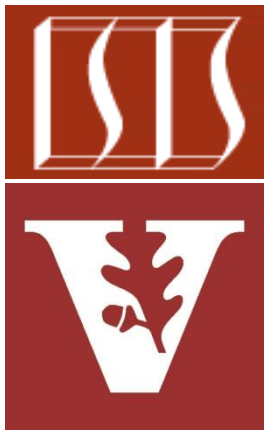
Java Atomic Classes & Operations: Usage Considerations



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
- Note a human known use of atomic operations
- Know how Java atomic operations are implemented
- Recognize how the Java AtomicLong & AtomicBoolean classes are implemented
- Be aware of how to apply Java AtomicLong in practice
- Appreciate Java atomic class & operation usage considerations

Usage Considerations for Java Atomic Operations

Usage Considerations for Java Atomic Operations

- Programs should use atomic operations carefully since they "busy wait"

See en.wikipedia.org/wiki/Busy_waiting

Usage Considerations for Java Atomic Operations

- Programs should use atomic operations carefully since they "busy wait"
  - Busy waiting needlessly wastes CPU cycles if contention is high

See www.ibm.com/support/knowledgecenter/en/SS3KLZ/com.ibm.java.diagnostics.healthcenter.doc/topics/resolving.html

Usage Considerations for Java Atomic Operations

- Programs should use atomic operations carefully since they "busy wait"
  - Busy waiting needlessly wastes CPU cycles if contention is high
  - However, some "spinning" is useful in multi-core processors

See www.youtube.com/watch?v=sq0MX3fHkro

Usage Considerations for Java Atomic Operations

- Programs should use atomic operations carefully since they "busy wait"
  - Busy waiting needlessly wastes CPU cycles if contention is high
  - However, some "spinning" is useful in multi-core processors
    - e.g., due to context switching overhead of sleep locks

See www.youtube.com/watch?v=sq0MX3fHkro

Usage Considerations for Java Atomic Operations

- The compareAndSet*() methods in the various Java Atomic* classes provide a portable means of accessing low-level CAS operations

compareAndSet

```
public final boolean compareAndSet(boolean expect,
                                   boolean update)
```

Atomically sets the value to the given updated value if the current value == the expected value.

Parameters:

expect - the expected value

update - the new value

Returns:

true if successful. False return indicates that the actual value was not equal to the expected value.

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

Usage Considerations for Java Atomic Operations

- The compareAndSet*() methods in the various Java Atomic* classes provide a portable means of accessing low-level CAS operations
  - Keep in mind that these methods are intended for very specific use cases

```
class Random ... {
  public Random()
  { this(seedUniquifier() ^ System.nanoTime()); }

  private static long seedUniquifier(){
    for (;;) {
      long s = seedUniquifier.get();
      long next = s * 1181783497276652981L;
      if (seedUniquifier.compareAndSet(s, next))
        return next;
    }
  }

  private static final AtomicLong seedUniquifier =
    new AtomicLong(8682522807148012L);
```

Try to set the computed next seed atomically, which will succeed only if **`s`** *is still the current seed value*

compareAndSet() is only called once per loop, per thread & only succeeds in one thread

End of Atomic Classes & Operations: Usage Considerations