Overview of Java Atomic Operations & Variables

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 Lesson

• Recognize Java programming language & class library features that provide atomic operations & variables
Overview of Java Atomic Operations & Variables
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity

See www.ibm.com/developerworks/library/j-jtp11234
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - *Volatile variables*

See upcoming lesson on "Java Volatile Variables"
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
    - Ensure a variable is read from & written to main memory & not cached

See [en.wikipedia.org/wiki/Volatile_variable#In_Java](en.wikipedia.org/wiki/Volatile_variable#In_Java)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
    - Ensure a variable is read from & written to main memory & not cached
    - e.g., sharing a field between two threads

```java
class PingPongTest {
    private volatile int val = 0;
    private int MAX = ...;

    public void playPingPong() {
        new Thread() -> { // T2 Listener.
            for (int lv = val; lv < MAX; )
                if (lv != val) {
                    print("pong(" + val + ")");
                    lv = val;
                }
        }).start();

        new Thread() -> { // T1 Changer.
            for (int lv = val; val < MAX; ) {
                val = ++lv;
                print("ping(" + lv + ")");
                ... Thread.sleep(500); ...
            }
        }).start();

    }
}
```

See [dzone.com/articles/java-volatile-keyword-0](dzone.com/articles/java-volatile-keyword-0)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
    - Ensure a variable is read from & written to main memory & not cached
    - e.g., sharing a field between two threads

```java
public class PingPongTest {
    private volatile int val = 0;
    private int MAX = ...;

    public void playPingPong() {
        new Thread(() -> { // T2 Listener.
            for (int lv = val; lv < MAX; )
                if (lv != val) {
                    print("pong(" + val + ")");
                    lv = val;
                }
        }).start();

        new Thread(() -> { // T1 Changer.
            for (int lv = val; val < MAX; )
                print("ping(" + ++lv + ")");
            val = lv;
            sleep(500);
        }).start();
    }
}
```

This program alternates printing "ping" & "pong" between threads $T_1$ & $T_2$

Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
    - Ensure a variable is read from & written to main memory & not cached
    - e.g., sharing a field between two threads

```java
class PingPongTest {
    private volatile int val = 0;
    private int MAX = ...;

    public void playPingPong() {
        new Thread(() -> { // T2 Listener.
            for (int lv = val; lv < MAX; )
                if (lv != val) {
                    print("pong(" + val + ")");
                    lv = val;
                }
        }).start();

        new Thread(() -> { // T1 Changer.
            for (int lv = val; val < MAX; )
                print("ping(" + ++lv + ")");
            val = lv;
            sleep(500);
        }).start();
    }
}
```

If volatile's omitted from `val`'s definition the program won't terminate since `val`'s not visible

By defining `val` as volatile reads & writes bypass local caches
Overview of Java Atomic Operations & Variables

• Java supports several types of atomicity, e.g.
  • **Volatile variables**
    • Ensure a variable is read from & written to main memory & not cached
    • e.g., sharing a field between two threads

```java
class PingPongTest {
    private volatile int val = 0;
    private int MAX = ...;

    public void playPingPong() {
        new Thread(() -> { // T2 Listener.
            for (int lv = val; lv < MAX; )
                if (lv != val) {
                    print("pong(" + val + ")");
                    lv = val;
                }
        }).start();
        new Thread(() -> { // T1 Changer.
            for (int lv = val; val < MAX; ) {
                print("ping(" + ++lv + ")");
                val = lv;
                sleep(500);
            }
        }).start();
    }
}
```

These reads from **val** are atomic
Java supports several types of atomicity, e.g.

Volatile variables

- Ensure a variable is read from & written to main memory & not cached

- e.g., sharing a field between two threads

```java
class PingPongTest {
    private volatile int val = 0;
    private int MAX = ...;

    public void playPingPong() {
        new Thread(() -> { // T2 Listener.
            for (int lv = val; lv < MAX; )
                if (lv != val) {
                    print("pong(" + val + ")");
                    lv = val;
                }
        }).start();

        new Thread(() -> { // T1 Changer.
            for (int lv = val; val < MAX; ) {
                print("ping(" + ++lv + ")");
                val = lv;
                sleep(500);
            }
        }).start();
    }
}
```

This write to `val` is atomic
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - *Volatile variables*
  - *Low-level atomic operations*

See upcoming lesson on “Java Atomic Operations & Classes”
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations, e.g.**
    - **The Java Unsafe class**
      - It’s designed for use only by the Java Class Library, not by normal app programs

**Concurrency**

And few words about concurrency with Unsafe.compareAndSwap methods are atomic and can be used to implement high-performance lock-free data structures.

For example, consider the problem to increment value in the shared object using lot of threads.

First we define simple interface `Counter`:

```java
interface Counter {
    void increment();
    long getCount();
}
```

Then we define worker thread `CounterClient`, that uses `Counter`:

```java
class CounterClient implements Runnable {
    private Counter c;
    private int num;

    public CounterClient(Counter c, int num) {
        this.c = c;
        this.num = num;
    }

    @Override
    public void run() {
        for (int i = 0; i < num; i++) {
            c.increment();
        }
    }
}
```

See [www.baeldung.com/java-unsafe](http://www.baeldung.com/java-unsafe)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations, e.g.**
    - **The Java Unsafe class**
      - It’s designed for use only by the Java Class Library, not by normal app programs
      - Its “compare & swap” (CAS) methods are quite useful

```java
int compareAndSwapInt(
    Object o, long offset,
    int expected, int updated) {
    START_ATOMIC();
    int *base = (int *) o;
    int oldValue = base[offset];
    if (oldValue == expected) {    
        base[offset] = updated;
    }
    END_ATOMIC();
    return oldValue;
}
```

Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations, e.g.**
    - **The Java Unsafe class**
      - It’s designed for use only by the Java Class Library, not by normal app programs
    - Its “compare & swap” (CAS) methods are quite useful

```java
int compareAndSwapInt(
    Object o, long offset,
    int expected, int updated) {
    START_ATOMIC();
    int *base = (int *) o;
    int oldValue = base[offset];
    if (oldValue == expected)
        base[offset] = updated;
    END_ATOMIC();
    return oldValue;
}
```

Atomically compare the contents of memory with a given value & modify contents to a new given value iff they are the same

See upcoming lesson on “Implementing Java Atomic Operations”
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations, e.g.**
    - **The Java Unsafe class**
      - It’s designed for use only by the Java Class Library, not by normal app programs
      - Its “compare & swap” (CAS) methods are quite useful
      - CAS methods can be used to implement efficient “lock free” algorithms

```java
void lock(Object o, long offset) {
    while (compareAndSwapInt (o, offset, 0, 1) > 0);
}

void unlock(Object o, long offset) {
    START_ATOMIC();
    int *base = (int *) o;
    base[offset] = 0;
    END_ATOMIC();
}
```

See [en.wikipedia.org/wiki/Non-blocking_algorithm](en.wikipedia.org/wiki/Non-blocking_algorithm)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations, e.g.**
    - *The Java Unsafe class*
      - It’s designed for use only by the Java Class Library, not by normal app programs
      - Its “compare & swap” (CAS) methods are quite useful
      - CAS methods can be used to implement efficient “lock free” algorithms

```java
void lock(Object o, long offset){
    while (compareAndSwapInt (o, offset, 0, 1) > 0);
}

void unlock(Object o, long offset){
    START_ATOMIC();
    int *base = (int *) o;
    base[offset] = 0;
    END_ATOMIC();
}
```

Uses CAS to implement a simple “mutex” spin-lock

See upcoming lesson on “Implementing Java Atomic Operations”
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations, e.g.**
    - **The Java Unsafe class**
      - It’s designed for use only by the Java Class Library, not by normal app programs
      - Its “compare & swap” (CAS) methods are quite useful
      - CAS methods can be used to implement efficient “lock free” algorithms
    - Synchronizers in the Java Class Library use CAS methods extensively

See [www.youtube.com/watch?v=sq0MX3fHkro](http://www.youtube.com/watch?v=sq0MX3fHkro)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations, e.g.**
    - The Java Unsafe class
    - The Java 9+ VarHandle class
      - Defines a standard for invoking equivalents of the `java.util.concurrent.atomic` & `sun.misc.Unsafe` operations on fields & array elements

See [docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html](docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations, e.g.**
    - The Java Unsafe class
    - The Java 9+ VarHandle class
      - Defines a standard for invoking equivalents of the java.util.concurrent.atomic & sun.misc.Unsafe operations on fields & array elements
      - Those operations are mostly atomic or ordered operations
        - e.g., CAS operations or incrementing atomic fields

See [www.baeldung.com/java-variable-handles](http://www.baeldung.com/java-variable-handles)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations, e.g.**
    - The Java Unsafe class
    - **The Java 9+ VarHandle class**
      - Defines a standard for invoking equivalents of the `java.util.concurrent.atomic` & `sun.misc.Unsafe` operations on fields & array elements
    - Those operations are mostly atomic or ordered operations
  - The VarHandle class is designed to be usable by apps, unlike the Java Unsafe class

---

**Using JDK 9 Memory Order Modes**

by Doug Lea.

Last update: Fri Nov 16 08:46:48 2018 Doug Lea

**Introduction**

This guide is mainly intended for expert programmers familiar with Java concurrency, but unfamiliar with the memory order modes available in JDK 9 provided by VarHandles. Mostly, it focuses on how to think about modes when developing parallel software. Feel free to first read the Summary.

To get the shockingly ugly syntactic details over with: A VarHandle can be associated with any field, array element, or static, allowing control over access modes. VarHandles should be declared as static final fields and explicitly initialized in static blocks. By convention, we give VarHandles for fields names that are uppercase versions of the field names. For example, in a Point class:

```java
import java.lang.invoke.MethodHandles;
import java.lang.invoke.VarHandle;

class Point {
    volatile int x, y;
    private static final VarHandle X;
    static {
        try {
            X = MethodHandles.lookup().
                findVarHandle(Point.class, "x",
                    int.class);
        } catch (ReflectiveOperationException e) {
            throw new Error(e);
        }
    }
}
```

See gee.cs.oswego.edu/dl/html/j9mm.html
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations**
  - **Atomic classes**

**Class Summary**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicBoolean</td>
<td>A boolean value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicInteger</td>
<td>An int value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicIntegerArray</td>
<td>An int array in which elements may be updated atomically.</td>
</tr>
<tr>
<td>AtomicIntegerFieldUpdater&lt;T&gt;</td>
<td>A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.</td>
</tr>
<tr>
<td>AtomicLong</td>
<td>A long value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicLongArray</td>
<td>A long array in which elements may be updated atomically.</td>
</tr>
<tr>
<td>AtomicLongFieldUpdater&lt;T&gt;</td>
<td>A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.</td>
</tr>
<tr>
<td>AtomicMarkableReference&lt;V&gt;</td>
<td>An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.</td>
</tr>
<tr>
<td>AtomicReference&lt;V&gt;</td>
<td>An object reference that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicReferenceArray&lt;E&gt;</td>
<td>An array of object references in which elements may be updated atomically.</td>
</tr>
</tbody>
</table>
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations**
  - **Atomic classes**
    - Use Java Unsafe or VarHandle classes internally to implement “lock-free” methods

**Package java.util.concurrent.atomic**
A small toolkit of classes that support lock-free thread-safe programming on single variables.
See: Description

<table>
<thead>
<tr>
<th>Class Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicBoolean</td>
<td>A boolean value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicInteger</td>
<td>An int value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicIntegerArray</td>
<td>An int array in which elements may be updated atomically.</td>
</tr>
<tr>
<td>AtomicIntegerFieldUpdater&lt;T&gt;</td>
<td>A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.</td>
</tr>
<tr>
<td>AtomicLong</td>
<td>A long value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicLongArray</td>
<td>A long array in which elements may be updated atomically.</td>
</tr>
<tr>
<td>AtomicLongFieldUpdater&lt;T&gt;</td>
<td>A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.</td>
</tr>
<tr>
<td>AtomicMarkableReference&lt;V&gt;</td>
<td>An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.</td>
</tr>
<tr>
<td>AtomicReference&lt;V&gt;</td>
<td>An object reference that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicReferenceArray&lt;E&gt;</td>
<td>An array of object references in which elements may...</td>
</tr>
</tbody>
</table>

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity, e.g.
  - **Volatile variables**
  - **Low-level atomic operations**
  - **Atomic classes**
    - Use Java Unsafe or Var Handle classes internally to implement “lock-free” methods
    - e.g., AtomicLong & AtomicBoolean

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html & docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html
End of Overview of Java
Atomic Operations & Variables