Exercises week 1
August 29 2019

Goal of the exercises

The goal of this week's exercises is to make sure that you can use Java threads and `synchronized` methods and statements; 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. Mandatory reading.
- "Bloch" means Bloch: *Effective Java*. Second edition, Addison-Wesley 2008. Recommended reading.
- "Herlihy" means Herlihy and Shavit: *The Art of Multiprocessor Programming*. Revised reprint, Morgan Kaufmann 2012. A few chapters are mandatory reading.

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 forum for the course in LearnIT to ask for help.

How to hand in

You should make hand-ins as simple as possible for you and for the teaching assistants. Please submit a single zipped (`*.zip`) folder containing the following:

- A text file `ANSWERS.TXT` with answers to all the week's exercises, preferably containing also relevant code snippets. Using Markdown is nice, but not necessary; plain ASCII is fine.
- A directory or folder `SRC/` with the source code, essentially the very same code provided to you, but with your additions. Please **do not** create extra directories or reorganize the files, keep it simple (as the code is given to you!).

Please do **not** submit:

- Microsoft Word documents (`*.doc` or `*.docx` files) or LibreOffice or OpenOffice documents (`.odt`).
- PDF documents (`*.pdf` files).
- Eclipse or Netbeans project metafiles (`*.proj` files and other junk).
- Compiled Java classes (`*.class` files).
- Exotic archive formats such as `.rar` files.
- Screenshots that just show code or text output. Better submit those as `.java` or `.txt` files.

Do this first

Make sure you have the Java Development Kit installed; **you will need 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 Neon (4.6).

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 (3,001,134)?

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 1.5 Consider the small artificial program in file TestLocking0.java. In class Mystery, the single mutable field `sum` is private, and all methods are synchronized, so superficially the class seems to be thread-safe.

1. Compile the program and run it several times. Show the results you get. Do they indicate that class Mystery is thread-safe or not?

2. Explain why class Mystery is not thread-safe. Hint: Consider (a) what it means for an instance method to be synchronized, and (b) what it means for a static method to be synchronized.

3. Explain how you could make the class thread-safe, *without* changing its sequential behavior. That is, you should not make any static field into an instance field (or vice versa), and you should not make any static method into an instance method (or vice versa). Make the class thread-safe, and rerun the program to see whether it works.

Exercise 1.6 Consider class DoubleArrayList in TestLocking1.java. It implements an array list of numbers, and like Java's ArrayList it dynamically resizes the underlying array when it has become full.

1. Explain the simplest natural way to make class DoubleArrayList thread-safe so it can be used from multiple concurrent threads.

2. Discuss how well the thread-safe version of the class is likely scale if a large number of threads call `get`, `add` and `set` concurrently.

3. Now your notorious colleague Ulrik Funder suggests to improve the code by introducing a separate lock for each method, roughly as follows:

```
private final Object sizeLock = new Object(), getLock = new Object(),
  addLock = new Object(), setLock = new Object(), toStringLock = ...;
public boolean add(double x) {
  synchronized (addLock) {
    if (size == items.length) {
      ...
    }
    items[size] = x;
    size++;
    return true;
  }
```

```
    }
    public double set(int i, double x) {
      synchronized (setLock) {
        if (0 <= i && i < size) {
          double old = items[i];
          items[i] = x;
          return old;
        } else
          throw new IndexOutOfBoundsException(String.valueOf(i));
      }
    }
```

Would this achieve thread-safety? Explain why not. Would it achieve visibility? Explain why not.

Exercise 1.7 Consider the extended class DoubleArrayList in TestLocking2.java. Like the class in the previous exercise it implements an array list of numbers, but now also has a static field `totalSize` that maintains a count of all the items ever added to any DoubleArrayList instance.

It also has a static field `allLists` that contains a hashset of all the DoubleArrayList instances created. There are corresponding changes in the `add` method and the constructor.

1. Explain how one can make the class thread-safe enough so that the `totalSize` field is maintained correctly even if multiple concurrent threads work on multiple DoubleArrayList instances at the same time. You may ignore the `allLists` field for now.

2. Explain how one can make the class thread-safe enough so that the `allLists` field is maintained correctly even if multiple concurrent threads create new DoubleArrayList instances at the same time.

Exercise 1.8 Consider the small artificial program in file TestLocking3.java. Since the single field and the three methods in classes MysteryA and MysteryB are all static, there is no confusion of locks on class and instance, so superficially the classes seem to be thread-safe.

```
class MysteryA {
  protected static long count = 0;
  public static synchronized void increment() {
    count++;
  }
  public static synchronized long get() {
    return count;
  }
}

class MysteryB extends MysteryA {
  public static synchronized void increment4() {
    count += 4;
  }
}
```

1. Explain why after 10 million calls to `MysteryB.increment()` and 10 million concurrent calls to `MysteryB.increment4()`, the resulting value of `count` is rarely the expected 50,000,000.

   Hint: Consider the actual meaning of the `synchronized` modifier when used on a static method.

2. Explain how one can use an explicit lock object and `synchronized` statements (not `synchronized` methods) to change the locking scheme, so that the result is always the expected 50,000,000.