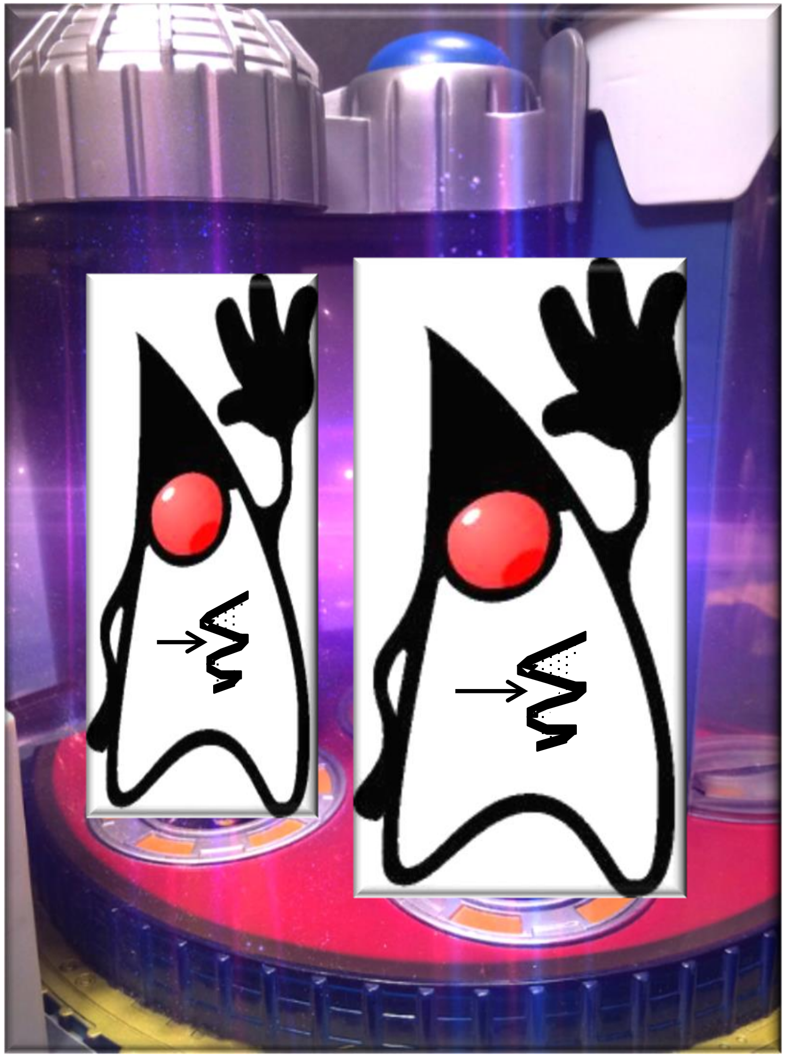
Java Atomic Classes & Operations: Introduction

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

- Understand how Java atomic classes & operations provide concurrent programs with lock-free, thread-safe mechanisms to read from & write to single variables



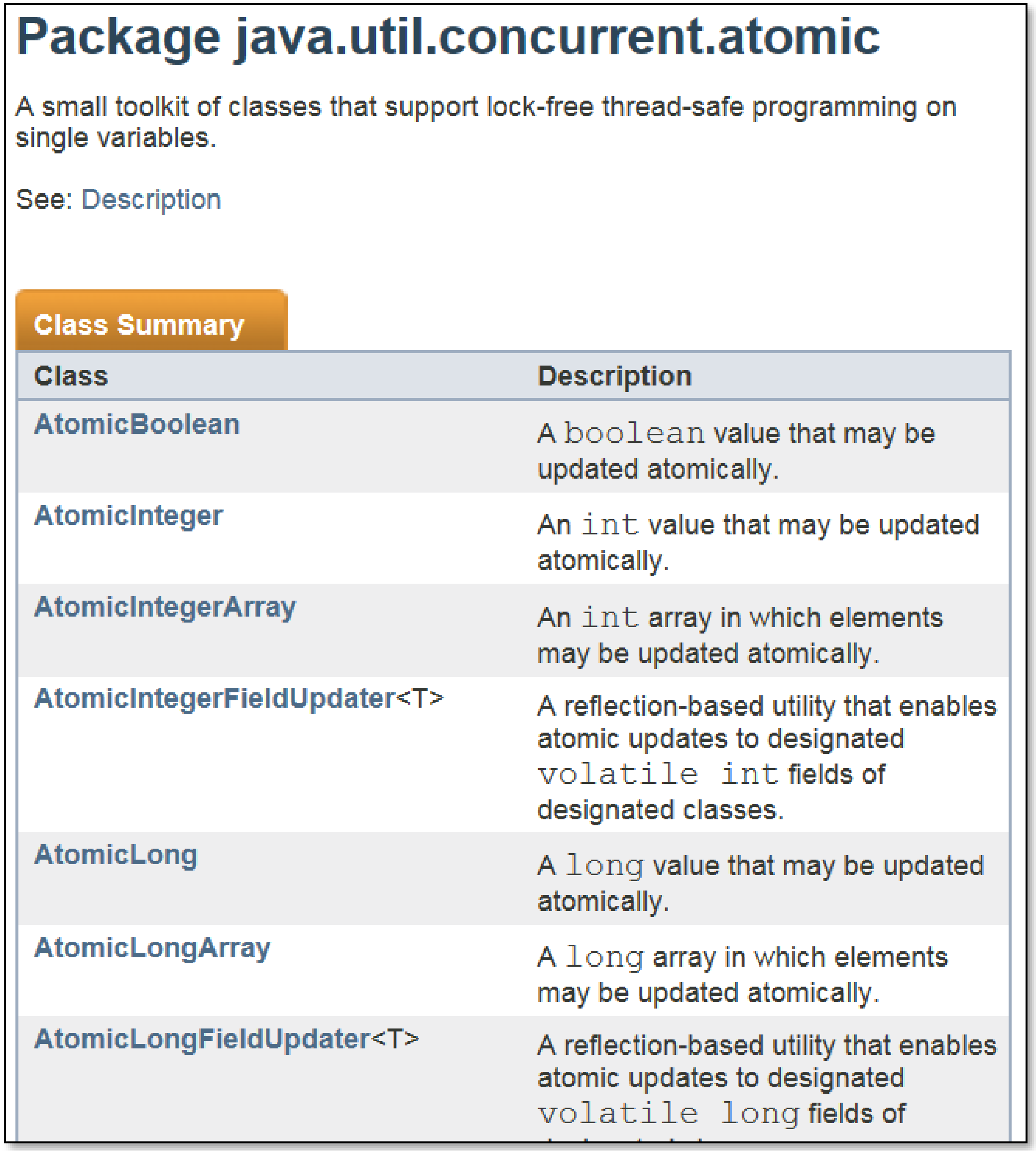
Learning Objectives in this Part of the Lesson

- Understand how Java atomic classes & operations provide concurrent programs with lock-free, thread-safe mechanisms to read from & write to single variables
- Note a human known use of atomic operations

Overview of Java Atomic Classes

Overview of Java Atomic Classes

- The java.util.concurrent.atomic package several types of atomic actions on objects



Package java.util.concurrent.atomic

A small toolkit of classes that support lock-free thread-safe programming on single variables.

See: Description

Class Summary

Class	Description
AtomicBoolean	A `boolean` value that may be updated atomically.
AtomicInteger	An `int` value that may be updated atomically.
AtomicIntegerArray	An `int` array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile int` fields of designated classes.
AtomicLong	A `long` value that may be updated atomically.
AtomicLongArray	A `long` array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile long` fields of

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

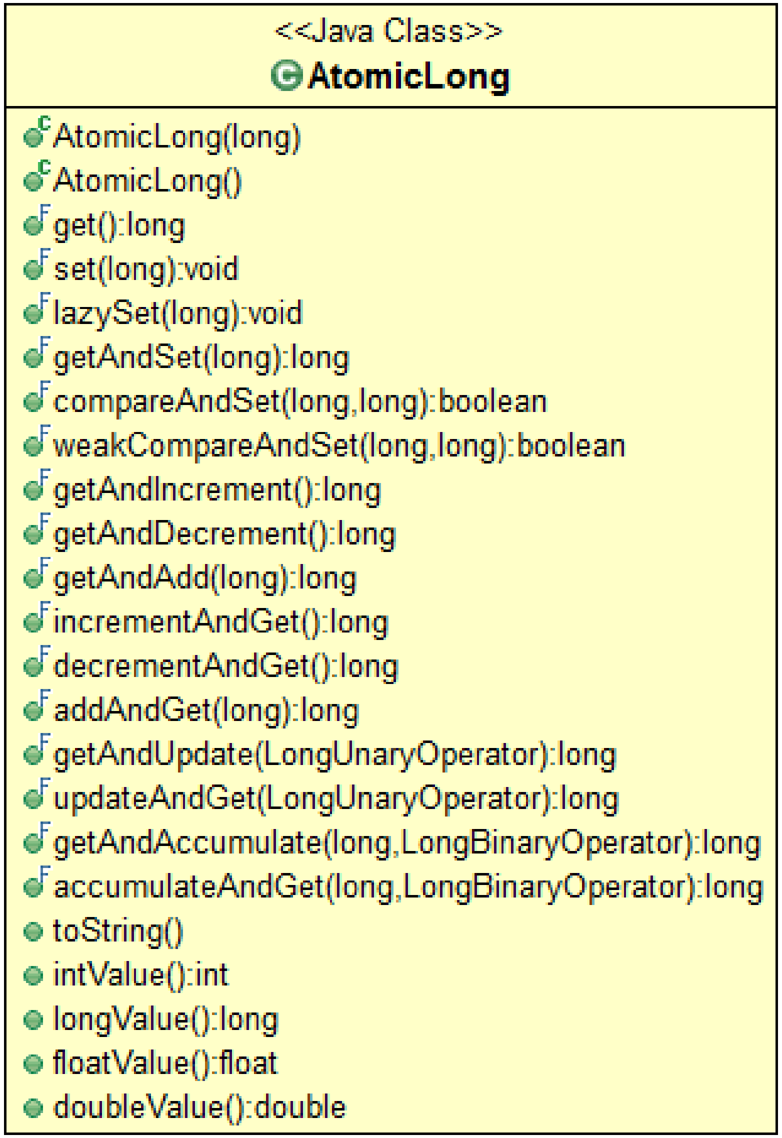
Overview of Java Atomic Classes

- The java.util.concurrent.atomic package several types of atomic actions on objects
  - *Atomic variables*
    - Provide lock-free & thread-safe operations on single variables

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

Overview of Java Atomic Classes

- The java.util.concurrent.atomic package several types of atomic actions on objects
  - *Atomic variables*
    - Provide lock-free & thread-safe operations on single variables
      - e.g., AtomicLong supports atomic "compare-and-swap" operations



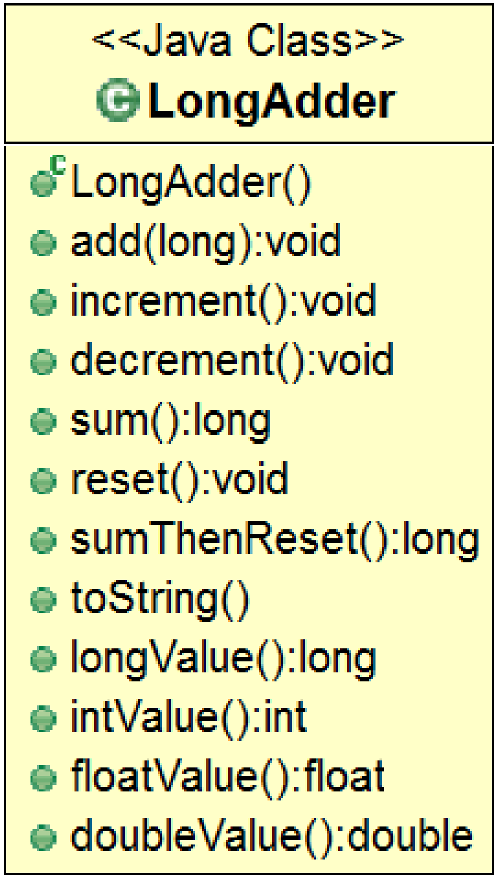
<<Java Class>>
AtomicLong

AtomicLong(long)
AtomicLong()
get():long
set(long):void
lazySet(long):void
getAndSet(long):long
compareAndSet(long,long):boolean
weakCompareAndSet(long,long):boolean
getAndIncrement():long
getAndDecrement():long
getAndAdd(long):long
incrementAndGet():long
decrementAndGet():long
addAndGet(long):long
getAndUpdate(LongUnaryOperator):long
updateAndGet(LongUnaryOperator):long
getAndAccumulate(long,LongBinaryOperator):long
accumulateAndGet(long,LongBinaryOperator):long
toString()
intValue():int
longValue():long
floatValue():float
doubleValue():double

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

Overview of Java Atomic Classes

- The java.util.concurrent.atomic package several types of atomic actions on objects
  - *Atomic variables*
  - *LongAdder*
    - Allows multiple threads to update a common sum efficiently under high contention



<<Java Class>>
LongAdder

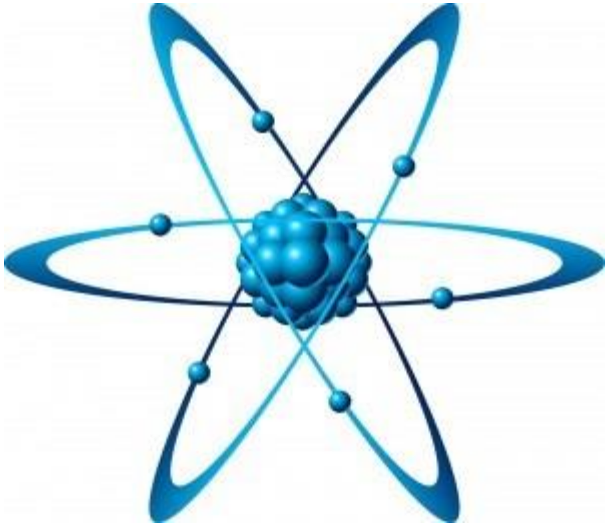
- LongAdder()
- add(long):void
- increment():void
- decrement():void
- sum():long
- reset():void
- sumThenReset():long
- toString()
- longValue():long
- intValue():int
- floatValue():float
- doubleValue():double

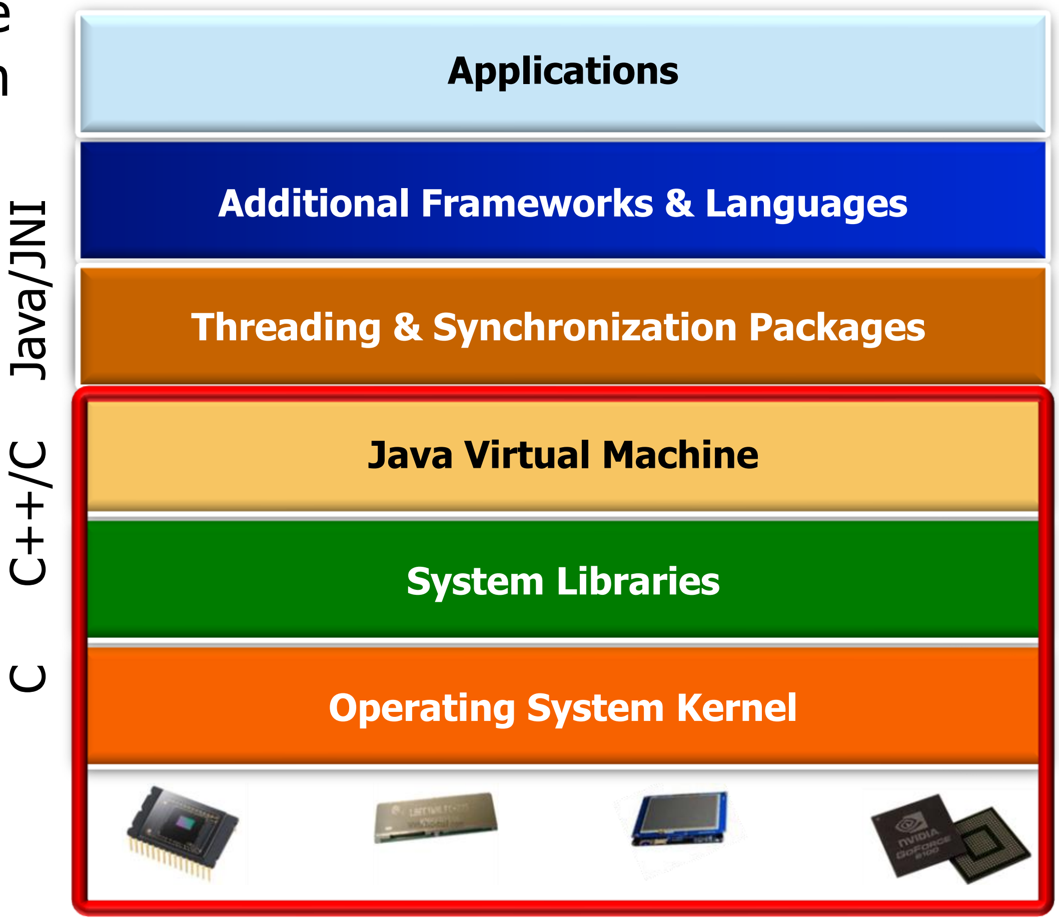
See docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html

Overview of Atomic Operations

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers





Applications

Additional Frameworks & Languages

Threading & Synchronization Packages

Java Virtual Machine

System Libraries

Operating System Kernel

See software.intel.com/en-us/node/506090

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

Compare-and-swap atomically compares the current contents of a memory location to a given value & iff they are the same it modifies the contents of that memory location to a given new value & returns the old value

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

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
    - CAS – “compare-and-swap”

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

Compare-and-swap atomically compares the current contents of a memory location to a given value & iff they are the same it modifies the contents of that memory location to a given new value & returns the old value

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

Compare-and-swap atomically compares the current contents of a memory location to a given value & iff they are the same it modifies the contents of that memory location to a given new value & returns the old value

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

Compare-and-swap atomically compares the current contents of a memory location to a given value & iff they are the same it modifies the contents of that memory location to a given new value & returns the old value

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

Compare-and-swap atomically compares the current contents of a memory location to a given value & iff they are the same it modifies the contents of that memory location to a given new value & returns the old value

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

```
void lock(int *mutex) {
  while (compareAndSwap(mutex, 0, 1) == 1)
    continue;
}
```

The lock() method uses compareAndSwap() to implement mutual exclusion (mutex) via a "spin-lock"

See en.wikipedia.org/wiki/Spinlock

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

```
void lock(int *mutex) {
  while (compareAndSwap(mutex, 0, 1) == 1)
    continue;
}
```

The lock() method uses compareAndSwap() to implement mutual exclusion (mutex) via a "spin-lock"

compareAndSwap() must be called only once per lock attempt

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

```
void lock(int *mutex) {
  while (compareAndSwap(mutex, 0, 1) == 1)
    continue;
}
```

compareAndSwap() checks if the location pointed to by mutex is 0 & iff that's true it sets the value to 1

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

```
void lock(int *mutex) {
  while (compareAndSwap(mutex, 0, 1) == 1)
    continue;
}
```

compareAndSwap() checks if the location pointed to by mutex is 0 & iff that's true it sets the value to 1

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

```
void lock(int *mutex) {
  while (compareAndSwap(mutex, 0, 1) == 1)
    continue;
}
```

compareAndSwap() checks if the location pointed to by mutex is 0 & iff that's true it sets the value to 1

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

```
void lock(int *mutex) {
  while (compareAndSwap(mutex, 0, 1) == 1)
    continue;
}
```

If compareAndSwap() returns 1 that means the mutex is "acquired" so the loop keeps spinning

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

```
void lock(int *mutex) {
  while (compareAndSwap(mutex, 0, 1) == 1)
    continue;
}

void unlock(int *mutex) {
  START_ATOMIC();
  *mutex = 0;
  END_ATOMIC();
}
```

The unlock() method atomically resets the mutex value to 0

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

```
void lock(int *mutex) {
  while (compareAndSwap(mutex, 0, 1) == 1)
    continue;
}

void unlock(int *mutex) {
  START_ATOMIC();
  *mutex = 0;
  END_ATOMIC();
}
```

The unlock() method atomically resets the mutex value to 0

Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  - CAS – "compare-and-swap"

```
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

```
void lock(int *mutex) {
  while (compareAndSwap(mutex, 0, 1) == 1)
    continue;
}

void unlock(int *mutex) {
  START_ATOMIC();
  *mutex = 0;
  END_ATOMIC();
}
```

The unlock() method atomically resets the mutex value to 0

Overview of Atomic Operations

- Atomic operations can be implemented other ways
  - e.g., "test-and-set"

```
int testAndSet(int *loc) {
  int oldValue;
  START_ATOMIC();
  oldValue = *loc;
  *loc = 1; // 1 == locked
  END_ATOMIC();
  return oldValue;
}
```

Test-and-set atomically modifies the contents of a memory location & returns its old value

See en.wikipedia.org/wiki/Test-and-set

Overview of Atomic Operations

- Atomic operations can be implemented other ways
  - e.g., “test-and-set”

```
int testAndSet(int *loc) {
  int oldValue;
  START_ATOMIC();
  oldValue = *loc;
  *loc = 1; // 1 == locked
  END_ATOMIC();
  return oldValue;
}
```

Test-and-set atomically modifies the contents of a memory location & returns its old value

Overview of Atomic Operations

- Atomic operations can be implemented other ways
  - e.g., "test-and-set"

```
int testAndSet(int *loc) {
   int oldValue;
   START_ATOMIC();
   oldValue = *loc;
   *loc = 1; // 1 == locked
   END_ATOMIC();
   return oldValue;
}
```

Test-and-set atomically modifies the contents of a memory location & returns its old value

Overview of Atomic Operations

- Atomic operations can be implemented other ways
  - e.g., "test-and-set"

```
int testAndSet(int *loc) {
  int oldValue;
  START_ATOMIC();
  oldValue = *loc;
  *loc = 1; // 1 == locked
  END_ATOMIC();
  return oldValue;
}
```

Test-and-set atomically modifies the contents of a memory location & returns its old value

Overview of Atomic Operations

- Atomic operations can be implemented other ways
  - e.g., "test-and-set"

```
int testAndSet(int *loc) {
  int oldValue;
  START_ATOMIC();
  oldValue = *loc;
  *loc = 1; // 1 == locked
  END_ATOMIC();
  return oldValue;
}
```

```
void lock(int *loc) {
  while (testAndSet(loc) == 1);
}

void unlock(int *loc) {
  START_ATOMIC();
  *loc = 0;
  END_ATOMIC();
}
```

Test-and-set can also be used to implement a spin-lock mutex

Overview of Atomic Operations

- compareAndSwap() provides a more general solution than testAndSet()

```
int testAndSet(int *loc) {
  int oldValue;
  START_ATOMIC();
  oldValue = *loc;
  *loc = 1; // 1 == locked
  END_ATOMIC();
  return oldValue;
}

int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

See pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf

Overview of Atomic Operations

- compareAndSwap() provides a more general solution than testAndSet()
  - e.g., it can set the value to something other than 1 or 0

```
int testAndSet(int *loc) {
  int oldValue;
  START_ATOMIC();
  oldValue = *loc;
  *loc = 1; // 1 == locked
  END_ATOMIC();
  return oldValue;
}

int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
  START_ATOMIC();
  int oldValue = *loc;
  if (oldValue == expected)
     *loc = updated;
  END_ATOMIC();
  return oldValue;
}
```

This capability is used by various Atomic* classes in Java

Human Known Use of Atomic Operations

Human Known Use of Atomic Operations

- One "human" known use of atomic operations is a Star Trek transporter

See en.wikipedia.org/wiki/Transporter_(Star_Trek)

Human Known Use of Atomic Operations

- One "human" known use of atomic operations is a Star Trek transporter
  - Converts a person/object into an energy pattern & "beams" them to a destination where they're converted back into matter

Human Known Use of Atomic Operations

- One "human" known use of atomic operations is a Star Trek transporter
  - Converts a person/object into an energy pattern & "beams" them to a destination where they're converted back into matter
  - This process must occur atomically or a horrible accident will occur!

See en.wikipedia.org/wiki/Transporter_(Star_Trek)#Transporter_accidents

Human Known Use of Atomic Operations

- Another “human” known use of atomic operations is “apparition” in Harry Potter

See harrypotter.fandom.com/wiki/Apparition

Human Known Use of Atomic Operations

- Another "human" known use of atomic operations is "apparition" in Harry Potter
  - If the user focuses properly they disappear from their current location & instantly reappear at the desired location

See harrypotter.fandom.com/wiki/Apparition

Human Known Use of Atomic Operations

- Another "human" known use of atomic operations is "apparition" in Harry Potter
  - If the user focuses properly they disappear from their current location & instantly reappear at the desired location
  - However, "spinching" occurs if a wizard or witch fails to apparate atomically!

See harrypotter.fandom.com/wiki/Splinching

End of Atomic Classes & Operations: Introduction