**Atomic Vs Volatile – Precise - 2022**

There are two important concepts in multithreading environment.

1. **Atomicity**: **An atomic operation will either be completed or not done at all**. Other threads will not be able to see partially complete state. The Greek word "atom" (ἄτομος; atomos) means *uncuttable*. **A task performed by a computer is said to be atomic when it is not divisible anymore**: it can't be broken into smaller steps. Conversely, when a thread atomically reads from shared data, it sees the value as it appeared at a single moment in time.
2. **Visibility**: **Values that are cached in one processor's local memory are not visible to threads executing on a different processor is the visibility issue**

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.**

# AtomicInteger

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.

**How Compare and Swap works Internally**

The implementation of CAS varies depending on the architecture and the programming language being used, but the basic steps are

1. Load the current value of the memory location into a register.
2. Compare the value in the register to the expected value.
3. If the values are equal, write the new value to the memory location and return a success flag. If the values are not equal, return a failure flag.