Atomic Boolean

Get a clear understanding and use cases of the AtomicBoolean class and its differences with volatile boolean variables.

Explanation

The `AtomicBoolean` class belonging to Java's `java.util.concurrency.atomic` package represents a boolean value that can be updated and modified atomically. The `Atomic*` family of classes extend the notion of `volatile` variables that are designed to be operated upon without locking using machine-level atomic instructions available on modern processors. However, on other platforms some form of internal locking may be used to serialize thread access.

`Atomic*` classes including `AtomicBoolean` offer a method `compareAndSet(expectedValue, updatedValue)` to conditionally update the value of the variable to `updatedValue` if it is set to `expectedValue` in one go, i.e. atomically. All read-and-update methods except for `lazySet()` and `weakCompareAndSet()` have memory effects equivalent of both reading and writing `volatile` variables.

The read and write methods i.e. `get()` and `set()` on instances of this class are similar in behavior to volatile variables i.e `get()` has the memory effect of reading a volatile variable and `set()` has the memory effect of writing a volatile variable.

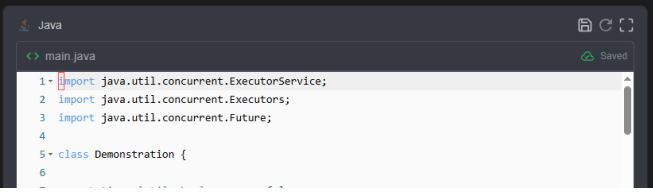
Difference between Volatile boolean and AtomicBoolean

It may be tempting to confuse and equate `volatile boolean` with `AtomicBoolean`, however, the two differ in several respects. Though it is fair to say that `volatile boolean` exhibits a subset of the functionality of `AtomicBoolean`. Recall, from our discussion on the `volatile` keyword that Java's memory model and compiler optimization can result in a situation where one thread doesn't see the latest value of a shared variable in main memory since it is using a cached stale value. Marking such a shared variable as `volatile` ensures that all threads see the latest state of the variable at all times. The quintessential example of this scenario found in online literature is as follows:

```
class SomeClass {

   boolean quit = false;

   void thread1() {

   |  while (!quit) {
   |  |  // ... Take some action
   |  }
   }

   void thread2() {
   |  // thread 1 may never see the updated value of quit!
   |  quit = true;
   }
}
```

Without `volatile` the thread executing the method `thread1` in the above program can spin in an infinite loop and never quit. Apart from delivering a consistent view of the memory, the `volatile` keyword doesn't promise much in synchronization guarantees. Specifically, multiple threads accessing a `volatile` keyword don't do so in a serialized manner. The onus of making a volatile variable's accesses synchronized and thread-safe is on the developer. This is where the `AtomicBoolean` classes come in. For instance, the methods `compareAndSet()` and `getAndSet()` exposed by the `AtomicBoolean` class represent a series of operations executed atomically, which otherwise would require synchronization on the part of the developer and are unachievable as an atomic transaction using `volatile` variables.

To make the distinction clearer let's work with an example. Say, we instantiate several threads and have them race to change a shared boolean variable's `won` value to true. The first one to do so is considered to have won the race. The rest print a message to indicate they lost the race. We mark the variable `won` as volatile and run the program below:



```
    static volatile boolean won = false;

    public static void main( String args[] ) throws Exception {
        ExecutorService es = Executors.newFixedThreadPool(25);
        try {
            int numThreads = 15;
```

Run

If you run the above program enough times especially on a machine with multiple processes you'll observe multiple threads printing the winning statement. The widget above may not show the faulty run since the VM executing the code may have. asingle vCPU. The `volatile` variable doesn't prevent multiple threads from accessing and then updating the variable `won`. A possible sequence resulting in multiple threads declaring themselves as winners can be:

1. Thread A reads the value of `won` which is false.
2. Thread B reads the value of `won` which is false.
3. Thread A changes the value of `won` to true and the variable is updated in main memory for all threads to see.
4. Thread B doesn't know Thread A had read the value of the variable `won` before Thread B accessed it. Thread B too updates the value of `won` to true.
5. Since `won` is a volatile variable, it always reflects its latest value to the next thread reading it but that's about it.

The above program can be fixed by using appropriate synchronization as shown in the widget below:

Java — Press ⇧ + ↵ to interact

main.java — Saved

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    static Object syncObj = new Object();
    static boolean won = false;

    public static void main( String args[] ) throws Exception {
        ExecutorService es = Executors.newFixedThreadPool(25);
        try {
```

Run

The above program synchronizes access to the `won` variable and only a single thread ever operates on the `won` variable at a time. Note that we have not marked `won` as `volatile` since the `synchronized` block establishes the *happens-before* relationship i.e. any thead reading the variables updated in the `synchronized` block see the latest value for those variables. We can rid of the `synchronized` block by using `AtomicBoolean` as shown in the following widget:

Java — Press ⇧ + ↵ to interact

main.java — Saved

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.atomic.AtomicBoolean;

class Demonstration {

    static AtomicBoolean won = new AtomicBoolean(false);

    public static void main( String args[] ) throws Exception {
        ExecutorService es = Executors.newFixedThreadPool(25);
        try {
```

Run

The method `compareAndSet(expectedValue, updatedValue)` atomically checks if the variable `won` is false and sets it to true and then returns true if the entire operation is successful. Since these compound steps occur atomically, we can be assured that only a single thread ever gets to set the variable `won` to true. Other threads all see the variable `won` as false and fall in the `else` clause. Using `AtomicBoolean` instead of `synchronized` block results in code that is more readable, concise and faster on machines with support for machine-level atomic operations.

In summary, you can think of the `AtomicBoolean` and its sibling classes as offering `volatile`-like capabilities and then some.

Technical Quiz

Consider the following statement and pick the correct statement:

1.

```
static volatile AtomicBoolean boolVar;
```

A. The statement won't compile as `AtomicBoolean` already mirrors the `volatile` functionality.

B. The `volatile` is superfluous and removing it doesn't break any functionality.

C. `volatile` is needed for synchronization purposes.

Reset Quiz

Q1 / Q2

Submit Answer

← Back lesson

StampedLock

Mark As Completed

Next →

AtomicInteger