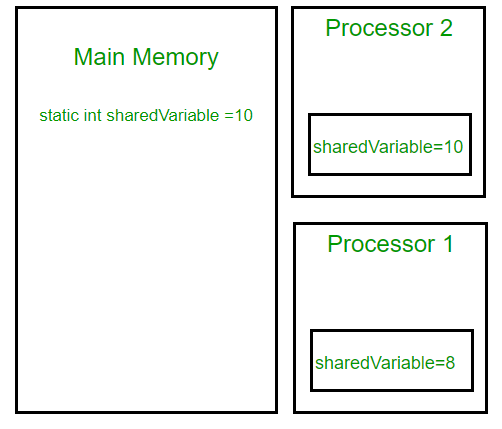
**Volatile keyword in java**Now, there is no qualifier concept in java. So, volatile in c is slightly different than volatile in java. Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

Suppose that two threads are working on SharedObj. If two threads run on different processors each thread may have its own local copy of sharedVariable. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the write policy of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.



Although most modern hardware provide good cache coherence therefore most probably the changes in one cache are reflected in other but it’s not a good practice to rely on hardware for to ‘fix’ a faulty application.

**Never Confuse between Static And Volatile**static variables are class members that are shared among all objects. There is only one copy of them in main memory.

Whereas, the for volatile class members, each thread can have it’s own copy of sharable objects (in cache) if they are running on different processors(all modern day computers have multi core processors)

**Note:** You can use both at the same time.

**An Example:**

public class ASingleton {

private static volatile ASingleton instance;

private static Object mutex = new Object();

private ASingleton() {

}

public static ASingleton getInstance() {

ASingleton result = instance;

if (result == null) {

synchronized (mutex) {

result = instance;

if (result == null)

instance = result = new ASingleton();

}

}

return result;

}

}

**volatile vs synchronized:**

Before we move on let’s take a look at two important features of locks and synchronization.

**Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.

**Visibility:** It means that changes made by one thread to shared data are visible to other threads.

Java’s synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modifies the value of shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other thread trying to enter the block at the same time will be blocked and put to sleep.

In some cases we may only desire the visibility and not atomicity. Use of synchronized in such situation is an overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, use of volatile is limited to very restricted set of cases as most of the times atomicity is desired. For example a simple increment statement such as x = x + 1; or x++ seems to be a single operation but is s really a compound read-modify-write sequence of operations that must execute atomically.

**The Java volatile Happens-Before Guarantee**

To address the instruction reordering challenge, the Java volatile keyword gives a "happens-before" guarantee, in addition to the visibility guarantee. The happens-before guarantee guarantees that:

Reads from and writes to other variables cannot be reordered to occur after a write to a volatile variable, if the reads / writes originally occurred before the write to the volatile variable.

The reads / writes before a write to a volatile variable are guaranteed to "happen before" the write to the volatile variable. Notice that it is still possible for e.g. reads / writes of other variables located after a write to a volatile to be reordered to occur before that write to the volatile. Just not the other way around. From after to before is allowed, but from before to after is not allowed.

Reads from and writes to other variables cannot be reordered to occur before a read of a volatile variable, if the reads / writes originally occurred after the read of the volatile variable. Notice that it is possible for reads of other variables that occur before the read of a volatile variable can be reordered to occur after the read of the volatile. Just not the other way around. From before to after is allowed, but from after to before is not allowed.

The above happens-before guarantee assures that the visibility guarantee of the volatile keyword are being enforced.

So, what does that mean? **That means compiler cannot optimize the code when volatile variable is there.**

**volatile is Not Always Enough:**

Even if the volatile keyword guarantees that all reads of a volatile variable are read directly from main memory, and all writes to a volatile variable are written directly to main memory, there are still situations 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 the 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 read 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 generate a new value for the shared volatile variable, a volatile variable is no longer enough to guarantee correct 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 where 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 to 1 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:

Two threads have read a shared counter variable into their local CPU caches and incremented it.

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.

**When is volatile Enough?**

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 a synchronized in that case to guarantee that the reading and writing of the variable is atomic. Reading or writing a volatile variable does not block 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 in the java.util.concurrent package. For instance, the AtomicLong or AtomicReference or one of the others.

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

**Read The Example Again:**

public class ASingleton {

private static volatile ASingleton instance;

private static Object mutex = new Object();

private ASingleton() {

}

public static ASingleton getInstance() {

ASingleton result = instance;

if (result == null) {

synchronized (mutex) {

result = instance;

if (result == null)

instance = result = new ASingleton();

}

}

return result;

}

}

Now, there are some factors which you should check:

**private static volatile ASingleton instance;**

Now, static is a storage class specifier and volatile is a qualifier in c. Here, the concept is little different. Still those can be mixed. So, both can be mixed. Now, why volatile? So, in multi core multithreading environment the threads don’t have a local cached version of this variable.

Local variable result seems unnecessary. But it’s there to improve performance of our code. In cases where instance is already initialized (most of the time), the volatile field is only accessed once (due to “return result;” instead of “return instance;”). This can improve the method’s overall performance by as much as 25 percent.

(because, local variable can be cached to improve performance probably. But, volatile variable cannot be optimized)

Also, note this optimization:

**instance = result = new ASingleton();**

result=instance=new ASingleton() this would make instance accessed twice. And we should reduce the access to volatile.