COMP 150-CCP
Concurrent Programming

Lecture 16:
Thread Safety in Java

Dr. Richard S. Hall
rickhall@cs.tufts.edu

Reference

The content of this lecture is based on Chapter 2 of the ***Java Concurrency in Practice*** book by Brian Goetz.

Java and Concurrency

- Threads are everywhere in Java
  - JVM housekeeping (e.g., garbage collection, finalization)
  - Main thread for running application
  - AWT & Swing threads for events
  - `Timer` class for deferred tasks
  - Component frameworks such as Servlets and RMI create pools of threads
- In Java your application is likely to be multi-threaded whether you know it or not
  - Thus, you have to be familiar with concurrency and thread safety

State Management

- Concurrent programming is not really about threads or locks, these are simply mechanisms
- At its core, it is about ***managing access to state***, particularly shared, mutable state
  - In Java, this state is the data fields of objects
  - An object's state encompasses any data that can affect its externally visible behavior

Need for Thread Safety

- Depends on whether object will be accessed from multiple threads
  - This is a property of ***how*** the object will be used, not ***what*** it does
- If multiple threads can access an object and one of them might write to it, then they ***all must coordinate*** access using synchronization
  - There are no *special* situations where this rule does not apply

Achieving Thread Safety

- If multiple threads access the same mutable state variable without appropriate synchronization, then ***your program is broken***

Achieving Thread Safety

- If multiple threads access the same mutable state variable without appropriate synchronization, then ***your program is broken***
- There are three ways to fix it
  - Don't share the state variable across threads
  - Make the state variable immutable
  - Use synchronization whenever accessing the state variable

Achieving Thread Safety

- If multiple threads access the same mutable state variable without appropriate synchronization, then ***your program is broken***
- There are three ways to fix it
  - Don't share the state variable across threads
  - Make the state variable immutable
  - Use synchronization whenever accessing the state variable
- None of these are necessarily as easy as they may sound

Thread-Safe Classes

- A class is ***thread safe*** when it continues to behave correctly when accessed from multiple threads

Thread-Safe Classes

- A class is ***thread safe*** when it continues to behave correctly when accessed from multiple threads
  - Regardless of scheduling or interleaving of execution

Thread-Safe Classes

- A class is ***thread safe*** when it continues to behave correctly when accessed from multiple threads
  - Regardless of scheduling or interleaving of execution
  - No set of operations performed sequentially or concurrently on instances of thread-safe classes can cause an instance to be in an invalid state

Thread-Safe Classes

- A class is ***thread safe*** when it continues to behave correctly when accessed from multiple threads
  - Regardless of scheduling or interleaving of execution
  - No set of operations performed sequentially or concurrently on instances of thread-safe classes can cause an instance to be in an invalid state
  - Any needed synchronization is encapsulated in the class so that clients need not provide their own
    - The concept of a thread-safe class only makes sense if the class fully encapsulates its state
      - Likewise for the entire body of code that comprises a thread-safe program

Thread-Safe Class vs Program

- Is a thread-safe program simply a program constructed of thread-safe classes?

Thread-Safe Class vs Program

- Is a thread-safe program simply a program constructed of thread-safe classes?
  - *No*
    - All thread-safe classes can still result in non-thread-safe programs
    - A thread-safe program may use non-thread-safe classes

Thread-Safety Example

- Stateless factorizing servlet

```
public class StatelessFactorizer implements Servlet {
  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    encodeIntoResponse(resp, factors);
  }
}
```

Thread-Safety Example

- Stateless factorizing servlet

```
public class StatelessFactorizer implements Servlet {
  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    encodeIntoResponse(resp, factors);
  }
}
```

Has no fields and references no fields from other classes; everything is on the stack. Therefore, it is thread safe.

Thread-Safety Example

- Stateless factorizing servlet

```
public class StatelessFactorizer implements Servlet {
  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    encodeIntoResponse(resp, factors);
  }
}
```

Stateless objects are always thread safe.

Thread-Safety Example

- Stateful factorizing servlet
  - Keeps track of how many times it has been invoked

```
public class CountingFactorizer implements Servlet {
  private long count = 0;

  public long getCount() { return count; }

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    ++count;
    encodeIntoResponse(resp, factors);
  }
}
```

Thread-Safety Example

- Stateful factorizing servlet
  - Keeps track of how many times it has been invoked

```
public class CountingFactorizer implements Servlet {
  private long count = 0;

  public long getCount() { return count; }

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i)
    ++count;
    encodeIntoRespo
  }
}
```

This would work fine with in a single-threaded program, but not in a multi-threaded one...this is susceptible to *lost updates*.

Thread-Safety Example

- Stateful factorizing servlet
  - Keeps track of how many times it has been invoked

```
public class CountingFactorizer implements Servlet {
  private long count = 0;

  public long getCount() { return count; }

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i)
    ++count;
    encodeIntoRespon
  }
}
```

This is a *read-modify-write* race condition.

Race Conditions

- A ***race condition*** is the possibility of incorrect results due to timing of execution

Race Conditions

- A ***race condition*** is the possibility of incorrect results due to timing of execution
- Most common form is ***check-then-act***
  - A stale observation is used to determine what to do next
    - We've seen this in our homework where we have used individually atomic actions to test and the perform some action

Race Conditions

- A *race condition* is the possibility of incorrect results due to timing of execution
- Most common form is *check-then-act*
  - A stale observation is used to determine what to do next
    - We've seen this in our homework where we have used individually atomic actions to test and the perform some action
- Similar to what happens in real-life if you try to meet someone...
  - Need to have some agreed upon protocol

Race Condition Example

- Lazy initialization

```
public class LazyInit {
  private ExpensiveObject instance = null;

  public ExpensiveObject getInstance() {
    if (instance == null)
      instance = new ExpensiveObject();
    return instance;
  }
}
```

Race Condition Example

- Lazy initialization

```
public class LazyInit {
  private ExpensiveObject instance = null;

  public ExpensiveObject getInstance() {
    if (instance == null)
      instance = new ExpensiveObject();
    return instance;
  }
}
```

Unfortunate timing could result in this method returning different instances.

Compound Actions

- *Read-modify-write* and *check-then-act* operation sequences are compound actions
  - To ensure thread safety, all constituent actions must be performed atomically
- An operation or sequence of operations is ***atomic*** if it is indivisible relative to other operations on the same state
  - i.e., other threads see it as either happening completely or not at all.

Thread-Safety Example

- Modified stateful factorizing servlet
  - Keeps track of how many times it has been invoked

```
public class CountingFactorizer implements Servlet {
  private final AtomicLong count = new AtomicLong(0);

  public long getCount() { return count.get(); }

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    count.incrementAndGet();
    encodeIntoResponse(resp, factors);
  }
}
```

Thread-Safety Example

- Modified stateful factorizing servlet
  - Keeps track of how many times it has been invoked

```
public class CountingFactorizer implements Servlet {
  private final AtomicLong count = new AtomicLong(0);

  public long getCount()    return count.get(); }

  public void service(Servl     uest req, ServletResponse resp) {
    BigInteger i = extractFr     st(req);
    BigInteger[] factors = fa
    count.incrementAndGet(
    encodeIntoRespons
  }
}
```

Since this uses a thread-safe `AtomicLong` type from `java.util.concurrent.atomic`, the class is once again thread safe.

Thread-Safety Example

- Modified stateful factorizing servlet
  - Keeps track of how many times it has been invoked

```
public class CountingFactorizer i
  private final AtomicLong coun

  public long getCount() { ret

  public void service(ServletRequ                          ) {
    BigInteger i = extractFromRe
    BigInteger[] factors = fac    (i);
    count.incrementAndGet();
    encodeIntoResponse(resp, factors);
  }
}
```

`AtomicLong` provides an atomic read-modify-write operation for incrementing the value.

Thread-Safety Example

- Modified stateful factorizing servlet
  - Keeps track of how many times it has been invoked

```
public class CountingFactorizer i
  private final AtomicLong coun

  public long getCount() { ret

  public void service(ServletRequ
    BigInteger i = extractFromRe
    BigInteger[] factors = fa
    count.incrementAndGet();
    encodeIntoResponse(resp, factors);
  }
}
```

Advice:
Where practical, use existing thread-safe objects to manage your state.

Side Note: AtomicLong

- `AtomicLong` replaces a `long`/`Long`
  - `get()` - Gets the current value.
  - `set(long newValue)` - Sets to the given value.
  - `lazySet(long newValue)` - Eventually sets to the given value.
  - `compareAndSet(long expect, long update)` - Atomically sets the value to the given updated value if the current value == the expected value.
  - `weakCompareAndSet(long expect, long update)` - Atomically sets the value to the given updated value if the current value == the expected value.
  - `getAndAdd(long delta)` - Atomically adds the given value to the current value.
  - `getAndDecrement()` - Atomically decrements by one the current value.
  - `getAndIncrement()` - Atomically increments by one the current value.
  - `getAndSet(long newValue)` - Atomically sets to the given value and returns the old value.
  - `addAndGet(long delta)` - Atomically adds the given value to the current value.
  - `incrementAndGet()` - Atomically increments by one the current value.
  - `decrementAndGet()` - Atomically decrements by one the current value.

Thread-Safety Example

- Caching factorizing servlet
  - Remembers last result

```
public class CachingFactorizer implements Servlet {
  private final AtomicReference<BigInteger> lastNumber
    = new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors
    = new AtomicReference<BigInteger[]>();

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber.get()))
      encodeIntoResponse(resp, lastFactors.get());
    else {
      BigInteger[] factors = factor(i);
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp, factors);
    }
  }
}
```

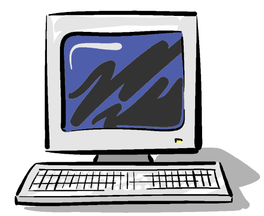
Thread-Safety Example

- Caching factorizing servlet
  - Remembers last result

```
public class CachingFactorizer implements Servlet {
  private final AtomicReference<BigInteger> lastNumber
    = new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors
    = new AtomicReference<BigInteger[]>();

  public void service(ServletRequest req, Servle      onse resp) {
    BigInteger i = extractFromRequest(re
    if (i.equals(lastNumber.get()))
      encodeIntoResponse(resp, las
    else {
      BigInteger[] factors = factor(i
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp, factors);
    }
  }
}
```

Even though our two references are atomic, they are not independent.



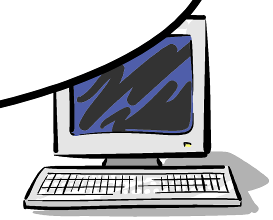
Thread-Safety Example

- Caching factorizing servlet
  - Remembers last result

```
public class CachingFactorizer implements Servlet {
  private final AtomicReference<BigInteger> lastNumber
    = new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors
    = new AtomicReference<BigInteger[]>();

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber.get()))
      encodeIntoResponse(resp, lastFactors.get());
    else {
      BigInteger[] factors = factor(
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp, fac
    }
  }
}
```

Thus, dependent operations on them must be done atomically...

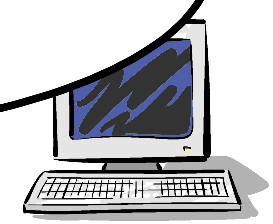


Thread-Safety Example

- Caching factorizing servlet
  - Remembers last result

```
public class CachingFactorizer implements Servlet {
  private final AtomicReference<BigInteger> lastNumber
    = new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors
    = new AtomicReference<BigInteger[]>();

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber.get()))
      encodeIntoResponse(resp, lastFactors.get());
    else {
      BigInteger[] factors = factor(i);
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp, fac
    }
  }
}
```

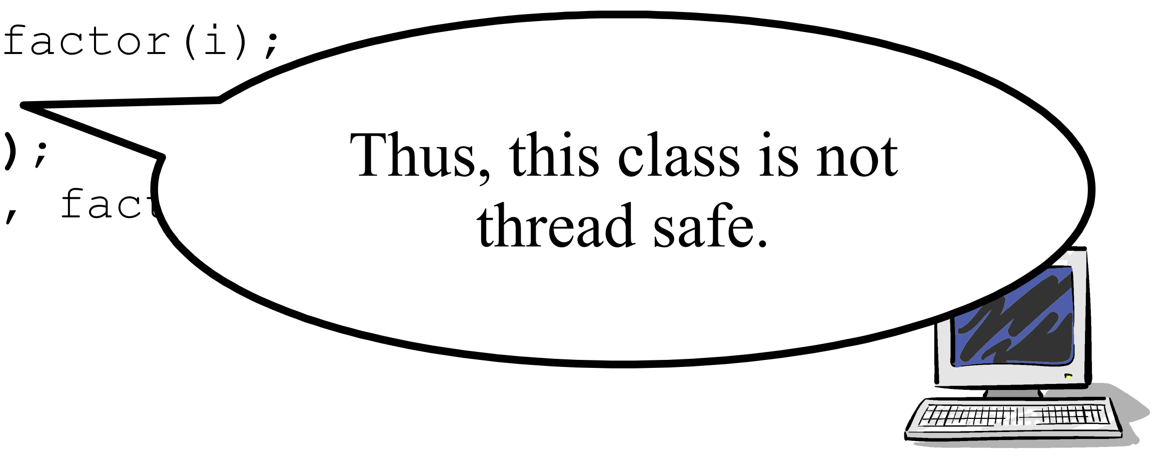


Thread-Safety Example

- Caching factorizing servlet
  - Remembers last result

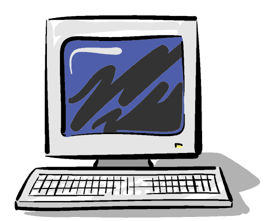
```
public class CachingFactorizer implements Servlet {
  private final AtomicReference<BigInteger> lastNumber
    = new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors
    = new AtomicReference<BigInteger[]>();

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber.get()))
      encodeIntoResponse(resp, lastFactors.get());
    else {
      BigInteger[] factors = factor(i);
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp, fac
    }
  }
}
```



Side Note: AtomicReference<V>

- `AtomicReference<V>` can be used in place of a reference to an object
  - `get()` - Gets the current value.
  - `set(V newValue)` - Sets to the given value.
  - `lazySet(V newValue)` - Eventually sets to the given value.
  - `getAndSet(V newValue)` - Atomically sets to the given value and returns the old value.
  - `compareAndSet(V expect, V update)` - Atomically sets the value to the given updated value if the current value == the expected value.
  - `weakCompareAndSet(V expect, V update)` - Atomically sets the value to the given updated value if the current value == the expected value.



Intrinsic Locks

- Java offers built-in locking to enforce atomicity via the `synchronized` block
  - One lock associated with each object instance
  - A synchronized block is comprised of
    - An object reference that is used as the lock
    - A block of code that is guarded by the lock
  - Only one thread at any given time can be inside of a block guarded by a given lock (i.e, mutual exclusion)
    - The lock is acquired/released by the thread on entry/exit
      - It may be blocked to wait to acquire the lock
  - Although each object has a lock, that lock can be used for any purpose
    - Not necessarily related to the object itself
    - Possibly spanning many objects

Intrinsic Locks

- A lock associated with an object does not restrict access to the object's state
  - It only restricts multiple threads from acquiring the lock at the same time
  - Having a built-in object lock is only a convenience so that we don't have to explicitly create locks
- Encapsulation with an appropriate locking protocol is the only way to restrict access to an object's state

Intrinsic Locks

```
public class Foo {
  public synchronized void bar() {
    ...
  }
}
```

Is equivalent to:

```
public class Foo {
  public void bar() {
    synchronized (this) {
      ...
    }
  }
}
```

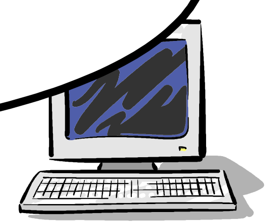
Intrinsic Locks

```java
public class Foo {
  public synchronized void bar() {
    ...
  }
}
```

Is equivalent to:

```java
public class Foo {
  public void bar() {
    synchronized (this) {
      ...
    }
  }
}
```

Why might you do this instead?



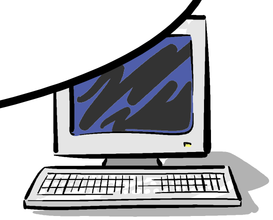
Intrinsic Locks

```
public class Foo {
  public synchronized void bar() {
    ...
  }
}
```

Is equivalent to:

```
public class Foo {
  public void bar() {
    synchronized (this) {
      ...
    }
  }
}
```

To provide more fine-grained locking.



Intrinsic Locks

- Assuming `Foo` is thread safe, what can we assume about `bar()` and `woz()`?

```
public class Foo {
  public synchronized void bar() {
    ...
  }

  public void woz() {
    ...
  }
}
```

Intrinsic Locks

- Assuming `Foo` is thread safe, what can we assume about `bar()` and `woz()`?

```
public class Foo {
  public synchronized void bar() {
    ...
  }

  public void woz() {
    ...
  }
}
```

Perhaps that the `woz()` method does not access shared state...

Intrinsic Locks

- Assuming `Foo` is thread safe, what can we assume about `bar()` and `woz()`?

```
public class Foo {
  public synchronized void bar() {
    ...
  }

  public void woz() {
    ...
  }
}
```

Or that `woz()` has more fine-grained locking...

Intrinsic Locks

- Assuming `Foo` is thread safe, what can we assume about `bar()` and `woz()`?

```
public class Foo {
  public synchronized void bar() {
    ...
  }

  public void woz() {
    ...
  }
}
```

At a minimum, we know that more than one thread can enter `woz()` and thus the `Foo` instance at a given time.

Intrinsic Locks

- How does sharing locks across objects work?

```
public class Foo {
  public Foo(String s) { ... }
  public void foo() {
    synchronized (s) {
      ...
    }
  }
}

public class Bar {
  public Bar(String s) { ... }
  public void bar() {
    synchronized (s) {
      ...
    }
  }
}
```

```
...
String s = new String(“1”);
Foo f = new Foo(s);
Bar b = new Bar(s);
...
```

Intrinsic Locks

- How does sharing locks across objects work?

```
public class Foo {
  public Foo(String s) { ... }
  public void foo() {
    synchronized (s) {
      ...
    }
  }
}

public class Bar {
  public Bar(String s) { ... }
  public void bar() {
    synchronized (s) {
      ...
    }
  }
}
```

```
...
String s = new String(“1”);
Foo f = new Foo(s);
Bar b = new Bar(s);
...
```

As you would expect, only one thread can be executing inside the guarded code blocks of `Foo` or `Bar` at a given time.

Thread-Safety Example

- Modified caching factorizing servlet
  - Remembers last result

```
public class CachingFactorizer implements Servlet {
  private final AtomicReference<BigInteger> lastNumber
    = new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors
    = new AtomicReference<BigInteger[]>();

  public synchronized void service(
   ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber.get()))
      encodeIntoResponse(resp, lastFactors.get());
    else {
      BigInteger[] factors = factor(i);
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp, factors);
    }
  }
}
```

Thread-Safety Example

- Modified caching factorizing servlet
  - Remembers last result

```
public class CachingFactorizer implements Servlet {
  private final AtomicReference<BigInteger> lastNumber
    = new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors
    = new AtomicReference<BigInteger[]>();

  public synchronized void service(
   ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req
    if (i.equals(lastNumber.get()))
      encodeIntoResponse(resp, la
    else {
      BigInteger[] factors = fa
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp, fa
    }
  }
}
```

Intrinsic Locks

- Would the following code deadlock?

```
public class Widget {
  public synchronized void doSomething() {
    ...
  }
}

public class LoggingWidget extends Widget {
  public synchronized void doSomething() {
    System.out.println("Calling doSomething");
    super.doSomething();
  }
}
```

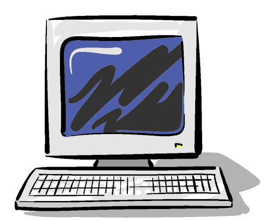
Intrinsic Locks

- Would the following code deadlock?

```
public class Widget {
  public synchronized void doSomething() {
    ...
  }
}

public class LoggingWidget extends Widget {
  public synchronized void doSomething() {
    System.out.println("Calling doSomething");
    super.doSomething();
  }
}
```

No, because intrinsic locks are reentrant.

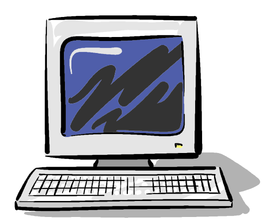


Intrinsic Locks

- Intrinsic locks are acquired per thread, not per invocation
  - Semaphores are acquired per invocation, for example

Intrinsic Locks

- Intrinsic locks are acquired per thread, not per invocation
  - Semaphores are acquired per invocation, for example
- Essentially, Java remembers the thread that owns a lock and keeps a lock counter
  - The counter value for unheld locks is zero
  - Each time a given thread acquires a lock, it increments the counter
    - Likewise, it decrements the counter each time it exits a synchronized block until it reaches zero and the lock is freed

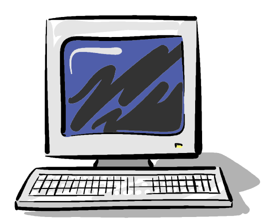


Intrinsic Locks

- Intrinsic locks are acquired per thread, not per invocation
  - Semaphores are acquired per invocation, for example
- Essentially, Java remembers the thread that owns a lock and keeps a lock counter
  - The counter value for unheld locks is zero
  - Each time a given thread acquires a lock, it increments the counter
    - Likewise, it decrements the counter each time it exits a synchronized block until it reaches zero and the lock is freed
- Reentrancy facilitates encapsulation of locking behavior and simplifies development object-oriented concurrent code

Locking Rules of Thumb

- Compound actions on shared state must be made atomic (i.e., *read-modify-write*, *check-then-act*)



Locking Rules of Thumb

- Compound actions on shared state must be made atomic (i.e., *read-modify-write*, *check-then-act*)
  - Combining individually atomic actions does not result in an atomic action

Locking Rules of Thumb

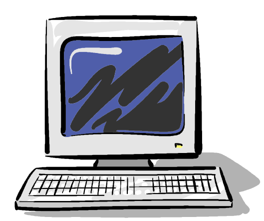
- Compound actions on shared state must be made atomic (i.e., *read-modify-write*, *check-then-act*)
  - Combining individually atomic actions does not result in an atomic action
- All access to shared state must be synchronized, not just modifications

Locking Rules of Thumb

- Compound actions on shared state must be made atomic (i.e., *read-modify-write*, *check-then-act*)
  - Combining individually atomic actions does not result in an atomic action
- All access to shared state must be synchronized, not just modifications
  - Use same lock wherever a specific variable is accessed
    - Variable is considered to be ***guarded by*** the specific lock
    - Also true if multiple variables make up a single invariant
  - You should clearly document which locks are used to guard which state

Locking Rules of Thumb

- Compound actions on shared state must be made atomic (i.e., *read-modify-write*, *check-then-act*)
  - Combining individually atomic actions does not result in an atomic action
- All access to shared state must be synchronized, not just modifications
  - Use same lock wherever a specific variable is accessed
    - Variable is considered to be ***guarded by*** the specific lock
    - Also true if multiple variables make up a single invariant
  - You should clearly document which locks are used to guard which state
- Only guard mutable state that is potentially accessed by multiple threads



Locking Rules of Thumb

- Need ***right*** amount of locking

Locking Rules of Thumb

- Need ***right*** amount of locking
  - Too little could result in invalidate states

Locking Rules of Thumb

- Need ***right*** amount of locking
  - Too little could result in invalidate states
  - Too much could result in deadlock

Locking Rules of Thumb

- Need ***right*** amount of locking
  - Too little could result in invalidate states
  - Too much could result in deadlock
  - Too coarse grained could result in poor performance

Locking Rules of Thumb

- Need ***right*** amount of locking
  - Too little could result in invalidate states
  - Too much could result in deadlock
  - Too coarse grained could result in poor performance
  - Too fine grained increases complexity and could also result in poor performance

Locking Rules of Thumb

- Need ***right*** amount of locking
  - Too little could result in invalidate states
  - Too much could result in deadlock
  - Too coarse grained could result in poor performance
  - Too fine grained increases complexity and could also result in poor performance
- Prefer simplicity over performance
  - Optimize later, if necessary

Locking Rules of Thumb

- Need ***right*** amount of locking
  - Too little could result in invalidate states
  - Too much could result in deadlock
  - Too coarse grained could result in poor performance
  - Too fine grained increases complexity and could also result in poor performance
- Prefer simplicity over performance
  - Optimize later, if necessary
- Avoid holding locks during lengthy computations

Locking Rules of Thumb

- Need ***right*** amount of locking
  - Too little could result in invalidate states
  - Too much could result in deadlock
  - Too coarse grained could result in poor performance
  - Too fine grained increases complexity and could also result in poor performance
- Prefer simplicity over performance
  - Optimize later, if necessary
- Avoid holding locks during lengthy computations
- Avoid calling external code while holding locks

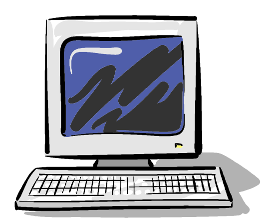
Thread-Safety Example

```
@ThreadSafe
public class CachedFactorizer implements Servlet {
  @GuardedBy("this") private BigInteger lastNumber, lastFactors[];
  @GuardedBy("this") private long hits, cacheHits;
  public synchronized long getHits() { return hits; }
  public synchronized double getCacheHitRatio()
    { return (double) cacheHits / (double) hits; }
  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = null;
    synchronized (this) {
      ++hits;
      if (i.equals(lastNumber)) {
        ++cacheHits; factors = lastFactors.clone();
      }
    }
    if (factors == null) {
      factors = factor(i);
      synchronized (this)  {
        lastNumber = i; lastFactors = factors.clone();
      }
    }
    encodeIntoResponse(resp, factors);
  }
}
```

Thread-Safety Example

```
@ThreadSafe
public class CachedFactorizer implements Servlet {
  @GuardedBy("this") private BigInteger lastNumber, lastFactors[];
  @GuardedBy("this") private long hits, cacheHits;
  public synchronized long getHits() { return hits; }
  public synchronized double getCacheHitRatio()
    { return (double) cacheHits / (double) hits; }
  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = null;
    synchronized (this) {
      ++hits;
      if (i.equals(lastNumber)) {
        ++cacheHits; factors = lastFactors.clone();
      }
    }
    if (factors == null) {
      factors = factor(i);
      synchronized (this)  {
        lastNumber = i; lastFactors = factors.clone();
      }
    }
    encodeIntoResponse(resp, factors);
  }
}
```

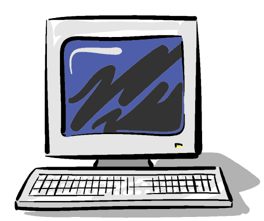
Is this class good now?



Thread-Safety Example

```
@ThreadSafe
public class CachedFactorizer implements Servlet {
  @GuardedBy("this") private BigInteger lastNumber, lastFactors[];
  @GuardedBy("this") private long hits, cacheHits;
  public synchronized long getHits() { return hits; }
  public synchronized double getCacheHitRatio()
    { return (double) cacheHits / (double) hits; }
  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = null;
    synchronized (this) {
      ++hits;
      if (i.equals(lastNumber)) {
        ++cacheHits; factors = lastFact
      }
    }
    if (factors == null) {
      factors = factor(i);
      synchronized (this)  {
        lastNumber = i; lastFactors = fac
      }
    }
    encodeIntoResponse(resp, factors);
  }
}
```

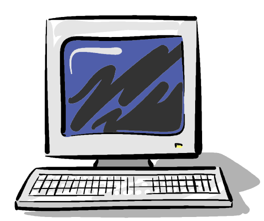
Yes. The compound actions are appropriately guarded in such a way that still allows for concurrency.



Thread-Safety Example

```
@ThreadSafe
public class CachedFactorizer implements Servlet {
  @GuardedBy("this") private BigInteger lastNumber, lastFactors[];
  @GuardedBy("this") private long hits, cacheHits;
  public synchronized long getHits() { return hits; }
  public synchronized double getCacheHitRatio()
    { return (double) cacheHits / (double) hits; }
  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = null;
    synchronized (this) {
      ++hits;
      if (i.equals(lastNumber)) {
        ++cacheHits; factors = lastFactors.clone();
      }
    }
    if (factors == null) {
      factors = factor(i);
      synchronized (this)  {
        lastNumber = i; lastFactors = factors.clone();
      }
    }
    encodeIntoResponse(resp, factors);
  }
}
```

Why do we no longer use `AtomicLong` variables?



Thread-Safety Example

```java
@ThreadSafe
public class CachedFactorizer implements Servlet {
  @GuardedBy("this") private BigInteger lastNumber, lastFactors[];
  @GuardedBy("this") private long hits, cacheHits;
  public synchronized long getHits() { return hits; }
  public synchronized double getCacheHitRatio()
    { return (double) cacheHits / (double) hits; }
  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = null;
    synchronized (this) {
      ++hits;
      if (i.equals(lastNumber)) {
        ++cacheHits; factors = lastFactors.clone();
      }
    }
    if (factors == null) {
      factors = factor(i);
      synchronized (this)  {
        lastNumber = i; lastFactors = factors.clone();
      }
    }
    encodeIntoResponse(resp, factors);
  }
}
```

Not necessary since access is already guarded.