Extracted from:

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors

This PDF file contains pages extracted from *Programming Concurrency on the JVM*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com .

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

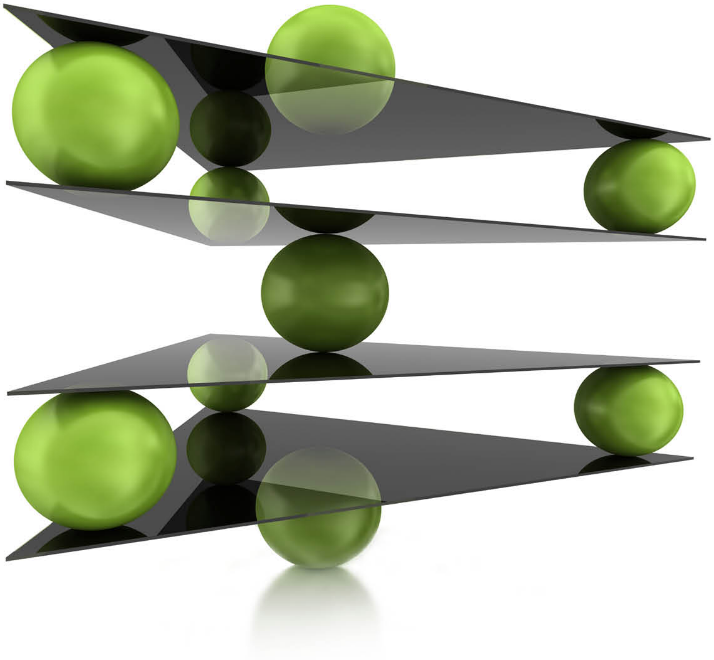
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The
Pragmatic
Programmers

Programming Concurrency on the JVM

*Mastering
Synchronization,
STM, and Actors*



Venkat Subramaniam

edited by Brian P. Hogan

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors

Venkat Subramaniam

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

Pragmatic
Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-76-0
Printed on acid-free paper.
Book version: P1.0—August 2011

To Mom and Dad, for teaching the values of integrity, honesty, and diligence.

I've discouraged shared mutability quite a few times so far in this book. You may ask, therefore, why I discuss it further in this chapter. The reason is quite simple: it's been the way of life in Java, and you're likely to confront legacy code that's using shared mutability.

I certainly hope you'll heavily lean toward isolated mutability or pure immutability for any new code, even in existing projects. My goal in this chapter is to help cope with legacy code—the menacing code you've soldiered to refactor.

5.1 Shared Mutability != public

Shared mutability is not restricted to `public` fields. You may be thinking "Gee, all my fields are `private`, so I have nothing to worry about," but it's not that simple.

A shared variable is accessed, for read or write, by more than one thread. On the other hand, a variable that's never accessed by more than one thread—ever—is isolated and not shared. Shared mutable variables can really mess things up if we fail to ensure visibility or avoid race conditions. It's rumored that shared mutability is the leading cause of insomnia among Java programmers.

Irrespective of access privileges, we must ensure that any value passed to other methods as parameters is thread safe. We must assume that the methods we call will access the passed instance from more than one thread. So, passing an instance that's not thread safe will not help you sleep better at night. The same concern exists with the references we return from methods. In other words, don't let any non-thread-safe references *escape*. See *Java Concurrency in Practice* [Goe06] for an extensive discussion of how to deal with escaping.

Escaping is tricky; we may not even realize until we closely examine the code that it's there. In addition to passing and returning references, variables may escape if we directly set references into other objects or into static fields. A variable may also escape if we passed it into a collection, like the `BlockingQueue` we discussed previously. Don't be surprised if the hair on the back of your neck stands up the next time you open code with mutable variables.

5.2 Spotting Concurrency Issues

Let's learn to identify the perils in shared mutability with an example and see how to fix those problems. We'll refactor a piece of code that controls a fancy energy source. It allows users to drain energy, and it automatically replenishes the source at regular intervals. Let's first glance at the code that's crying for our help:

Download tamingSharedMutability/originalcode/EnergySource.java

```
//Bad code
public class EnergySource {
  private final long MAXLEVEL = 100;
  private long level = MAXLEVEL;
  private boolean keepRunning = true;

  public EnergySource() {
    new Thread(new Runnable() {
      public void run() { replenish(); }
    }).start();
  }

  public long getUnitsAvailable() { return level; }

  public boolean useEnergy(final long units) {
    if (units > 0 && level >= units) {
      level -= units;
      return true;
    }
    return false;
  }

  public void stopEnergySource() { keepRunning = false; }

  private void replenish() {
    while(keepRunning) {
      if (level < MAXLEVEL) level++;

      try { Thread.sleep(1000); } catch(InterruptedException ex) {}
    }
  }
}
```

Identify the concurrency issues in the `EnergySource` class. There are a few easy-to-spot problems but some hidden treasures as well, so take your time.

Done? OK, let's go over it. The `EnergySource`'s methods may be called from any thread. So, the nonfinal `private` variable `level` is a shared mutable variable, but it's not thread safe. We have unprotected access to it, from a thread-safety viewpoint, in most of the methods. That leads to both the visibility concern—calling thread may not see the change, because it was not asked to cross the memory barrier—and race condition.

That was easy to spot, but there's more.

The `replenish()` method spends most of the time sleeping, but it's wasting an entire thread. If we try to create a large number of `EnergySources`, we'll get an `OutOfMemoryError` because of the creation of too many threads—typically the JVM will allow us to create only a few thousand threads.

The `EnergySource` breaks the class invariant.[1] A well-constructed object ensures that none of its methods is called before the object itself is in a valid state. However, the `EnergySource`'s constructor violated invariant when it invoked the `replenish()` method from another thread before the constructor completed. Also, `Thread`'s `start()` method automatically inserts a memory barrier, and so it escapes the object before its initiation is complete. Starting threads from within constructors is a really bad idea, as we'll discuss in the next section.

That's quite a few issues for such a small piece of code, eh? Let's fix them one by one. I prefer not to fix problems concurrently so I can focus on solving each in turn.

5.3 Preserve Invariant

We may be tempted to start threads from constructors to get background tasks running as soon as an object is instantiated. That's a good intention with undesirable side effects. The call to `start()` forces a memory barrier, exposing the partially created object to other threads. Also, the thread we started may invoke methods on the instance before its construction is complete.

An object should preserve its invariant, and therefore starting threads from within constructors is forbidden.

`EnergySource` is clearly in violation on this count. We could move the thread-starting code from the constructor to a separate instance method. However, that creates a new set of problems. We have to deal with method calls that may arrive before the thread-starting method is called, or a programmer may simply forget to call it. We could put a flag to deal with that, but that'd lead to ugly duplicated code. We also have to prevent the thread-starting method from being called more than once on any instance.

On the one hand, we shouldn't start threads from constructors, and on the other hand, we don't want to open up the instance for any use without fully creating it to satisfaction. There's gotta be a way to get out of this pickle.

---

[1] Class invariant is a condition that every object of the class must satisfy at all times—see *What Every Programming Should Know About Object-Oriented Design* [Pag95] and *Object-Oriented Software Construction* [Mey97]. In other words, we should never be able to access an object in an invalid state.

The answer is in the first item in *Effective Java* [Blo08]: "Consider static factory methods instead of constructors." Create the instance in the static factory method and start the thread before returning the instance to the caller.

Download tamingSharedMutability/fixingconstructor/EnergySource.java

```
//Fixing constructor...other issues pending
private EnergySource() {}

private void init() {
  new Thread(new Runnable() {
    public void run() { replenish(); }
  }).start();
}

public static EnergySource create() {
  final EnergySource energySource = new EnergySource();
  energySource.init();
  return energySource;
}
```

We keep the constructor private and uncomplicated. We could perform simple calculations in the constructor but avoid any method calls here. The `private` method `init()` does the bulk of the work we did earlier in the constructor. Invoke this method from within the static factory method `create()`. We avoided the invariant violation and, at the same time, ensured that our instance is in a valid state with its background task started upon creation.

Look around your own project; do you see threads being started in constructors? If you do, you have another cleanup task to add to your refactoring tasks list.

5.4 Mind Your Resources

Threads are limited resources, and we shouldn't create them arbitrarily. `EnergySource`'s `replenish()` method is wasting a thread and limits the number of instances we can create. If more instances created their own threads like that, we'd run into resource availability problems. The replenish operation is short and quick, so it's an ideal candidate to run in a timer.

We could use a `java.util.Timer`. For a better throughput, especially if we expect to have a number of instances of `EnergySource`, it's better to reuse threads from a thread pool. `ScheduledThreadPoolExecutor` provides an elegant mechanism to run periodic tasks. We must ensure that the tasks handle their exceptions; otherwise, it would result in suppression of their future execution.

Let's refactor `EnergySource` to run the `replenish()` method as a periodic task.

Download tamingSharedMutability/periodictask/EnergySource.java

```
//Using Timer...other issues pending
public class EnergySource {
  private final long MAXLEVEL = 100;
  private long level = MAXLEVEL;
  private static final ScheduledExecutorService replenishTimer =
    Executors.newScheduledThreadPool(10);
  private ScheduledFuture<?> replenishTask;

  private EnergySource() {}

  private void init() {
    replenishTask = replenishTimer.scheduleAtFixedRate(new Runnable() {
      public void run() { replenish(); }
    }, 0, 1, TimeUnit.SECONDS);
  }

  public static EnergySource create() {
    final EnergySource energySource = new EnergySource();
    energySource.init();
    return energySource;
  }

  public long getUnitsAvailable() { return level; }

  public boolean useEnergy(final long units) {
    if (units > 0 && level >= units) {
      level -= units;
      return true;
    }
    return false;
  }

  public void stopEnergySource() { replenishTask.cancel(false); }

  private void replenish() { if (level < MAXLEVEL) level++; }
}
```

In addition to being kind on resource usage, the code got simpler. We got rid of the `keepRunning` field and simply canceled the task in the `stopEnergySource()` method. Instead of starting a thread for each instance of `EnergySource`, the `init()` method scheduled the timer to run the `replenish()` method. This method, in turn, got even simpler—we're not concerned about the sleep or the timing, so instead we focus on the logic to increase the energy level.

We made the reference `replenishTimer` a `static` field. This allows us to share a pool of threads to run the `replenish()` operation on multiple instances of `EnergySource`. We can vary the number of threads in this thread pool, currently set to `10`, based on the duration of the timed task and the number of instances. Since the `replenish()` task is very small, a small pool size is adequate.

Making the `replenishTimer` field static helped us share the pool of threads in the `ScheduledThreadPoolExecutor`. However, this leads to one complication: we have to figure out a way to shut it down. By default the executor threads run as nondaemon threads and will prevent the shutdown of the JVM if we don't explicitly shut them down. There are at least two ways[2] to handle this:

- Provide a static method in the `EnergySource` class, like so:

  ```
  public static void shutdown() { replenishTimer.shutdown(); }
  ```

  There are two problems with this approach. The users of the `EnergySource` have to remember to call this method. We also have to add logic to deal with instances of `EnergySource` being created after the call to `shutdown()`.

- We may pass an additional `ThreadFactory` parameter to the `newScheduledThreadPool()` method. This factory can ensure all the threads created are daemon threads, like so:

  ```
  private static final ScheduledExecutorService replenishTimer =
    Executors.newScheduledThreadPool(10,
        new java.util.concurrent.ThreadFactory() {
      public Thread newThread(Runnable runnable) {
        Thread thread = new Thread(runnable);
        thread.setDaemon(true);
        return thread;
      }
    });
  ```

  The main disadvantage of this approach is more code for us to write and maintain.

Our `EnergySource` just lost a few pounds and is more scalable than when we created the thread internally.

Examine your own project to see where you're creating threads, especially using the `Thread` class. Evaluate those situations to see whether you can use a periodic task scheduler like we did.

---

[2] The Google Guava API (http://code.google.com/p/guava-libraries/), which provides quite a few convenience wrappers on top of the JDK concurrency API, also provides a method to create pools that exits automatically.