

# Learning from a Bad Example

---

CSCI 5828: Foundations of Software Engineering  
Lecture 26 — 11/18/2014

# Goals

---

- Demonstrate techniques on dealing with shared mutability
  - Show an example where multiple threads access an “EnergySource”
    - it is poorly designed and has many concurrency-related defects
  - We will refactor the program
    - until we’ve tamed shared mutability and have thread safe code
- The examples in this lecture come from the excellent book: Programming Concurrency on the JVM from Pragmatic Programmers
  - Source Code is available here (I’m not allowed to distribute it):
    - <https://pragprog.com/book/vspcon/programming-concurrency-on-the-jvm#links>

# Shared Mutability

---

- We've been talking about the dangers of shared mutability a lot this semester
  - When we use the word “danger”, we mean that the code has the potential to be unstable
    - there may be deadlocks hiding in the code
    - there may be race conditions
      - leading the values of shared variables to behave unpredictably
  - The danger, then, is that you spend a lot of time trying to debug this code
- If you work with concurrent code that uses shared mutability, then you need to be able to identify the types of code structures that can lead to problems
  - and learn how to eliminate them

# Controlling your variables (I)

---

- In a shared mutability design, you need to have a clear sense of which threads can access which variables
  - You can then design into the program the ways in which these variables can be protected using the synchronization constructs discussed in previous lectures
    - we want to avoid being heavy-handed with the use of *synchronized*
    - Instead, we should look for ways to make use of the Lock interface from `java.util.concurrent` for fine-grained access control
- Note: this example is written in Java but its lessons are more general and will apply to other languages that provide access to low-level threading primitives

# Controlling your variables (II)

---

- If you ensure that all mutable variables are either
  - accessed by only one thread
  - or accessed by multiple threads using Locks
- then you can be (more) confident that your program will be free from thread-related dangers;
- If, however, a thread can access one of these variables
  - without passing through the protections you put in place
  - then the variable is said to have “**escaped**” and you are open to race conditions and non-stable code

# Controlling your variables (III)

---

- A complex aspect to this analysis is the different ways in which values can escape
- Imagine Class A creates an instance of a collection class
- and Class A ensures that the collection is accessed in a thread safe way
  - the instance variable is private
  - all methods that update the collection make use of the Lock interface

# Controlling your variables (IV)

---

- All of these protections are ***null and void*** if one of Class A's methods returns a pointer to the collection
  - `public List getRecords() { return records; }`
- At this point, Class A cannot protect this collection
  - Any class that calls this method has a pointer to the collection and can directly update it without using Class A
  - Class B might call `getRecords()` and make its handle to Class A's collection class visible to other threads
    - At this point the *records* variable has **escaped** and is no longer protected
      - in other words, it becomes vulnerable to race conditions

# Controlling your variables (V)

---

- The same is true if Class A decides to pass records to some other method as input
  - `{ ... ; records = foo.update(records); ... }`
- If foo decides to keep a pointer to all of the collections passed to its `update()` method, then records has escaped and all of Class A's protections are, again, useless
- Finally, if a class has *public instance variables* or *public static variables* then any of these variables can easily escape
  - Code can simply reach in and update the instances without the host class knowing about it



# Controlling your variables (VI)

---

- By now it should be clear that the visibility specifications
  - *public, protected, private*
- have ***nothing*** to do with protecting a variable from access by multiple threads
  - The values pointed at by “private” variables can be passed to other classes which can then point at those values
    - *stripping them of their protection*
- If you have a very small program, then you should be able to conduct the analysis of whether a variable has escaped its protection
  - but as your programs get larger, it becomes more and more difficult to keep track of all the ways a variable is accessed
    - and this is what causes the pain of debugging shared mutability designs

# Example: Step 1

---

- To demonstrate these issues, let's look at a “bad example” of shared mutability design
  - EnergySource is a resource that maintains a certain amount of energy
    - Clients can make use of this energy by calling `useEnergy()` and specifying how much energy they need
    - Internally, EnergySource starts a thread that will slowly replenish the EnergySource if its energy is consumed
- I have augmented this example with client code that makes use of the EnergySource
  - a monitor prints out the current level of the source on a periodic basis
  - consumers read the current level and then consume a random amount of energy

**DEMO**

# Discussion

---

- EnergySource is a HORRIBLE instance of concurrent design
  - it does pretty much **everything wrong**
    - the internal thread is started incorrectly
      - because the internal thread has the potential to access the class before it has been initialized
  - its internal instance variable is mutable and unprotected (race condition)
  - the internal thread loops forever until a boolean flag changes state
    - changing the boolean flag may not cross the memory barrier
    - thread is stuck endlessly looping and sleeping, consuming resources
  - one internal thread is created per instance; threads are expensive!

## Step 2: Fix creation of internal thread (I)

---

- We do not want to create threads in our constructor
  - If we call start() on those threads in the constructor
    - they may start accessing our object before it exits the constructor!
    - as a result, they will be accessing the object in an inconsistent state
- We want the call to the constructor to complete before any other object accesses the energy source
  - This allows us to make sure the energy source is in a consistent state
    - then, we can design the class such that each method
      - starts in a consistent state, performs its service, and ensures that it is leaving the object in a consistent state before it returns

## Step 2: Fix creation of internal thread (II)

---

- To address this problem, we make use of a factory pattern
  - The *constructor* of the class is made **private**
    - This prevents other classes from creating instances of EnergySource
  - A *private instance method* (init) is used to create the internal thread
  - A *public static method* is created to provide an instance of EnergySource
    - a static “factory” method
      - creates an instance of the class (constructor will fully initialize class)
      - calls the init method to start the thread
      - returns the instance to the caller

**DEMO**

## Step 3: Get rid of internal thread

---

- The internal thread was created so that periodically the EnergySource would be replenished
  - The author thought that a thread was the only way to accomplish this
- However, Java has a class called Timer that can be used to fire events on a periodic basis
  - but creating one Timer per instance of EnergySource is wasteful
- Instead, we'll use a ScheduledThreadPoolExecutor
  - It can allocate a fixed number of threads and then reuse them to handle the task of replenishing multiple energy sources
  - The thread pool will be static, so it will be shared across all instances

**DEMO**

# Discussion (I)

---

- As a result of adding an instance of `ScheduledExecutorService` to `EnergySource`
  - the private `init()` method is changed such that
    - instead of creating a thread
    - it now creates an instance of a task that it submits to the thread pool
    - the task simply calls `replenish`
    - we ask that the task be run every second
  - the `replenish` method is now simplified: check level, increment if needed
    - no more loop, no more sleeping
  - the boolean flag goes away
    - the request to stop the energy source, now just cancels the task

# Discussion (II)

---

- One complication
  - With the addition of a static thread pool, we need to come up with a way to shut the thread pool down
    - We have two options
      - Add a static shutdown ( ) method to EnergySource
        - Call this method when its time to shut our program down
      - Configure the pool with a thread factory that sets all threads to be daemon threads
  - The former is simpler (at least for this example program)



## Step 4: Ensure visibility

---

- Our shared mutable instance variable (level) is not protected
  - changes to it may not pass the memory barrier
  - race conditions exist since multiple threads may try to read the value of level at the same time and then try to consume energy based on that value
    - Our Consumer thread has a transaction problem in this regard that we'll fix later
- We'll start by fixing this problem by adding the synchronized keyword to all methods that access the shared instance variable
  - This protects the variable but greatly reduces performance
    - If we have a lot of threads accessing EnergySource, most of them will be blocked while one thread is inside one of these methods

# Step 5: Enhance Concurrency

---

- Use of the synchronized keyword is too restrictive in terms of performance
  - We'll change our instance variable from a long to an AtomicLong
  - We can then get rid of our synchronized keyword and allow the threads to access the energy source as fast as possible
    - The AtomicLong will ensure that the minimum amount of synchronization is used to protect its value from multiple threads
- Note: use of AtomicLong.compareAndSet(**expected**, **new**) in useEnergy()
  - a thread says “here is the value that I think is current;
  - if it is current, then change it to this **new** value
- Protects against situations where a thread reads a value and it gets updated before it can write a new value; the update fails, if it gets **expected** wrong

# We still need a transaction

---

- Even with these protections, our consumers still get into problems
  - Consumer 0 tries to consume 23: SUCCESS!
  - Consumer 2 tries to consume 94: FAIL!
  - Consumer 1 tries to consume 89: FAIL!
- Even though Consumer 0 had updated the EnergySource
  - Consumer 1 and Consumer 2 both read the level of EnergySource at the same time and tried to consume an invalid amount of energy
- We now need to address this problem with our consumers

# Step 6: Add a notion of transaction to consumer

---

- Our consumers are designed to
  - read the value of the energy source
  - use that value to generate a random amount of energy to consume
  - and then consume that amount of energy
- The problem?
  - they do not do this read/update in a transaction
  - as a result, they can all read the same amount at the same time and then all move on to consume different amounts, some of which will be invalid
- All of the work we've done in EnergySource does **not** solve this problem
  - We'll solve it via a **shared lock**; if we had more than one type of thread, we'd have to place this lock in EnergySource; for now, we will create it in Consumer

# Step 7: Fix the problem with replenish

---

- We do have a problem
  - even with the transaction, it's possible that the replenish task slips in between a Consumer's read and write, incrementing the value, and causing the Consumer's write to fail
  - This would manifest in the step06 program like this
    - Consumer 7 tries to consume 2: FAIL!
  - It's very hard to make this happen, but it's possible
- So, we need to share the lock between the consumers and the replenish task
  - We add a public lock to EnergySource and update Consumers to use that lock instead (deleting the lock inside of Consumer) and updating replenish() to use that lock as well

## Step 8: Update semantics of replenish

---

- The way the program is written currently, we consume the energy of the EnergySource very quickly
  - Let's allow replenish to do more than increment the level
    - It can do this safely since all consumers will be blocked during its update
  - Let's change the consumers to be more modest in their consumption
- We should now have a program in which the EnergySource stays at a reasonable level, rather than stuck down at one or two units constantly
  - Now, it will just be ***MOSTLY*** stuck down at one or two units

## Step 9: Ensure Atomicity

---

- The last change that the book makes is to add another mutable variable to `EnergySource`
  - This variable will track the number of times that the `EnergySource` is used
- The purpose of this change is to show that `AtomicLong` is insufficient to keep changes to two separate variables coordinated
  - Instead, we need a lock to ensure that both variables are updated in tandem
- We'll change our `Lock` to a `ReadWriteLock`, get rid of the `AtomicLong`, and update `Consumer`, `Monitor`, and the `replenish` task to make use of the new `ReadWriteLock`
  - Everything works fine and we get the maximum amount of concurrency that can occur, given our need to protect the two variables

# Summary

---

- Learned useful lessons about taming shared mutability
  - Do not create threads in constructors; create in static factory methods
  - Do not create arbitrary threads (replenish thread); use thread pools
  - Ensure updates to mutable variables cross memory barrier
  - Evaluate the granularity of locks to promote concurrency
    - avoid synchronized if at all possible
  - Ensure atomicity of multiple mutable variables via locks
- Note: the final program is thread safe and as performant as we can make it
  - unfortunately, the code is quite complex; an unavoidable aspect of the shared mutability approach to the design of concurrent software systems