

Lecture 3 Thread Safety

9/9/2012

©2012, 2013 University of Maryland

Anomaly from Lecture 1

- IncThread.java

```
public class BadCounter implements Runnable {

    private static int count = 0; // Shared variable
    private String name = "";    // Name of thread

    BadCounter (String name) { this.name = name; }

    public void run () {
        int c = count;
        System.out.println (name + " read shared = " + c);
        c++;
        count = c;
        System.out.println (name + " assigned to shared: " + c);
    }
}
```
- Main thread created two instances, T1 and T2, and started both

9/9/2012

©2012, 2013 University of Maryland

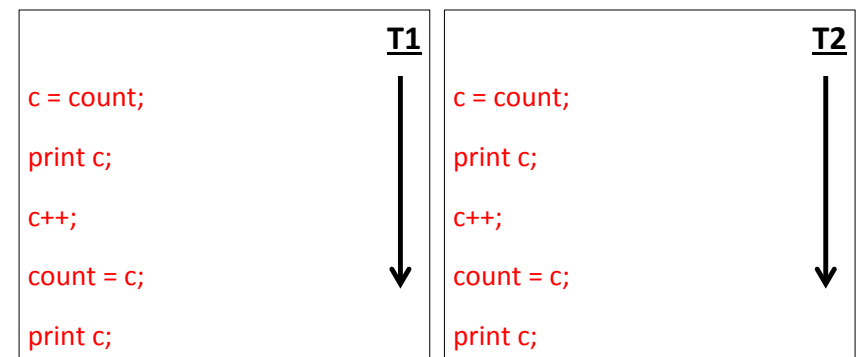
Thread Anomalies

- Scheduler determines when threads execute
 - Thread computation can be interleaved on a single processor, or
 - Threads computations can be on different processors, or
 - Some combination of both
- Programmer can have some influence via `yield()`, `setPriority()`, etc.
- But most decisions are outside user control, leading to possibilities for
 - Nondeterminism
 - Interference*: threads overwrite each other's work

9/9/2012

©2012, 2013 University of Maryland

Two Threads



- Different schedules can leave `count = 2`, `count = 1`
- This is an example of a *data race*

9/9/2012

©2012, 2013 University of Maryland

Data Races and Race Conditions

- A **data race** occurs when the same memory location can be accessed simultaneously by two threads, with at least one of accesses a write.
- They “seem bad” ... but why?
 - In previous example, if it does not matter if count is 1 or 2, then is there an error?
 - On the other hand, if count should only be 2, then there is an error
- A **race condition** occurs when a program’s correctness depends on scheduling decisions
 - If the correct outcome of the previous example is count = 2, then the data race induces a race condition
 - If the correct outcome is count = 1 or count = 2, then there is no race condition!

9/9/2012

©2012, 2013 University of Maryland

Correctness?

- Definition of race condition mentions program correctness
- We will adopt a class-based view:
A class is correct if it satisfies its specification
- So what is a “class specification”?

9/9/2012

©2012, 2013 University of Maryland

Class Specifications

- Classes are used to define objects
- Classes contain static members
- Objects contain instance members
- Some members are fields, while others are methods
- Classes generally enforce consistency constraints on static, instance members
 - Field values should be “consistent”
 - Methods should preserve consistency, compute the right thing

9/9/2012

©2012, 2013 University of Maryland

Example: Points and Lines

- Point.java

```
public class Point {  
  
    private final double x; private final double y;  
  
    Point (int x, int y) { this.x = x; this.y = y; }  
  
    double getX () { return x; }  
    double getY () { return y; }  
}
```
- BuggyLine.java

```
public class BuggyLine {  
  
    private Point p1; private Point p2;  
  
    BuggyLine (Point p1, Point p2) { this.p1 = p1; this.p2 = p2; }  
  
    public double slope () {  
        return ((p1.getY() - p2.getY()) / (p1.getX() + p2.getX()));  
    }  
}
```

9/9/2012

©2012, 2013 University of Maryland

Notions of Consistency for Line?

- Would like to know that points are different!
- **Invariants** capture notion of consistency
 - Invariants describe properties that must always hold among instance variables
 - They reflect relationships you can “rely on”
- Here is an invariant for lines:
p1 and p2 must be different points
- Is BuggyLine class correct? No!
 - Constructor does not check that points are different
 - So constructor can construct objects violating invariant

9/9/2012

©2012, 2013 University of Maryland

Corrected Line Class

- Line.java – change constructor to:

```
Line (Point p1, Point p2) throws IllegalArgumentException {
    if ( (p1.getX() != p2.getX()) ||
        (p1.getY() != p2.getY())
    ) {
        this.p1 = p1;
        this.p2 = p2;
    }
    else {
        throw new IllegalArgumentException (
            "Points to Line constructor must differ: "
            + p1.toString() + "given twice.");
    }
}
```
- Note that when invariant violation is detected, no updating is performed, and exception is thrown!

9/9/2012

©2012, 2013 University of Maryland

Now it's almost correct ...

- Some would say yes ...
- ... and yet there is one more issue: division by zero!
 - If p1, p2 have the same x-value, then the slope calculation involves dividing by 0
 - This can throw a run-time exception!
- This is not a consistency issue among fields, but instead a property of methods.

9/9/2012

©2012, 2013 University of Maryland

Class Specifications: Preconditions / Postconditions / Exception Conditions

- To specify the behavior of methods, need
 - **Preconditions:** what should hold of inputs, fields in order to ensure correct termination
 - **Postconditions:** what will hold when method exits normally
 - **Exceptions:** what happens when precondition violated

9/9/2012

©2012, 2013 University of Maryland

Class spec: slope method

- Specification should indicate that
 - if points form a vertical line, then method will throw an exception; otherwise, slope is returned
 - Header for method should be changed to reflect this

9/9/2012

©2012, 2013 University of Maryland

Corrected slope() Method

- CorrectedLine.java

```
// Precondition: p1, p2 do not form vertical line
// Postcondition: return slope of line thru p1, p2
// Exception: if p1, p2 form vertical line, throw
//      ArithmeticException

public double slope () throws ArithmeticException {
    return ((p1.getY() - p2.getY()) / (p1.getX() +
p2.getX()));
}
```

9/9/2012

©2012, 2013 University of Maryland

Class Specifications: Field invariants

- These are relationships between fields that hold before and after every method invocation of the class
 - May temporarily violate invariant during call to method, as long as no calls to other methods, which assume the invariant, are made in the meantime

9/9/2012

©2012, 2013 University of Maryland

Class Correctness

- A class is correct when it satisfies its specification
 - All pre/post/exception conditions are satisfied
 - Fields always maintain invariant at calls to and from class methods
- Note: Specifications (including pre/post/exception) should be in documentation
 - Ideally, they can be tested or verified directly
 - Matter of ongoing research

9/9/2012

©2012, 2013 University of Maryland

Establishing Correctness in the Sequential Case

- Check that each constructor returns an object satisfying the invariant
- Check that each method leaves the invariant true if it starts with the invariant true
- Check preconditions / postcondition / exceptions
- Works because of validity of procedural abstraction!
 - Method call can be viewed as one *atomic* operation that is equivalent to executing body of method
 - So analyzing correctness can be done on a method-by-method basis

9/9/2012

©2012, 2013 University of Maryland

Problems with Threads

- Even if a class is correct with respect to a specification, threads can break invariants!
- This happens because:
 - A class can be correct even though methods might break the invariants in the middle of their execution
Methods only have to make sure the invariants hold when they terminate.
 - Concurrency breaks procedural abstraction!
 - One thread can see the intermediate results of another thread's execution
 - If the second thread is in the middle of a method call, the class's invariants might not be true
 - The first thread then gets an inconsistent view of the corresponding object

9/9/2012

©2012, 2013 University of Maryland

Example: BadCounter Revisited

- IncThread.java

```
...
private static int count = 0; // Shared variable
...
public void run () {
    int c = count;
    c++;
    count = c;
}
```
- Specification
 - Invariant: `count` records the number of times `run()` has been invoked
 - Precondition / postcondition / exception for `run()`: no requirements
- BadCounter is correct (sequentially!)
 - Initially, invariant is true, since `count == 0`
 - `run()` increments `count`, so invariant is true when `run()` finishes if it is true when `run()` starts
- There are erroneous runs when there are multiple threads!
 - Until `run()` increments `count` invariant is not true
 - Another thread can then read an inconsistent value of `count`!

9/9/2012

©2012, 2013 University of Maryland

Thread Safety

A correct class is *thread-safe* if every execution of any threaded application using the class preserves the specification's invariants and method specifications

- Thread safety only makes sense if you have a class specification!
- This fact is crucial but often overlooked
 - Default view: a class is thread-safe if its methods execute atomically (i.e., their multithreaded behavior can be boiled down to single-threaded behavior)

9/9/2012

©2012, 2013 University of Maryland

Example Re-revisited

- Suppose BadCounter invariant is changed to:
The value of `count` is $\leq$ the number of times `run()` is executed
- Then BadCounter is thread-safe!
 - Every value any thread might read of `count` is $\leq$ the number of times `run()` has been invoked
 - Every thread increments `count`
 - Even though there is a data race, the class can be used as is in a threaded application, *for this specification*
- Again: thread-safety is a property of a class **and its specification**, not just of a class

9/9/2012

©2012, 2013 University of Maryland

Recap

- A class can be correct with respect to its specification and still not be thread-safe
- Why?
 - The methods in a correct class will preserve the specifications invariants before and after each executes
 - During execution of a method, the invariants might not be true
 - In a multi-threaded application, another thread might see this inconsistent state of an object, since procedural abstraction is violated!
- Implication: if a class is not thread-safe, it cannot be counted on to be correct in a multi-threaded execution

9/9/2012

©2012, 2013 University of Maryland

Fixing Thread Safety Problems

- Thread-safety is assured for immutable objects
 - In immutable objects, the fields never change after construction
 - Enforce this using the **final** field qualifier
 - So if the fields of an object satisfy an invariant after it is built, it will never violate the invariant
- So: when feasible, use immutable objects

9/9/2012

©2012, 2013 University of Maryland

Implementing Immutable Points

- Immutable: `Point.class`

```
public class Point {  
    public final double x;  
    public final double y;  
    public Point (double x, double y)  
        { this.x = x; this.y = y; }  
}
```

9/9/2012

©2012, 2013 University of Maryland

Implementing Points

- Mutable: MutablePoint.class

```
public class MutablePoint {  
    private double x;  
    private double y;  
    public MutablePoint (double x, double y)  
    { this.x = x; this.y = y; }  
    double getX() { return x; }  
    void setX(double z) { x = z; }  
    ... // same for Y  
}
```

This class is not be thread safe, but could be made so by removing setters (but then it's effectively immutable)

9/9/2012

©2012, 2013 University of Maryland

Fixing Thread-Safety Problems: Locks

- Thread-safety problems are often related to methods inducing invariant errors while “in flight”
 - The invariant errors are fixed before the method terminates
 - If another thread sees this intermediate erroneous data, it can use it without realizing it.
- The issue: procedural abstraction
 - We would like to think of method calls as *atomic*, i.e. as either not having started or having finished, like single machine instructions
 - This perspective is valid in a sequential program
 - It is not in a multi-threaded program
- A solution: use *locks* to give illusion of atomicity!

9/9/2012

©2012, 2013 University of Maryland

Lock Fundamentals

- Examples of a *concurrency-control* primitive
 - As the name suggests, concurrency-control primitives are intended to control concurrency!
 - The idea: eliminate the possibility of concurrency while critical operations are taking place
- A lock is a data structure
 - Two states: *locked*, *unlocked*
 - Two operations: *acquire*, *release*
 - acquire: block execution until the state of the lock is unlocked, then set state to locked.
 - release: set status of lock to unlocked
 - Both operations are *atomic*
 - Variations:
 - Releasing a lock whose status is unlocked may or may not throw an exception
 - Some locks have more states (e.g. *read-locked*)

9/9/2012

©2012, 2013 University of Maryland

Using Locks to Fix Thread-Safety Issues

- Idea
 - Associate lock(s) with classes
 - Methods must acquire appropriate locks before performing internal operations that may violate invariants
 - Methods release locks when invariant is restored
- This ensures that multiple threads cannot see intermediate changes that methods make to fields during execution!

9/9/2012

©2012, 2013 University of Maryland

Locks in Java

- Several types
 - Intrinsic / monitor locks
 - Various classes whose objects are locks
- We will first study intrinsic / monitor locks (both terms are used)

9/9/2012

©2012, 2013 University of Maryland

Intrinsic / Monitor Locks

- Every object in Java has a lock associated with it, called the *monitor (lock)* or *intrinsic lock*
- No explicit acquire / release operations; rather the state of an intrinsic lock is modified using *synchronized* blocks
 - Basic form:
`synchronized (obj) { statements }`
 - Semantics
 - Acquire intrinsic lock of obj
 - Execute statements
 - Release intrinsic lock of obj when block exits (terminates, throws an exception, breaks, etc.)

9/9/2012

©2012, 2013 University of Maryland

Fixing BadCounter.java

- Counter.java

```
public class Counter implements Runnable {  
  
    private static int count = 0;  
    static Object lock = new Object ();    // Lock must be static!  
    ...  
    public void run () {  
        synchronized (lock) {  
            int myShared = shared;  
            myShared++;  
            shared = myShared;  
        }  
    }  
}
```
- The specification invariant that `shared` is the number of invocations of `run()`.
- The class-wide object `lock` is used to “guard” the part of `run()` where the invariant is violated (i.e. where `shared` is not yet updated).
- When one thread is executing its `synchronized` block, all other threads are waiting outside theirs.
- After `run` updates `shared` the invariant has been restored, and the lock can be released.

9/9/2012

©2012, 2013 University of Maryland

Synchronized Instance Methods

- In many cases we want entire methods to occur atomically
- Java provides the following short-hand for this by allowing methods to be declared `synchronized`
 - E.g.

```
public synchronized void setP1 (Point p1) {  
    this.p1 = p1;  
}
```
 - This is an abbreviation for the following, since the method is an instance method

```
public boolean setP2 (Point p2) {  
    synchronized (this) {  
        this.p2 = p2;  
    }  
}
```

9/9/2012

©2012, 2013 University of Maryland

Synchronized Static Methods

- Static (class) methods may also be synchronized
 - For example, could add following method to `SyncIncThread`

```
public synchronized static void incShared () {  
    ++shared;  
}
```
 - What object's intrinsic lock is used in this case?
 - Answer: the class object associated with the relevant class!
 - In this case, here is equivalent code:

```
public static void altIncShared () {  
    synchronized (SyncIncThread.class) {  
        ++shared;  
    }  
}
```

9/9/2012

©2012, 2013 University of Maryland

Reentrant Locking

- Intrinsic locks are reentrant!
 - If a thread acquires an intrinsic lock, it can acquire it again without blocking
 - A thread with multiple acquisitions on an intrinsic lock frees it only when the number of releases equals the number of acquisitions
- Huh?
 - Consider following code used to do atomic updating of a bounded counter

```
public synchronized boolean isMaxed () {  
    return (value == upperBound);  
}  
  
public synchronized void inc () {  
    if (!isMaxed()) ++inc;  
}
```
 - Without reentrant locking, every call to `inc()` would block forever!

9/9/2012

©2012, 2013 University of Maryland

Example: Bounded Counter Class

- `BoundedCounter.java`: a correct, but not thread-safe class.
- How do we make it thread safe?

9/9/2012

©2012, 2013 University of Maryland

BoundedCounter.java (some method specs elided)

```
public class BoundedCounter {  
    private int value = 0;  
    private int upperBound = 0;  
  
    //INVARIANT: in all instances 0 <= value <= upperBound  
  
    //Precondition: argument must be >= 0  
    //Postcondition: object created  
    //Exception: If argument < 0, IllegalArgumentException thrown  
    BoundedCounter (int upperBound) throws IllegalArgumentException {  
        if (this.upperBound >= 0) this.upperBound = upperBound;  
        else throw new IllegalArgumentException  
            ("Bad argument to BoundedCounter: " + upperBound + "; must be >= 0");  
    }  
  
    //Precondition: none  
    //Postcondition: current value returned  
    //Exception: none  
    public int current () { return value; }  
  
    public void reset () { value = 0; }  
  
    public boolean isMaxed () { return (value == upperBound); }  
  
    //Precondition: none  
    //Postcondition: increment value if not maxed; otherwise, do nothing.  
    //Exception: none  
    public void inc () { if (!isMaxed()) ++value; }  
}
```

9/9/2012

©2012, 2013 University of Maryland

Design Considerations

- Whose job is it to enforce correctness?
 - Class? Or User
 - In `BoundedCounter.java`, could have incremented `inc` as:

```
public void inc () { ++value; }
```

 - This would put burden on maintaining correctness on user
 - But it is more efficient
 - A better perspective
 - Class should enforce correctness
 - Class designer, though, can choose what notion of correctness is
 - In the `inc` example, invariant could be relaxed to say that only correctness criterion is $0 \leq \text{value}$
- A similar question: whose job is it to enforce thread safety
 - So far: we have said class
 - A common alternative: it is user's job to implement correct synchronization (reason: performance!)
 - Argument: The "better perspective" comment applies here also!
 - Commit to a notion of correctness
 - Make class thread-safe with respect to that notion
 - Not what Java designers have done, e.g., with `Collection` classes