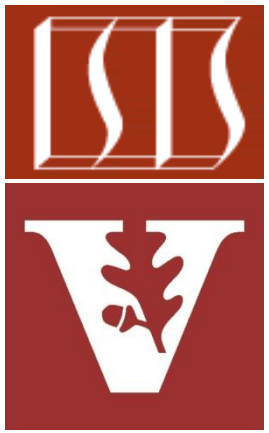
Java StampedLock: Usage Considerations



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 the structure, functionality of the Java StampedLock class
- Know the key methods in Java StampedLock
- Recognize how to apply Java StampedLock in practice
- Appreciate Java StampedLock usage considerations

We'll also compare/contrast StampedLock with other Java synchronizers

Java StampedLock Usage Considerations

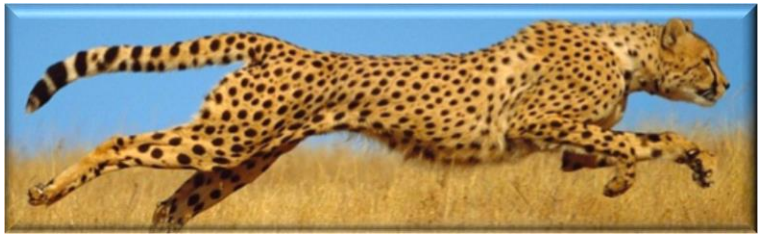
Java StampedLock Usage Conventions

- StampedLock often *much* faster than ReentrantReadWriteLock

SYNCHRONIZED	OPTIMISTIC	RWLOCK	STAMPED
1996.6	1174	116393	64077
2312.7	1174	116617	47897
2100.9	1122	117746	65921
2285.1	1182.9	115605	73500
2173.6	1184.9	118346	32857
2173.78	**1167.56**	**116941.4**	**56850.4**

19 readers & 1 writer

Optimistic read mode works very well with little/no contention



See www.takipiblog.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized

Java StampedLock Usage Conventions

- StampedLock often *much* faster than ReentrantReadWriteLock

SYNCHRONIZED	OPTIMISTIC	RWLOCK	STAMPED
1996.6	1174	116393	64077
2312.7	1174	116617	47897
2100.9	1122	117746	65921
2285.1	1182.9	115605	73500
2173.6	1184.9	118346	32857
2173.78	**1167.56**	**116941.4**	**56850.4**

19 readers & 1 writer

ReentrantReadWriteLock is very slow..

See www.takipiblog.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized

Java StampedLock Usage Conventions

- StampedLock often *much* faster than ReentrantReadWriteLock

SYNCHRONIZED	OPTIMISTIC	RWLOCK	STAMPED
1996.6	1174	116393	64077
2312.7	1174	116617	47897
2100.9	1122	117746	65921
2285.1	1182.9	115605	73500
2173.6	1184.9	118346	32857
2173.78	**1167.56**	**116941.4**	**56850.4**

19 readers & 1 writer

StampedLock with "reading mode" works better than ReentrantReadWriteLock

See www.takipiblog.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized

Java StampedLock Usage Conventions

- StampedLock often *much* faster than ReentrantReadWriteLock

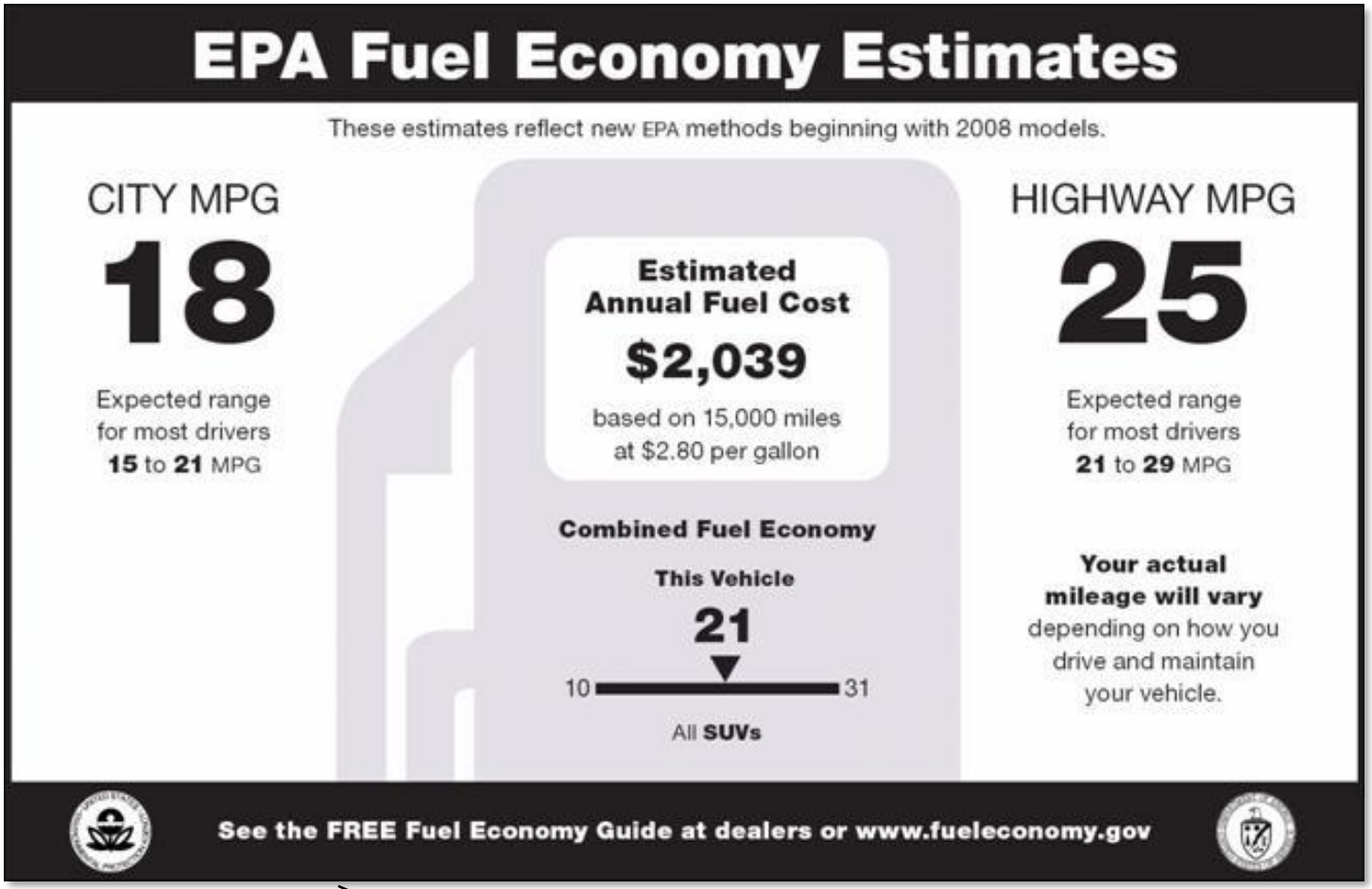
SYNCHRONIZED	OPTIMISTIC	RWLOCK	STAMPED
1996.6	1174	116393	64077
2312.7	1174	116617	47897
2100.9	1122	117746	65921
2285.1	1182.9	115605	73500
2173.6	1184.9	118346	32857
2173.78	**1167.56**	**116941.4**	**56850.4**

Synchronized statements perform quite well

See www.takipiblog.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized

Java StampedLock Usage Conventions

- StampedLock often *much* faster than ReentrantReadWriteLock



However, your mileage may vary!

See en.wiktionary.org/wiki/your_mileage_may_vary

Java StampedLock Usage Conventions

- StampedLock often *much* faster than ReentrantReadWriteLock

RWLOCK	STAMPED	SYNCHRONIZED	OPTIMISTIC
1960.8	165.1	177.4	387.9
1473.6	111.3	192.1	382.8
2119.7	216.8	173.3	403.6
2772.2	221.9	205.4	403.9
2721.4	189.3	181.2	394.2
2209.54	**180.88**	**185.88**	**394.48**

10 readers & 10 writers

Optimistic read mode works less well with more contention

See www.takipiblog.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized

Java StampedLock Usage Conventions

- StampedLock often *much* faster than ReentrantReadWriteLock

RWLOCK	STAMPED	SYNCHRONIZED	OPTIMISTIC
1960.8	165.1	177.4	387.9
1473.6	111.3	192.1	382.8
2119.7	216.8	173.3	403.6
2772.2	221.9	205.4	403.9
2721.4	189.3	181.2	394.2
2209.54	**180.88**	**185.88**	**394.48**

10 readers & 10 writers

However, ReentrantReadWriteLock is still much slower..

See www.takipiblog.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized

Java StampedLock Usage Conventions

- StampedLock often *much* faster than ReentrantReadWriteLock

RWLOCK	STAMPED	SYNCHRONIZED	OPTIMISTIC
1960.8	165.1	177.4	387.9
1473.6	111.3	192.1	382.8
2119.7	216.8	173.3	403.6
2772.2	221.9	205.4	403.9
2721.4	189.3	181.2	394.2
2209.54	**180.88**	**185.88**	**394.48**

10 readers & 10 writers

StampedLock & synchronized statements both do quite well

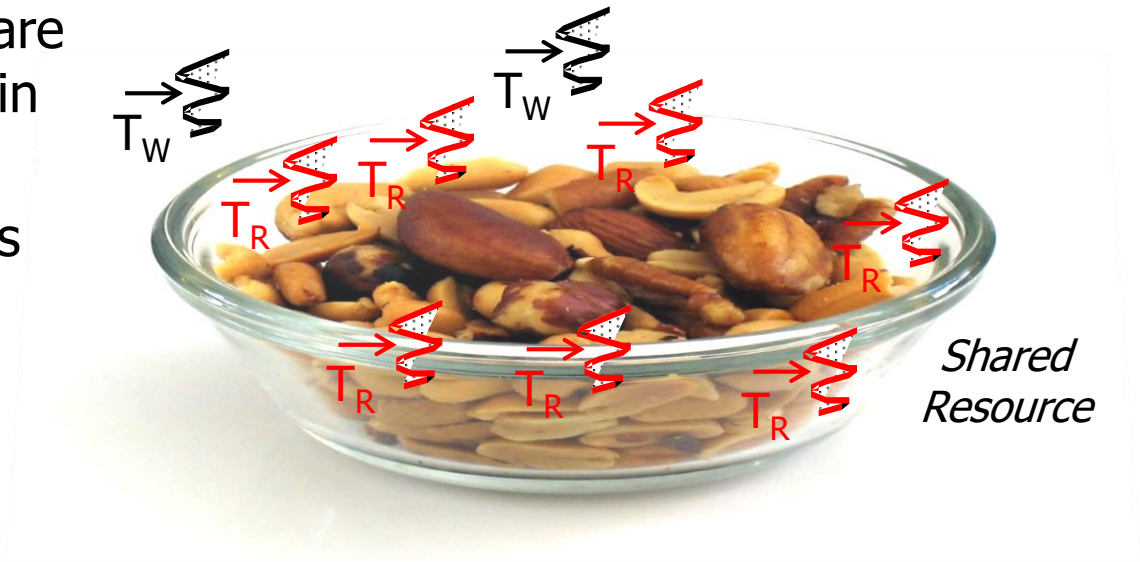
See www.takipiblog.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized

Java StampedLock Usage Conventions

- Java StampedLock speedups are only fully realized under certain conditions

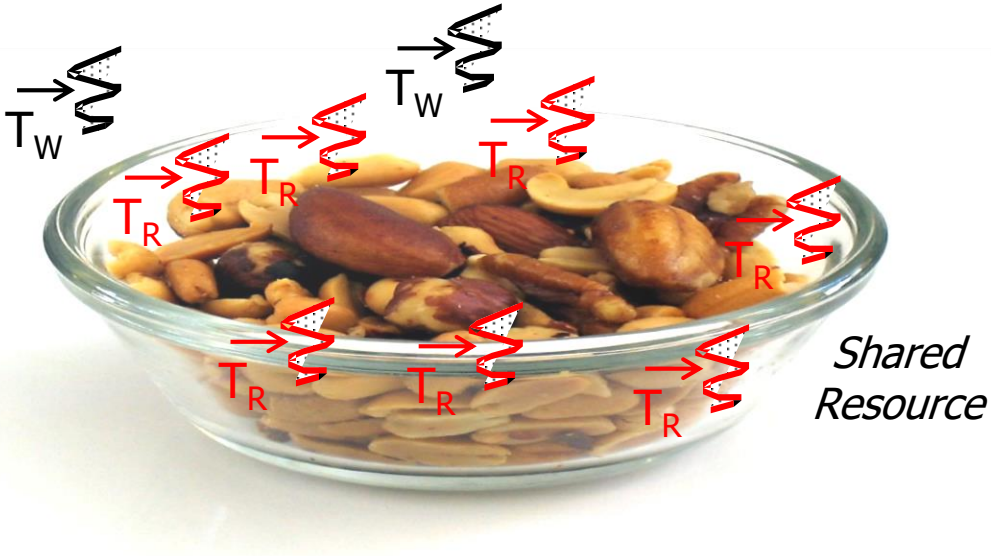
Java StampedLock Usage Conventions

- Java StampedLock speedups are only fully realized under certain conditions, e.g.
  - Frequency of reads to writes
    - Ideally, *many* more reads than writes



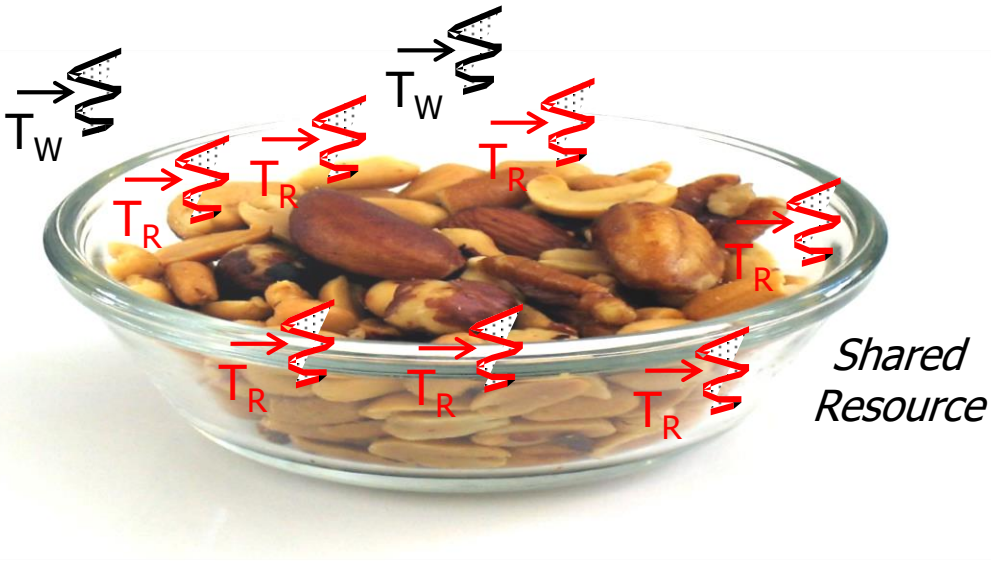
Java StampedLock Usage Conventions

- Java StampedLock speedups are only fully realized under certain conditions, e.g.
  - Frequency of reads to writes
  - Duration of read & write operations
    - Ideally, read operations should be non-trivial or else locking costs may dominate



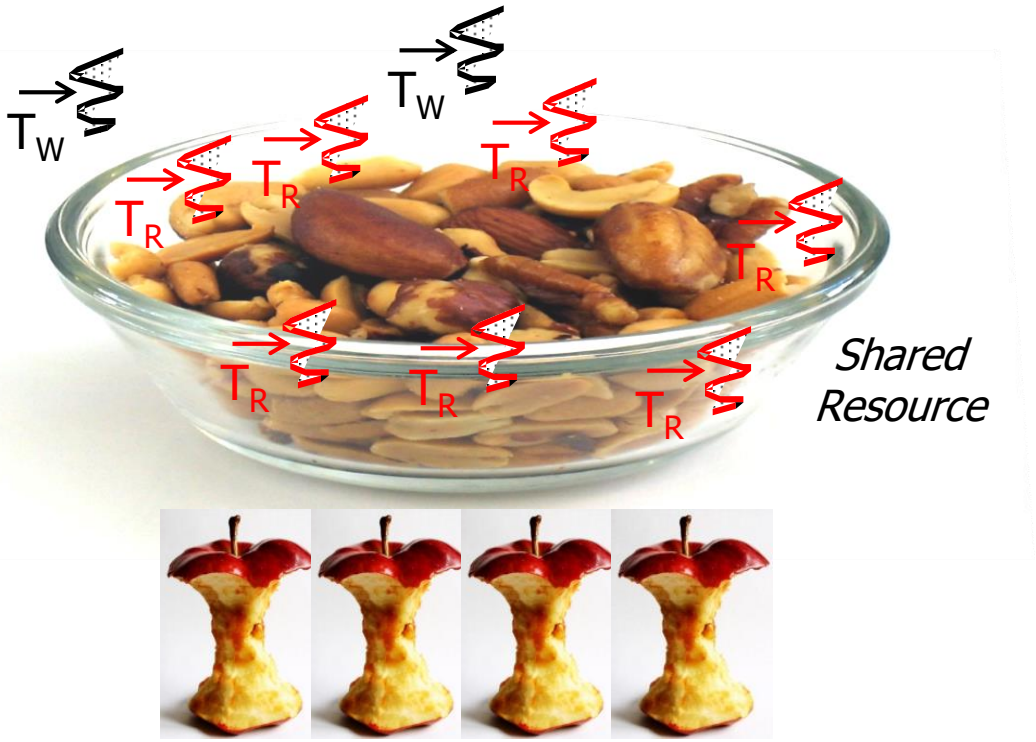
Java StampedLock Usage Conventions

- Java StampedLock speedups are only fully realized under certain conditions, e.g.
  - Frequency of reads to writes
  - Duration of read & write operations
  - "Contention" for the data
    - Ideally, *many* concurrent readers



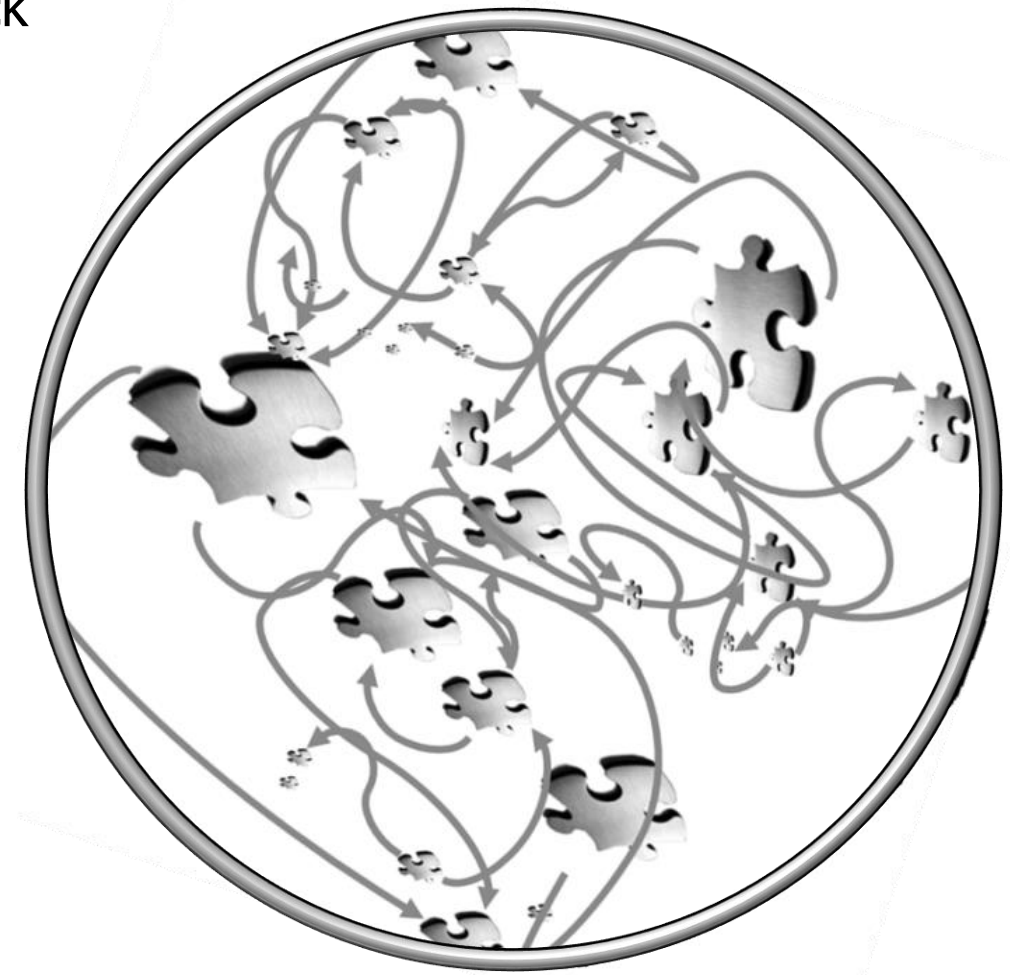
Java StampedLock Usage Conventions

- Java StampedLock speedups are only fully realized under certain conditions, e.g.
  - Frequency of reads to writes
  - Duration of read & write operations
  - "Contention" for the data
  - Number of processor cores
    - Ideally, *many* cores



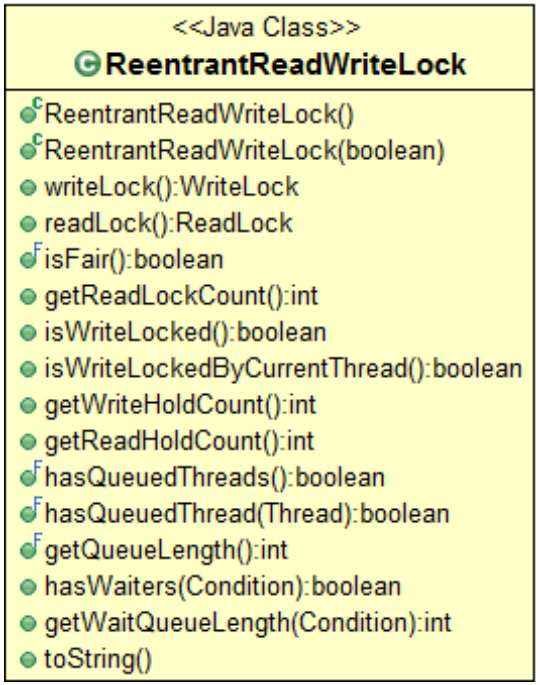
Java StampedLock Usage Conventions

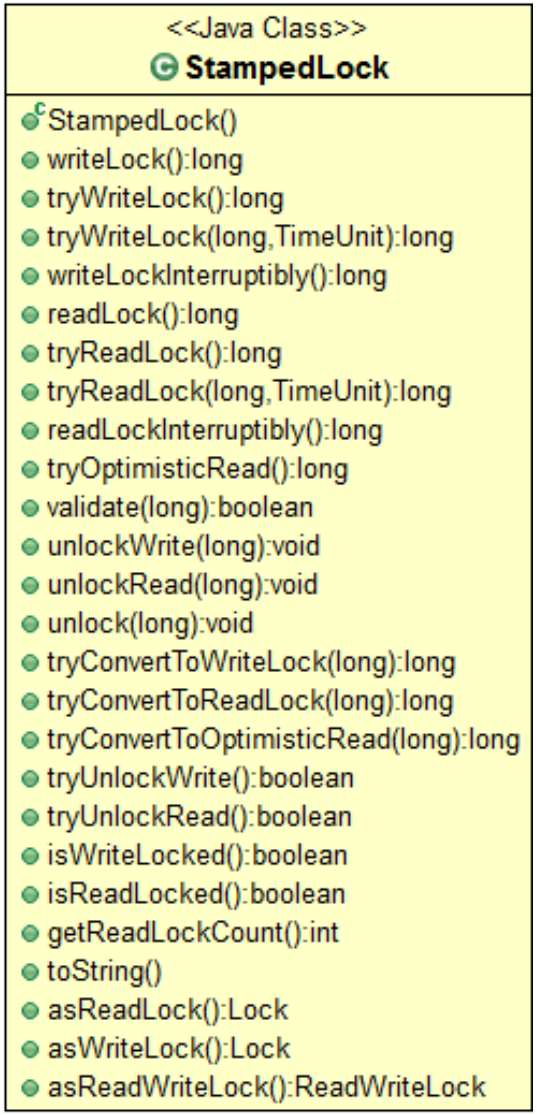
- StampedLock can be harder to use than ReentrantReadWriteLock



Java StampedLock Usage Conventions

- StampedLock can be harder to use than ReentrantReadWriteLock
  - Many more methods





Java StampedLock Usage Conventions

- StampedLock can be harder to use than ReentrantReadWriteLock
  - Many more methods
  - More intricate semantics & usage patterns

```
void moveIfAtOrigin(double newX,
                    double newY) {
  long stamp = sl.readLock();
    try {
      while (x == 0.0 && y == 0.0) {
        long ws =
          sl.tryConvertToWriteLock
              (stamp);
        if (ws != 0L) {
          stamp = ws;
          x = newX; y = newY;
          break;
        } else {
           sl.unlockRead(stamp);
           stamp = sl.writeLock();
         }
      }
    } finally
    { sl.unlock(stamp); }
  ...
```

Conditional writes & lock upgrades are tricky to program

See www.techevents.online/using-java-8-lambdas-stampedlock-manage-thread-safety

Java StampedLock Usage Conventions

- StampedLock can be harder to use than ReentrantReadWriteLock
  - Many more methods
  - More intricate semantics & usage patterns
  - Invariants are tricky with optimistic read locks

```
class Boooom {
  StampedLock mS =
    new StampedLock();
  int mX = 0;
  int mY = 1;
  ...

  // Thread T1
  while (true) {
    mS.writeLock();
    mX++; mY++;
    mS.writeUnlock();
  }

  // Thread T2
  do {
    stamp = mS.tryOptimisticRead();
    z = 1 / (mX - mY);
  } while (mS.validate(stamp));
```

See concurrencyfreaks.blogspot.com/2013/11/stampedlocktryoptimisticread-and.html

Java StampedLock Usage Conventions

- StampedLock can be harder to use than ReentrantReadWriteLock
  - Many more methods
  - More intricate semantics & usage patterns
  - Invariants are tricky with optimistic read locks

```
class Boooom {
  StampedLock mS =
    new StampedLock();
  int mX = 0;
  int mY = 1;
  ...

  // Thread T1
  while (true) {
    mS.writeLock();
    mX++; mY++;
    mS.writeUnlock();
  }

  // Thread T2
  do {
    stamp = mS.tryOptimisticRead();
    z = 1 / (mX - mY);
  } while (mS.validate(stamp));
```

Create a StampedLock to protect two fields

Java StampedLock Usage Conventions

- StampedLock can be harder to use than ReentrantReadWriteLock
  - Many more methods
  - More intricate semantics & usage patterns
  - Invariants are tricky with optimistic read locks
    - Fields read in optimistic mode may be inconsistent since their values can change unpredictably

```
class Boooom {
  StampedLock mS =
    new StampedLock();
  int mX = 0;
  int mY = 1;
  ...

  // Thread T1
  while (true) {
    mS.writeLock();
    mX++; mY++;
    mS.writeUnlock();
  }

  // Thread T2
  do {
    stamp = mS.tryOptimisticRead();
    z = 1 / (mX - mY);
  } while (mS.validate(stamp));
```

Want to establish the invariant mX == mY – 1

Java StampedLock Usage Conventions

- StampedLock can be harder to use than ReentrantReadWriteLock
  - Many more methods
  - More intricate semantics & usage patterns
  - Invariants are tricky with optimistic read locks
    - Fields read in optimistic mode may be inconsistent since their values can change unpredictably

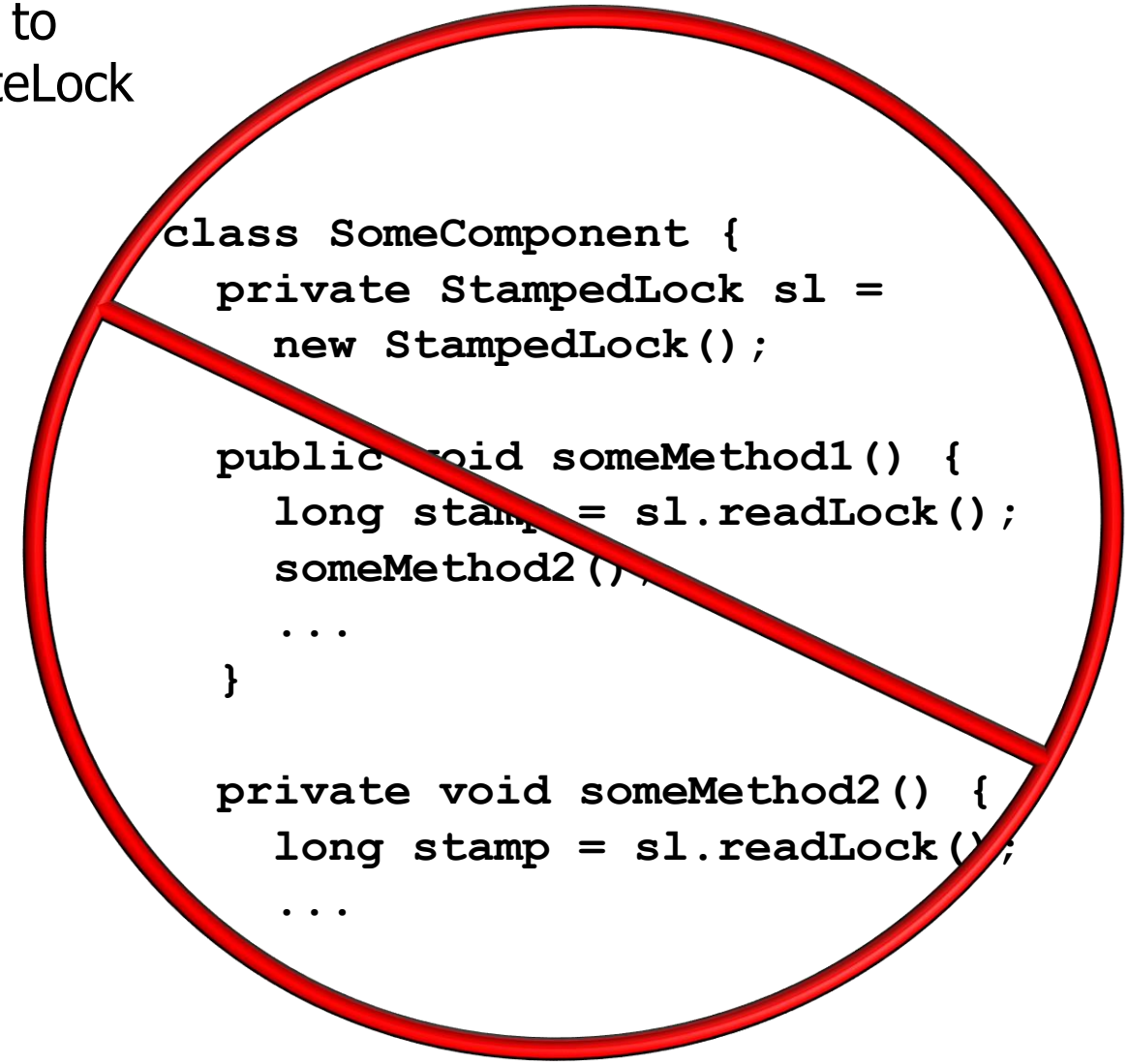
```
class Boooom {
  StampedLock mS =
    new StampedLock();
  int mX = 0;
  int mY = 1;
  ...

  // Thread T1
  while (true) {
    mS.writeLock();
    mX++; mY++;
    mS.writeUnlock();
  }

  // Thread T2
  do {
    stamp = mS.tryOptimisticRead();
    z = 1 / (mX - mY);
  } while (mS.validate(stamp));
```

Since no read lock is held, mX & mY may be reordered, such that invariant mX == mY – 1 may not hold

See concurrencyfreaks.blogspot.com/2013/11/stampedlocktryoptimisticread-and.html

Java StampedLock Usage Conventions

- StampedLock can be harder to use than ReentrantReadWriteLock
  - Many more methods
  - More intricate semantics & usage patterns
  - Invariants are tricky with optimistic read locks
  - Non-reentrant



Java StampedLock Usage Conventions

- StampedLock is usually the best choice for readers-writer locks in Java 8+!
  - Despite its complexity & lack of reentrant semantics

Class StampedLock

java.lang.Object
    java.util.concurrent.locks.StampedLock

All Implemented Interfaces:

`Serializable`

```
public class StampedLock
extends Object
implements Serializable
```

A capability-based lock with three modes for controlling read/write access. The state of a StampedLock consists of a version and mode. Lock acquisition methods return a stamp that represents and controls access with respect to a lock state; "try" versions of these methods may instead return the special value zero to represent failure to acquire access. Lock release and conversion methods require stamps as arguments, and fail if they do not match the state of the lock. The three modes are:

- **Writing.** Method `writeLock()` possibly blocks waiting for exclusive access, returning a stamp that can be used in method `unlockWrite(long)` to release the lock. Untimed and timed versions of `tryWriteLock` are also provided. When the lock is held in write mode, no read locks may be obtained, and all optimistic read validations will fail.
- **Reading.** Method `readLock()` possibly blocks waiting for non-exclusive access, returning a stamp that can be used in method `unlockRead(long)` to release the lock. Untimed and timed versions of `tryReadLock` are also provided.
- **Optimistic Reading.** Method `tryOptimisticRead()` returns a non-zero stamp only if the lock is not currently held in write mode. Method `validate(long)` returns true if the lock has not been acquired in write mode since obtaining a given stamp. This mode can be thought of as an extremely weak version of a read-lock, that can be broken by a writer at any time. The use of optimistic mode for short read-only code segments

See www.javaspecialists.eu/archive/Issue215.html

End of Java StampedLock: Usage Considerations