| | [**Overview**](http://docs.google.com/overview-summary.html) | **Package** | Class | [**Use**](http://docs.google.com/package-use.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV PACKAGE**](http://docs.google.com/java/util/concurrent/package-summary.html)   [**NEXT PACKAGE**](http://docs.google.com/java/util/concurrent/locks/package-summary.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/atomic/package-summary.html)    [**NO FRAMES**](http://docs.google.com/package-summary.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |

## Package java.util.concurrent.atomic

A small toolkit of classes that support lock-free thread-safe programming on single variables.

**See:**

[**Description**](#3znysh7)

| **Class Summary** | |
| --- | --- |
| [**AtomicBoolean**](http://docs.google.com/java/util/concurrent/atomic/AtomicBoolean.html) | A boolean value that may be updated atomically. |
| [**AtomicInteger**](http://docs.google.com/java/util/concurrent/atomic/AtomicInteger.html) | An int value that may be updated atomically. |
| [**AtomicIntegerArray**](http://docs.google.com/java/util/concurrent/atomic/AtomicIntegerArray.html) | An int array in which elements may be updated atomically. |
| [**AtomicIntegerFieldUpdater<T>**](http://docs.google.com/java/util/concurrent/atomic/AtomicIntegerFieldUpdater.html) | A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes. |
| [**AtomicLong**](http://docs.google.com/java/util/concurrent/atomic/AtomicLong.html) | A long value that may be updated atomically. |
| [**AtomicLongArray**](http://docs.google.com/java/util/concurrent/atomic/AtomicLongArray.html) | A long array in which elements may be updated atomically. |
| [**AtomicLongFieldUpdater<T>**](http://docs.google.com/java/util/concurrent/atomic/AtomicLongFieldUpdater.html) | A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes. |
| [**AtomicMarkableReference<V>**](http://docs.google.com/java/util/concurrent/atomic/AtomicMarkableReference.html) | An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically. |
| [**AtomicReference<V>**](http://docs.google.com/java/util/concurrent/atomic/AtomicReference.html) | An object reference that may be updated atomically. |
| [**AtomicReferenceArray<E>**](http://docs.google.com/java/util/concurrent/atomic/AtomicReferenceArray.html) | An array of object references in which elements may be updated atomically. |
| [**AtomicReferenceFieldUpdater<T,V>**](http://docs.google.com/java/util/concurrent/atomic/AtomicReferenceFieldUpdater.html) | A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes. |
| [**AtomicStampedReference<V>**](http://docs.google.com/java/util/concurrent/atomic/AtomicStampedReference.html) | An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically. |

## Package java.util.concurrent.atomic Description

A small toolkit of classes that support lock-free thread-safe programming on single variables. In essence, the classes in this package extend the notion of volatile values, fields, and array elements to those that also provide an atomic conditional update operation of the form:

boolean compareAndSet(expectedValue, updateValue);

This method (which varies in argument types across different classes) atomically sets a variable to the updateValue if it currently holds the expectedValue, reporting true on success. The classes in this package also contain methods to get and unconditionally set values, as well as a weaker conditional atomic update operation weakCompareAndSet described below.

The specifications of these methods enable implementations to employ efficient machine-level atomic instructions that are available on contemporary processors. However on some platforms, support may entail some form of internal locking. Thus the methods are not strictly guaranteed to be non-blocking -- a thread may block transiently before performing the operation.

Instances of classes [AtomicBoolean](http://docs.google.com/java/util/concurrent/atomic/AtomicBoolean.html), [AtomicInteger](http://docs.google.com/java/util/concurrent/atomic/AtomicInteger.html), [AtomicLong](http://docs.google.com/java/util/concurrent/atomic/AtomicLong.html), and [AtomicReference](http://docs.google.com/java/util/concurrent/atomic/AtomicReference.html) each provide access and updates to a single variable of the corresponding type. Each class also provides appropriate utility methods for that type. For example, classes AtomicLong and AtomicInteger provide atomic increment methods. One application is to generate sequence numbers, as in:

class Sequencer {  
 private final AtomicLong sequenceNumber  
 = new AtomicLong(0);  
 public long next() {  
 return sequenceNumber.getAndIncrement();  
 }  
}

The memory effects for accesses and updates of atomics generally follow the rules for volatiles, as stated in  [The Java Language Specification, Third Edition (17.4 Memory Model)](http://java.sun.com/docs/books/jls/):

* get has the memory effects of reading a volatile variable.
* set has the memory effects of writing (assigning) a volatile variable.
* lazySet has the memory effects of writing (assigning) a volatile variable except that it permits reorderings with subsequent (but not previous) memory actions that do not themselves impose reordering constraints with ordinary non-volatile writes. Among other usage contexts, lazySet may apply when nulling out, for the sake of garbage collection, a reference that is never accessed again.
* weakCompareAndSet atomically reads and conditionally writes a variable but does *not* create any happens-before orderings, so provides no guarantees with respect to previous or subsequent reads and writes of any variables other than the target of the weakCompareAndSet.
* compareAndSet and all other read-and-update operations such as getAndIncrement have the memory effects of both reading and writing volatile variables.

In addition to classes representing single values, this package contains *Updater* classes that can be used to obtain compareAndSet operations on any selected volatile field of any selected class. [AtomicReferenceFieldUpdater](http://docs.google.com/java/util/concurrent/atomic/AtomicReferenceFieldUpdater.html), [AtomicIntegerFieldUpdater](http://docs.google.com/java/util/concurrent/atomic/AtomicIntegerFieldUpdater.html), and [AtomicLongFieldUpdater](http://docs.google.com/java/util/concurrent/atomic/AtomicLongFieldUpdater.html) are reflection-based utilities that provide access to the associated field types. These are mainly of use in atomic data structures in which several volatile fields of the same node (for example, the links of a tree node) are independently subject to atomic updates. These classes enable greater flexibility in how and when to use atomic updates, at the expense of more awkward reflection-based setup, less convenient usage, and weaker guarantees.

The [AtomicIntegerArray](http://docs.google.com/java/util/concurrent/atomic/AtomicIntegerArray.html), [AtomicLongArray](http://docs.google.com/java/util/concurrent/atomic/AtomicLongArray.html), and [AtomicReferenceArray](http://docs.google.com/java/util/concurrent/atomic/AtomicReferenceArray.html) classes further extend atomic operation support to arrays of these types. These classes are also notable in providing volatile access semantics for their array elements, which is not supported for ordinary arrays.

The atomic classes also support method weakCompareAndSet, which has limited applicability. On some platforms, the weak version may be more efficient than compareAndSet in the normal case, but differs in that any given invocation of the weakCompareAndSet method may return false *spuriously* (that is, for no apparent reason). A false return means only that the operation may be retried if desired, relying on the guarantee that repeated invocation when the variable holds expectedValue and no other thread is also attempting to set the variable will eventually succeed. (Such spurious failures may for example be due to memory contention effects that are unrelated to whether the expected and current values are equal.) Additionally weakCompareAndSet does not provide ordering guarantees that are usually needed for synchronization control. However, the method may be useful for updating counters and statistics when such updates are unrelated to the other happens-before orderings of a program. When a thread sees an update to an atomic variable caused by a weakCompareAndSet, it does not necessarily see updates to any *other* variables that occurred before the weakCompareAndSet. This may be acceptable when, for example, updating performance statistics, but rarely otherwise.

The [AtomicMarkableReference](http://docs.google.com/java/util/concurrent/atomic/AtomicMarkableReference.html) class associates a single boolean with a reference. For example, this bit might be used inside a data structure to mean that the object being referenced has logically been deleted. The [AtomicStampedReference](http://docs.google.com/java/util/concurrent/atomic/AtomicStampedReference.html) class associates an integer value with a reference. This may be used for example, to represent version numbers corresponding to series of updates.

Atomic classes are designed primarily as building blocks for implementing non-blocking data structures and related infrastructure classes. The compareAndSet method is not a general replacement for locking. It applies only when critical updates for an object are confined to a *single* variable.

Atomic classes are not general purpose replacements for java.lang.Integer and related classes. They do *not* define methods such as hashCode and compareTo. (Because atomic variables are expected to be mutated, they are poor choices for hash table keys.) Additionally, classes are provided only for those types that are commonly useful in intended applications. For example, there is no atomic class for representing byte. In those infrequent cases where you would like to do so, you can use an AtomicInteger to hold byte values, and cast appropriately. You can also hold floats using Float.floatToIntBits and Float.intBitstoFloat conversions, and doubles using Double.doubleToLongBits and Double.longBitsToDouble conversions.

**Since:** 1.5

| | [**Overview**](http://docs.google.com/overview-summary.html) | **Package** | Class | [**Use**](http://docs.google.com/package-use.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV PACKAGE**](http://docs.google.com/java/util/concurrent/package-summary.html)   [**NEXT PACKAGE**](http://docs.google.com/java/util/concurrent/locks/package-summary.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/atomic/package-summary.html)    [**NO FRAMES**](http://docs.google.com/package-summary.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |

[Submit a bug or feature](http://bugs.sun.com/services/bugreport/index.jsp)

For further API reference and developer documentation, see [Java SE Developer Documentation](http://docs.google.com/webnotes/devdocs-vs-specs.html). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2006 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](http://docs.google.com/legal/license.html). Also see the [documentation redistribution policy](http://java.sun.com/docs/redist.html).