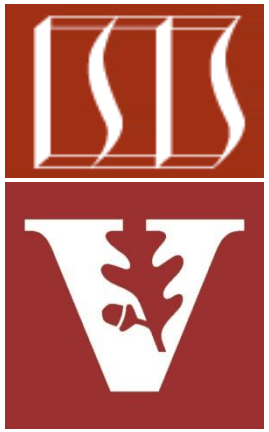
Overview of Atomic Operations



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

- Understand what atomic operations are

Learning Objectives in this Lesson

- Understand what atomic operations are
- Recognize key concepts associated with atomic operations in Java

Overview of Atomic Operations

Overview of Atomic Operations

- Atomic operations ensure 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 Operations

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

See en.wikipedia.org/wiki/Linearizability

Overview of Atomic Operations

- Atomic operations ensure changes to a field are always consistent & visible to other threads
  - An *atomic* operation 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 Operations

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

Key Concepts Related to Java Atomic Operations

Key Concepts Related to Java Atomic Operations

- Three key concepts are associated with atomic operations in Java

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

Key Concepts Related to Java Atomic Operations

- Three key concepts are associated with atomic operations in Java
  - *Atomicity* deals w/which operations have indivisible effects

```
class NonAtomicOps {
  long mCounter = 0;

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

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

  ...
}
```

Key Concepts Related to Java Atomic Operations

- Three key concepts are associated with atomic operations in Java
  - *Atomicity* deals w/which operations have indivisible effects

```
class NonAtomicOps {
  long mCounter = 0;

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

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

  ...
}
```

Mutable shared state

Key Concepts Related to Java Atomic Operations

- Three key concepts are associated with atomic operations in Java
  - *Atomicity* deals w/which operations have indivisible effects

```
class NonAtomicOps {
  long mCounter = 0;

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

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

  ...
}
```

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

Key Concepts Related to Java Atomic Operations

- Three key concepts are associated with atomic operations in Java
  - *Atomicity* deals w/which operations have indivisible effects
  - *Visibility* determines when a thread can see 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;
  }

  ...
}
```

Key Concepts Related to Java Atomic Operations

- Three key concepts are associated with atomic operations in Java
  - *Atomicity* deals w/which operations have indivisible effects
  - *Visibility* determines when a thread can see 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;
  }

  ...
}
```

Mutable shared state

Key Concepts Related to Java Atomic Operations

- Three key concepts are associated with atomic operations in Java
  - *Atomicity* deals w/which operations have indivisible effects
  - *Visibility* determines when a thread can see 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;
  }

  ...
}
```

Thread T_2 may never stop, even after Thread T_1 sets mDone to true..

Key Concepts Related to Java Atomic Operations

- Three key concepts are associated with atomic operations in Java
  - *Atomicity* deals w/which operations have indivisible effects
  - *Visibility* determines when a thread can see the effects of another
  - *Ordering* determines when the operations in one thread occur out of order wrt to other thread(s)


```
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
  }
}
```

Key Concepts Related to Java Atomic Operations

- Three key concepts are associated with atomic operations in Java
  - *Atomicity* deals w/which operations have indivisible effects
  - *Visibility* determines when a thread can see the effects of another
  - *Ordering* determines when the operations in one thread occur out of order wrt to other thread(s)


```
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
  }
}
```

Mutable shared state

Key Concepts Related to Java Atomic Operations

- Three key concepts are associated with atomic operations in Java
  - *Atomicity* deals w/which operations have indivisible effects
  - *Visibility* determines when a thread can see the effects of another
  - *Ordering* determines when the operations in one thread occur out of order wrt to other thread(s)


```
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
  }
}
```

Fields a & b may appear in thread T_2 in an order different than set in thread T_1!

End of Overview of Java Atomic Operations