# Atomic Classes

Understand the performance benefits of atomic classes over locks.

## Introduction

As part of your senior engineer interview, you may be quizzed on the atomic classes and their utility. Though this is an advanced topic, understanding of it broadens your depth about, and command over concurrent systems and their workings.

So far we have seen locks that allow shared data to be manipulated safely among multiple threads. However, locking doesn't come for free and takes a toll on performance especially when the shared data/state is being contented for access by multiple threads.

## Cons of Locking

Locking comes with its downsides some of which include:

### Thread scheduling vs useful work

JVM is very efficient when it comes to acquiring and releasing a lock that is requested by a single thread. However, when multiple threads attempt to acquire the same lock, only one wins and the rest must be suspended. The suspension and resumption of threads is costly and introduces significant overhead and this can be an issue for scenarios where several threads contend for the same lock but execute very little functionality. In such cases, the time spent in scheduling overhead overwhelms the useful work done. This is true of synchronized collections where the majority of methods perform very few operations.

### Priority inversion

A higher priority thread can be blocked by a lower priority thread that holds the lock and itself is blocked because of a page fault, scheduling delay etc. This situation effectively downgrades the priority of the higher-priority thread to that of the lower-priority thread since the former can't make progress until the latter releases the lock. In general, all threads that require a particular lock can't make progress until the thread holding the lock releases it.

### Liveness issues

Locking also introduces the possibility of liveness issues such as deadlocks, livelock or simply programming bugs that have threads caught in infinite loops blocking other threads from making progress.

### Locking, a heavyweight mechanism

In general locking is a heavyweight mechanism, especially for fine-grained tasks such as manipulating a counter. Locking is akin to assuming the worst or preparing for the worst possible scenario, i.e. the thread assumes it would necessarily run into contention with another thread and acquires a lock to manipulate shared state. Another approach could be to update shared state hoping it would complete without contention/interference from other participants. In case contention is detected, the update operation can be failed, and if desired, reattempted later. We'll see how this approach is supported by hardware later.

## Atomic vs `volatile`

Short of locking, we have volatile variables that promise the same visibility guarantees as a lock, however, volatile variables can't be used for:

- Executing compound actions, .e.g. Decrementing a counter involves fetching the counter value, decrementing it and then writing the updated value for a total of three steps.
- When the value of a variable depends on another or the new value of a variable depends on its older value.

The above limitations are addressed by atomic classes, which offer similar memory visibility guarantees as volatile variables and also allow operations such as read-modify-write to be executed atomically. As an aside, consider the following snippet:

```
volatile AtomicInteger atomicInteger = new AtomicInteger();
```

Note that the marking the `AtomicInteger` variable above `volatile` isn't superfluous and implies that when the variable `atomicInteger` is updated to a new reference, the updated value `atomicInteger` holds will be observed by all threads that read the variable . In the absence of `volatile` the `atomicInteger` variable's value (which points to a memory location) may get cached by a processor and the new object the variable points to after the update, may not be visible to the processor that cached the old value.

Atomic variables can also be thought of as “better volatiles”. Atomic variables make ideal counters, sequence generators, and variables that accumulate running statistics. They offer the same memory semantics as volatile variables with additional support for atomic updates and may be better choices than volatile variables in most circumstances.

## Atomic Processor Instructions

Modern processors have instructions that can atomically execute compound operations offering a compromise between locking and volatile variables. Hardware support for concurrency is ubiquitous in present-day processors and the most well-known of these instructions is the *Compare and Swap* instruction or CAS for short. The CAS instruction is the secret sauce behind atomically executing compound operations.

### Compare and Swap

In general, the CAS instruction has three operands:

1. A memory location, say M,representing the variable we desire to manipulate.
2. An expected value for the variable, say A. This is the latest value seen for the variable.
3. The new value, say B, which we want the variable to update to.

CAS instruction works by performing the following actions atomically:

1. Check the latest value of the memory location M.
2. If the memory location has a value other than A, then it implies that another thread changed the variable since the last time we examined it and therefore the requested update operation should be aborted.
3. If the variable’s value is indeed A, then it implies that no other thread has had a chance to change the variable to a different value than A, since we last examined the variable’s value and therefore we can proceed to update the variable/memory location to the new value B.

CAS takes the *optimistic approach* when performing an update on a shared variable. Expressed in prose, the instruction says *I have seen and expect the variable’s value to be A, if that is still true, update the variable to the new value B, otherwise fail my request and let me know. When multiple threads invoke CAS to manipulate a shared variable, only a single thread succeeds and the rest fail.* However, the crucial difference between CAS and locking is that with CAS the threads that fail executing the CAS command have a choice to retry or go do some other useful work, unlike locking where all the threads unsuccessful in acquiring the lock will block. This may sound trivial but can improve throughput and performance many fold.

The idiomatic usage of CAS usually takes the form of reading the value *A* of a shared variable, deriving a new value *B* from the value *A*, and finally invoking CAS to update the variable from *A* to *B* if it hasn’t been changed to another value in the meantime.

### ABA Problem

The astute reader would recognize that CAS succeeds even if the value of a shared variable is changed from A to B and then back to A. Consider the following sequence:

1. A thread T1 reads the value of a shared variable as A and desires to change it to B. After reading the variable’s value, thread
2. T1 undergoes a context switch. Another thread, T2 comes along, changes the value of the shared variable from A to B and then back to A from B.
3. Thread T1 is scheduled again for execution and invokes CAS with A as the expected value and B as the new value. CAS succeeds since the current value of the variable is A, even though it changed to B and then back to A in the time thread T1 was context switched.

For some algorithms the *ABA* problem may not be an issue but for others changing the value from A to B and then back to A may require re-executing some step(s) of an algorithm. This problem usually occurs when a program manages its own memory rather than leaving it to the Garbage Collector. For example, you may want to recycle the nodes in your linked list for performance reasons. Thus noting that the head of the list still references the same node, may not necessarily imply that the list wasn't changed. One solution to this problem is to attach a version number with the value, i.e. instead of storing the value as A, we store it as a pair (A, V1). Another thread can change the value to (B, V1) but when it changes it back to A the associated version is different i.e. (A, V2). In this way, a collision can be detected. There are two classes in Java that can be used to address the ABA problem:

1. `AtomicStampedReference` (please see this class in our reference section for detailed explanation of ABA problem)
2. `AtomicMarkableReference`

Spurious Wakeups

More on Atomics