[Volatile Vs Atomic](http://stackoverflow.com/questions/19744508/volatile-vs-atomic)

I know some of you guys will consider it as a repeated question... :) but my only point is how Atomic / Volatile / Synchronize internally works ??

What is the difference between following code blocks....

Code :1

private int counter;

public int getNextUniqueIndex() {

return counter++;

}

Code :2

private AtomicInteger counter;

public int getNextUniqueIndex() {

return counter.getAndIncrement();

}

Code :3

private Volatile int counter;

public int getNextUniqueIndex() {

return counter++;

}

does the Volatile works in following way ??

volatile int i = 0;

void incIBy5() {

i += 5;

}

is equal to

void incIBy5() {

int temp;

synchronized(i) { temp = i }

synchronized(i) { i = temp + 5 }

}

I know that two threads can not enter in Synchronize block at the same time.. am i right ?? if this is true than How this atomic.incrementAndGet() works without Synchronize ?? and is thread safe ??

And what is difference between internal reading and writing to Volatile Variable / Atomic Variable ?? i read in some article that thread has local copy of variables what is that ??

Thank you for reading my question. Hardik

**Answer**

You are specifically asking about how they internally work, so here you are:

# No synchronization

private int counter;

public int getNextUniqueIndex() {

return counter++;

}

It basically reads value from memory, increments it and puts back to memory. This works in single thread but nowadays, in the era of multi-core, multi-CPU, multi-level caches it won't work correctly. First of all it introduces race condition (several threads can read the value at the same time), but also visibility problems. The value might only be stored in "local" CPU memory (some cache) and not be visible for other CPUs/cores (and thus - threads). This is why many refer to local copy of a variable in a thread. It is very unsafe, consider very popular but broken thread-stopping code:

private boolean stopped;

public void run() {

while(!stopped) {

//do some work

}

}

public void pleaseStop() {

stopped = true;

}

Add volatile to stopped variable and it works fine - if any other thread modifies stopped variable via pleaseStop() method, you are guaranteed to see that change immediately in working thread's while(!stopped) loop. BTW this is not a good way to interrupt a thread either, see: [How to stop a thread that is running forever without any use](http://stackoverflow.com/questions/6410721) and [Stopping a specific java thread](http://stackoverflow.com/questions/7786305).

# AtomicInteger

private AtomicInteger counter = new AtomicInteger();

public int getNextUniqueIndex() {

return counter.getAndIncrement();

}

The AtomicInteger class uses CAS ([compare-and-swap](http://en.wikipedia.org/wiki/Compare-and-swap)) low-level CPU operations (no synchronization needed!) They allow you to modify particular variable only if the present value is equal to something else (and return it it succeed). So when you execute getAndIncrement() it actually runs in a loop (simplified real implementation):

int current;

do {

current = get();

} while(!compareAndSet(current, current + 1)

So basically: read, try to store incremeneted value, if not succeeded (the value is no longer equal to current) read and try again. The tryChanging() is implemented in native code (assembly).

# volatile without synchronization

private volatile int counter;

public int getNextUniqueIndex() {

return counter++;

}

This code is not correct. It fixed visibility issue (volatile makes sure other threads can see change made to counter) but still introduces race condition. This has been explained multiple times: pre/post-incrementation is not atomic.

The only side-effect of volatile is "flushing" caches so that all other parties see the most fresh version of the data. This is too strict in most situations, that's why volatile is not default.

# volatile without synchronization (2)

volatile int i = 0;

void incIBy5() {

i += 5;

}

The same problem as above, even worse, because i is not private. Still race condition is present. Why is it a problem? If, say, two threads run this code simultaneously, the output might be + 5 as well as + 10 - but at least you are guaranteed to see the change.

# Multiple independent synchronized

void incIBy5() {

int temp;

synchronized(i) { temp = i }

synchronized(i) { i = temp + 5 }

}

Surprise, this code is incorrect as well. In fact it is completely wrong. First of all you are synchronizing on i, which is about to be changed (moreover, i is a primitive, so I guess you are synchronizing on a temporary Integer created via autoboxing...) Completely flawed. You could as well write:

synchronized(new Object()) {

//thread-safe, SRSLy?

}

No two threads can enter the same synchronized block **with the same lock**. In this case (and similarily in your code) the lock object changes upon every execution, so synchronized effectively has no effect.

Even if you have used final variable (or this) for synchronization, the code is still incorrect. Two threads can first read i to temp synchronously (having the same value locally in temp) then first assigns new value to i (say, from 1 to 6) and another one does the same thing (from 1 to 6).

The synchronization must span from reading to assigning a value. Your first synchronization has no effect (reading an int is atomic) and the second as well. These are correct forms if you ask me:

void synchronized incIBy5() {

i += 5

}

void incIBy5() {

synchronized(this) {

i += 5

}

}

void incIBy5() {

synchronized(this) {

int temp = i;

i = temp + 5;

}

}

**Answer**

I know that two threads can not enter in Synchronize block at the same time

Two thread cannot enter a synchronized block on the same object twice. This means that two threads can enter the same block on different objects. This confusion can lead to code like this.

private Integer i = 0;

synchronized(i) {

i++;

}

This will not behave as expected as it could be locking on a different object each time.

if this is true than How this atomic.incrementAndGet() works without Synchronize ?? and is thread safe ??

yes. It doesn't use locking to achieve thread safety.

If you want to know how they work in more detail, you can read the code for them.

And what is difference between internal reading and writing to Volatile Variable / Atomic Variable ??

Atomic class uses volatile **fields.** There is no difference in the field. The difference is the operations performed. The Atomic classes use CompareAndSwap or CAS operations.

i read in some article that thread has local copy of variables what is that ??

I can only assume that it referring to the fact that each CPU has its own cached view of memory which can be different from every other CPU. To ensure that your CPU has a consistent view of data, you need to use thread safety techniques.

This is only an issue when memory is shared at least one thread updates it.

The effect of the volatile keyword is approximately that each individual read or write operation on that variable is atomic.

Notably, however, an operation that requires more than one read/write -- such as i++, which is equivalent to i = i + 1, which does one read and one write -- is not atomic, since another thread may write to i between the read and the write.

The Atomic classes, like AtomicInteger and AtomicReference, provide a wider variety of operations atomically, specifically including increment for AtomicInteger.

**Answer**

Volatile and Atomic are two different concepts. Volatile ensures, that a certain, expected (memory) state is true across different threads, while Atomics ensure that operation on variables are performed atomically.

Take the following example of two threads in Java:

Thread A:

value = 1;

done = true;

Thread B:

if (done)

System.out.println(value);

Starting with value = 0 and done = false the rule of threading tells us, that it is undefined whether or not Thread B will print value. **Furthermore value is undefined at that point as well!** To explain this you need to know a bit about Java memory management (which can be complex), in short: Threads may create local copies of variables, and the JVM can reorder code to optimize it, therefore there is no guarantee that the above code is run in exactly that order. Setting done to true and **then**setting value to 1 would be a possible outcome of the JIT.

volatile only ensures, that at the moment of access of such a variable, the new value will be immediately visible to all other threads **and** the order of execution ensures, that the code is at the state you would expect it to be. So in case of the code above, defining done as volatile will ensure that whenever Thread B checks the variable, it is either false, or true, and if it is true, then value has been set to 1 as well.

As a side-effect of volatile, the value of such a variable is set thread-wide atomically (at a very minor cost of execution speed). This is however only important on 32-bit systems that i.E. use long (64-bit) variables (or similar), in most other cases setting/reading a variable is atomic anyways. But there is an important difference between an atomic access and an atomic operation. Volatile only ensures that the access is atomically, while Atomics ensure that the operation is atomically.

Take the following example:

i = i + 1;

No matter how you define i, a different Thread reading the value just when the above line is executed might get i, or i + 1, because the operation is not atomically. If the other thread sets i to a different value, in worst case i could be set back to whatever it was before by thread A, because it was just in the middle of calculating i + 1 based on the old value, and then set i again to that old value + 1. Explanation:

Assume i = 0

Thread A reads i, calculates i+1, which is 1

Thread B sets i to 1000 and returns

Thread A now sets i to the result of the operation, which is i = 1

Atomics like AtomicInteger ensure, that such operations happen atomically. So the above issue cannot happen, i would either be 1000 or 1001 once both threads are finished.

**Answer**

There are two important concepts in multithreading environment.

1. atomicity
2. visibility

Volatile eradicates visibility problem but it does not deal with atomicity. Volatile will prevent compiler to reorder the instruction which involves write and subsequent read of a volatile variable. e.g.k++ Here k++ is not a single machine instruction rather it is three machine instructions.

1. copy the value to register
2. increment it
3. place it back

So even though you declare variable to volatile it will not make this operation atomic that means another thread can see a intermediate result which is a stale or unwanted value for the other thread.

But AtomicInteger, AtomicReference are based on the **Compare and swap instruction**. CAS has three operands a memory location V on which to operate, the expected old value A, and the new value B. CAS atomically updates V to the new value B, but only if the value in V matches the expected old value A; otherwise it does nothing. In either case, it returns the value currently in V. This is used by JVM in AtomicInteger, AtomicReference and they call the function as compareAndSet() if this functionality is not supported by underlying processor then JVM implements it by **spin lock**.

**Answer**

As Trying as indicated, volatile deals only with visibility.

Consider this snippet in a concurrent environment:

boolean isStopped = false;

:

:

while (!isStopped) {

// do some kind of work

}

The idea here is that some thread could change the value of isStopped from false to true in order to indicate to the subsequent loop that it is time to stop looping.

Intuitively, there is no problem. Logically if another thread makes isStopped equal to true, then the loop must terminate. The reality is that the loop will likely never terminate even if another thread makesisStopped equal to true.

The reason for this is not intuitive, but consider that modern processors have multiple cores and that each core has multiple registers and multiple levels of cache memory that **are not accessible to other processors**. In other words, values that are cached in one processor's local memory are **not visisble** to threads executing on a different processor. Herein lies one of the central problems with concurrency: visibility.

The Java Memory Model makes no guarantees whatsoever about when changes that are made to a variable in on thread may become visible to other threads. In order to guarantee that updates are visisble as soon as they are made, you must synchronize.

The volatile keyword is a weak form of synchronization. While it does nothing for mutual exclusion or atomicity, it does provide a guarantee that changes made to a variable in one thread will become visible to other threads as soon as it is made. Because individual reads and writes to variables that are not 8-bytes are atomic in Java, declaring variables volatile provides an easy mechanism for providing visibility in situations where there are no other atomicity or mutual exclusion requirements.

<http://blog.vinceliu.com/2010/05/difference-between-atomic-and-volatile.html>

Saturday, May 15, 2010

### **[Difference between Atomic and Volatile?](http://blog.vinceliu.com/2010/05/difference-between-atomic-and-volatile.html)**

There seems to be a tendency for people to confuse an atomic action with a volatile memory access. This is quite natural, since they exhibit similar properties and makes it hard for people to distinguish between the two.  
  
So what is the common property between an atomic action and a volatile memory? They both exhibit the property of total ordering. Let's assume for \forall a, b, c \in A where A is the set of possible actions a program can execute, and a \rightarrow b denotes that an action a precedes action b, then the property of total order means:

a \rightarrow b  \wedge b \rightarrow a \Rightarrow a = b  
  
a \rightarrow b , b  \rightarrow c \Rightarrow a \rightarrow c  
  
a \rightarrow b \vee b  \rightarrow a

Literally, the set of rules mean (1) that for any action a that precedes b and any action b that precedes a, then the action a must necessarily be the same action as b. (2) If a precedes b, and b precedes c, then a must precede c. (3) a must precede b or b must precede a. These are just some mathematical (or convoluted) formalisms of saying any action is guaranteed to be in sequential order.  
  
But that's where the similarity ends.  
  
An atomic action is literally what it means: an action that is indivisible. Volatile memory only provides a guarantee of sequential order, but offers no guarantee that an interleaving of actions cannot occur. For an illustration, lets take a look at the following code:  
  
a = 0;  
a++;  
  
Assume that 'a' is a volatile variable. From the code, we naturally think that the increment operator (++) will atomically increment the value of 'a' to 1, but this assumption is slightly faulty:  
  
[a] = 0;  
r0 = [a];  
r0 = r0 + 1;  
[a] = r0;  
  
The square brackets between [a] denotes a value gotten from memory, while r0 represents a register in which the addition operation actually occurs. An increment operation is actually a *composite* action, composed of a *read*, then a *write* action. (By the way, it is implied that register actions are atomic, despite being a *composite*action. This is because an operation within a register cannot be directly observed).  
  
So increment is a *composite* action, compared to being an *atomic* action, what's the harm? Lets go through the same example, but with 2 concurrent processors executing simultaneously:

|  |  |
| --- | --- |
| **CPU1** | **CPU2** |
| a++; | a++; |

Lets assume that a is initially 0 to start. Given that iff the actions from CPU1 and CPU2 are both *volatile* and *atomic*, then the only possible transition value that can be seen after execution can only be 2.  
  
However, if the actions from CPU1 and CPU2 are only *volatile*, then the actions can translate to this:

|  |  |
| --- | --- |
| **CPU1** | **CPU2** |
| r0 = a; | r1 = a; |
| r0 = r0 + 1; | r1= r1 + 1 |
| a = r0; | a = r1 |

one can see that it is possible that both registers r0 and r1 both read a as 0 simultaneously which causes the resultant possible values to be either 1 or 2. Clearly seeing 1 is *wrong*, since if the operation is atomic, then 1 cannot be a legal value at the end of the execution.  
  
Conversely, a volatile variable *can* be deemed atomic. Consider the example below:

|  |  |
| --- | --- |
| **CPU1** | **CPU2** |
| a=1; | a=2; |

In this case, the only possible set of terminal values are 1 or 2, which is exactly the same as you would get for an atomic action. Therefore, a volatile memory access is atomic iff you perform only a *read* or *write* action, which are atomic in nature. But a volatile variable is no guarantee of atomicity if a composite action is executed.  
  
Hence the label atomic actions is a slight misnomer, because it doesn't imply that the action is atomic, but rather it means that it guarantees a *composite action is executed as a single atomic action*. A well known example of such a composite action is the[Compare-and-Swap](http://en.wikipedia.org/wiki/Compare-and-swap) instruction.  
  
For simplicity, just remember that atomicity is not a property of memory, but a property its actions, to which a volatile memory access may or may not uphold.

<http://jeremymanson.blogspot.in/2007/08/volatile-does-not-mean-atomic.html>

### **Volatile Does Not Mean Atomic!**

Here's a question I get a lot. Is the following code snippet "thread safe"?

volatile int v = 0;  
  
Thread 1:  
v++;  
  
Thread 2:  
v--;

The question asks what the possible results of this code are; the questioners usually want the answer "v can only be 0 after this code is run".  
  
This isn't the way it works! If you do an increment of a volatile integer, you are actually performing three separate operations:

Read the integer to a local.

Increment the local.

Write the integer back out to the volatile field.

So what you really have is this:

volatile int v = 0;  
  
Thread 1:  
r1 = v;  
r2 = r1 + 1;  
v = r2;  
  
Thread 2:  
r3 = v;  
r4 = r3 - 1;  
v = r4;

So, if Threads 1 and 2 both read v and see the value 0, then Thread 1 will write 1 to it and Thread 2 will write -1 to it. You are not guaranteed to see the value 0!  
  
If you want an atomic increment (or decrement), you have to use [the java.util.concurrent.atomic classes](http://java.sun.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html), which allow you to create object that represent numbers that can be incremented or decremented atomically. The VM is smart enough to replace the objects with plain ol' numbers (a process that I claim is called*intrinsification*), which it then uses atomic machine instructions to manipulate.  
  
So beware!  
  
*ETA: There was a bit of confusion about atomicity after I posted this. For more on atomicity, visibility and ordering, check out*[*this post*](http://jeremymanson.blogspot.com/2007/08/atomicity-visibility-and-ordering.html)*.*

### **Atomicity, Visibility and Ordering**

(Note: I've cribbed this from my doctoral dissertation. I tried to edit it heavily to ease up on the mangled academic syntax required by thesis committees, but I may have missed some / badly edited in places. Let me know if there is something confusingly written or just plain confusing, and I'll try to untangle it.)  
  
There are these three concepts, you see. And they are fundamental to correct concurrent programming. When a concurrent program is not correctly written, the errors tend to fall into one of the three categories: *atomicity*, *visibility*, or *ordering*.   
  
*Atomicity* deals with which actions and sets of actions have indivisible effects. This is the aspect of concurrency most familiar to programmers: it is usually thought of in terms of mutual exclusion. *Visibility* determines when the effects of one thread can be seen by another. *Ordering* determines when actions in one thread can be seen to occur out of order with respect to another. Let's talk about them.

## **ATOMICITY** Everyone doing any serious concurrent programming knows what atomicity is (or will when I describe it) — I'm just putting it in for completeness's sake. If an action is (or a set of actions are) *atomic*, its result must be seen to happen ``all at once'', or indivisibly. Atomicity is the traditional bugbear of concurrent programming. Enforcing it usually means using locking to enforce mutual exclusion. To see atomicity in action (or in inaction, perhaps), consider this code:

class BrokenBankAccount {  
 private int balance;  
  
 synchronized int getBalance() {  
 return balance;  
 }  
  
 synchronized void setBalance(int x)   
 throws IllegalStateException {  
 balance = x;  
 if (balance < 0) {  
 throw new IllegalStateException("Negative Balance");  
 }  
 }  
  
 void deposit(int x) {  
 int b = getBalance();  
 setBalance(b + x);  
 }  
  
 void withdraw(int x) {  
 int b = getBalance();  
 setBalance(b - x);  
 }  
}

Since all accesses to the shared variable balance are guarded by locks, this code is free of what are called *data races*, which are basically what happens when you access a variable concurrently without some use of synchronization or volatile or the like (one of those accesses has to be a write to be a data race in the true sense). When code is free from data races, we say it is *correctly synchronized*. So the code is correct, right?  
  
No, of course not. This code is not at all correct, in the sense that it doesn't necessarily do what we want it to do. Think about what happens if one thread callsdeposit(5) and another calls withdraw(5); there is an initial balance of 10. Ideally, at the end of these two calls, there would still be a balance of 10. However, consider what would happen if:

1. The deposit() method sees a value of 10 for the balance, then
2. The withdraw() method sees a value of 10 for the balance  
   **and** withdraws 5, leaving a balance of 5, and finally
3. The deposit() method uses the balance it originally saw (10) to  
   calculate a new balance of 15.

As a result of this lack of "atomicity", the balance is 15 instead of 10. This effect is often referred to as a *lost update*, because the withdrawal is lost. A programmer writing multi-threaded code must use synchronization carefully to avoid this sort of error. In Java, if the deposit() and withdraw() methods are declared synchronized, it will ensure that locks are held for their duration: the actions of those methods will be seen to take place atomically.  
  
Atomicity, of course, is only guaranteed when all the threads use synchronization correctly. If someone comes along and decides to read the balance without acquiring a lock, it can end up with all sorts of confusing results.  
  
Atomicity is the most common problem you get when using synchronization. It is a common mistake to think that it is the only problem; it is not. Here's another one:

## **VISIBILITY** What's visibility, you ask? Well, if an action in one thread is *visible* to another thread, then the result of that action can be observed by the second thread. In order to guarantee that the results of one action are observable to a second action, then you have to use some form of synchronization to make sure that the second thread sees what the first thread did.  (Note: when I say synchronization in this post, I don't actually mean locking. I mean anything that guarantees visibility or ordering in Java. This can include final and volatile fields, as well as class initialization and thread starts and joins and all sorts of other good stuff.) Here's an example of a code with visibility problems:

class LoopMayNeverEnd {   
 boolean done = false;   
  
 void work() {   
 while (!done) {   
 // do work   
 }   
 }   
   
 void stopWork() {   
 done = true;   
 }   
}

In this code, imagine that two threads are created; one thread calls work, and at some point, the other thread calls stopWork on the same object. Because there is no synchronization between the two, the thread in the loop may never see the update to done performed by the other thread. In practice, this may happen if the compiler detects that no writes are performed to done in the first thread; the compiler may decide that the program only has to read done once, transforming it into an infinite loop.  
  
(By the way, "compiler" in this context doesn't mean javac — it actually means the JVM itself, which plays lots of games with your code to get it to run more quickly. In this case, if the compiler decides that you are reading a variable that you don't have to read, it can eliminate the read quite nicely. As described above.)  
  
To ensure that this does not happen, you have to use a mechanism that provides synchronization between the two threads. In LoopMayNeverEnd, if you want to do this, you can declare done to be volatile. Conceptually, all actions on volatiles happen in a single order, where each read sees the last write in that order. In other words, the compiler can't prevent a read of a volatile from seeing a write performed by another thread.  
  
There is a side issue here; some architectures and virtual machines may execute this program without providing a guarantee that the thread that executes work will ever give up the CPU and allow other threads to execute. This would prevent the loop from ever terminating because of scheduling guarantees, not because of a lack of visibility guarantees. This is typically called *cooperative multithreading*. The only implementation I know that does this is the Oracle VM — check the box for details.  
  
There's one more problem that crops up:

## **ORDERING** *Ordering* constraints describe what order things are seen to occur. You only get intuitive ordering constraints by synchronizing correctly. Here's an example of when ordering problems can bite you:

class BadlyOrdered {  
 boolean a = false;  
 boolean b = false;  
  
 void threadOne() {  
 a = true;  
 b = true;  
 }  
  
 boolean threadTwo() {  
 boolean r1 = b; // sees true  
 boolean r2 = a; // sees false  
 boolean r3 = a; // sees true  
 return (r1 && !r2) && r3; // returns true  
 }  
}

Consider what happens if threadOne() gets invoked in one thread and threadTwo() gets invoked on the same object in another. Would it be possible for threadTwo() to return the value true? If threadTwo() returns true, it means that the thread saw both updates by threadOne, but that it saw the change to b before the change to a.  
  
Well, this code fragment does not use synchronization correctly, so surprising things can happen! It turns out that Java allows this result, contrary to what a programmer might have expected.   
  
The assignments to a and b in threadOne() can be seen to be performed out of order. Compilers have a lot of freedom to reorder code in the absence of synchronization; they could either reorder the writes in threadOne or the reads in threadTwo freely.   
  
How do you fix it? **Synchronize your code carefully!** In this case, you can throw a lock around threadOne or threadTwo, or you can declare them both to be volatile, and get the ordering you want.