**Atomic Vs Volatile – Precise - 2022**

private int counter;

public int getNextUniqueIndex() {

return counter++;

}

A single thread reads value from memory, increments it and puts back to memory. First of all it introduces race condition (several threads can read the value at the same time), but also visibility problems. 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 visible** to threads executing on a different processor. Herein lies one of the central problems with concurrency: visibility.

# 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 incremented value, if not succeeded (the value is no longer equal to current) read and try again.

# 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. 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.

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

**Atomic\* actually gives both atomicity and volatility.**