

Java Concurrency

- 1 [Java Concurrency / Multithreading Tutorial](#)
- 2 [Multithreading Benefits](#)
- 3 [Multithreading Costs](#)
- 4 [Creating and Starting Java Threads](#)
- 5 [Race Conditions and Critical Sections](#)
- 6 [Thread Safety and Shared Resources](#)
- 7 [Thread Safety and Immutability](#)
- 8 [Java Memory Model](#)
- 9 [Java Synchronized Blocks](#)
- 10 **[Java's Volatile Keyword](#)**
- 11 [Java ThreadLocal](#)
- 12 [Thread Signaling](#)
- 13 [Deadlock](#)
- 14 [Deadlock Prevention](#)
- 15 [Starvation and Fairness](#)
- 16 [Nested Monitor Lockout](#)
- 17 [Slipped Conditions](#)
- 18 [Locks in Java](#)
- 19 [Read / Write Locks in Java](#)
- 20 [Reentrance Lockout](#)
- 21 [Semaphores](#)
- 22 [Blocking Queues](#)
- 23 [Thread Pools](#)
- 24 [Compare and Swap](#)
- 25 [Anatomy of a Synchronizer](#)

Java's Volatile Keyword



By Jakob Jenkov

Connect with me:



Rate article:



Share article:



Table of Contents

- [Java volatile Guarantees Variable Visibility](#)
- [The Java volatile Guarantee](#)
- [When is volatile Enough?](#)
- [Performance Considerations of volatile](#)

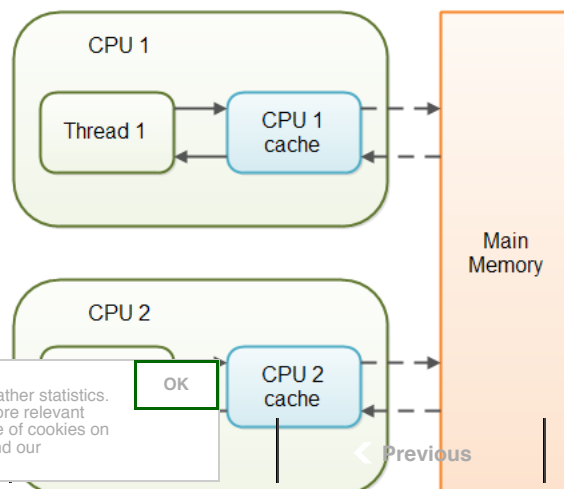
The Java `volatile` keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

Actually, since Java 5 the `volatile` keyword guarantees more than just that volatile variables are written and read from main memory. I will explain that in the following sections.

Java volatile Guarantees Variable Visibility

The Java `volatile` keyword guarantees visibility of changes to variables across threads. This may sound abstract, so let me elaborate.

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs. This diagram illustrates such a situation:



With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. Let me explain what problems that can cause with an example:

Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

What Does

Find out in the Retail Tech Forecast

Free Download

This website uses cookies to improve the user experience and gather statistics. Our advertisers use cookies too (3rd party cookies), to provide more relevant ads. Continued use of this website implies that you accept the use of cookies on this website. We do not share our cookies with our advertisers, and our advertisers do not share cookies with us.



Get all my free tips & tutorials!

Connect with me, or sign up for my news letter or [RSS feed](#), and get all my tips that help you become a more skilled and efficient developer.



Newsletter

First Name * Last Name *

Email *

☐ Yes, give me tips!

```
public class SharedObject {
    public int counter = 0;
}
```

Thread 1 could read a shared `counter` variable with the value 0 into its CPU cache, increment it to 1 and write the changed value back into main memory. Thread 2 could then read the same `counter` variable from main memory where the value of the variable is still 0, into its own CPU cache. Thread 2 could then also increment the counter to 1, and also not write it back to main memory. Thread 1 and Thread 2 are now practically out of sync. The real value of the shared `counter` variable should have been 2, but each of the threads has the value 1 for the variable in their CPU caches, and in main memory the value is still 0. It is a mess! Even if the threads eventually write their value for the shared `counter` variable back to main memory, the value will be wrong.

By declaring the shared `counter` variable `volatile` the JVM guarantees that every read of the variable always be read from main memory, and that all writes to the variable will always be written back to main memory. Here is how the `volatile` declaration looks:

```
public class SharedObject {
    public volatile int counter = 0;
}
```

In some cases simply declaring a variable `volatile` may be enough to assure that multiple threads accessing a variable see the latest written value. I will get back to which cases `volatile` is sufficient later.

In the situation with the two threads reading and writing the same variable, simply declaring the variable `volatile` is not enough. Thread 1 may read the `counter` value 0 into a CPU register in CPU 1. At the same time (after) Thread 2 may read the `counter` value 0 into a CPU register in CPU 2. Both threads have read the value directly from main memory. Now both threads increase the value and write the value back to main memory. They both increment their register version of `counter` to 1, and both write the value 1 back to main memory. The value should have been 2 after two increments.

The problem with multiple threads that do not see the latest value of a variable because that value has not been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

The Java volatile Guarantee

Since Java 5 the `volatile` keyword guarantees more than just the reading and writing of a variable from/to main memory. Actually, the `volatile` keyword guarantees this:

- If Thread A writes to a `volatile` variable and Thread B subsequently reads the same `volatile` variable, all variables visible to Thread A before writing the `volatile` variable, will also be visible to Thread B.
- The reading and writing instructions of `volatile` variables cannot be reordered by the JVM (the JVM may reorder instructions for performance reasons as long as the JVM detects no change in program behavior from the reordering). Instructions before and after can be reordered, but the `volatile` read or write cannot be mixed with these instructions. Whatever instructions follow a read or write of a `volatile` variable are guaranteed to happen after the read or write.

Look at this example:

```
Thread A:
sharedObject.nonVolatile = 123;
sharedObject.counter = sharedObject.counter + 1;

Thread B:
int counter = sharedObject.counter;
int nonVolatile = sharedObject.nonVolatile;
```

Since Thread A writes the non-volatile variable `sharedObject.nonVolatile` before writing to the volatile `sharedObject.counter`, then both `sharedObject.nonVolatile` and `sharedObject.counter` are written to main memory.

Since Thread B starts by reading the volatile `sharedObject.counter`, then both the `sharedObject.counter` and `sharedObject.nonVolatile` are read from main memory.

The reading and writing of the non-volatile variable cannot be reordered to happen before or after the reading and writing of the volatile variable.

When is volatile Enough?

As I have mentioned earlier, if two threads are both reading and writing to a shared variable, then using the `volatile` keyword for that is not enough. You need to use **synchronization** in that case to guarantee that reading and writing of the variable is atomic.

But in case one thread reads and writes the value of a `volatile` variable, and other threads only read the value, then the reading threads are guaranteed to see the latest value written to the `volatile` variable. Without making the variable `volatile`, this would not be guaranteed.

Performance Considerations of volatile

Reading and writing of volatile variables causes the variable to be read or written to main memory. Reading and writing to main memory is more expensive than accessing the CPU cache. Accessing volatile variables prevent instruction reordering which is a normal performance enhancement technique. Thus, you should use volatile variables when you really need to enforce visibility of variables.

Next: [Java ThreadLocal](#)

Connect with me:



Newsletter - Get all my free tips!

<input type="text" value="First Name *"/>	<input type="text" value="Last Name *"/>	<input type="text" value="Email *"/>
<input type="button" value="Yes, give me tips!"/>		

NCache Goes Open Source!
Most Popular .NET Cache Now FREE




Released under Apache License, Version 2.0
[Download Now >](#)