Exercises week 1
Friday 28 August 2015

Goal of the exercises

The goal of this week's exercises is to make sure that you can use Java threads and `synchronized` methods; 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.

The following abbreviations are used in the exercise sheets:

- "Goetz" means Goetz et al.: *Java Concurrency in Practice*, Addison-Wesley 2006.
- "Bloch" means Bloch: *Effective Java*. Second edition, Addison-Wesley 2008.
- "Herlihy" means Herlihy and Shavit: *The Art of Multiprocessor Programming*. Revised reprint, Morgan Kaufmann 2012.

The exercises let you try yourself the ideas and concepts that were introduced in the lectures. Some exercises may be challenging, but they are not supposed to require days of work.

If you get stuck with an exercise outside the exercise sessions, you may use the News Forum for the course in LearnIT to ask for help. This is better than emailing the teaching assistants individually.

Exercises may be solved and solutions handed in in groups of 1 or 2 students.

Exercise solutions should be **handed in through LearnIT** no later than 23:55 on the Thursday following the exercise date.

How to hand in

You should make hand-ins as simple as possible for you and for the teaching assistants. For instance, hand in a zip-file containing the Java source files written to answer the programming questions. Use Java comments to clearly indicate which part of the code relates to which exercise.

You may also use Java comments in the source files to reply to the text questions of the exercises, and to present output from experiments. Alternatively use simple text files for this purpose, but then name the files to make it completely clear what files contain solutions to what questions. In general, do not waste your time formatting everything beautifully with LaTeX or MS Word, unless this is actually faster for you.

Do **not** submit code in the form of screenshots. Do **not** hand in rar files and other exotic archive formats. Do **not** hand in zip-files of complete Eclipse workspaces and similar; they contain extraneous junk.

Do this first

Make sure you the Java Development Kit installed; **you will Java version 8 for this course**. Type `java -version` in a console on Windows, MacOS or Linux to see what version you have. From inside Eclipse you may instead inspect `Preferences > Java > Installed JREs`.

You may want to install a recent version of an integrated development environment such as Eclipse Mars (4.5).

Get and unpack this week's example code in zip file pcpp-week01.zip on the course homepage.

Exercise 1.1 Consider the lecture's LongCounter example found in file TestLongCounterExperiments.java, and **remove** the `synchronized` keyword from method `increment` so you get this class:

```
class LongCounter {
  private long count = 0;
  public void increment() {
    count = count + 1;
  }
  public synchronized long get() {
    return count;
  }
}
```

1. The `main` method creates a LongCounter object. Then it creates and starts two threads that run concurrently, and each increments the `count` field 10 million times by calling method `increment`.

   What kind of final values do you get when the increment method is **not** synchronized?

2. Reduce the `counts` value from 10 million to 100, recompile, and rerun the code. It is now likely that you get the correct result (200) in every run. Explain how this could be. Would you consider this software correct, in the sense that you would guarantee that it always gives 200?

3. The `increment` method in LongCounter uses the assignment

   ```
   count = count + 1;
   ```

   to add one to `count`. This could be expressed also as `count += 1` or as `count++`.

   Do you think it would make any difference to use one of these forms instead? Why? Change the code and run it, do you see any difference in the results for any of these alternatives?

4. Extend the LongCounter class with a `decrement()` method which subtracts 1 from the `count` field.

   Change the code in `main` so that `t1` calls `decrement` 10 million times, and `t2` calls `increment` 10 million times, on a LongCounter instance. In particular, initialize `main`'s `counts` variable to 10 million as before.

   What should the final value be, after both threads have completed?

   Note that `decrement` is called only from one thread, and `increment` is called only from another thread. So do the methods have to be `synchronized` for the example to produce the expected final value? Explain why (or why not).

5. Make four experiments: (i) Run the example without `synchronized` on any of the methods; (ii) with only `decrement` being synchronized; (iii) with only `increment` being synchronized; and (iv) with both being synchronized. List some of the final values you get in each case. Explain how they could arise.

Exercise 1.2 Consider this class, whose `print` method prints a dash "`-`", waits for 50 milliseconds, and then prints a vertical bar "`|`":

```
class Printer {
  public void print() {
    System.out.print("-");
    try { Thread.sleep(50); } catch (InterruptedException exn) { }
    System.out.print("|");
  }
}
```

1. Write a program that creates a Printer object `p`, and then creates and starts two threads. Each thread must call `p.print()` forever. You will observe that most of the time the dash and bar symbols alternate neatly as in `-|-|-|-|-|-|-|`.

   But occasionally two bars are printed in a row, or two dashes are printed in a row, creating small "weaving faults" like those shown below:

```
|-|-|-|-|-|-||--|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-||--
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-||--|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
```

Since each thread always prints a dash after printing a bar, and vice versa, this phenomenon can be caused only by one thread printing a bar and then the other thread printing a bar before the first one gets to print its dash.

Describe a scenario involving the two threads where this happens.

2. Making method `print` synchronized should prevent this from happening. Explain why. Compile and run the improved program to see whether it works.
3. Rewrite `print` to use a `synchronized` statement in its body instead of the method being synchronized.
4. Make the `print` method static, and change the `synchronized` statement inside it to lock on the Print class's reflective Class object instead.

   For beauty, you should also change the threads to call static method `Print.print()` instead of instance method `p.print()`.

Exercise 1.3 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 1.4 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?