Principles of Software Construction:
Concurrency, Part 2

Josh Bloch Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 5a due now
- You will get early feedback tomorrow!
  - Thank your TAs
- 2nd midterm exam returned today, after class

Outline

I. "It's bigger on the outside" exam question
II. Static Analysis – (I should covered this earlier)
III. `Wait/Notify` – primitives for cooperation
IV. The dangers of over-synchronization

Specification

```
/**
 * Returns an immutable list consisting of n consecutive
 * copies of the elements in the specified list. The
 * returned list logically contains n * source.size()
 * elements (as reported by its size method), but its
 * memory consumption does not depend on the value of n.
 *
 * @param n      the number of "virtual copies" of source in result
 * @param source the elements to appear repeatedly in result
 * @throws       IllegalArgumentException if n < 0
 * @throws       NullPointerException if source is null
 */
 public static <T> List<T> nCopiesOfList(int n, List<T> source) { }
```

Hint given: use AbstractList

```
/**
 * This class provides a skeletal implementation of the List
 * interface to minimize the effort required to implement it.
 * To implement an unmodifiable list, you need only to extend this
 * class and provide implementations of get(int) and size().
 */
public abstract class AbstractList<E> implements List<E> {
    protected AbstractList() { }

    /**
     * Returns the element at the specified position in this list.
     *
     * @throws IndexOutOfBoundsException if index is out of range
     * (index < 0 || index >= size())
     */
    public abstract E get(int index);

    /** Returns the number of elements in this list. */
    public abstract int size();
}
```

The **entire** solution

```
public static <T> List<T> nCopiesOfList(int n, List<T> source) {
    if (n < 0)
        throw new IllegalArgumentException("n < 0: " + n);

    return new AbstractList<T>() {
        private final List<T> src = new ArrayList<>(source);
        private final int size = n * src.size(); // Optimization

        public T get(int index) {
            if (index < 0 || index >= size)
                throw new IndexOutOfBoundsException();
            return src.get(index % src.size());
        }

        public int size() { return size; }
    };
}
```

Another optimization

It's nice to share!

```
public static <T> List<T> nCopiesOfList(int n, List<T> source) {
    if (n < 0)
        throw new IllegalArgumentException("n < 0: " + n);

    List<T> src = new ArrayList<>(source);  // Moved out of class
    int size = n * src.size();              //   "     "   "    "
    if (size == 0)
        return Collections.emptyList();

    return new AbstractList<T>() {
        // No explicit fields necessary! Remainder unchanged.
        ...
    }
}
```

Top level class is a bit wordier

Static factory omitted for brevity

```
class MultiCopyList<T> extends AbstractList<T> {
    private final List<T> src;
    private final int size;
    MultiCopyList(int n, List<T> source) {
        if (n < 0)
            throw new IllegalArgumentException("n < 0: " + n);
        src = new ArrayList<>(source);
        size = n * src.size();
    }

    public T get(int index) {
        if (index < 0 || index >= size)
            throw new IndexOutOfBoundsException();
        return src.get(index % src.size());
    }
    public int size() { return size; }
}
```

Common problems

- Problem specification
  - List must be "bigger on the outside" (virtual copies)
- Correctness
  - **Parameter validity checking**
- Immutability
  - Fields should be `final` and `private`
  - Need defensive copy of source
  - No explicit mutators
  - Class must not be extendable

Outline

I. "It's bigger on the outside" exam question
II. Static Analysis – (I should have covered earlier)
III. `Wait/Notify` – primitives for cooperation
IV. The dangers of over-synchronization

Remember this bug?

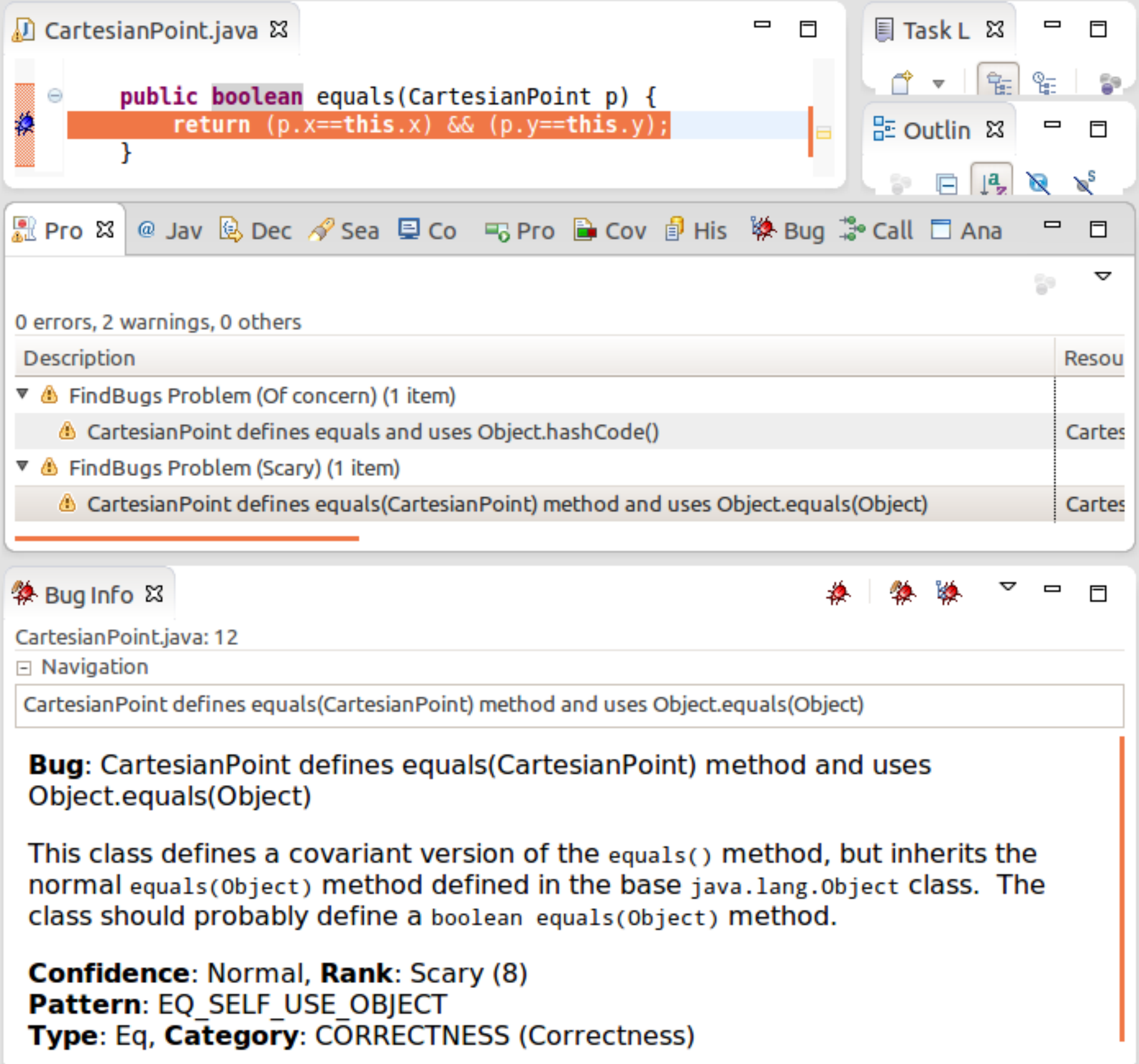
```
public class Name {
    private final String first, last;
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first; this.last = last;
    }
    public boolean equals(Name o) { // Accidental overloading
        return first.equals(o.first) && last.equals(o.last);
    }
    public int hashCode() {           // Overriding
        return 31 * first.hashCode() + last.hashCode();
    }
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(
            s.contains(new Name("Mickey", "Mouse")));
    }
}
```

Here's the fix

Replace the **overloaded** `equals` method with an **overriding** `equals` method

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Name))
        return false;
    Name n = (Name)o;
    return n.first.equals(first) && n.last.equals(last);
}
```

FindBugs


CartesianPoint.java
public boolean equals(CartesianPoint p) {
return (p.x==this.x) && (p.y==this.y);
}
Task L
Outlin
Pro
Jav
Dec
Sea
Co
Pro
Cov
His
Bug
Call
Ana
0 errors, 2 warnings, 0 others
Description
Resou
FindBugs Problem (Of concern) (1 item)
CartesianPoint defines equals and uses Object.hashCode()
Cartes
FindBugs Problem (Scary) (1 item)
CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)
Cartes
Bug Info
CartesianPoint.java: 12
Navigation
CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)
Bug: CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)
This class defines a covariant version of the equals() method, but inherits the normal equals(Object) method defined in the base java.lang.Object class. The class should probably define a boolean equals(Object) method.
Confidence: Normal, Rank: Scary (8)
Pattern: EQ_SELF_USE_OBJECT
Type: Eq, Category: CORRECTNESS (Correctness)

Static analysis

- Analyzing code without executing it
  - Also known as *automated inspection*
- Some tools looks for *bug patterns*
- Some formally verify specific aspects
- Typically integrated into IDE or build process
- Type checking by compiler is static analysis!

Static analysis: a formal treatment

- Static analysis is the systematic examination of an abstraction of a program's state space
- By abstraction we mean
  – Don't track everything!
  – Consider only an important attribute

	Error exists	No error exists
Error Reported	True positive (correct analysis result)	False positive (annoying noise)
No Error Reported	False negative (false confidence)	True negative (correct analysis result)

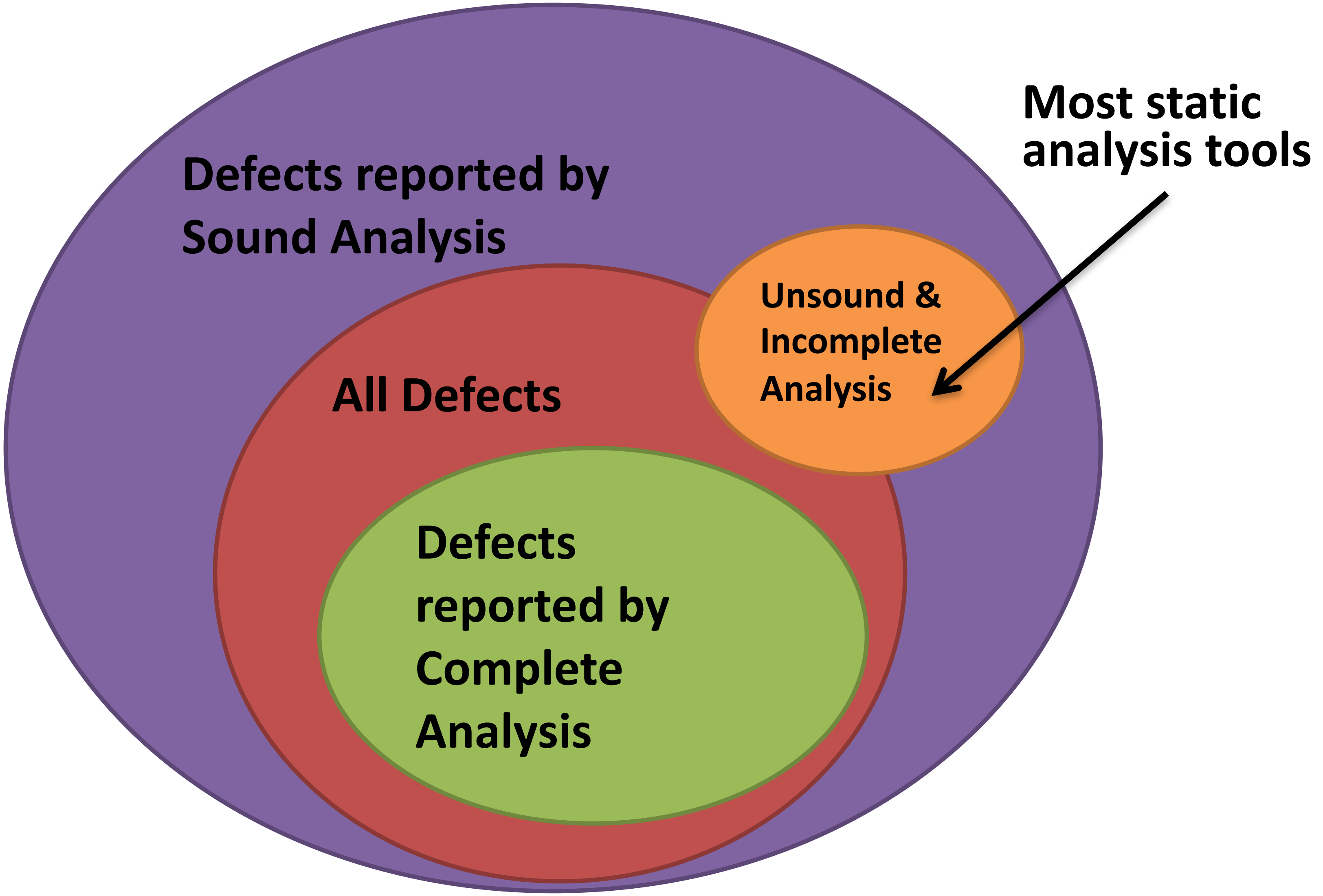
Results of static analysis can be classified as

- **Sound:**
  - Every reported defect is an actual defect
    - **No false positives**
  - Typically underestimated
- **Complete:**
  - Reports all defects
    - **No false negatives**
  - Typically overestimated

The bad news: Rice's theorem

- There are limits to what static analysis can do
- Every static analysis is necessarily incomplete, unsound, or undecidable

> **“Any nontrivial property about the language recognized by a Turing machine is undecidable.”**
>
> Henry Gordon Rice, 1953


Most static
analysis tools
Defects reported by
Sound Analysis
Unsound &
Incomplete
Analysis
All Defects
Defects
reported by
Complete
Analysis

Back to our regularly scheduled programming – concurrency!

Key concepts from Tuesday…

- `Runnable` interface represents work to be done
- To create a thread: `new Thread(Runnable)`
- To start thread: `thread.start();`
- To wait for thread to finish: `thread.join();`
- One `sychronized` static method runs at a time
- `volatile` – communication sans mutual exclusion
- ***Must*** synchronize access to shared mutable state
  - Else program will suffer safety and liveness failures

Pop quiz – what's wrong with this?

It's from last lecture, but I broke it

```
public class StopThread {
    private static boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }
    private static boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested())
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
```

Answer – you must synchronize writes **and reads!**

```
public class StopThread {
    private static boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }
    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested())
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
```

Outline

I. "It's bigger on the outside" exam question
II. Static Analysis – (I should covered this earlier)
III. `Wait/Notify` – primitives for cooperation
IV. The dangers of over-synchronization

The basic idea is simple…

- State (fields) protected by lock (`synchronized`)
- Sometimes, thread can't proceed till state is right
  - So it waits with `wait`
  - Automatically drops lock while waiting
- Thread that makes state right wakes waiting thread(s) with `notify`
  - Waking thread must hold lock when it calls `notify`
  - Waiting thread automatically gets lock when woken

But the devil is in the details
Never invoke wait outside a loop!

- Loop tests condition before and after waiting
- Test before skips wait if condition already holds
  - Necessary to ensure **liveness**
  - Without it, thread can wait forever!
- Testing after waiting ensure **safety**
  - Condition may not be true when thread wakens
  - If thread proceeds with action, it can destroy invariants!

All of your waits should look like this

```
synchronized (obj) {
    while (<condition does not hold>) {
        obj.wait();
    }

    ... // Perform action appropriate to condition
}
```

Why can a thread wake from a `wait` when condition does not hold?

- Another thread can slip in between `notify` & wake
- Another thread can invoke `notify` accidentally or maliciously when condition does not hold
  - This is a flaw in java locking design!
  - Can work around flaw by using private lock object
- Notifier can be liberal in waking threads
  - Using `notifyAll` is good practice, but causes this
- Waiting thread can wake up without a `notify`(!)
  - Known as a *spurious wakeup*

Example: read-write locks (API)

Also known as shared/exclusive mode locks

```
private final RwLock lock = new RwLock();

lock.readLock();
try {
    // Do stuff that requires read (shared) lock
} finally {
    lock.unlock();
}

lock.writeLock();
try {
    // Do stuff that requires write (exclusive) lock
} finally {
    lock.unlock();
}
```

Example: read-write locks (Impl. 1/2)

```
public class RwLock {
    // State fields are protected by RwLock's intrinsic lock

    /** Num threads holding lock for read. */
    private int numReaders = 0;

    /** Whether lock is held for write. */
    private boolean writeLocked = false;

    public synchronized void readLock() throws InterruptedException {
        while (writeLocked) {
            wait();
        }
        numReaders++;
    }
```

Example: read-write locks (Impl. 2/2)

```
    public synchronized void writeLock() throws InterruptedException {
        while (numReaders != 0 || writeLocked) {
            wait();
        }
        writeLocked = true;
    }

    public synchronized void unlock() {
        if (numReaders > 0) {
            numReaders--;
        } else if (writeLocked) {
            writeLocked = false;
        } else {
            throw new IllegalStateException("Lock not held");
        }
        notifyAll(); // Wake any waiters
    }
}
```

Caveat: `RwLock` is just a toy!

- It has poor fairness properties
  - Readers can starve writers!
- `java.util.concurrent` provides an industrial strength `ReadWriteLock`
- More generally, avoid `wait/notify`
  - In the early days it was all you had
  - Nowadays, higher level concurrency utils are better

Outline

I. "It's bigger on the outside" exam question
II. Static Analysis – (I should covered this earlier)
III. `Wait/Notify` – primitives for cooperation
IV. The dangers of over-synchronization

Broken Work Queue (1)

```
public class WorkQueue {
    private final Queue<Runnable> queue = new ArrayDeque<>();
    private boolean stopped = false;
    public WorkQueue() {
        new Thread(() -> {
            while (true) { // Main loop
                synchronized (queue) { // Locking on private obj.
                    try {
                        while (queue.isEmpty() && !stopped)
                            queue.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                    if (stopped) return;  // Causes thread to end
                    queue.remove().run(); // BROKEN - LOCK HELD!
                }
            }
        }).start();
    }
```

Broken Work Queue (2)

```
Broken Work Queue (2)
    public final void enqueue(Runnable workItem) {
        synchronized (queue) {
            queue.add(workItem);
            queue.notify();
        }
    }
    public final void stop() {
        synchronized (queue) {
            stopped = true;
            queue.notify();
        }
    }
}
```

Perverse use that demonstrates flaw

```java
public static void main(String[] args) {
    WorkQueue wq = new WorkQueue();

    // Enqueue task that starts thread that enqueues task...
    wq.enqueue(() -> {
        Thread t = new Thread(() -> {
            wq.enqueue(() -> { System.out.println("Hi Mom!"); });
        });
        t.start();

        // ...and waits for thread to finish
        try {
            t.join();
        } catch (InterruptedException e) {
            throw new AssertionError(e);
        }
    });
}
```

Luckily, it's easy to fix the deadlock

```
public WorkQueue() {
    new Thread(() -> {
        while (true) { // Main loop
            Runnable task = null;
            synchronized (queue) {
                try {
                    while (queue.isEmpty() && !stopped)
                        queue.wait();
                } catch (InterruptedException e) {
                    return;
                }
                if (stopped) return;  // Causes thread to terminate
                task = queue.remove();
            }
            task.run(); // Fixed! "Open call" (no lock held)
        }
    }).start();
}
```

Never do callbacks while holding lock

- It is *over-synchronization*
- We saw it deadlock
- And it can do worse!
  – If the callback goes back into the module holding the lock, it will not block, and can damage invariants!
- So always drop any locks before callbacks
  – You may have to copy the callbacks under lock

Summary

- Validate input parameters
- **Never use wait outside of a while loop!**
  - Think twice before using it at all
- **Neither an under- nor an over-synchronizer be**
  - Under-synchronization causes safety (& liveness) failures
  - Over-synchronization causes liveness (& safety) failures