Exercises week 2
Mandatory handin 1
Friday 5 September 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you have an initial understanding of using multiple threads for better performance, a good understanding of visibility of field updates between threads, and the advantages of immutability. You should be able to use locking (`synchronized`) and the `volatile` field modifier to ensure visibility between threads and use the `final` modifier to properly create and publish immutable objects.

Do this first

Get and unpack this week's example code in zip file pcpp-week02.zip on the course homepage.

Exercise 2.1 Consider the lecture's example in file TestMutableInteger.java, which contains this definition of class MutableInteger:

```
class MutableInteger {       // WARNING: USELESS IN THIS FORM
  private int value = 0;
  public void set(int value) {
    this.value = value;
  }
  public int get() {
    return value;
  }
}
```

As said in the Goetz book and the lecture, this cannot be used to reliably communicate an integer from one thread to another, as attempted here:

```
final MutableInteger mi = new MutableInteger();
Thread t = new Thread(new Runnable() { public void run() {
      while (mi.get() == 0) { }
      System.out.println("I completed, mi = " + mi.get());
  }});
t.start();
System.out.println("Press Enter to set mi to 42:");
System.in.read();                    // Wait for enter key
mi.set(42);
System.out.println("mi set to 42, waiting for thread ...");
try { t.join(); } catch (InterruptedException exn) { }
System.out.println("Thread t completed, and so does main");
```

1. Compile and run the example as is. Do you observe the same problem as in the lecture, where the `"main"` thread's write to `mi.value` remains invisible to the `t` thread, so that it loops forever?

2. Now declare both the `get` and `set` methods `synchronized`, compile and run. Does thread `t` terminate as expected now?

3. Now remove the `synchronized` modifier from the `get` methods. Does thread `t` terminate as expected now? If it does, is that something one should rely on? Why is `synchronized` needed on **both** methods for the reliable communication between the threads?

4. Remove both `synchronized` declarations and instead declare field `value` to be `volatile`. Does thread `t` terminate as expected now? Why should it be sufficient to use `volatile` and not `synchronized` in class MutableInteger?

Exercise 2.2 Consider the lecture's example in file TestCountPrimes.java.

1. Run the sequential version on your computer and measure its execution time. From a Linux or MacOS shell you can time it with `time java TestCountPrimes`; within Windows Powershell you can probably use `Measure-Command java TestCountPrimes`; from a Windows Command Prompt you probably need to use your wristwatch or your cellphone's timer.
2. Now run the 10-thread version and measure its execution time; is it faster or slower than the sequential version?
3. Try to remove the synchronization from the `increment()` method and run the 2-thread version. Does it still produce the correct result (664,579)?
4. In this particular use of LongCounter, does it matter in practice whether the `get` method is synchronized? Does it matter in theory? Why or why not?

Exercise 2.3 Consider the potentially computation-intensive problem of counting the number of prime number factors of an integer. This Java method from file TestCountFactors.java finds the number of prime factors of `p`:

```
public static int countFactors(int p) {
  if (p < 2)
    return 0;
  int factorCount = 1, k = 2;
  while (p >= k * k) {
    if (p % k == 0) {
      factorCount++;
      p /= k;
    } else
      k++;
  }
  return factorCount;
}
```

How this method works is not important, only that it may take some time to compute the number of prime factors. Actually the time is bounded by a function proportional to the square root of `p`, in other words $O(\sqrt{p})$.

1. Write a sequential program to compute the total number of prime factors of the integers in range 0 to 4,999,999. The result should be 18,703,729. How much time does this take?
2. For use in the next subquestion you will need a MyAtomicInteger class that represents a thread-safe integer. It must have a method `int addAndGet(int amount)` that atomically adds `amount` to the integer and returns its new value, and a `int get()` method that returns the current value.

   Write such a MyAtomicInteger class.
3. Write a parallel program that uses 10 threads to count the total number of prime factors of the integers in range 0 to 4,999,999. Divide the work so that the first thread processes the numbers 0–499,999, the second thread processes the numbers 500,000–999,999, the third thread processes the numbers 1,000,000–1,499,999, and so on, using your MyAtomicInteger class. Do you still get the correct answer? How much time does this take?
4. Could one implement MyAtomicInteger without synchronization, just using a volatile field? Why or why not?
5. Solve the same problem but use the AtomicInteger class from the java.util.concurrent.atomic package instead of MyAtomicInteger. Is there any noticeable difference in speed or result? Should the AtomicInteger field be declared `final`?

Exercise 2.4 Consider the lecture's versions of Goetz's factorization examples in file TestFactorizer.java.

1. In the VolatileCachingFactorizer class, why is it important that the `cache` field is declared volatile?
2. In the OneValueCache class, why is it important that both fields are declared `final`?