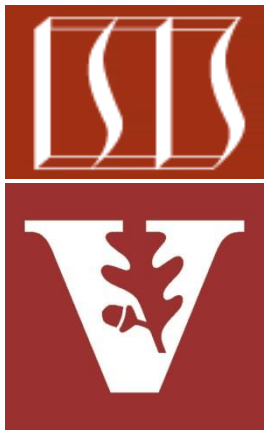
Java Atomic Classes & Operations (Part 2)



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

All Implemented Interfaces:

Serializable

```
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

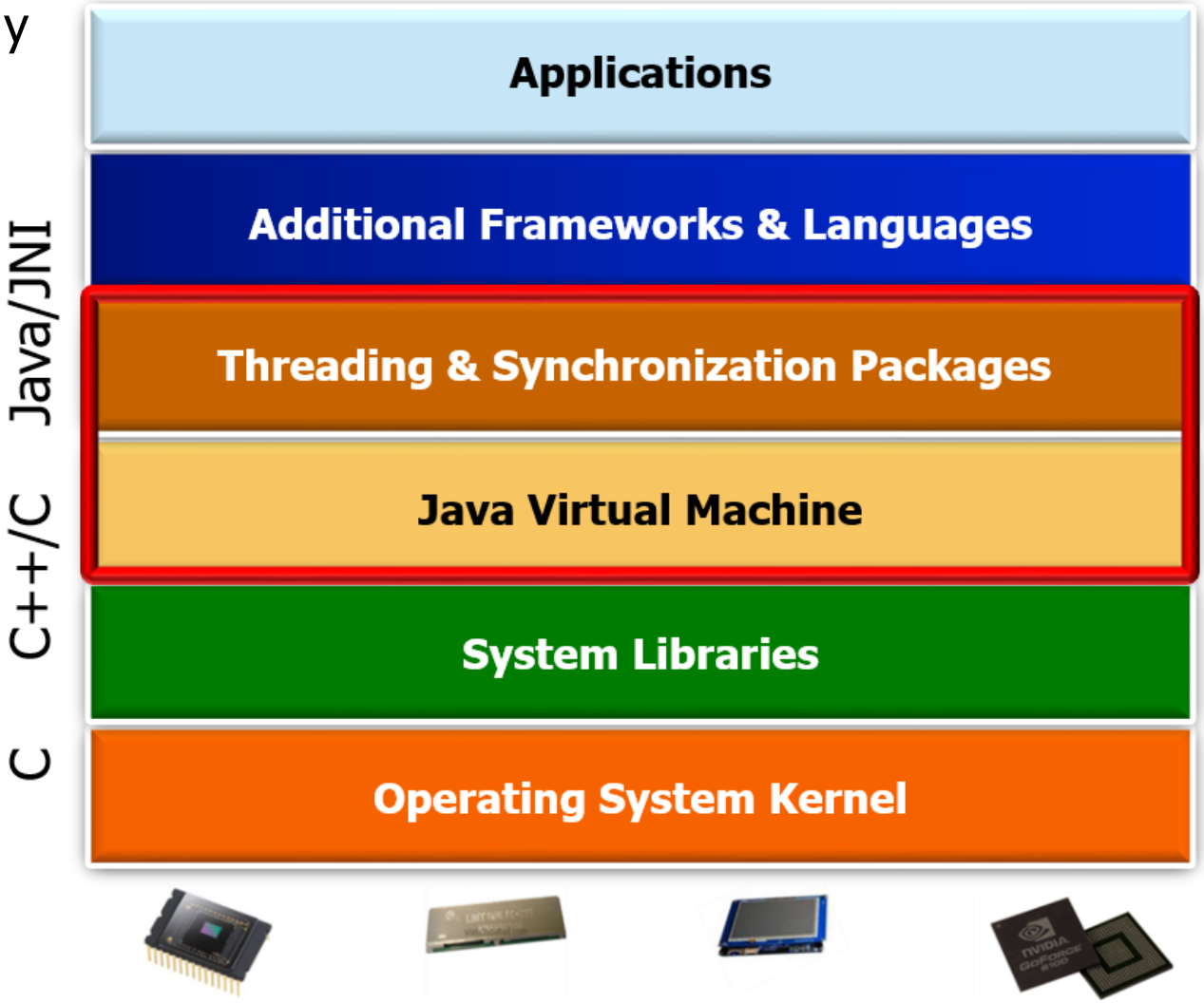
Learning Objectives in this Part of the Lesson

- 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

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()

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

See www.docjar.com/html/api/sun/misc/Unsafe.java.html

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

See www.docjar.com/html/api/sun/misc/Unsafe.java.html

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*

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*

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*

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***

Implementation of Java AtomicLong

Implementation of Java AtomicLong

- AtomicLong contains a value that is updated atomically

Class AtomicLong

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

All Implemented Interfaces:

`Serializable`

```
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

Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```
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:
26:     // setup to use Unsafe.compareAndSwapLong for updates
27:     private static final Unsafe unsafe = Unsafe.getUnsafe();
28:     private static final long valueOffset;
29:
30:     /**
31:      * Records whether the underlying JVM supports lockless
32:      * CompareAndSet for longs. While the unsafe.CompareAndSetLong
33:      * method works in either case, some constructions should be
34:      * handled at Java level to avoid locking user-visible locks.
35:      */
36:     static final boolean VM_SUPPORTS_LONG_CAS = VMSupportsCS8();
37:
38:     /**
39:      * Returns whether underlying JVM supports lockless CompareAndSet
40:      * for longs. Called only once and cached in VM_SUPPORTS_LONG_CAS.
41:      */
42:     private static native boolean VMSupportsCS8();
43:
44:     static {
45:       try {
46:         valueOffset = unsafe.objectFieldOffset
47:             (AtomicLong.class.getDeclaredField("value"));
48:       } catch (Exception ex) { throw new Error(ex); }
49:     }
50:
51:     private volatile long value;
```

See java/util/concurrent/atomic/AtomicLong.java

Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

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

```
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"));
     ...
   }
...
```

See en.wikipedia.org/wiki/Volatile_(computer_programming)#In_Java

Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```
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"));
     ...
   }
...
```

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

See docs.oracle.com/javase/tutorial/reflect

Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```
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

See www.docjar.com/html/api/sun/misc/Unsafe.java.html

Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```
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()

```
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'

Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```
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;
    }
```

'v' is the value at 'offset' into object 'o'

Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```
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

Implementation of Java AtomicLong

- 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);
   }
```

The Unsafe.getAndAddLong() method is used to increment & decrement values atomically!

Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```
public class AtomicLong
   ... {

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

Atomically sets value to given updated value if current value equals the expected value

Implementation of Java AtomicLong

- AtomicLong uses method compareAndSwapLong()

```
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!

Implementation of Java AtomicLong

- AtomicInteger & AtomicBoolean are implemented very similarly to AtomicLong

Class AtomicBoolean

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

All Implemented Interfaces:

Serializable

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

All Implemented Interfaces:

Serializable

```
public class AtomicInteger
extends Number
implements Serializable
```

An `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"

See 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

See 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

compareAndSet

```
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

End of Atomic Classes & Operations (Part 2)