Java Atomic Classes & Operations

(Part 2)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Be aware of the Java memory model
- Understand how Java atomic operations provide concurrent programs with lock-free & thread-safe mechanisms to read from & write to single variables
- Recognize how Java atomic classes & operations are implemented

**Class AtomicLong**

```java
public class AtomicLong
extends Number
implements Serializable
```

A long value that may be updated atomically. See the java.util.concurrent.atomic package specification for description of the properties of atomic variables. An AtomicLong is used in applications such as atomically incremented sequence numbers, and cannot be used as a replacement for a Long. However, this class does extend Number to allow uniform access by tools and utilities that deal with numerically-based classes.

**Since:**
1.5

**See Also:**
Serialized Form
Learning Objectives in this Part of the Lesson

• Be aware of the Java memory model
• Understand how Java atomic operations provide concurrent programs with lock-free & thread-safe mechanisms to read from & write to single variables
• Recognize how Java atomic classes & operations are implemented
• Appreciate Java atomic class & operation usage considerations
Implementation of Java Atomic Operations
Java uses CAS extensively in the JVM & portions of java.util.concurrent*
Implementation of Java Atomic Operations

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*
- E.g., compareAndSwapLong()

```java
public final class Unsafe {
    public final native boolean compareAndSwapLong(Object o, long offset, long expected, long updated);
}
```

Implementation of Java Atomic Operations

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*
  - e.g., compareAndSwapLong()

```
public final class Unsafe {
    public final native boolean compareAndSwapLong(Object o,
                                                   long offset,
                                                   long expected,
                                                   long updated) {
        START_ATOMIC();
        int *base = (int *) o;
        int oldValue = base[offset];
        if (oldValue == expected)
            base[offset] = updated;
        END_ATOMIC();
        return oldValue;
    }
    ...
```

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value.
Java uses CAS extensively in the JVM & portions of java.util.concurrent

- e.g., compareAndSwapLong()

```java
public final class Unsafe {
    public final native boolean compareAndSwapLong(Object o,
                                                    long offset,
                                                    long expected,
                                                    long updated) {

        START_ATOMIC();
        int *base = (int *) o;
        int oldValue = base[offset];
        if (oldValue == expected)
            base[offset] = updated;
        END_ATOMIC();
        return oldValue;
    }

    ...
}
```

*This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value*
Java uses CAS extensively in the JVM & portions of java.util.concurrent*

- e.g., compareAndSwapLong()

```java
public final class Unsafe {
    public final native boolean compareAndSwapLong(Object o, long offset, long expected, long updated) {
        START_ATOMIC();
        int *base = (int *) o;
        int oldValue = base[offset];
        if (oldValue == expected) { 
            base[offset] = updated;
        END_ATOMIC();
        return oldValue;
    }
    ...
}
```

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value.
Implementation of Java Atomic Operations

• Java uses CAS extensively in the JVM & portions of java.util.concurrent*
• e.g., compareAndSwapLong()

```java
public final class Unsafe {
    public final native boolean compareAndSwapLong(Object o,
                                                   long offset,
                                                   long expected,
                                                   long updated) {

        START_ATOMIC();
        int *base = (int *) o;
        int oldValue = base[offset];
        if (oldValue == expected)
            base[offset] = updated;
        END_ATOMIC();
        return oldValue;
    }
    ...

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value
```
Java uses CAS extensively in the JVM & portions of java.util.concurrent
  
  e.g., compareAndSwapLong()

Implementation of Java Atomic Operations

```java
public final class Unsafe {
    public final native boolean
    compareAndSwapLong(Object o,
                        long offset,
                        long expected,
                        long updated) {

        START_ATOMIC();
        int *base = (int *) o;
        int oldValue = base[offset];
        if (oldValue == expected)
            base[offset] = updated;
        END_ATOMIC();
        return oldValue;
    }

    ...
}
```

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value.
Implementation of Java AtomicLong
Implementation of Java AtomicLong

- AtomicLong contains a value that is updated atomically

Class AtomicLong

```java
java.lang.Object
    java.lang.Number
        java.util.concurrent.atomic.AtomicLong
```

All Implemented Interfaces:
Serializable

```java
public class AtomicLong
extends Number
implements Serializable
```

A long value that may be updated atomically. See the java.util.concurrent.atomic package specification for description of the properties of atomic variables. An AtomicLong is used in applications such as atomically incremented sequence numbers, and cannot be used as a replacement for a Long. However, this class does extend Number to allow uniform access by tools and utilities that deal with numerically-based classes.

Since:
1.5

See Also:
Serialized Form

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html)
Implementation of Java AtomicLong

- AtomicLong uses method `compareAndSwapLong()`

```java
10: /**
11: * A <tt>long</tt> value that may be updated atomically. See the
12: * {link java.util.concurrent.atomic} package specification for
13: * description of the properties of atomic variables. An
14: * <tt>AtomicLong</tt> is used in applications such as atomically
15: * incremented sequence numbers, and cannot be used as a replacement
16: * for a {link java.lang.Long}. However, this class does extend
17: * <tt>Number</tt> to allow uniform access by tools and utilities that
18: * deal with numerically-based classes.
19: *
20: * @since 1.5
21: * @author Doug Lea
22: */
23: public class AtomicLong extends Number implements java.io.Serializable {
24:     private static final long serialVersionUID = 1927816293512124184L;
25:     // setup to use Unsafe.compareAndSwapLong for updates
26:     private static final Unsafe unsafe = Unsafe.getUnsafe();
27:     private static final long valueOffset;
28:     /**
29:      * Records whether the underlying JVM supports lockless
30:      * CompareAndSet for longs. While the unsafe.compareAndSetLong
31:      * method works in either case, some constructions should be
32:      * handled at Java level to avoid locking user-visible locks.
33:      */
34:     static final boolean VM_SUPPORTS_LONG_CAS = VMSupportsCS8();
35:     /**
36:      * Returns whether underlying JVM supports lockless CompareAndSet
37:      * for longs. Called only once and cached in VM_SUPPORTS_LONG_CAS.
38:      */
39:     private static native boolean VMSupportsCS8();
40:     static {
41:         try {
42:             valueOffset = unsafe.objectFieldOffset
43:                 (AtomicLong.class.getDeclaredField("value"));
44:         } catch (Exception ex) { throw new Error(ex); }
45:     }
46:     private volatile long value;
```
Implementation of Java AtomicLong

- AtomicLong uses method `compareAndSwapLong()`

```java
public class AtomicLong {
    ...
    private volatile long value;
    ...
    
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    
    private static final long valueOffset;
    static {
        ...
        valueOffset = unsafe.objectFieldOffset(
            AtomicLong.class.
            getDeclaredField("value"));
        ...
    }
    
    ...
}
```

This volatile field will be read from & written to atomically via CAS operations

See [en.wikipedia.org/wiki/Volatile_(computer_programming)#In_Java](en.wikipedia.org/wiki/Volatile_(computer_programming)#In_Java)
AtomicLong uses method `compareAndSwapLong()`

```java
public class AtomicLong {
    private volatile long value;
    ...}

private static final Unsafe unsafe = Unsafe.getUnsafe();

private static final long valueOffset;

static {
    ...}

static {
    ... valueOffset = unsafe.objectFieldOffset(
        (AtomicLong.class.
        getDeclaredField("value")));
    ...}

...}
```

Java reflection is used to determine & store the offset of volatile 'value'

See [docs.oracle.com/javase/tutorial/reflect](docs.oracle.com/javase/tutorial/reflect)
Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```java
public final class Unsafe {
    public final long getAndAddLong(
        Object o,
        long offset,
        long delta) {
        long v;
        do {
            v = getIntVolatile(o, offset);
        } while (!compareAndSwapLong(
            o, offset,
            v, v + delta));
        return v;
    }
}
```

*Unsafe.getAndAddLong() atomically updates a value at an offset in the object*

AtomicLong uses method `compareAndSwapLong()` implementation of Java AtomicLong:

```java
public final class Unsafe {
    public final long getAndAddLong
        (Object o,
         long offset,
         long delta) {
        long v;
        do {
            v = getIntVolatile (o, offset);
        } while (!compareAndSwapLong
                   (o, offset,
                    v, v + delta));
        return v;
    }
```

This “lock-free” call runs atomically.
Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```java
public final class Unsafe {
    public final long getAndAddLong
        (Object o, long offset, long delta) {
        long v;
        do {
            v = getIntVolatile (o, offset);
        } while (!compareAndSwapLong
            (o, offset,
             v, v + delta));
        return v;
    }
}
```

The 'offset' is relative to the start of object 'o'
public final class Unsafe {
    public final long getAndAddLong
        (Object o, long offset, long delta) {
        long v;
        do {
            v = getIntVolatile
                (o, offset);
        } while (!compareAndSwapLong
            (o, offset,
             v, v + delta));
        return v;
    }
}

• AtomicLong uses method compareAndSwapLong()
Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```java
public final class Unsafe {
    public final long getAndAddLong(
        Object o,
        long offset,
        long delta) {
        long v;
        do { v = getIntVolatile(o, offset); } while (!
            compareAndSwapLong(o, offset,
            v, v + delta));
        return v;
    }
}
```

'delta' is atomically added to value 'v' iff 'v' hasn't changed since it was read
AtomicLong uses method compareAndSwapLong()

```
public class AtomicLong {
    ...

    private volatile long value;
    ...

    public final long getAndIncrement() {
        return unsafe
            .getAndAddLong(this, valueOffset, 1L);
    }

    public final long getAndDecrement() {
        return unsafe
            .getAndAddLong(this, valueOffset, -1L);
    }
```
Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()
Implementation of Java AtomicLong

- AtomicLong uses method `compareAndSwapLong()`

```java
public class AtomicLong {
    ... {

        private volatile long value;
    ... public final Boolean compareAndSet
        (long expect, long update) {
            return unsafe
                .compareAndSwapLong
                (this,
                 valueOffset,
                 expect,
                 update);

        }
    ...}
```

*Unsafe.compareAndSwapLong() attempts to update value atomically!*
AtomicInteger & AtomicBoolean are implemented very similarly to AtomicLong.

**Class AtomicBoolean**

```java
java.lang.Object
    java.util.concurrent.atomic.AtomicBoolean

All Implemented Interfaces:
Serializable
```

```java
public class AtomicBoolean
extends Object
implements Serializable

A boolean value that may be updated atomically. See the
java.util.concurrent.atomic package specification for
description of the properties of atomic variables. An
AtomicBoolean is used in applications such as atomically
updated flags, and cannot be used as a replacement for a
Boolean.
```

**Class AtomicInteger**

```java
java.lang.Object
    java.lang.Number
        java.util.concurrent.atomic.AtomicInteger

All Implemented Interfaces:
Serializable
```

```java
public class AtomicInteger
extends Number
implements Serializable

A int value that may be updated atomically. See the
java.util.concurrent.atomic package specification for
description of the properties of atomic variables. An
AtomicInteger is used in applications such as atomically
incremented counters, and cannot be used as a replacement
for an Integer. However, this class does extend Number to
allow uniform access by tools and utilities that deal with
numerically-based classes.
```
Usage Considerations for Java Atomic Operations
Usage Considerations for Java Atomic Operations

- Programs should use atomic operations carefully since they “busy wait”
Usage Considerations for Java Atomic Operations

- Programs should use atomic operations carefully since they “busy wait”
- Busy waiting needlessly wastes CPU cycles if contention is high

See [en.wikipedia.org/wiki/Busy_waiting](en.wikipedia.org/wiki/Busy_waiting)
Usage Considerations for Java Atomic Operations

- Programs should use atomic operations carefully since they “busy wait”
  - Busy waiting needlessly wastes CPU cycles if contention is high
- However, some “spinning” is useful in multi-core processors

See www.youtube.com/watch?v=sq0MX3fHkro
Usage Considerations for Java Atomic Operations

• Programs should use atomic operations carefully since they “busy wait”
  • Busy waiting needlessly wastes CPU cycles if contention is high
  • However, some “spinning” is useful in multi-core processors
    • e.g., due to context switching overhead of sleep locks

See www.youtube.com/watch?v=sq0MX3fHkro
Usage Considerations for Java Atomic Operations

- The `compareAndSet*()` methods in the various Java Atomic* classes provide a portable means of accessing low-level CAS operations.

```java
public final boolean compareAndSet(boolean expect, boolean update)
```

Atomically sets the value to the given updated value if the current value == the expected value.

**Parameters:**
- `expect` - the expected value
- `update` - the new value

**Returns:**
- `true` if successful. False return indicates that the actual value was not equal to the expected value.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html#compareAndSet](docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html#compareAndSet)
End of Atomic Classes & Operations (Part 2)