lcs.mit.edu/~earwig/), [Jeremy Manson](http://www.cs.umd.edu/~jmanson), [John D. Mitchell (jGuru)](http://www.jguru.com/johnm) Kelvin Nilsen, [Bill Pugh](http://www.cs.umd.edu/~pugh), [Emin Gun Sirer](http://www.cs.washington.edu/homes/egs/)

Double-Checked Locking is widely cited and used as an efficient method for implementing lazy initialization in a multithreaded environment.

Unfortunately, it will not work reliably in a platform independent way when implemented in Java, without additional synchronization. When implemented in other languages, such as C++, it depends on the memory model of the processor, the reorderings performed by the compiler and the interaction between the compiler and the synchronization library. Since none of these are specified in a language such as C++, little can be said about the situations in which it will work. Explicit memory barriers can be used to make it work in C++, but these barriers are not available in Java.

To first explain the desired behavior, consider the following code:

|  |
| --- |
| // Single threaded version  class Foo {  private Helper helper = null;  public Helper getHelper() {  if (helper == null)  helper = new Helper();  return helper;  }  // other functions and members...  } |

If this code was used in a multithreaded context, many things could go wrong. Most obviously, two or more Helper objects could be allocated. (We'll bring up other problems later). The fix to this is simply to synchronize the getHelper() method:

|  |
| --- |
| // Correct multithreaded version  class Foo {  private Helper helper = null;  public synchronized Helper getHelper() {  if (helper == null)  helper = new Helper();  return helper;  }  // other functions and members...  } |

The code above performs synchronization every time getHelper() is called. The double-checked locking idiom tries to avoid synchronization after the helper is allocated:

|  |
| --- |
| // Broken multithreaded version  // "Double-Checked Locking" idiom  class Foo {  private Helper helper = null;  public Helper getHelper() {  if (helper == null)  synchronized(this) {  if (helper == null)  helper = new Helper();  }  return helper;  }  // other functions and members...  } |

Unfortunately, that code just does not work in the presence of either optimizing compilers or shared memory multiprocessors.

## It doesn't work

There are lots of reasons it doesn't work. The first couple of reasons we'll describe are more obvious. After understanding those, you may be tempted to try to devise a way to "fix" the double-checked locking idiom. Your fixes will not work: there are more subtle reasons why your fix won't work. Understand those reasons, come up with a better fix, and it still won't work, because there are even more subtle reasons.

Lots of very smart people have spent lots of time looking at this. There is no way to make it work without requiring each thread that accesses the helper object to perform synchronization.

### The first reason it doesn't work

The most obvious reason it doesn't work it that the writes that initialize the Helper object and the write to the helper field can be done or perceived out of order. Thus, a thread which invokes getHelper() could see a non-null reference to a helper object, but see the default values for fields of the helper object, rather than the values set in the constructor.

If the compiler inlines the call to the constructor, then the writes that initialize the object and the write to the helper field can be freely reordered if the compiler can prove that the constructor cannot throw an exception or perform synchronization.

Even if the compiler does not reorder those writes, on a multiprocessor the processor or the memory system may reorder those writes, as perceived by a thread running on another processor.

Doug Lea has written a [more detailed description of compiler-based reorderings](http://gee.cs.oswego.edu/dl/cpj/jmm.html).

#### A test case showing that it doesn't work

Paul Jakubik found an example of a use of double-checked locking that did not work correctly. [A slightly cleaned up version of that code is available here](http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckTest.java).

When run on a system using the Symantec JIT, it doesn't work. In particular, the Symantec JIT compiles

singletons[i].reference = new Singleton();

to the following (note that the Symantec JIT using a handle-based object allocation system).

0206106A mov eax,0F97E78h

0206106F call 01F6B210 ; allocate space for

; Singleton, return result in eax

02061074 mov dword ptr [ebp],eax ; EBP is &singletons[i].reference

; store the unconstructed object here.

02061077 mov ecx,dword ptr [eax] ; dereference the handle to

; get the raw pointer

02061079 mov dword ptr [ecx],100h ; Next 4 lines are

0206107F mov dword ptr [ecx+4],200h ; Singleton's inlined constructor

02061086 mov dword ptr [ecx+8],400h

0206108D mov dword ptr [ecx+0Ch],0F84030h

As you can see, the assignment to singletons[i].reference is performed before the constructor for Singleton is called. This is completely legal under the existing Java memory model, and also legal in C and C++ (since neither of them have a memory model).

#### A fix that doesn't work

Given the explanation above, a number of people have suggested the following code:

|  |
| --- |
| // (Still) Broken multithreaded version  // "Double-Checked Locking" idiom  class Foo {  private Helper helper = null;  public Helper getHelper() {  if (helper == null) {  Helper h;  synchronized(this) {  h = helper;  if (h == null)  synchronized (this) {  h = new Helper();  } // release inner synchronization lock  helper = h;  }  }  return helper;  }  // other functions and members...  } |

This code puts construction of the Helper object inside an inner synchronized block. The intuitive idea here is that there should be a memory barrier at the point where synchronization is released, and that should prevent the reordering of the initialization of the Helper object and the assignment to the field helper.

Unfortunately, that intuition is absolutely wrong. The rules for synchronization don't work that way. The rule for a monitorexit (i.e., releasing synchronization) is that actions before the monitorexit must be performed before the monitor is released. However, there is no rule which says that actions after the monitorexit may not be done before the monitor is released. It is perfectly reasonable and legal for the compiler to move the assignment helper = h; inside the synchronized block, in which case we are back where we were previously. Many processors offer instructions that perform this kind of one-way memory barrier. Changing the semantics to require releasing a lock to be a full memory barrier would have performance penalties.

#### More fixes that don't work

There is something you can do to force the writer to perform a full bidirectional memory barrier. This is gross, inefficient, and is almost guaranteed not to work once the Java Memory Model is revised. Do not use this. In the interests of science, [I've put a description of this technique on a separate page.](http://www.cs.umd.edu/~pugh/java/memoryModel/BidirectionalMemoryBarrier.html) Do not use it.

However, even with a full memory barrier being performed by the thread that initializes the helper object, it still doesn't work.

The problem is that on some systems, the thread which sees a non-null value for the helper field also needs to perform memory barriers.

Why? Because processors have their own locally cached copies of memory. On some processors, unless the processor performs a cache coherence instruction (e.g., a memory barrier), reads can be performed out of stale locally cached copies, even if other processors used memory barriers to force their writes into global memory.

I've created [a separate web page](http://www.cs.umd.edu/~pugh/java/memoryModel/AlphaReordering.html) with a discussion of how this can actually happen on an Alpha processor.

## Is it worth the trouble?

For most applications, the cost of simply making the getHelper() method synchronized is not high. You should only consider this kind of detailed optimizations if you know that it is causing a substantial overhead for an application.

Very often, more high level cleverness, such as using the builtin mergesort rather than handling exchange sort (see the SPECJVM DB benchmark) will have much more impact.

## Making it work for static singletons

If the singleton you are creating is static (i.e., there will only be one Helper created), as opposed to a property of another object (e.g., there will be one Helper for each Foo object, there is a simple and elegant solution.

Just define the singleton as a static field in a separate class. The semantics of Java guarantee that the field will not be initialized until the field is referenced, and that any thread which accesses the field will see all of the writes resulting from initializing that field.

|  |
| --- |
| class HelperSingleton {  static Helper singleton = new Helper();  } |

## It will work for 32-bit primitive values

Although the double-checked locking idiom cannot be used for references to objects, it can work for 32-bit primitive values (e.g., int's or float's). Note that it does not work for long's or double's, since unsynchronized reads/writes of 64-bit primitives are not guaranteed to be atomic.

|  |
| --- |
| // Correct Double-Checked Locking for 32-bit primitives  class Foo {  private int cachedHashCode = 0;  public int hashCode() {  int h = cachedHashCode;  if (h == 0)  synchronized(this) {  if (cachedHashCode != 0) return cachedHashCode;  h = computeHashCode();  cachedHashCode = h;  }  return h;  }  // other functions and members...  } |

In fact, assuming that the computeHashCode function always returned the same result and had no side effects (i.e., idempotent), you could even get rid of all of the synchronization.

|  |
| --- |
| // Lazy initialization 32-bit primitives  // Thread-safe if computeHashCode is idempotent  class Foo {  private int cachedHashCode = 0;  public int hashCode() {  int h = cachedHashCode;  if (h == 0) {  h = computeHashCode();  cachedHashCode = h;  }  return h;  }  // other functions and members...  } |

## Making it work with explicit memory barriers

It is possible to make the double checked locking pattern work if you have explicit memory barrier instructions. For example, if you are programming in C++, you can use the code from Doug Schmidt et al.'s book:

|  |
| --- |
| // C++ implementation with explicit memory barriers  // Should work on any platform, including DEC Alphas  // From "Patterns for Concurrent and Distributed Objects",  // by Doug Schmidt  template <class TYPE, class LOCK> TYPE \*  Singleton<TYPE, LOCK>::instance (void) {  // First check  TYPE\* tmp = instance\_;  // Insert the CPU-specific memory barrier instruction  // to synchronize the cache lines on multi-processor.  asm ("memoryBarrier");  if (tmp == 0) {  // Ensure serialization (guard  // constructor acquires lock\_).  Guard<LOCK> guard (lock\_);  // Double check.  tmp = instance\_;  if (tmp == 0) {  tmp = new TYPE;  // Insert the CPU-specific memory barrier instruction  // to synchronize the cache lines on multi-processor.  asm ("memoryBarrier");  instance\_ = tmp;  }  return tmp;  } |

## Fixing Double-Checked Locking using Thread Local Storage

Alexander Terekhov (TEREKHOV@de.ibm.com) came up clever suggestion for implementing double checked locking using thread local storage. Each thread keeps a thread local flag to determine whether that thread has done the required synchronization.

|  |
| --- |
| class Foo {  /\*\* If perThreadInstance.get() returns a non-null value, this thread  has done synchronization needed to see initialization  of helper \*/  private final ThreadLocal perThreadInstance = new ThreadLocal();  private Helper helper = null;  public Helper getHelper() {  if (perThreadInstance.get() == null) createHelper();  return helper;  }  private final void createHelper() {  synchronized(this) {  if (helper == null)  helper = new Helper();  }  // Any non-null value would do as the argument here  perThreadInstance.set(perThreadInstance);  }  } |

The performance of this technique depends quite a bit on which JDK implementation you have. In Sun's 1.2 implementation, ThreadLocal's were very slow. They are significantly faster in 1.3, and are expected to be faster still in 1.4. [Doug Lea analyzed the performance of some techniques for implementing lazy initialization](http://www.cs.umd.edu/~pugh/java/memoryModel/DCL-performance.html).

## Under the new Java Memory Model

As of JDK5, there is [a new Java Memory Model and Thread specification](http://www.cs.umd.edu/~pugh/java/memoryModel).

### Fixing Double-Checked Locking using Volatile

JDK5 and later extends the semantics for volatile so that the system will not allow a write of a volatile to be reordered with respect to any previous read or write, and a read of a volatile cannot be reordered with respect to any following read or write. See [this entry in Jeremy Manson's blog](http://jeremymanson.blogspot.com/2008/05/double-checked-locking.html) for more details.

With this change, the Double-Checked Locking idiom can be made to work by declaring the helper field to be volatile. This does not work under JDK4 and earlier.

|  |
| --- |
| // Works with acquire/release semantics for volatile  // Broken under current semantics for volatile  class Foo {  private volatile Helper helper = null;  public Helper getHelper() {  if (helper == null) {  synchronized(this) {  if (helper == null)  helper = new Helper();  }  }  return helper;  }  } |

### Double-Checked Locking Immutable Objects

If Helper is an immutable object, such that all of the fields of Helper are final, then double-checked locking will work without having to use volatile fields. The idea is that a reference to an immutable object (such as a String or an Integer) should behave in much the same way as an int or float; reading and writing references to immutable objects are atomic.

# Descriptions of double-check idiom

* [Reality Check](http://www.cs.wustl.edu/~schmidt/editorial-3.html), Douglas C. Schmidt, C++ Report, SIGS, Vol. 8, No. 3, March 1996.
* [Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects](http://www.cs.wustl.edu/~schmidt/DC-Locking.ps.gz), Douglas Schmidt and Tim Harrison. 3rd annual Pattern Languages of Program Design conference, 1996
* [Lazy instantiation](http://www.javaworld.com/javaworld/javatips/jw-javatip67.html), Philip Bishop and Nigel Warren, JavaWorld Magazine
* [Programming Java threads in the real world, Part 7](http://www.javaworld.com/javaworld/jw-04-1999/jw-04-toolbox-3.html), Allen Holub, Javaworld Magazine, April 1999.
* [Java 2 Performance and Idiom Guide](http://www.phptr.com/ptrbooks/ptr_0130142603.html), Craig Larman and Rhett Guthrie, p100.
* [Java in Practice: Design Styles and Idioms for Effective Java](http://www.google.com/search?q=Java+Design+Styles+Nigel+Bishop), Nigel Warren and Philip Bishop, p142.
* Rule 99, [The Elements of Java Style](http://www.google.com/search?q=elements+java+style+Ambler), Allan Vermeulen, Scott Ambler, Greg Bumgardner, Eldon Metz, Trvor Misfeldt, Jim Shur, Patrick Thompson, SIGS Reference library
* [Global Variables in Java with the Singleton Pattern](http://gamelan.earthweb.com/journal/techfocus/022300_singleton.html), Wiebe de Jong, Gamelan