CSx55: Distributed Systems [Thread Safety]

On impending code breaks, putting the brakes, you are …

Let a reference escape, have you?
Misbehave, your code will, out of the blue

Get out, you will, of this bind
If, your objects, you have confined

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- How does the runtime for `wait/notify` contrast with that of the `Lock` interface?
- Is the `wait/notify` construct applicable beyond producer-consumer with shared buffer?
- Why can't we override the `wait/notify` methods?
- Use of `private static` fields
  - Not allowed in interfaces, but allowed in classes; most common use case is for constants and singleton instances where you *also* add final i.e., `private static final`; also in `ThreadLocal` which is a specific use-case.
- Errors if I use `Lock` inside a `synchronized` method? Which lock is acquired when that synchronized method is invoked?
- Why `synchronized` blocks? Doesn't the `synchronized` method do it all?
- Is waiting to acquire a `Lock` a blocking call?

Topics covered in this lecture

- Atomicity
- Locks& Reentrancy
- Guarding state with locks
- Sharing Objects
- Thread confinement

When should an object be thread-safe?

- Will it be accessed from multiple threads?
- The key here is *how* the object is **used**
  - Not *what* it **does**

How to make an object thread-safe

- Use *synchronization* to **coordinate** access to mutable state
- Failure to do this?
  - Data corruptions
  - Problems that manifest themselves in myriad forms

Mechanisms for synchronization in Java

- One way to achieve this is via the `synchronized` keyword
  - Exclusive locking
- Other approaches include:
  - `volatile` variables
  - Explicit **locks**
  - **Atomic** variables

Programs that omit synchronizations

- Might work for some time
  - But it ***will break*** at some point
- Far easier to design a class to be thread-safe *from the start*
  - Retrofitting it to be thread-safe is extremely hard

Thread-safety: Encapsulate your state

- Fewer code should have access to a particular variable
  - Easier to reason about *conditions* under which it might be accessed
- **DON'T:**
  - Store state in `public` fields
  - Publish reference to an **internal** object

Fixing access to mutable state variables from multiple threads

- *Don't share* state variables across threads
- Make state variables *immutable*
- Use *synchronization* to coordinate access to the state variable

Correctness of classes

- Class conforms to **specification**
- **Invariants** constrain object's state
- **Post conditions** describe the effects of operations

A Thread-safe class

- **Behaves correctly** when accessed from multiple threads
- Regardless of *scheduling or interleaving* of execution of those threads
  - By the runtime environment
- No additional synchronization or coordination by the calling code

Really?

- Thread safe classes encapsulate *any needed* synchronization
- Clients **should not** have to provide their own

Stateless objects are always thread-safe

```
public class StatelessClass implements Servlet {

     public void factorizer(ServletRequest req,
                            ServletResponse resp) {
         BigInteger i = extractFromReq(req);
         BigInteger[] factors  = factorize(i);
         encodeIntoResponse(resp, factors);
     }

}
```

Stateless objects are always thread-safe

- **Transient state** for a particular computation exists solely in *local variables*
  - Stored on the thread's stack
  - Accessible only to the executing thread

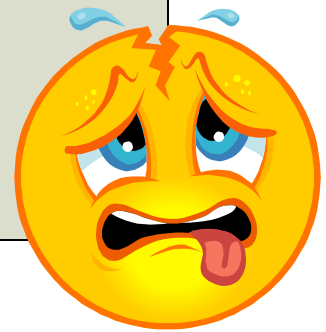
- One thread cannot influence the result of another
  - The threads have no shared state

Atomicity

- Let's look at two operations **A** and **B**
- From the perspective of thread executing **A**
- When another thread executes **B**
  - Either all of **B** has executed or none of it has
- Operations **A** and B are **atomic** *with respect to each other*

Initializing Objects

```
public class LazyInitialization {

    private ExpensiveObject instance = null;

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

}
```



Thread-safe initialization

```
public class Singleton {
   private static final Singleton instance = new Singleton();

   // Private constructor prevents instantiation from other
   // classes
   private Singleton() { }

   public static Singleton getInstance() {
      return instance;
   }
}
```

The `final` keyword

- You cannot extend a `final` class
  - E.g., `java.lang.String`
- You cannot override a `final` method
- You can only initialize a `final` variable **once**
  - Either via an initializer or an assignment statement

Blank `final` instance variable of a class

- Must be assigned *within every constructor* of the class
- Attempting to set it outside the constructor will result in a compilation error
- The value of a `final` variable is not necessarily known at compile time

Atomicity with compound operations

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

   public long getCount() {return count;}

   public void factorizer(int i) {
     int[] factors = factor(i);
     count++;
   }

}
```

Atomicity with compound operations

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

   public long getCount() {return count;}

   public void factorizer(int i) {
     int[] factors = factor(i);
     count.incrementAndGet();
   }

}
```

Compound actions & thread-safety

- Compound actions
  - Check-then-act
  - Read-modify-write
- Must be executed atomically for thread-safety

Locks & Reentrancy

Reentrancy

- When thread requests lock held by another thread?
  - Requesting thread blocks
- If a thread attempts to acquire a lock it already holds?
  - Succeeds
- Locks are acquired on a **per-thread** rather than on a per-invocation basis

How reentrancy works [1/2]

- For each lock two items are maintained
  - Acquisition count
  - Owning thread
- When the count is zero?
  - Lock is free
- If a thread acquires lock for the first time?
  - Count is one

How reentrancy works [2/2]

- If owning thread acquires lock again, count is incremented
- When owning thread exits synchronized block, count is decremented
  - If it is zero …. Lock is released

Does this result in a deadlock?

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

}

public class LoggingWidget extends Widget {

  public synchronized void doSomething() {
    System.out.println(toString()+"Calling doSomething());
    super.doSomething();
  }
}
```

No! Intrinsic locks are reentrant

GUARDING STATE WITH LOCKS

Guarding state with locks

- A *mutable*, *shared* variable that may be accessed by multiple threads must be guarded by the **same lock**

- For every invariant that involves more than one variable?
  - *All variables* must be guarded by the **same lock**

Watch for indiscriminate use of synchronization

- Every method in `Vector` is synchronized
- But this does not render compound actions on `Vector` atomic

```
if (!vector.contains(element)) {
   vector.add(element);
}
```

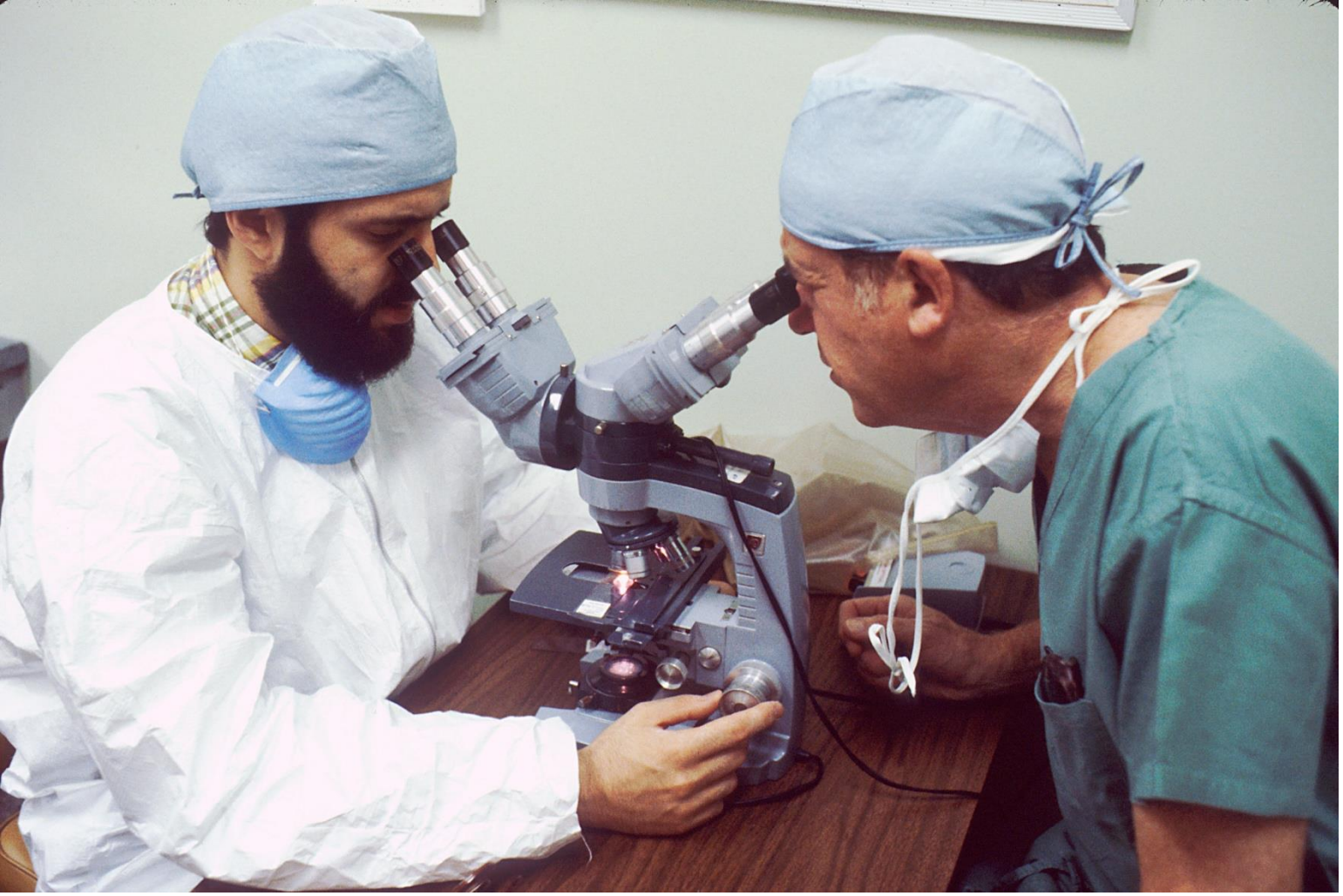
- Snippet has *race condition* even though `add` and `contains` are atomic
- **Additional locking needed for compound actions**

Pitfalls of over synchronization

- Number of simultaneous invocations?
  - Not limited by processor resources, but is limited by the application structure
  - **Poor concurrency**

Antidote for poor concurrency

- Control the **scope** of the lock
  - Too large: Invocations become sequential
  - Don't make it too small either
    - Operations that are atomic should not be in `synchronized` block



Sharing Objects

What we will be looking at

- Techniques for sharing and publishing objects
  - Safe access from multiple threads
- Together with synchronization, sharing objects lays foundation for thread-safe classes

Synchronization

- What we have seen so far:
  - Atomicity and demarcating *critical sections*
- But it is also about **memory visibility**
  - We *prevent* one thread from modifying object state while another is using it
  - When *state of an object is modified*, other thread can **see** the changes that were made

Publication and Escape

- Publishing an object
  - Makes it available *outside* current scope
    - Storing a reference to it, returning from a non-private method, passing it as an argument to another method
- **Escape**
  - An object that is published when it should not have been

Pitfalls in publication

- Publishing internal state variables
  - Makes it **difficult** to preserve invariants
- Publishing objects before they are constructed
  - Compromises thread-safety

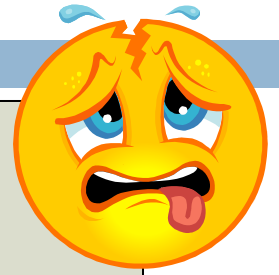
Most blatant form of publication

- Storing a reference in a `public static` field

```
public static Set<Secret> knownSecrets;

public void initialize() {
   knownSecrets = new HashSet<Secret>();
}
```

  - If you add a `Secret` to `knownSecrets`?
    - You also end up publishing that `Secret`

Allowing internal mutable state to escape

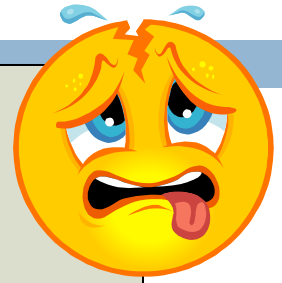


```
public class PublishingState {
   private String[] states = new String[] {
       "AK", "AL", …
   };

   public String[] getStates() {return states;}
}
```

- `states` has *escaped* its intended scope
  - What should have been private is now public
- **Any caller can modify its contents**

Another way to publish internal state

```
public class ThisEscape {

   public ThisEscape(EventSource source) {
      source.registerListener(
         new EventListener() {
               public void onEvent(Event e) {
                  doSomething(e);
               }
          });
   }
}
```



- When `EventListener` is published, it publishes the enclosing `ThisEscape` instance
- **Inner class instances contain hidden reference to enclosing instance**

Abbreviated view of the classes generated by the javac

```
public class ThisEscape {

    public ThisEscape(EventSource source) {
        source.registerListener(new ThisEscape$1(this));
    }

    private void doSomething(Event e) {
        .....
    }

    static void access$000(ThisEscape _this, Event event) {
        _this.doSomething(event);
    }
}
```

```
class ThisEscape$1 implements EventListener {
    final ThisEscape this$0;

    ThisEscape$1(ThisEscape thisescape) {
        this$0 = thisescape;    super();  }

    public void onEvent(Event e) {
        ThisEscape.access$000(this$0, e);  }
}
```

Safe construction practices

- An object is in a predictable, consistent state *only after its constructor returns*
- Publishing an object within its constructor?
  - You are publishing an incompletely constructed object
  - Even if you are doing so in the last line of the constructor
- RULE: Don't allow `this` to escape during construction

A common mistake is to start a thread from a constructor

- When an object creates a thread in its constructor
  - Almost always shares its `this` reference with the new thread
    - Explicitly: Passing it to the constructor
    - Implicitly: The `Thread` or `Runnable` is an inner class of the owning object
- Nothing wrong with creating a thread in a constructor
  - Just don't start the `Thread`
  - Expose an `initialize()` method

Thread Confinement

Thread confinement

- Accessing shared, mutable data requires synchronization
  - Avoid this by *not sharing*
- If data is only accessed from a single thread?
  - No synchronization is needed
- When an object is **confined** to a thread?
  - Usage is **thread-safe** *even if the object is not*

Thread confinement

- Language has no means of confining an object to a thread
- Thread confinement is an element of a **program's design**
  - Enforced by implementation
- Language and core libraries provide mechanisms to help with this
  - Local variables and the `ThreadLocal` class

Stack confinement

- Object can only be reached through local variables
- Local variables are **intrinsically confined** to the executing thread
  - Exist on executing thread's stack
  - Not accessible to other threads

Thread confinement of reference variables

```
public int loadTheArk() {
   SortedSet<Animal> animals;

   // animals confined to method don't let
   // them escape

   return numPairs;
}
```

If you were to publish a reference to `animals`**, stack confinement would be violated**

ThreadLocal

- Allows you to associate a per-thread value with a value-holding object
- Provides `set` and `get` accessor methods
  - Maintains a separate copy of value for each thread that uses it
  - `get` returns the most recent value passed to `set`
    - From the currently executing thread

Using `ThreadLocal` for thread confinement

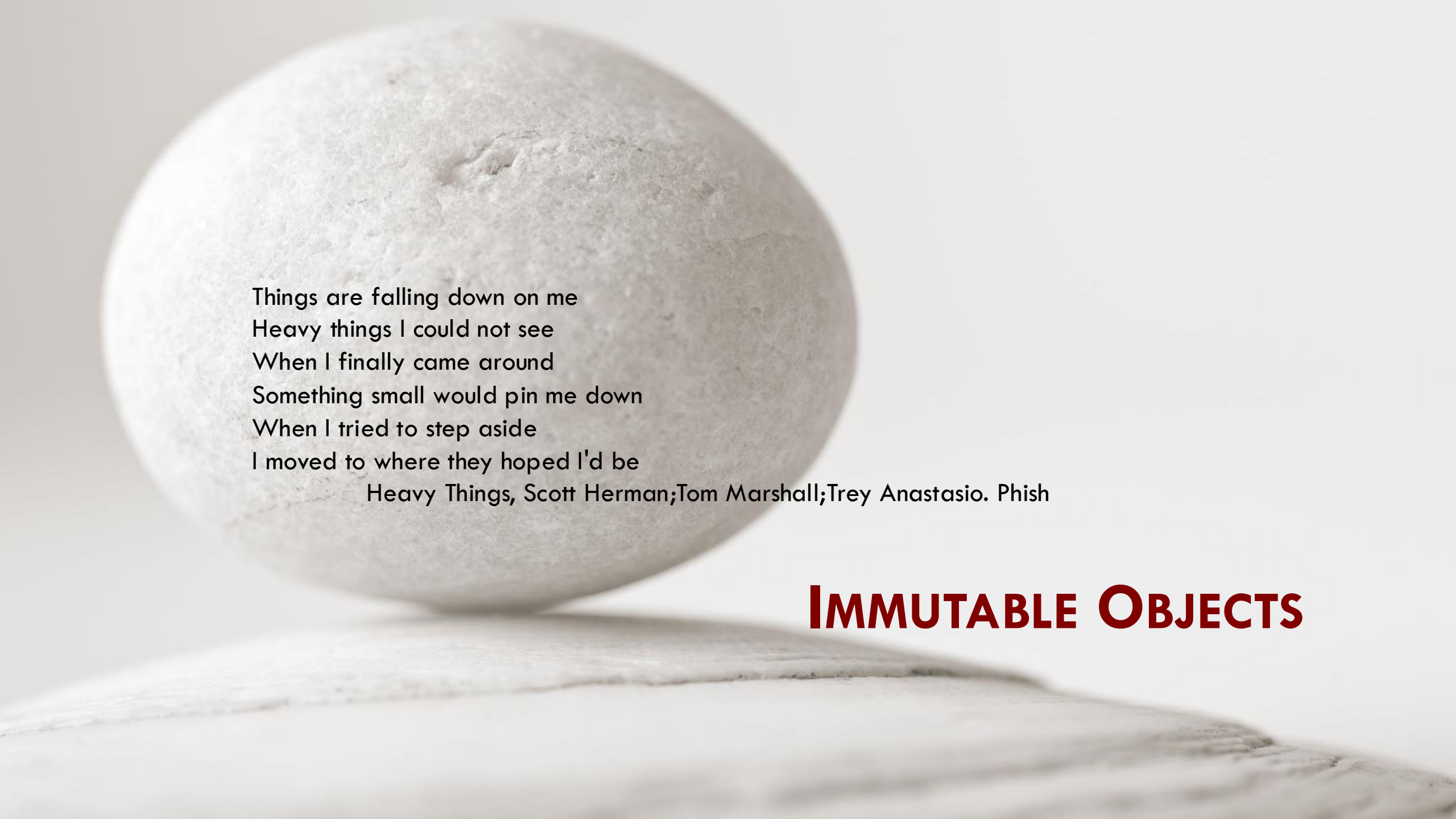
```
private static ThreadLocal<Connection> connectionHolder
   = new ThreadLocal<Connection>() {
        public Connection initialValue() {
          return DriverManager.getConnection(DB_URL);
        }
      };

public static Connection getConnection() {
    return connectionHolder.get();
}
```

Each thread will have its own connection

When thread calls ThreadLocal.get for the first time?
initialValue() provides the initial value

Common use of `ThreadLocal`

- Used when a frequently used operation requires a temporary object
  - Wish to avoid reallocating temporary object on each invocation
- `Integer.toString()`
  - Before 5.0 used `ThreadLocal` to store a 12-byte buffer for formatting result



Things are falling down on me
Heavy things I could not see
When I finally came around
Something small would pin me down
When I tried to step aside
I moved to where they hoped I'd be

Heavy Things, Scott Herman;Tom Marshall;Trey Anastasio. Phish

Immutable Objects

Immutable objects

- State cannot be modified after construction
- All its fields are `final`
- Properly constructed
  - The `this` reference does not escape during construction

Immutable objects

```
public final class ThreeStooges {
   private final Set<String> stooges = new HashSet<String>();

   public ThreeStooges() {
     stooges.add("Moe");
     stooges.add("Larry");
     stooges.add("Curly");
   }

   public boolean isStooge(String name) {
        return stooges.contains(name);
   }
}
```

Design makes it impossible to modify after construction

The stooges reference is final

All object state reached through a final field

Safe publication of objects

- Storing reference to an object into a public field is **not enough** to publish that object safely

```
public Holder holder;

public void initialize() {
   holder = new Holder(42);
}
```

`Holder` **could appear to be in an inconsistent state**

Even though invariants may have been established by constructor

Class at risk of failure if not published properly

```
public class Holder {
   private int n;

   public Holder(int n) {this.n = n}

   public void assertSanity() {
      if (n != n) {
         throw new AssertionError("Statement is false");
      }
   }
}
```

Thread may see a stale value first time it reads the field and an up-to-date value the next time

Pearls and swine bereft of me
Long and weary my road has been
I was lost in the cities
Alone in the hills
No sorrow or pity for leaving, I feel, yeah

I am not your rolling wheels
I am the highway
I am not your carpet ride
I am the sky

I Am the Highway, Audioslave

Composing Objects

Composing Objects

- We don't want to have to analyze *each memory access* to ensure program is thread-safe
- We wish to take thread-safe components and **compose** them into larger components or programs

Basic elements of designing a thread-safe class

- Identify **variables** that *form* the object's state
- Identify **invariants** that *constrain* the state variables
- Establish a **policy** for managing *concurrent access* to the object's state

Synchronization policy

- Defines how object *coordinates access* to its state
  - Without violating its invariants or post-conditions
- Specifies a combination of:
  - Immutability
  - Thread confinement
  - Locking

Looking at a counter

```
public final class Counter {
   private long value=0;

   public synchronized long getValue() {
        return value;
   }

   public synchronized long increment() {
     if (value == Long.MAX_VALUE) {
        throw new IllegalStateException("Counter Overflow");
     }
     value++;
     return value;
   }
}
```

Making a class thread-safe

- Ensure that invariants hold under concurrent access
  - We need to *reason* about state
- Object and variables have **state space**
  - ***Range*** of possible states
  - ***Keep this small*** so that it is easier to reason about

Classes have invariants that tag certain states as valid or invalid

- Looking back at our **`Counter`** example
- The `value` field is a `long`
- The state space ranges from `Long.MIN_VALUE` to `Long.MAX_VALUE`
- The class places constraints on `value`
  - Negative values are not allowed

Operations may have post conditions that tag state transitions as invalid

- Looking back at our **`Counter`** example
- If the current state of `Counter` is `17`
  - The *only* valid next state is `18`
  - When the next state is *derived from the current state*?
    - **Compound action**
- Not all operations impose state transition constraints
  - For e.g., if a variable tracks current temperature? Previous state doesn't impact current state

Constraints and synchronization requirements

- If certain states are invalid?
  - Underlying state variables should be **encapsulated**
    - If not, client code can put it in an *inconsistent* state
- If an operation has invalid state transitions?
  - It must be made **atomic**

The contents of this slide-set are based on the following references

- *Java Concurrency in Practice. Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Addison-Wesley Professional. ISBN: 0321349601/978-0321349606. [Chapters 1, 2, 3 and 4]*
- https://www.javaspecialists.eu/archive/Issue192b.html