**A Guide to AtomicInteger in Java**

The **AtomicInteger** class protects an underlying int value by providing methods that perform **atomic operations** on the value. It shall not be used as a replacement for an Integer class.

The AtomicInteger class is part of the java.util.concurrent.atomic package since Java 1.5.

**1. Create, get and set value of AtomicInteger**

The creation of AtomicInteger is straight forward by calling a constructor. The AtomicInteger provides two methods to get and set the values of it’s instances.

|  |
| --- |
| AtomicInteger example |
| //Initial value is 0  AtomicInteger atomicInteger = new AtomicInteger();    //Initial value is 100  AtomicInteger atomicInteger = new AtomicInteger(100);    int currentValue = atomicInteger.get();         //100    atomicInteger.set(1234);                        //Now value is 1234 |

**2. When to use AtomicInteger in Java**

In real life uses, we will need AtomicInteger in two cases:

1. As an **atomic counter** which is being used by multiple threads concurrently.
2. In [**compare-and-swap**](https://howtodoinjava.com/java/multi-threading/compare-and-swap-cas-algorithm/) operations to implement non-blocking algorithms.

**2.1. AtomicInteger as atomic counter**

To use it as counter, AtomicInteger class provides few methods which perform the addition and subtraction operations **atomically**.

* addAndGet() – Atomically adds the given value to the current value and returns new value *after* the addition.
* getAndAdd() – Atomically adds the given value to the current value and returns old value.
* incrementAndGet() – Atomically increments the current value by 1 and returns new value *after* the increment. It is equivalent to **++i** operation.
* getAndIncrement() – Atomically increment the current value and returns old value. It is equivalent to **i++** operation.
* decrementAndGet() – Atomically decrements the current value by 1 and returns new value *after* the decrement. It is equivalent to **i- –** operation.
* getAndDecrement() – Atomically decrements the current value and returns old value. It is equivalent to **– -i** operation.

|  |
| --- |
| AtomicInteger - add and subtract values |
| public class Main  {      public static void main(String[] args)      {          AtomicInteger atomicInteger = new AtomicInteger(100);            System.out.println(atomicInteger.addAndGet(2));         //102          System.out.println(atomicInteger);                      //102            System.out.println(atomicInteger.getAndAdd(2));         //102          System.out.println(atomicInteger);                      //104            System.out.println(atomicInteger.incrementAndGet());    //105          System.out.println(atomicInteger);                      //105            System.out.println(atomicInteger.getAndIncrement());    //105          System.out.println(atomicInteger);                      //106            System.out.println(atomicInteger.decrementAndGet());    //105          System.out.println(atomicInteger);                      //105            System.out.println(atomicInteger.getAndDecrement());    //105          System.out.println(atomicInteger);                      //104      }  } |

**2.2. Compare and swap operations**

A compare and swap operation compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This is done as a single atomic operation.

The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail.

To support compare and swap operations, this class provides a method which atomically sets the value to the given updated value if the current value == the expected value.

|  |
| --- |
| boolean compareAndSet(int expect, int update) |

We can see many real time uses of compareAndSet() method in Java concurrent collection classes such as [ConcurrentHashMap](https://howtodoinjava.com/java/multi-threading/best-practices-for-using-concurrenthashmap/).

|  |
| --- |
| AtomicInteger compareAndSet() example |
| import java.util.concurrent.atomic.AtomicInteger;    public class Main  {      public static void main(String[] args)      {          AtomicInteger atomicInteger = new AtomicInteger(100);            boolean isSuccess = atomicInteger.compareAndSet(100,110);   //current value 100            System.out.println(isSuccess);      //true            isSuccess = atomicInteger.compareAndSet(100,120);       //current value 110            System.out.println(isSuccess);      //false        }  } |

Program output.

|  |
| --- |
| Console |
| true  false |

**3. Conclusion**

As discussed above, the primary use of AtomicInteger is when we are in **multi-threaded context** and we need to perform **atomic operations** on an int value without using [synchronized](https://howtodoinjava.com/java/multi-threading/java-synchronized/) keyword.

Using the AtomicInteger is equally faster and more readable than performing the same using synchronization.