Exercises week 1

Last update 2020-07-24

Goal of the week

The goal of this week is that you:

- Are able to give examples of the three major usages of concurrency.
- Write code that illustrate race conditions, liveness and compound actions, and explain how the problems arise.
- Can use `synchronized` and locks to prevent such concurrency errors.

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 are a way for you to get a practical understanding of the material. In addition, they serve as the outset for weekly feedback.

How is feedback given

Feedback is given on zoom. You are divided into small teams of two or three students. Two such teams form a feedback group. Each feedback group is allocated a timesslot for feedback. Feedback will happen on Thursdays.

The feedback sessions has two aspects to them. Feedback on your solutions, and time for answering questions which you have not been able to find an answer for in your team, group or by other means.

Each exercise explains what you should prepare to demo at feedback.

Active participation in feedback corresponds to hand-in in other classes, that is, it is mandatory to actively participate in a number of the feedback sessions (11 out of 14).

Ambition levels

We acknowlege that different students strive to strike different balances between the different classes and between study and other aspects of their lives. We operate with three ambition levels:

- **green**. This ambition is for students who wants to make sure they pass the class, but clearly wants to prioritize other aspects of their life and studies highere. Sort of *I want a 4, but can live with 2*.
- **yellow**. This ambition is for students who want to learn the major part of the syllabus, but do not intend to get all the finer details straight. Sort of *I aim for 7, but hope for a 10*.
- **red**. This ambition is for students who want not only to get the major part of the syllabus, but also have a keen interest in all the finer details and suptle concepts. Sort of *I will be disappointed if I do not get a 10*.

Teams and groups should be made up of students of the same ambition level.

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.

You may want to install a recent version of an integrated development environment.

Get this week's example code from the course github page.

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;
  }
}
```

Green

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?

Yellow

4. Decompile the methods increment from above to see the byte code in the three versions (as is, +=, ++). The basic decompiler is `javap`. Does that verify or refuse the explanation you made earlier?

5. 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).

Red

6. 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 The example code "TestCountPrimes" show how one can speed up a program by using more parallel threads - up to a certain level which is hardware dependent. Try with different values of the parameter `threadCount` to show how many hardware threads your implementation of Java can support.

Yellow

Some of you might be able to get a result indicating you have eight hardware threads on a four core CPU. How can that be?

If you change the `range` to something large (so it takes minutes to compute) you might experience a drop in the number of hardware threads your java program can utilize. What happens (if anything)?

Exercise 1.3 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("|");
  }
}
```

Green

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:

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

2. Describe a scenario involving the two threads where this happens.
3. Making method `print` synchronized should prevent this from happening. Compile and run the improved program to see whether it works. Explain why.
4. Rewrite `print` to use a `synchronized` statement in its body instead of the method being synchronized.

Yellow

5. 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()`.

Red

6. Explain subquestion 3 in terms of the *happens-before* relation from Goetz section 16.1.3.

Exercise 1.4 In *Goetz* chapter 1.1, three motivations for concurrency is given: resource utilization, fairness and convenience. One of the inventers of object oriented programming, Kristen Nygaard, once gave an alternative characterization of the different ways to use concurrency:

- **Exploitation** of multiprocessors. The goal here is to exploit that the computer has multiple cores (or we have access to a number of physical computers). A good exploitation scheme makes it easy to write programs to make efficient use of true parallelism.
- **Concealed** parallelism. The goal here is to make it possible for several programs to share some resources in a manner where each can act if they had sole ownership.
- **Intrinsic** parallelism. The real world is intrinsically parallel. Computers who interact with the real world need to deal with this. The goal here is to make it easy to write programs which responds (in time) to input sensors or other connected devices.

Green
Compare the categories of Kristen and Goetz, try to find three examples (if possible - if not possible, argue why) of systems which are included in the categories of Goetz, but not in those of Kristen, and vice versa.

Yellow
Find examples of 3 systems in each of Kristens categories which you have used yourself (as a programmer or user).

Red
Explain how the concerns of each of Kristens categories might help in the support of the other two (if possible).

Exercise 1.5 *Yellow*
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 *Red*
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.

```java
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.

Exercise 1.7 *Red*

The keyword `synchronized` can be used to as a method modifier and as a statement. Consider this sample code:

```
class ThisSync{
  private long l;
  public synchronized void increment(){
    l++;
  }

  public void decrement(){
    synchronized(this){
      l--;
    };
  }
}
```

Examine the bytecode, and explain differences and/or similarities between the two ways to use use `synchronized`.