Java Volatile Keyword

Jakob Jenkov
Last update: 2018-02-23

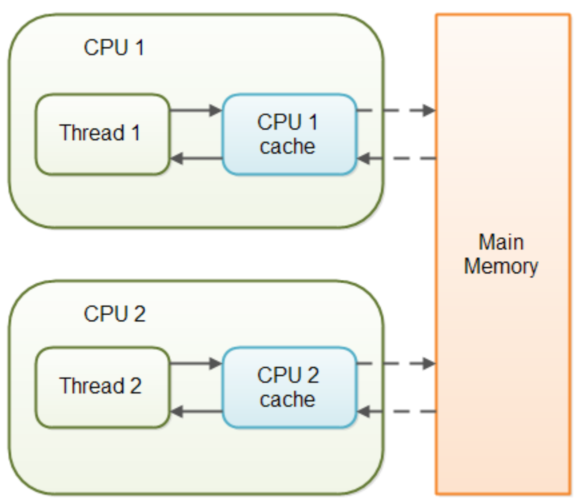
The Java `volatile` keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main mem and not from the CPU cache, and that every write to a volatile variable will be written to main memory,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

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

The Java volatile Visibility Guarantee

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

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



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. This can cause several problems which I will explain in the following sections.

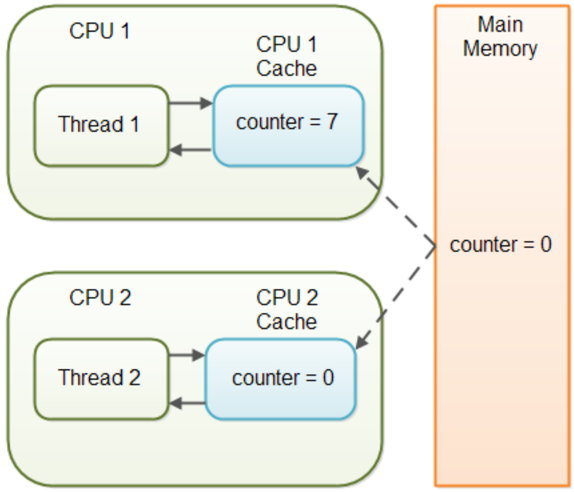
Imagine a situation in which two or more threads have access to a shared object which contains a cou variable declared like this:

```
public class SharedObject {
```

```
}
```

Imagine too, that only Thread 1 increments the `counter` variable, but both Thread 1 and Thread 2 may read the `counter` variable from time to time.

If the `counter` variable is not declared `volatile` there is no guarantee about when the value of the `cour` variable is written from the CPU cache back to main memory. This means, that the `counter` variable va in the CPU cache may not be the same as in main memory. This situation is illustrated here:



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

By declaring the `counter` variable `volatile` all writes to the `counter` variable will be written back to mai memory immediately. Also, all reads of the `counter` variable will be read directly from main memory. H is how the `volatile` declaration of the `counter` variable looks:

```
public class SharedObject {

    public volatile int counter = 0;

}
```

Declaring a variable `volatile` thus *guarantees the visibility* for other threads of writes to that variable.

Still a Race Condition

Please notice, that the above example counter is just an example I used to explain how the Java `vola` keyword affects when values are read and written from and to main memory. However, simply declarir the `counter` variable `volatile` is not enough to create a fully thread safe shared counter. Let me explai why:

When two threads first read, test and then write a new value based on an older value to a variable, ev the variable is declared `volatile` you will still have a race condition. Imagine this situation where two threads named T1 and T2 are reading, testing and writing to a shared `volatile` variable:

```
T1: Read counter variable - value is 6
T2: Read counter variable - value is 6

T1: if counter < 100 add 1 to counter
    -> so adds 1 to counter which is now 7 in T1
    -> counter is immediately written back to main memory

T2: if counter < 100 add 1 to counter
    -> so adds 1 to counter which is now 7 in T2
    -> counter is immediately written back to main memory

Value of counter in main memory is now 7,
but should have been 8
```

Notice how the two threads T1 and T2 read the shared `counter` variable directly from main memory, a write the updated values directly back to main memory, and yet because the read-test-write operation not carried out as a single atomic operation, both threads read the pre-incremented value of the share counter which is 7. Both threads then increment the counter and write their incremented value directly back to main memory, where T2 does not see T1's incrementation of the `counter` in main memory. T2 effectively "ignores" T1's incrementation, and sets the value to 7, even though the correct value would

So, as you can see, if two threads are incrementing the shared counter simultaneously, an error can s occur, even if both write the incremented value back to main memory as soon as it has been incremer

To make the read-test-write operation a single atomic operation, you could use one of the atomic primitives in the `java.util.concurrent` package. For instance, you could use **AtomicInteger** or **AtomicLong** .

I also have a discussion of atomic compare-and-swap operations in my **Java compare and swap tutorial**.

The Java volatile Happens-Before Guarantee

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

- If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile varia then all variables visible to Thread A *before* writing the volatile variable, will also be visible to Thr B *after* it has read the volatile variable.

- The reading and writing instructions of volatile variables cannot be reordered by the JVM (the JV may reorder instructions for performance reasons as long as the JVM detects no change in prog behaviour from the reordering). Instructions before and after can be reordered, but the volatile re 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.

These statements require a deeper explanation.

When a thread writes to a volatile variable, then not just the volatile variable itself is written to main memory. Also all other variables changed by the thread before writing to the volatile variable are also flushed to main memory. When a thread reads a volatile variable it will also read all other variables fro main memory which were flushed to main memory together with the volatile variable.

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 m memory when Thread A writes to `sharedObject.counter` (the `volatile` variable).

Since Thread B starts by reading the volatile `sharedObject.counter`, then both the `sharedObject.count` and `sharedObject.nonVolatile` are read from main memory into the CPU cache used by Thread B. By time Thread B reads `sharedObject.nonVolatile` it will see the value written by Thread A.

Developers may use this extended visibility guarantee to optimize the visibility of variables between threads. Instead of declaring each and every variable `volatile`, only one or a few need be declared `volatile`. Here is an example of a simple `Exchanger` class written after that principle:

```
public class Exchanger {

    private Object   object       = null;
    private volatile hasNewObject = false;

    public void put(Object newObject) {
        while(hasNewObject) {
            //wait - do not overwrite existing new object
        }
        object = newObject;
        hasNewObject = true; //volatile write
    }

    public Object take(){
        while(!hasNewObject){ //volatile read
            //wait - don't take old object (or null)
        }
        Object obj = object;
        hasNewObject = false; //volatile write
        return obj;
    }
}
```

Thread A may be putting objects from time to time by calling `put()`. Thread B may take objects from ti to time by calling `take()`. This `Exchanger` can work just fine using a `volatile` variable (without the use o `synchronized` blocks), as long as only Thread A calls `put()` and only Thread B calls `take()`.

However, the JVM may reorder Java instructions to optimize performance, if the JVM can do so withou

```
while(hasNewObject) {
    //wait - do not overwrite existing new object
}
hasNewObject = true; //volatile write
object = newObject;
```

Notice the write to the `volatile` variable `hasNewObject` is now executed before the new object is actual set. To the JVM this may look completely valid. The values of the two write instructions do not depend each other.

However, reordering the instruction execution would harm the visibility of the `object` variable. First of a Thread B might see `hasNewObject` set to `true` before Thread A has actually written a new value to the `object` variable. Second, there is now not even a guarantee about when the new value written to `obje` will be flushed back to main memory (well - the next time Thread A writes to a volatile variable somewhere...).

To prevent situations like the one described above from occurring, the `volatile` keyword comes with a "*happens before guarantee*". The happens before guarantee guarantees that read and write instructio of `volatile` variables cannot be reordered. Instructions before and after can be reordered, but the vola read/write instruction cannot be reordered with any instruction occurring before or after it.

Look at this example:

```
sharedObject.nonVolatile1 = 123;
sharedObject.nonVolatile2 = 456;
sharedObject.nonVolatile3 = 789;

sharedObject.volatile     = true; //a volatile variable

int someValue1 = sharedObject.nonVolatile4;
int someValue2 = sharedObject.nonVolatile5;
int someValue3 = sharedObject.nonVolatile6;
```

The JVM may reorder the first 3 instructions, as long as all of them *happens before* the `volatile` write instruction (they must all be executed before the volatile write instruction).

Similarly, the JVM may reorder the last 3 instructions as long as the volatile write instruction *happens before* all of them. None of the last 3 instructions can be reordered to before the volatile write instructi

That is basically the meaning of the Java volatile happens before guarantee.

volatile is Not Always Enough

Even if the `volatile` keyword guarantees that all reads of a `volatile` variable are read directly from m memory, and all writes to a `volatile` variable are written directly to main memory, there are still situati where it is not enough to declare a variable `volatile`.

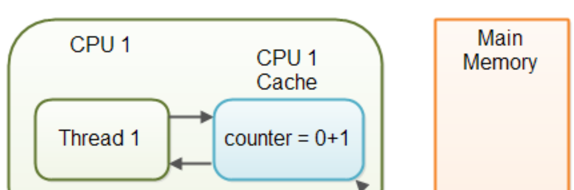
In the situation explained earlier where only Thread 1 writes to the shared `counter` variable, declaring `counter` variable `volatile` is enough to make sure that Thread 2 always sees the latest written value.

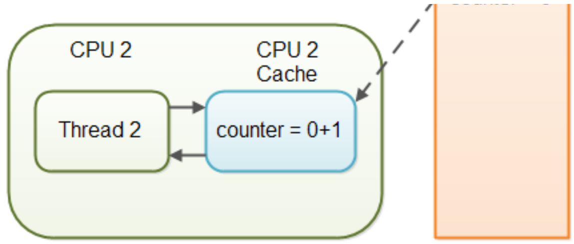
In fact, multiple threads could even be writing to a shared `volatile` variable, and still have the correct value stored in main memory, if the new value written to the variable does not depend on its previous value. In other words, if a thread writing a value to the shared `volatile` variable does not first need to its value to figure out its next value.

As soon as a thread needs to first read the value of a `volatile` variable, and based on that value gene a new value for the shared `volatile` variable, a `volatile` variable is no longer enough to guarantee cc visibility. The short time gap in between the reading of the `volatile` variable and the writing of its new value, creates an **race condition** where multiple threads might read the same value of the `volatile` variable, generate a new value for the variable, and when writing the value back to main memory - overwrite each other's values.

The situation where multiple threads are incrementing the same counter is exactly such a situation wh a `volatile` variable is not enough. The following sections explain this case in more detail.

Imagine if Thread 1 reads a shared `counter` variable with the value 0 into its CPU cache, increment it t and not 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. This situation is illustrated in the diagram below:





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 main memory the value is still 0. It is a mess! Even if the threads eventually write their value for the sh `counter` variable back to main memory, the value will be wrong.

When is volatile Enough?

As I have mentioned earlier, if two threads are both reading and writing to a shared variable, then usin the `volatile` keyword for that is not enough. You need to use a **synchronized** in that case to guarant that the reading and writing of the variable is atomic. Reading or writing a volatile variable does not bl threads reading or writing. For this to happen you must use the `synchronized` keyword around critical sections.

As an alternative to a `synchronized` block you could also use one of the many atomic data types found the **`java.util.concurrent` package**. For instance, the **`AtomicLong`** or **`AtomicReference`** or one of the oth

In case only one thread reads and writes the value of a volatile variable and other threads only read th variable, 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.

The `volatile` keyword is guaranteed to work on 32 bit and 64 variables.

Performance Considerations of volatile

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

Next: **Java ThreadLocal**

Jakob Jenkov