## <http://www.java2s.com/Tutorials/Java/Java_Thread/0200__Java_Atomic_Variables.htm>

## New Thread Concurrency Packages

java.util.concurrent, and java.util.concurrent.atomic and java.util.concurrent.locks include very useful concurrency constructs.

The Thread Concurrency Packages support the concurrency in four ways.

* Atomic variables
* Locks
* Synchronizers
* Concurrent collections

## Atomic Variables

Atomic variable classes are named like AtomicXxx, for example, AtomicInteger class is used to represent an int variable.

Atomic variable can be used to execute multiple instructions on a single variable atomically without using any lock.

* Scalar Atomic Variable Classes

The AtomicInteger, AtomicLong, and AtomicBoolean classes support operations on primitive data types int, long, and boolean, respectively.

The AtomicReference class is used to work with a reference data type when a reference variable needs to be updated atomically.

* Atomic Arrays Classes

There are three classes called AtomicIntegerArray, AtomicLongArray, and AtomicReferenceArray that represent an array of int, long, and reference types whose elements can be updated atomically.

* Atomic Field Updater Classes

There are three classes called AtomicLongFieldUpdater, AtomicIntegerFieldUpdater, and AtomicReferenceFieldUpdater that can be used to update a volatile field of a class atomically using reflection.

To get a reference to an object of these classes, you need to use their factory method called newUpdater().

* Atomic Compound Variable Classes

Atomic Variables

Operations on atomic variables get turned into the hardware primitives that the platform provides for concurrent access, such as compare-and-set.

=================

<http://www.baeldung.com/java-atomic-variables>

**3. Atomic Operations**

There is a branch of research focused on creating non-blocking algorithms for concurrent environments. These algorithms exploit low-level atomic machine instructions such as compare-and-swap (CAS), to ensure data integrity.

A typical CAS operation works on three operands:

1. The memory location on which to operate (M)
2. The existing expected value (A) of the variable
3. The new value (B) which needs to be set

## ****4. Atomic Variables in Java****

The most commonly used atomic variable classes in Java are [AtomicInteger](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html), [AtomicLong](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html), [AtomicBoolean](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html), and  [AtomicReference](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicReference.html). These classes represent an int, long, boolean and object reference respectively which can be atomically updated. The main methods exposed by these classes are:

* get() – gets the value from the memory, so that changes made by other threads are visible; equivalent to reading a volatile variable
* set() – writes the value to memory, so that the change is visible to other threads; equivalent to writing a volatile variable
* lazySet() – eventually writes the value to memory, may be reordered with subsequent relevant memory operations. One use case is nullifying references, for the sake of garbage collection, which is never going to be accessed again. In this case, better performance is achieved by delaying the null volatile write
* compareAndSet() – same as described in section 3, returns true when it succeeds, else false
* weakCompareAndSet() – same as described in section  3, but weaker in the sense, that it does not create happens-before orderings. This means that it may not necessarily see updates made to other variables

A thread safe counter implemented with AtomicInteger is shown in the example below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class SafeCounterWithoutLock {      private final AtomicInteger counter = new AtomicInteger(0);        public int getValue() {          return counter.get();      }      public void increment() {          while(true) {              int existingValue = getValue();              int newValue = existingValue + 1;              if(counter.compareAndSet(existingValue, newValue)) {                  return;              }          }      }  } |

As you can see, we retry the compareAndSet operation and again on failure, since we want to guarantee that the call to the increment method always increases the value by 1.

## ****5. Conclusion****

In this quick tutorial, we described an alternate way of handling concurrency where disadvantages associated with locking can be avoided. We also looked at the main methods exposed by the atomic variable classes in Java.

====================

https://stackoverflow.com/questions/9749746/what-is-the-difference-between-atomic-volatile-synchronized

# 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 a particular variable only if the present value is equal to something else (and is returned successfully). 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 successful (the value is no longer equal to current), read and try again. The compareAndSet() is implemented in native code (assembly).

=========================================================================

# http://www.java2s.com/Tutorials/Java/Java\_Thread/0220\_\_Java\_Synchronizers.htm

# Java Thread Tutorial - Java Synchronizers

A synchronizer object is used with a set of threads.

It maintains a state, and depending on its state, it lets a thread pass through or forces it to wait.

This section will discuss four types of synchronizers:

* Semaphores
* Barriers
* Latches
* Exchangers