



76% completed  Search Course

Multithreading in Java ^

- Atomic Assignments
- Thread Safety & Synchronized
- Wait & Notify
- Interrupting Threads
- Volatile
- Reentrant Locks & Condition Variables
- Missed Signals
- Semaphore in Java
- Spurious Wakeups
- Atomic Classes
- More on Atomics
- Non-blocking Synchronization
- Miscellaneous Topics

Java's Memory Model v

Interview Practice Problems v

Practice Mock Interview →

Interview Practice Problems v

Practice Mock Interview →

Java Multithreading for Senior Engineering Interviews / ... / More on Atomics

More on Atomics

Learn about the different categories of atomic classes and their performance in comparison to locks.

Taxonomy of atomic classes

There are a total of sixteen atomic classes divided into four groups:

- Scalars
- Field updaters
- Arrays
- Compound variables

Most well-known and commonly used are the scalar ones such as `AtomicInteger`, `AtomicLong`, `AtomicReference`, which support the CAS (compare-and-set). Other primitive types such as double and float can be simulated by casting `short` or `byte` values to and from `int` and using methods `floatToIntBits()` and `doubleToLongBits()` for floating point numbers. Atomic scalar classes extend from `Number` and don't redefine `hashCode()` or `equals()`.

Atomics are not primitivies

The following widget highlights these differences between `Integer` and `AtomicInteger`. Note, that the `Integer` class has the same hashCode for the same integer value but that's not the case for `AtomicInteger`. Thus `Atomic*` scalar classes are unsuitable as keys for collections

The following widget highlights these differences between `Integer` and `AtomicInteger`. Note, that the `Integer` class has the same hashCode for the same integer value but that's not the case for `AtomicInteger`. Thus `Atomic*` scalar classes are unsuitable as keys for collections that rely on hashCode.

```
Java
Press ⌘ + ⇧ to interact
main.java
1- import java.util.concurrent.atomic.AtomicInteger;
2
3- class Demonstration {
4-     public static void main( String args[] ) {
5-         AtomicInteger atomicFive = new AtomicInteger(5);
6-         AtomicInteger atomicAlsoFive = new AtomicInteger(5);
7
8-         System.out.println("atomicFive.equals(atomicAlsoFive) : " + atomicFive.equals(atomicAlsoFive));
9-         System.out.println("atomicFive.hashCode() == atomicAlsoFive.hashCode() : " + (atomicFive.hashCode() == atomicAlsoFive.hashCode()));
10
11
12-         Integer integer1 = new Integer(23235);
```

Atomics provides the users with an option to back off when faced with contention. For instance the `AtomicInteger` class has a `compareAndSet()` method that returns false if the operation doesn't succeed. The invoking thread now has the opportunity to either retry the operation immediately or use a custom retry strategy. In essence, responding to contention or *contention management* is pushed to the invoking thread and the JVM doesn't make a decision for the caller, as it does in case of locking by suspending the thread.

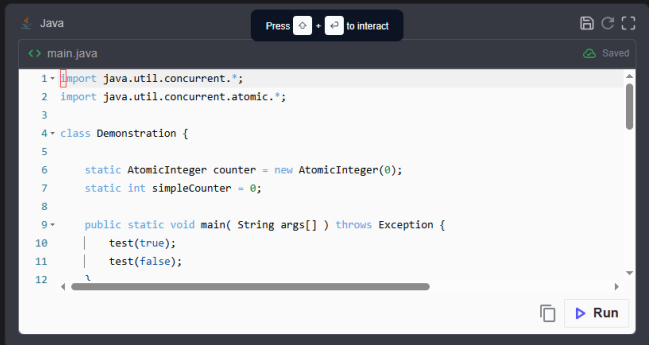
Performance of atomics vs locks

In the case of a single thread, i.e. zero contention environment, an operation that relies on CAS (e.g. updating an `AtomicInteger`) will be faster than an operation that involves locking first. On single CPU machines, CAS operations almost always succeed other than in the very rare case of a thread being interrupted in the middle of a read-modify-write operation. Even with moderate contention, CAS operations are faster as they avoid thread suspension and resumption, which locking solutions must deal with.

In benchmark tests, it has been observed that atomics perform better than locks under low to moderate contention, which is representative of real-life programs. However, in a highly contended environment, the majority of threads will waste CPU cycles retrying CAS operations but using locks in the same situation would have the threads suspended and then resumed later. Granted, thread suspension/resumption incurs overhead but at some point the CAS-retry expense dwarfs the overhead of suspending and resuming threads. In reality such high contention is unlikely and atomics are always preferable over lock-based solutions.

We can conduct a crude test to measure the performance of an `AtomicInteger` counter versus an ordinary counter. The test involves creating a thread pool of ten threads and then

having each thread increment the same counter a million times so that the counter value reaches ten million at the end of the test. We time the two scenarios in the widget below:



```
1 import java.util.concurrent.*;
2 import java.util.concurrent.atomic.*;
3
4 class Demonstration {
5
6     static AtomicInteger counter = new AtomicInteger(0);
7     static int simpleCounter = 0;
8
9     public static void main( String args[] ) throws Exception {
10         test(true);
11         test(false);
12     }
```

Keeping state thread local

Finally, the best choice for scalability and performance is to share as little state as possible among threads. Keeping variables and state, thread local results in maximum performance and elimination of contention. Even though atomics achieve better scalability than using locks, choosing not to share any state among threads will result in the best scalability.