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 & library features that provide atomic operations & variables

Overview of Atomic Actions

Overview of Atomic Actions

- Atomic actions ensure that changes to a field are always consistent & visible to other threads

Atomic Access

In programming, an *atomic* action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as `c++`, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except `long` and `double`).
- Reads and writes are atomic for *all* variables declared `volatile` (*including* `long` and `double` variables).

See docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html

Overview of Atomic Actions

- Atomic actions ensure that changes to a field are always consistent & visible to other threads
  - An *atomic* action is one that effectively happens all at once or it doesn't happen at all

See en.wikipedia.org/wiki/Linearizability

Overview of Atomic Actions

- Atomic actions ensure that changes to a field are always consistent & visible to other threads
  - An *atomic* action is one that effectively happens all at once or it doesn't happen at all
    - i.e., it can't stop in the middle & leave an inconsistent state

Overview of Atomic Actions

- Atomic actions ensure that changes to a field are always consistent & visible to other threads
  - An *atomic* action is one that effectively happens all at once or it doesn't happen at all
  - Any side effects of an atomic action aren't visible until the action completes

Overview of Java Atomic Actions

- Three key concepts are associated with atomic actions in Java

See jeremymanson.blogspot.com/2007/08/atomicity-visibility-and-ordering.html

Overview of Java Atomic Actions

- Three key concepts are associated with atomic actions in Java
  - *Atomicity* deals with which (sets of) actions have indivisible effects

```
class NonAtomicOps {
  long mCounter = 0;

  void increment() { // Thread T2
    for (;;) {
      mCounter++;
    }
  }

  void decrement() { // Thread T1
    for (;;) {
      mCounter--;
    }
  }

  ...
}
```

The behavior of running increment() & decrement() concurrently is undefined & not predictable..

Overview of Java Atomic Actions

- Three key concepts are associated with atomic actions in Java
  - *Atomicity* deals with which (sets of) actions have indivisible effects
  - *Visibility* determines when one thread can the effects of another


```
class LoopMayNeverEnd {
  boolean mDone = false;

  void work() {
    // Thread T2 read
    while (!mDone) {
      // do work
    }
  }

  void stopWork() {
    // Thread T1 write
    mDone = true;
  }

  ...
}
```

It's possible that thread T_2 will never stop even after Thread T_1 sets mDone to true..

Overview of Java Atomic Actions

- Three key concepts are associated with atomic actions in Java
  - *Atomicity* deals with which (sets of) actions have indivisible effects
  - *Visibility* determines when one thread can the effects of another
  - *Ordering* determines when actions in one thread occur out of order with respect to another thread

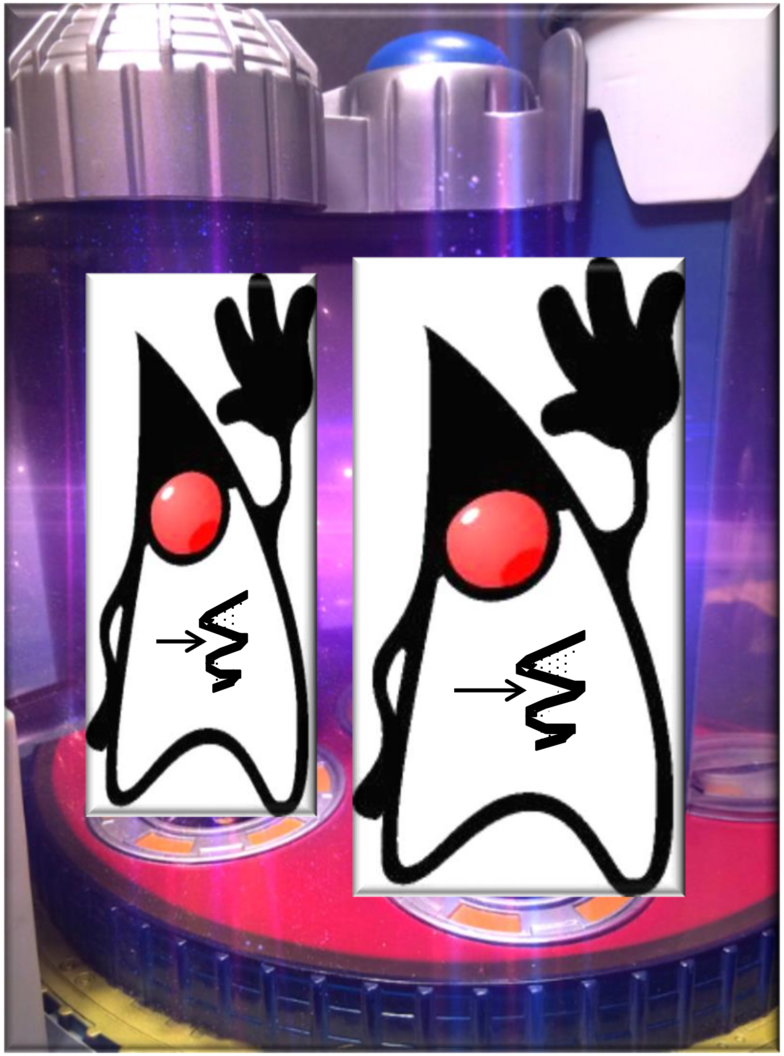

```
class BadlyOrdered {
  boolean a = false;
  boolean b = false;

  void method1() { // Thread T1
    a = true;
    b = true;
  }

  boolean method2() { // Thread T2
    boolean r1 = b; // sees true
    boolean r2 = a; // sees false
    boolean r3 = a; // sees true
    return (r1 && !r2) && r3;
    // returns true
  }
}
```

The order that fields a & b appear in thread T_2 may differ from the order they were set in Thread T_1!

Overview of Java Atomic Variables

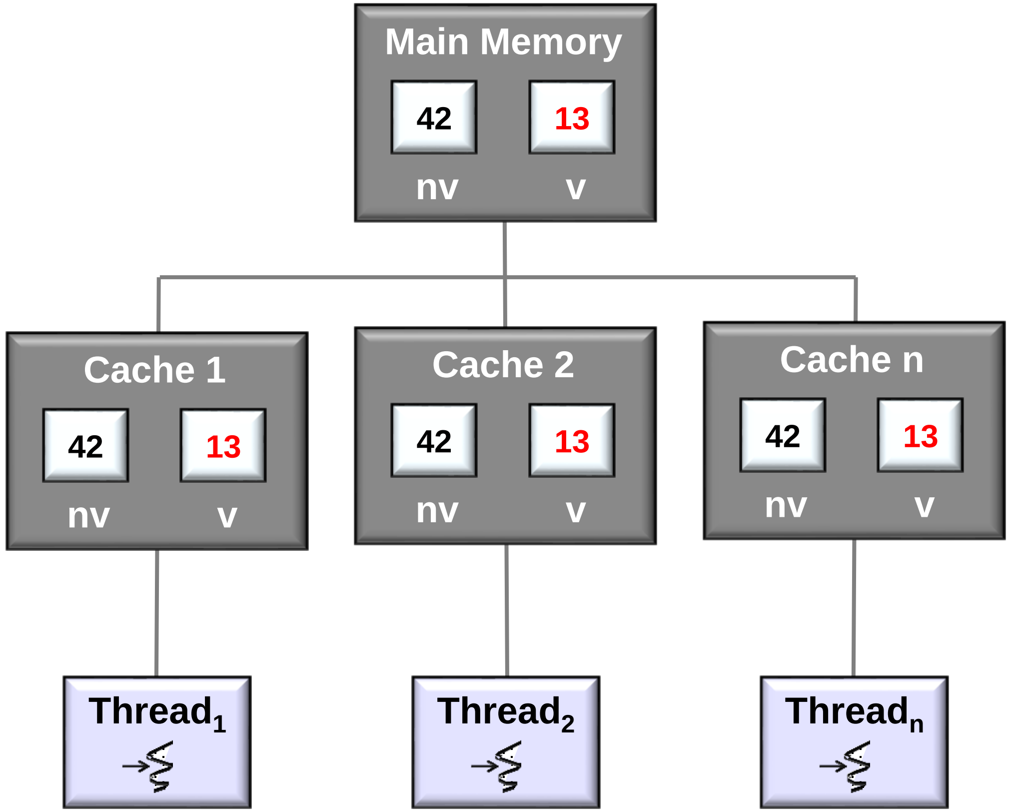
Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions



Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, 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

Overview of Java Atomic Operations & Variables

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

This program alternates printing "ping" & "pong" between two threads

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

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

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

See dzone.com/articles/java-volatile-keyword-0

Overview of Java Atomic Operations & Variables

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

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

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


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

If volatile is omitted from the definition of 'val' then the program doesn't terminate..

See dzone.com/articles/java-volatile-keyword-0

Overview of Java Atomic Operations & Variables

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

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

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

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

These reads from 'val' are atomic

See dzone.com/articles/java-volatile-keyword-0

Overview of Java Atomic Operations & Variables

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

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

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


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

This write to 'val' is atomic

See dzone.com/articles/java-volatile-keyword-0

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*

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`:

```
interface Counter {
    void increment();
    long getCounter();
}
```

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

```
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 mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*
    - It's designed for use only by the Java Class Library, not by normal 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`:

```
interface Counter {
    void increment();
    long getCounter();
}
```

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

```
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

Overview of Java Atomic Operations & Variables

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

```
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;
}
```

See en.wikipedia.org/wiki/Compare-and-swap

Overview of Java Atomic Operations & Variables

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

```
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;
}
```

This C-like pseudo-code compares contents of memory with a given value & modifies contents to a new given value iff they are the same

See en.wikipedia.org/wiki/Compare-and-swap

Overview of Java Atomic Operations & Variables

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

```
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

Overview of Java Atomic Operations & Variables

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

```
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();
}
```

Implements a simple "mutex" spin-lock

See en.wikipedia.org/wiki/Spinlock

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*
    - It's designed for use only by the Java Class Library, not by normal 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

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*
  - *Atomic classes*
    - Use Java Unsafe internally to implement "lock-free" algorithms

```
public class AtomicBoolean ... {
  private static final Unsafe unsafe
    = ...;

  private static final long
    valueOffset;

  private volatile int value;

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

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*
  - *Atomic classes*
    - Use Java Unsafe internally to implement "lock-free" algorithms

```
public class AtomicBoolean ... {
  private static final Unsafe unsafe
    = ...;

  private static final long
    valueOffset;

  private volatile int value;

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

Compute the offset of the 'value' field from the beginning of the object

See www.docjar.com/docs/api/sun/misc/Unsafe.html#objectFieldOffset

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*
  - *Atomic classes*
    - Use Java Unsafe internally to implement "lock-free" algorithms

```
public class AtomicBoolean ... {
  private static final Unsafe unsafe
    = ...;

  private static final long
    valueOffset;

  private volatile int value;

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

Uses the Java reflection API

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

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*
  - *Atomic classes*
    - Use Java Unsafe internally to implement "lock-free" algorithms

```
public class AtomicBoolean ... {
  private static final Unsafe unsafe
    = ...;

  private static final long
    valueOffset;

  private volatile int value;

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

Note the "value" field is volatile

See en.wikipedia.org/wiki/Volatile_variable#In_Java

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*
  - *Atomic classes*
    - Use Java Unsafe internally to implement "lock-free" algorithms
    - compareAndSet() uses Unsafe .compareAndSwapInt()

```
public class AtomicBoolean ... {
  ...
  public final boolean compareAndSet
                    (boolean expected,
                     boolean updated){
    int e = expected ? 1 : 0;
    int u = updated ? 1 : 0;
    return unsafe.compareAndSwapInt
        (this, valueOffset, e, u);
  }
  ...
```

See www.docjar.com/docs/api/sun/misc/Unsafe.html#compareAndSwapInt

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*
  - *Atomic classes*
    - Use Java Unsafe internally to implement "lock-free" algorithms
    - compareAndSet() uses Unsafe .compareAndSwapInt()

```
public class AtomicBoolean ... {
  ...
  public final boolean compareAndSet
                     (boolean expected,
                      boolean updated){
    int e = expected ? 1 : 0;
    int u = updated ? 1 : 0;
    return unsafe.compareAndSwapInt
           (this, valueOffset, e, u);
  }
  ...
```

Atomically updated field at valueOffset to 'updated' iff it's currently holding 'expected'

See www.docjar.com/docs/api/sun/misc/Unsafe.html#compareAndSwapInt

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*
  - *Atomic classes*
    - Use Java Unsafe internally to implement "lock-free" algorithms
    - compareAndSet() uses Unsafe .compareAndSwapInt()

```
public class AtomicBoolean ... {
  ...
  public final boolean compareAndSet
                          (boolean expected,
                           boolean updated){
    int e = expected ? 1 : 0;
    int u = updated ? 1 : 0;
    return unsafe.compareAndSwapInt
             (this, valueOffset, e, u);
  }
  ...
```

Returns true if successful, whereas false indicates that the actual value was not equal to the expected value

See www.docjar.com/docs/api/sun/misc/Unsafe.html#compareAndSwapInt

Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions, e.g.
  - *Volatile variables*
  - *Low-level atomic operations in the Java Unsafe class*
  - *Atomic classes*
    - Use Java Unsafe internally to implement "lock-free" algorithms
    - compareAndSet() uses Unsafe .compareAndSwapInt()

```
public class AtomicBoolean ... {
  ...
  public final boolean compareAndSet
                       (boolean expected,
                        boolean updated){
    int e = expected ? 1 : 0;
    int u = updated ? 1 : 0;
    return unsafe.compareAndSwapInt
           (this, valueOffset,
            e, u);
  }

  public final void set(boolean
                           newValue) {
    value = newValue ? 1 : 0;
  }
  ...
```

Unconditionally sets 'value' to given newValue via an atomic write on the volatile 'value'

End of Overview of Java Atomic Operations & Variables