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

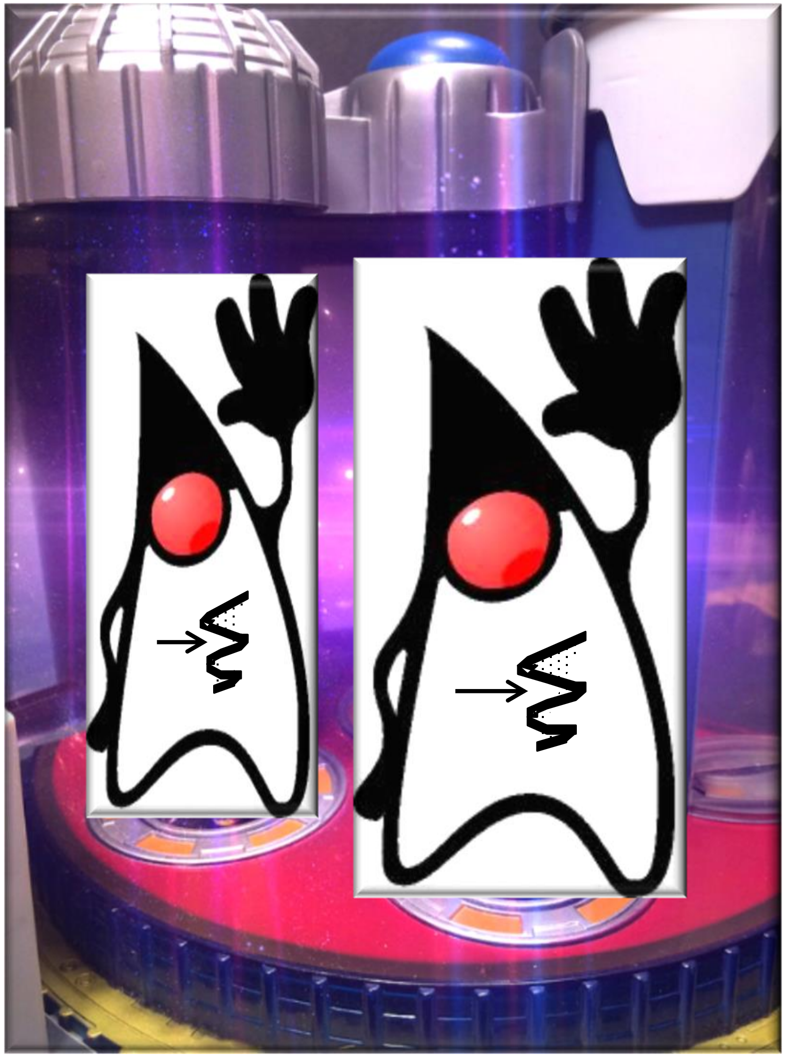
Learning Objectives in this Lesson

- Recognize Java programming language & library features that provide atomic operations & variables

Overview of Java Atomic Operations & Variables

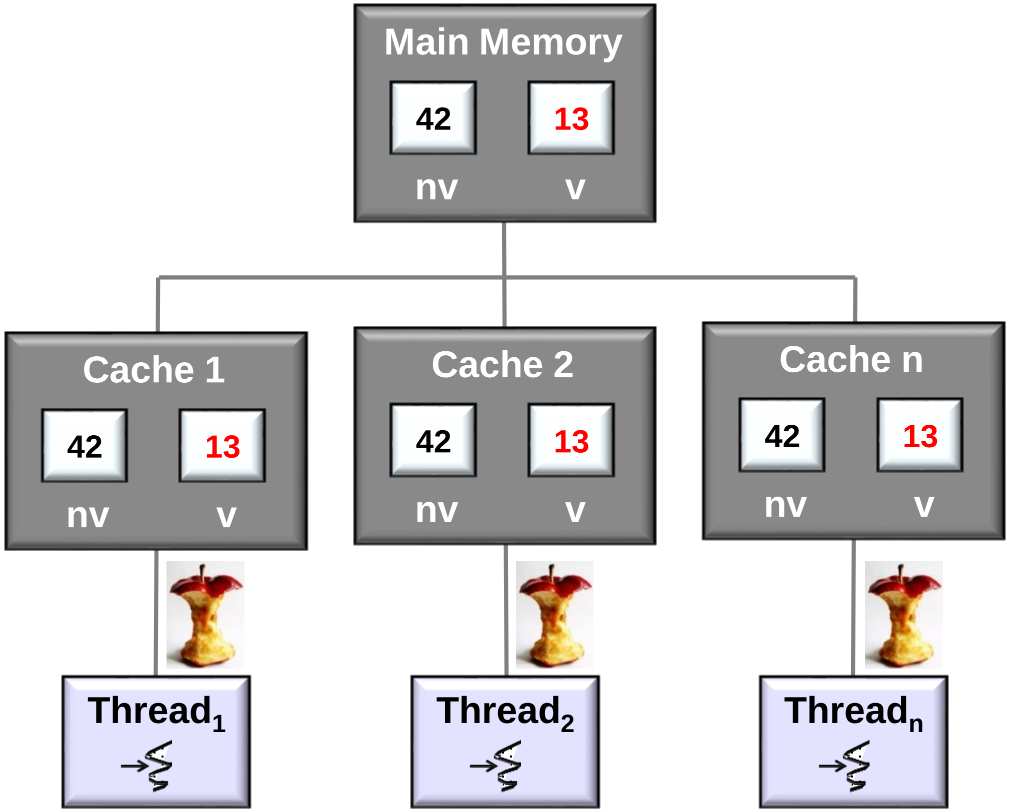
Overview of Java Atomic Operations & Variables

- Java supports several types of atomicity



Overview of Java Atomic Operations & Variables

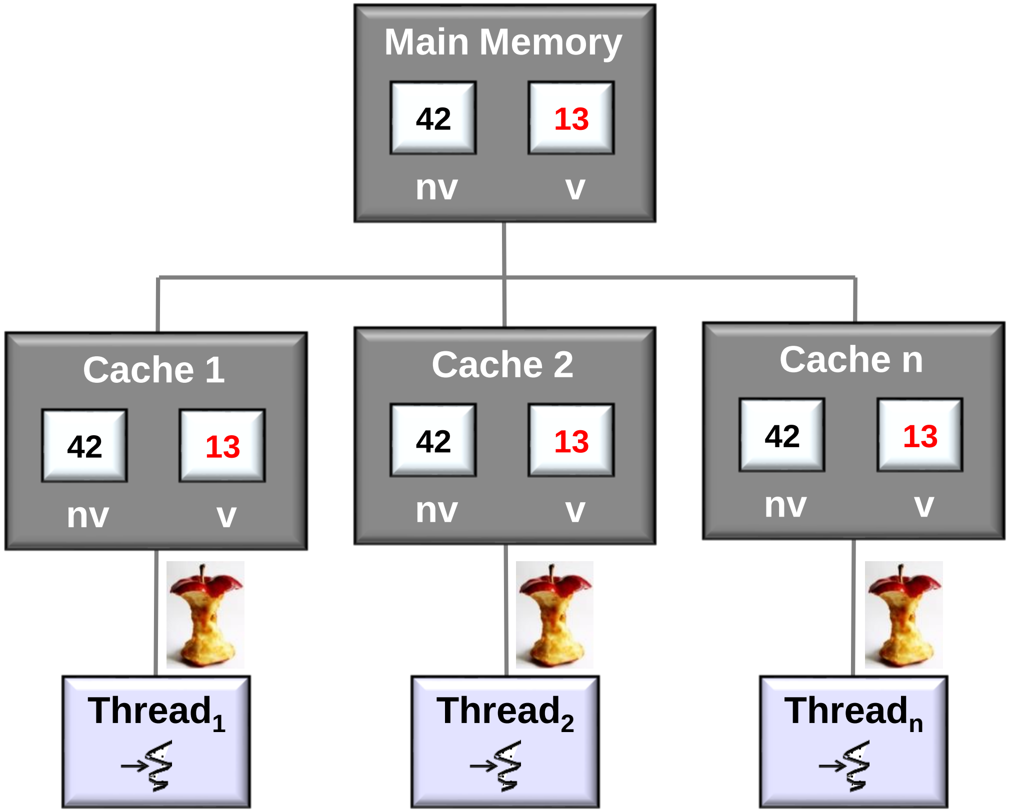
- Java supports several types of atomicity, e.g.
  - *Volatile variables*



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

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

```
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();
    ...
```

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

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

```
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

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

```
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();
    ...
```

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

```
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();
    ...
```

These reads from **val** *are atomic*

Overview of Java Atomic Operations & Variables

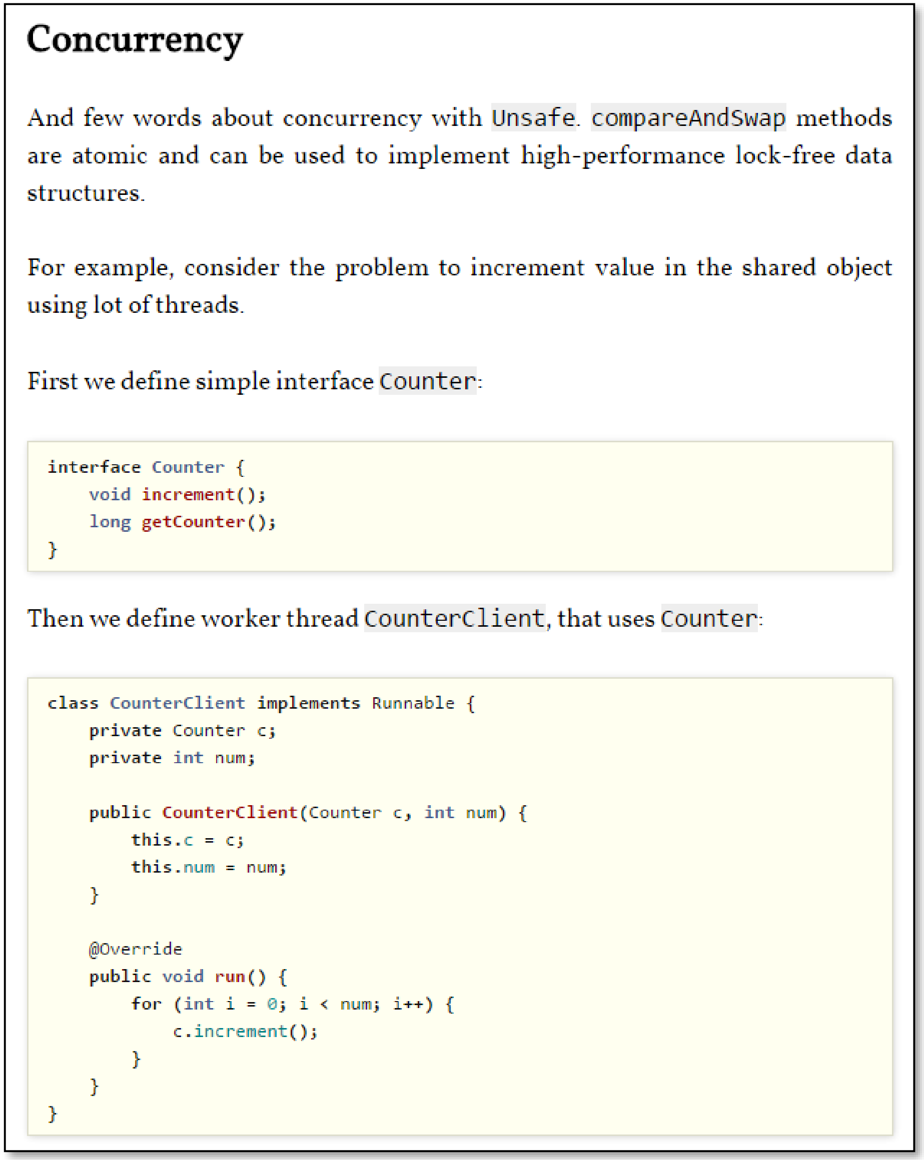
- Java supports several types of atomicity, e.g.
  - *Volatile variables*
    - Ensure a variable is read from & written to main memory & not cached
      - e.g., sharing a field between two threads

```
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();
    ...
```

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 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 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 in 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`:

```
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 atomicity, 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 app 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 atomicity, 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 app 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;
}
```

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 in 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

```
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 atomicity, 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 app 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);
}
```

Uses CAS to implement a simple "mutex" spin-lock

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

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 in 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

Overview of Java Atomic Operations & Variables

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

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 designated classes.
AtomicMarkableReference<V>	An `AtomicMarkableReference` maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may

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 in the Java Unsafe class*
  - *Atomic classes*
    - Use Java Unsafe 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

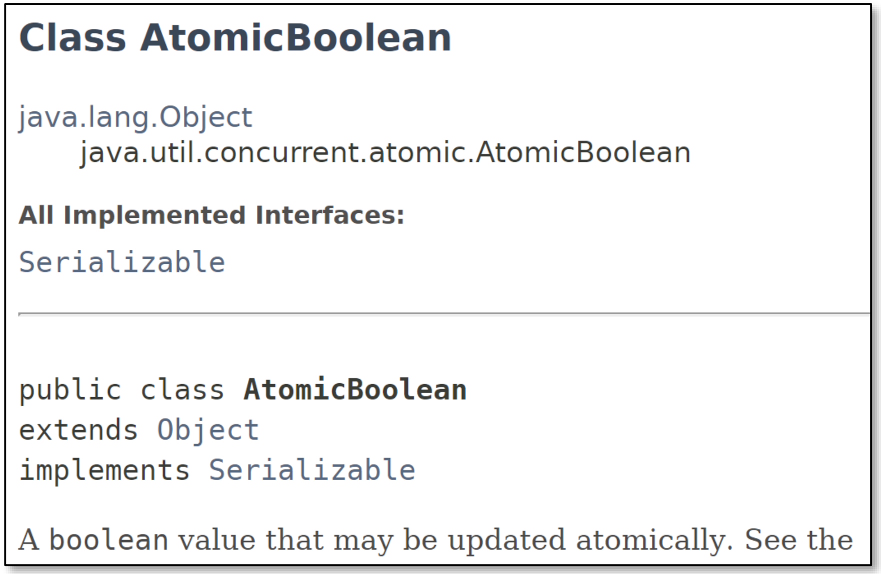
Class Summary

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 designated classes.
AtomicMarkableReference<V>	An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may

See 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 in the Java Unsafe class*
  - *Atomic classes*
    - Use Java Unsafe internally to implement "lock-free" methods
      - e.g., AtomicLong & AtomicBoolean


Class AtomicBoolean

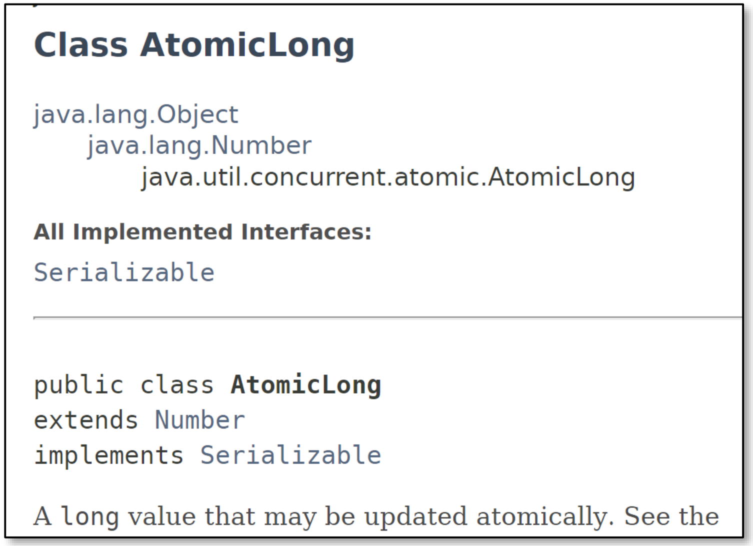
java.lang.Object
    java.util.concurrent.atomic.AtomicBoolean

All Implemented Interfaces:

Serializable

```
public class AtomicBoolean
extends Object
implements Serializable
```

A boolean value that may be updated atomically. See the


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

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