**Non-Access modifiers**

Non-access modifiers provide information about the characteristics of a class, method, or variable to the JVM. Seven types of Non-Access modifiers are present in Java. They are –

1. **final**
2. **static**
3. **abstract**
4. **synchronized**
5. **volatile**
6. **transient**
7. **native**

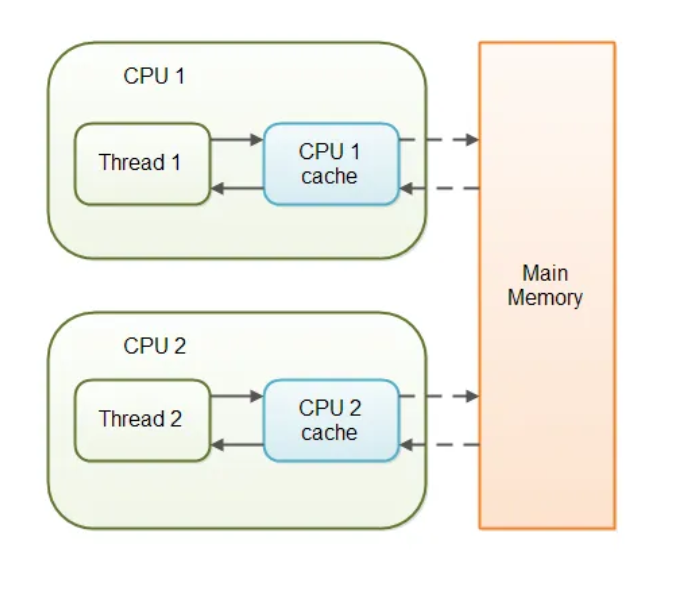
**the WHY:**

The volatile keyword in Java is used to mark a Java variable as “being stored in main memory”.

Every thread that accesses a volatile variable will read it from main memory, and not from the CPU cache. This way, all threads see the same value for the volatile variable.

The volatile keyword is often used with flags that indicate that a thread needs to stop running. For example, a thread might have a boolean flag called “done”, and when another thread sets this flag to “true”, the first thread will know to stop running. Without the volatile keyword, the first thread might keep running indefinitely, because it would never see the updated value of the “done” flag.

There are other uses for the volatile keyword as well, but this is one of the most common. In general, if you need to make sure that all threads see the same value for a certain variable, then you should mark that variable as being volatile.

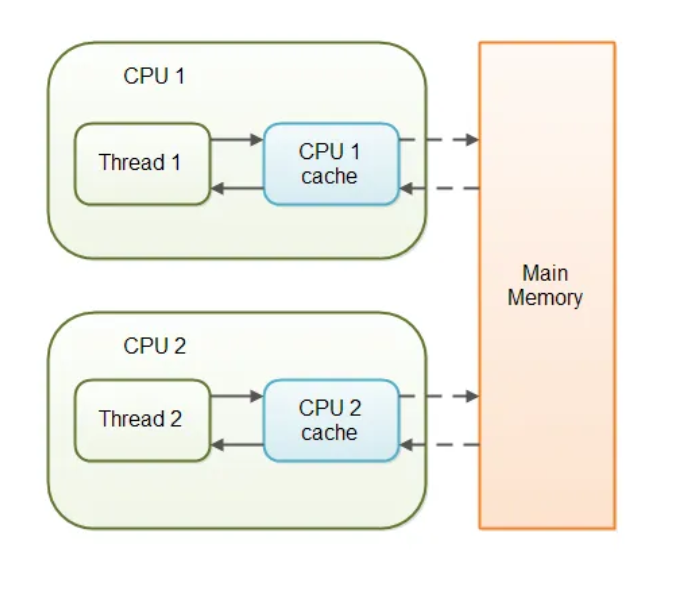


The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU registers, and that every write to a volatile variable will be written to main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

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

The Java volatile keyword guarantees visibility of changes to variables across threads. This may sound a bit 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 registers 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 registers 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 registers, or writes data from CPU registers to main memory. This can cause several problems

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

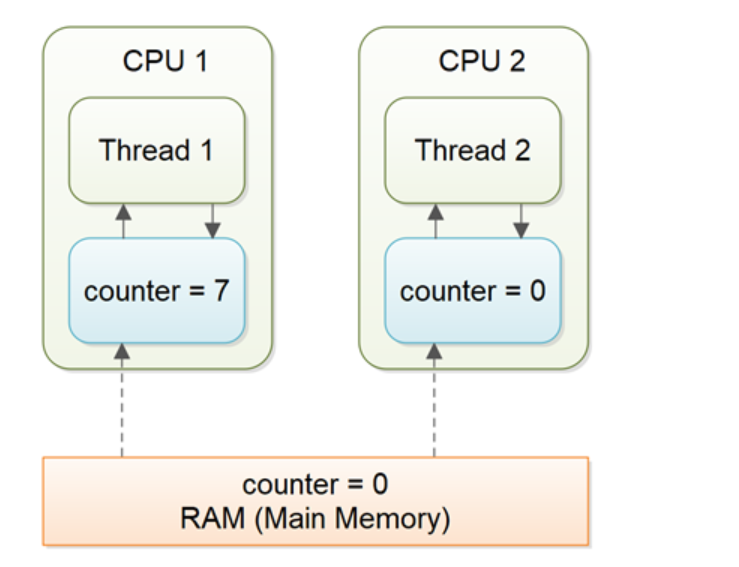
public class SharedObject {

public int counter = 0;

}

magine 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 counter variable is written from the CPU registers back to main memory. This means, that the counter variable value in the CPU register may not be the same as in main memory.



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 not visible to other threads.

The Java volatile keyword is intended to address variable visibility problems. By declaring the counter variable volatile all writes to the counter variable will be written back to main memory immediately. Also, all reads of the counter variable will be read directly from main memory.

Here 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.

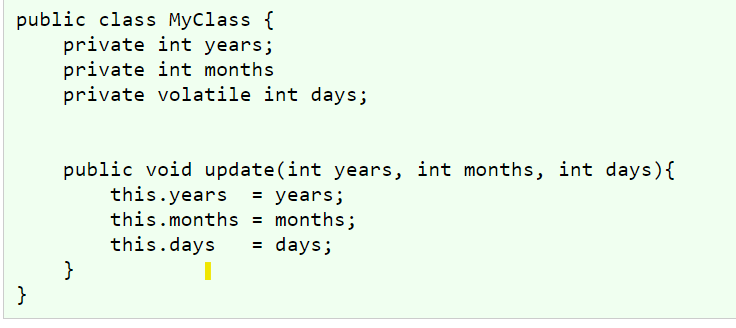
In the scenario given above, where one thread (T1) modifies the counter, and another thread (T2) reads the counter (but never modifies it), declaring the counter variable volatile is enough to guarantee visibility for T2 of writes to the counter variable.

If, however, both T1 and T2 were incrementing the counter variable, then declaring the counter variable volatile would not have been enough

**Full volatile Visibility Guarantee**

Actually, the visibility guarantee of Java volatile goes beyond the volatile variable itself. The visibility guarantee is as follows:

* If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B after it has read the volatile variable.
* If Thread A reads a volatile variable, then all all variables visible to Thread A when reading the volatile variable will also be re-read from main memory.



The udpate() method writes three variables, of which only days is volatile.

The full volatile visibility guarantee means, that when a value is written to days, then all variables visible to the thread are also written to main memory. That means, that when a value is written to days, the values of years and months are also written to main memory.

When reading the values of years, months and days you could do it like this:

A screenshot of a computer program

Description automatically generated

Notice the totalDays() method starts by reading the value of days into the total variable. When reading the value of days, the values of months and years are also read into main memory. Therefore you are guaranteed to see the latest values of days, months and years with the above read sequence.

**Instruction Reordering Challenges**

The Java VM and the CPU are allowed to reorder instructions in the program for performance reasons, as long as the semantic meaning of the instructions remain the same. For instance, look at the following instructions:

int a = 1;

int b = 2;

a++;

b++;

These instructions could be reordered to the following sequence without losing the semantic meaning of the program:

'

int a = 1;

a++;

int b = 2;

b++;

However, instruction reordering presents a challenge when one of the variables is a volatile variable. Let us look at the MyClass class from the example earlier in this Java volatile tutorial:

public class MyClass {

private int years;

private int months

private volatile int days;

public void update(int years, int months, int days){

this.years = years;

this.months = months;

this.days = days;

}

}

Once the update() method writes a value to days, the newly written values to years and months are also written to main memory. But, what if the Java VM reordered the instructions, like this:

public void update(int years, int months, int days){

this.days = days;

this.months = months;

this.years = years;

}

The values of months and years are still written to main memory when the days variable is modified, but this time it happens before the new values have been written to months and years. The new values are thus not properly made visible to other threads. The semantic meaning of the reordered instructions has changed.

Java has a solution for this problem

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.

**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**](https://jenkov.com/tutorials/java-concurrency/race-conditions-and-critical-sections.html) 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.

Imagine if Thread 1 reads a shared counter variable with the value 0 into its CPU register, 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 register. 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:

A diagram of a computer program

Description automatically generated

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 registers, 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?**

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 a [**synchronized**](https://jenkov.com/tutorials/java-concurrency/synchronized.html) 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**](https://jenkov.com/java-util-concurrent/index.html). For instance, the [**AtomicLong**](https://jenkov.com/java-util-concurrent/atomiclong.html) or [**AtomicReference**](https://jenkov.com/java-util-concurrent/atomicreference.html) 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 register. 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.

In practice, CPU register values will typically just be written to the CPU L1 cache, which is reasonably fast. Not as fast as writing to a CPU register, but still fast. Synchronization from the L1 cache down through L2 and L3 cache and back to main memory (RAM) is done by separate chips than the CPU (as far as I understand), so the CPU is not burdened with that.

Even so, try only to use volatile variables when you actually need them. That will also force you to understand in detail how Java volatile variables work!