Java Concurrency Exercises - Day2

*Important: You are NOT expected to complete all these exercises day2. Remember, Fridays are coding-days, so you have Friday to complete exercises for the full week, including a few more if you just “can't get enough” ;-)*

### **Exercise 1(race condition)**

**a)** The method next () in the Even class below should always return an even number. Demonstrate, in a test program, that this is not always true in a multi-threaded program.

**public class Even {**

**private int n = 0;**

**public int next() {**

**n++;**

**n++;**

**return n;**

**}**

**}**

Hint: Implement two threads, which both should call next () several times on the **SAME** Even object and test whether the return value is equal.

* Explain what is happening?
* How frequent is the error?

**b)** Add synchronization so that the instructions of the method runs atomically



### **Exercise 2 (race condition-2)**

The package turnstile (given with the code for day-1) contains the start code for this task. It simulates a large football stadium with many counters (turnstiles) where you always should be able to see how many spectators have entered the stadium. Initially, the code is set up with 40 counters (turnstiles), each simulating that 1000 spectators go through. Observe that calculations take place in a separate thread for each turnstile

Execute the code and observe the result of the shared Counter.

If the result is not as expected, solve the problem:

1. Using traditional Java-1 synchronization
2. Using an AtomicInteger
3. Using a java.util.concurrent.locks.ReentrantLock

### **Exercise 3 Threads and Performance** ([video introduktion](https://www.youtube.com/watch?v=5fz2gCyYEKA) til opgaven)

Yesterday we saw how we could make a non-responsive program responsive (the ball demo)

This exercise will (hopefully) demonstrate how threads can be used to improve performance.

We will use a **very naive** recursive algorithm to calculate numbers in the Fibonacci sequence. The algorithm is of NO importance for this exercise, but just for your info:

In maths, the Fibonacci sequence is described as*: the sequence of numbers where the first two numbers are 0 and 1, with each subsequent number being defined as the sum of the previous two numbers in the sequence*.

**The Fibonacci sequence looks like this: *0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233 and so on.***

*Ref:* [*https://medium.com/launch-school/recursive-fibonnaci-method-explained-d82215c5498e*](https://medium.com/launch-school/recursive-fibonnaci-method-explained-d82215c5498e)

This is the algorithm you should use for this exercise. Again the algorithm itself is not important, but what is important for this exercise is that the time spent on calculating a number grows exponentially.

int fibonacciRecursive(int n) {

if (n < 2) return n;

return FibonacciRecursive(n - 1) + FibonacciRecursive(n - 2);

}

**1)** Create a project (or use an existing one) for this exercise and add a simple class with a main which should verify that if you call the method above 14 times (with the number 0, 1, 2 .. 13) it should provide the results given above.

Try also to calculate the fibonacci number for a higher number like 45 (don't go beyond 50) and observe the result (time spent)

**2)** Implement a class that implements the Runnable interface, takes a number, via the constructor, and in the run() method calculates the corresponding fibonacci number. It should add this number to a private variable in the class, and leave the run-method.

Provide the class with a *getter*, so you can get the result.

**3)** Create a class with a main method.

In this method, create FOUR instances of your Runnable class (use the value 46 for ALL three) and THREE Threads using these runnable instances.

**4)** Run your thread instances like this, assuming you have called them t1, t2 and t3. OBSERVE: call **run( )** NOT start()

**long start = System.nanoTime();**

**t1.run(); //Runs sequential, NOT in a thread**

**t2.run();**

**t3.run();**

**t4.run();**

**long end = System.nanoTime();**

**System.out.println("Time Spent: "+(end-start)/1\_000\_000);**

This should provide you with the time spent in milliseconds. Execute the code a few times to get an "average" number, and MAKE sure you understand that what you did here was purely sequential, that is, work was NOT done by separate threads.

**5)** Now let's try and see whether executing the calculations in separate threads can improve performance (time spent).

Change the code above to call **start()** and not run

You should also add the necessary code to ensure that BEFORE you get to the part that calculates the end time, ALL threads have completed their work.

Do this a few times to get and "average" time and compare the result with what you got above.

EXPLAIN what you have observed.

### 

### 

### 

### **Exercise 4 Producer-Consumer Intro** ([watch this video](https://www.youtube.com/watch?v=N87AlojMiUY) (8 min.) before you start)

*This exercise does not require you to write any code. What you have to do is LOOK AT the code, Run the code, and explain what you see.*

This is tricky, but you have to get used to predicting the outcome of code, without always having to execute it first.

**1)** Can you predict EXACTLY what will be printed to the terminal when the code executes.

In order to do that you probably need to read the documentation for BlockingQueus’s methods:

* put() : One of several possible methods to place new items into the list
* take() :One of several possible methods to take items out of the list

When you read the documentation, focus on what will happen if the queue is **full** when you try to add a new item, or **empty** if you try to take out an item.

Also observe that the queue is created with a max capacity of **5** and count the numbers of “produces” made in the code.

**2)** If part 1 was (to) hard, execute the code and now explain the order of what you see printed.

**3)** Write down a small summary to yourself, so you will remember what you observed next time you come back to this exercise

### **Exercise 5 Deadlocks** ([watch this video](https://www.youtube.com/watch?v=XyGSwo3AViY) before you start (12 min)

The main purpose with this exercise is to illustrate the effect of a deadlock and the (wrong) code that led to the deadlock. So if you “had enough” of threads for this day/week, part 1 below is enough.

1. Execute the code in deadlock.simpledemo package. Observe how it “always” deadlocks.
2. The code you have cloned includes a DeadLockDetector class, which if it detects a deadlock will terminate the program. See whether you can use it to detect if the program actually deadlocks, and if, terminate the program.

**Hint:** See line 39

1. Can you fix the code so it will not deadlock.

**Hint:** Jenkov has the solution ;-)

<http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html>