Overview of Java Concurrency Hazards

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand the meaning of key concurrent programming concepts
- Recognize how Java supports concurrent programming concepts
- Be aware of common concurrency hazards faced by Java programmers

Learning Objectives in this Part of the Lesson

- Understand the meaning of key concurrent programming concepts
- Recognize how Java supports concurrent programming concepts
- Be aware of common concurrency hazards faced by Java programmers
  - Including hazards stemming from synchronizers themselves!

Common Concurrent Programming Hazards

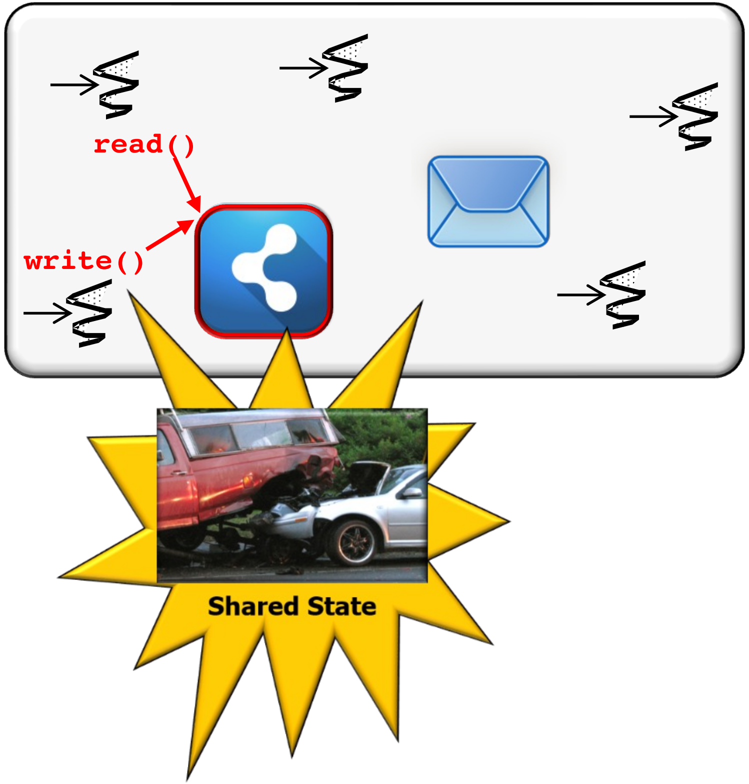
Common Concurrent Programming Hazards & Solutions

- Java shared objects & message passing mechanisms help share resources safely & avoid concurrency hazards, e.g.
  - Race conditions
  - Memory inconsistencies

See en.wikipedia.org/wiki/Thread_safety

Common Concurrent Programming Hazards & Solutions

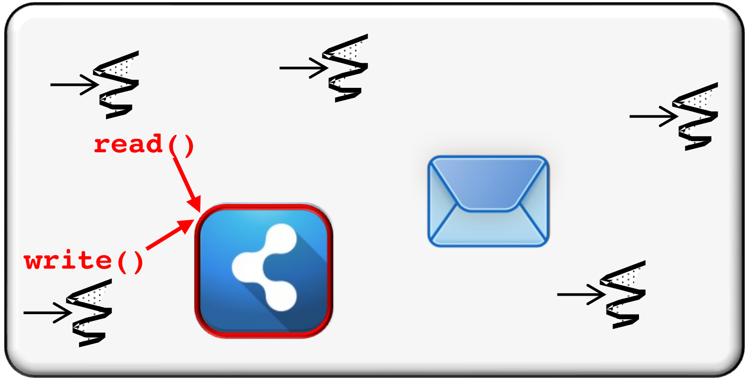
- Race conditions
  - Occur when a program depends on the sequence or timing of threads to operate properly



See en.wikipedia.org/wiki/Race_condition#Software

Common Concurrent Programming Hazards & Solutions

- Race conditions
  - Occur when a program depends on the sequence or timing of threads to operate properly



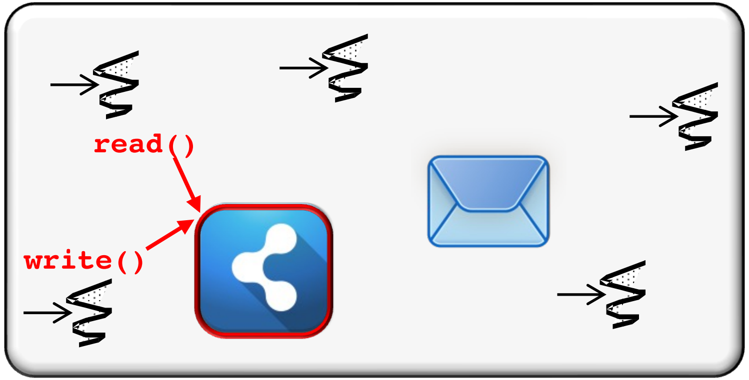

```
class BuggyQueue<E> {
  List<E> l = new ArrayList<>();
  public void offer(E e) {
    if (!isFull())
    { l.add(e); return true; }
    else return false;
  }
  public E poll() {
    return !isEmpty() ? l.remove(0) : null;
  } ...
```

This program induces race conditions between producer & consumer threads accessing an unsynchronized bounded queue

See github.com/douglascraigschmidt/LiveLessons/tree/master/BuggyQueue

Common Concurrent Programming Hazards & Solutions

- Race conditions
  - Occur when a program depends on the sequence or timing of threads to operate properly



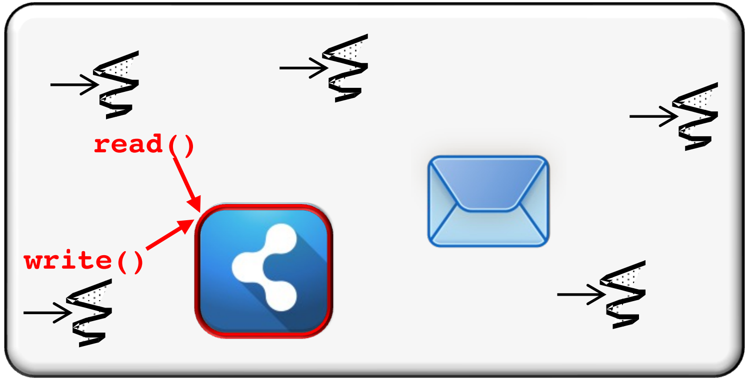

```
class BuggyQueue<E> {
  List<E> l = new ArrayList<>();
  public void offer(E e) {
    if (!isFull())
    { l.add(e); return true; }
    else return false;
  }
  public E poll() {
    return !isEmpty() ? l.remove(0) : null;
  } ...
```

Chaos & insanity may result if offer() & poll() are called concurrently!

See henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html

Common Concurrent Programming Hazards & Solutions

- Race conditions
  - Occur when a program depends on the sequence or timing of threads to operate properly




```
class BuggyQueue<E> {
  List<E> l = new ArrayList<>();
  public synchronized void offer(E e) {
    if (!isFull())
    { l.add(e); return true; }
    else return false;
  }
  public synchronized E poll() {
    return !isEmpty() ? l.remove(0) : null;
  } ...
```

Avoid via Java mutual exclusion mechanisms

e.g., synchronized statement/method, ReentrantLock, StampedLock, etc.

Common Concurrent Programming Hazards & Solutions

- Memory inconsistencies
  - Occur when different threads have inconsistent views of what should be the same data

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

Common Concurrent Programming Hazards & Solutions

- Memory inconsistencies
  - Occur when different threads have inconsistent views of what should be the same data

```
class LoopMayNeverEnd {
  boolean mDone;

  void work() {
    // Thread T2 read
    while (!mDone) {
      // do work
    }
  }

  void stopWork() {
    mDone = true;
    // Thread T1 write
  }
  ...
```

Common Concurrent Programming Hazards & Solutions

- Memory inconsistencies
  - Occur when different threads have inconsistent views of what should be the same data

Unsynchronized & mutable shared data (boolean fields are initialized to false by default)

```
class LoopMayNeverEnd {
  boolean mDone;

  void work() {
    // Thread T2 read
    while (!mDone) {
      // do work
    }
  }

  void stopWork() {
    mDone = true;
    // Thread T1 write
  }
  ...
```

See howtodoinjava.com/java/keywords/java-boolean

Common Concurrent Programming Hazards & Solutions

- Memory inconsistencies
  - Occur when different threads have inconsistent views of what should be the same data

```
class LoopMayNeverEnd {
  boolean mDone;

  void work() {
    // Thread T2 read
    while (!mDone) {
      // do work
    }
  }

  void stopWork() {
    mDone = true;
    // Thread T1 write
  }
  ...
```

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

Common Concurrent Programming Hazards & Solutions

- Memory inconsistencies
  - Occur when different threads have inconsistent views of what should be the same data

Avoid via Java mechanisms that ensure atomic operations

```
class LoopMayNeverEnd {
  volatile boolean mDone;

  void work() {
    // Thread T2 read
    while (!mDone) {
      // do work
    }
  }

  void stopWork() {
    mDone = true;
    // Thread T1 write
  }
  ...
```

e.g., volatile, AtomicBoolean, AtomicInteger, AtomicLock, etc.

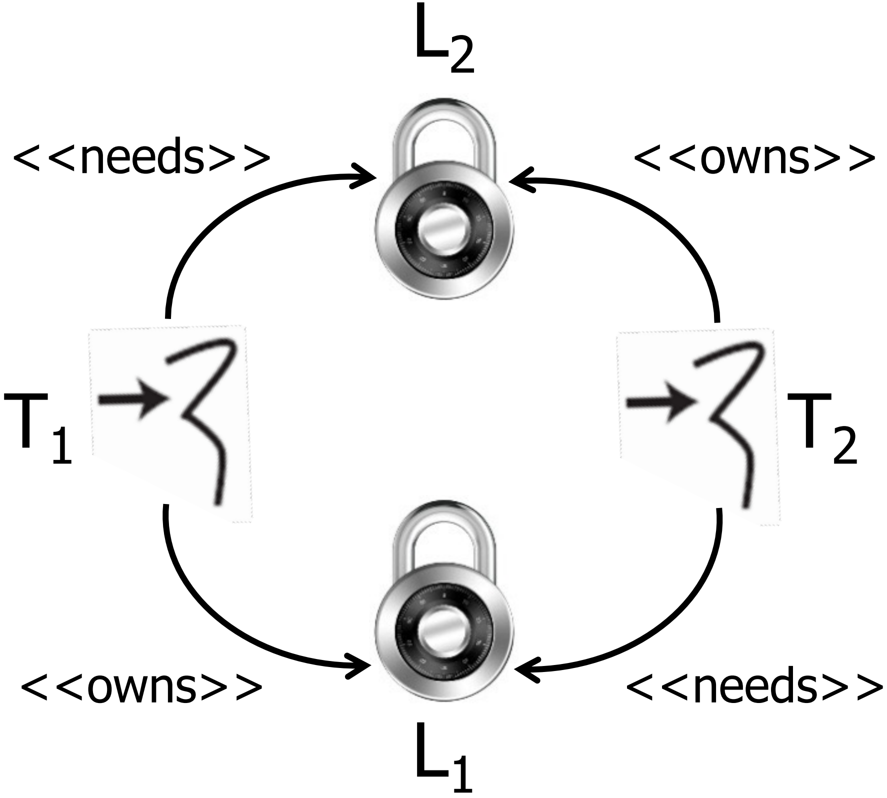
How Synchronizers Cause Concurrent Programming Hazards

How Synchronizers Cause Concurrent Programming Hazards

- Ironically, synchronizers can also enable concurrency hazards, e.g.
  - Deadlock

An Overview of Concurrent Programming Hazards

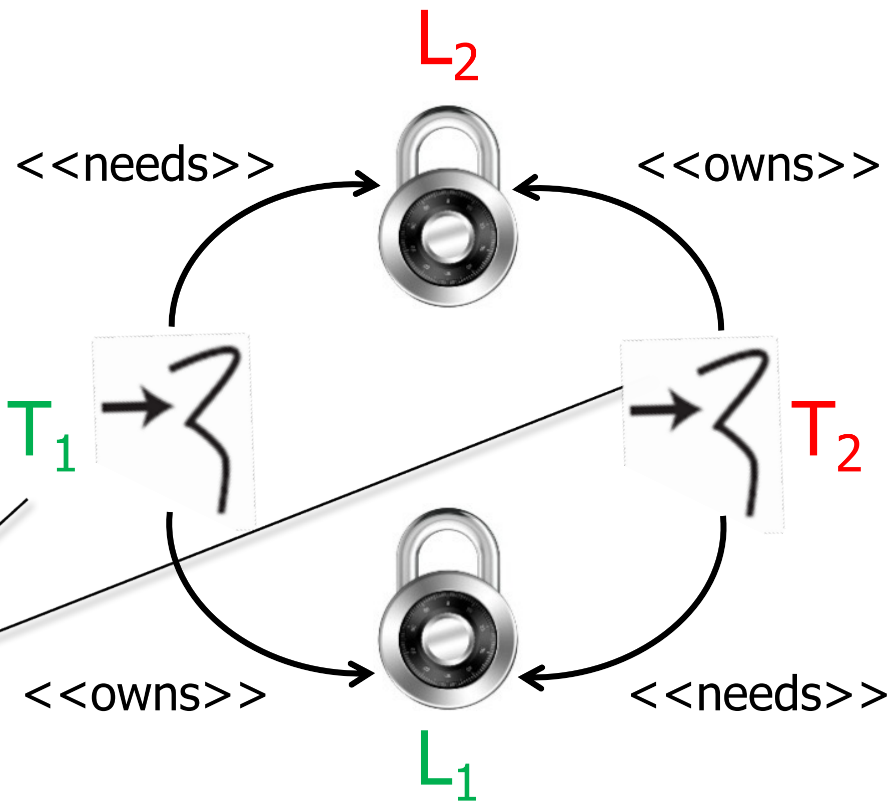
- Deadlock
  - Occurs when 2+ competing threads are waiting for the other(s) to finish, & thus none ever do



See en.wikipedia.org/wiki/Deadlock

An Overview of Concurrent Programming Hazards

- Deadlock
  - Occurs when 2+ competing threads are waiting for the other(s) to finish, & thus none ever do



T_2 & T_1 will be stuck in a "deadly embrace"

See www.computerworld.com/article/2585107/the-deadly-embrace.html

An Overview of Concurrent Programming Hazards

- Deadlock
  - Occurs when 2+ competing threads are waiting for the other(s) to finish, & thus none ever do


```
public void method1() {
  synchronized (String.class) {
    synchronized (Integer.class) { ... }
  }
}
public void method2() {
  synchronized (Integer.class) {
    synchronized (String.class) { ... }
  }
}
```

Deadlock will likely occur if method1() & method2() are called from thread T_1 & thread T_2 concurrently

See stackoverflow.com/a/14555496

An Overview of Concurrent Programming Hazards

- Deadlock
  - Occurs when 2+ competing threads are waiting for the other(s) to finish, & thus none ever do


```
public void method1() {
  synchronized (Integer.class) {
    synchronized (String.class) { ... }
  }
}
public void method2() {
  synchronized (Integer.class) {
    synchronized (String.class) { ... }
  }
}
```

Deadlock can be avoided by always acquiring locks in the same order!

See docs.oracle.com/cd/E19455-01/806-5257/6je9h0347/index.html

An Overview of Concurrent Programming Hazards

- Deadlock
  - Occurs when 2+ competing threads are waiting for the other(s) to finish, & thus none ever do


```
void transfer(SimpleQueue<String> src,
              SimpleQueue<String> dest)... {
   synchronized(src) {
     synchronized(dest) {
       while(!src.isEmpty())
         dest.put(src.take());
     }
   }
}
```

This program shows how deadlock may occur when transfer() is called concurrently from thread T_1 & thread T_2 with the src & dest params swapped

See github.com/douglascraigschmidt/LiveLessons/tree/master/DeadlockQueue

End of Overview Java Concurrency Hazards