Practice Script 2

Summary:

- Native concurrent programming in Java.

Exercise 2.1

Create a class `CounterThread`, to be attached with a thread, to count up to a given value (starting with 1). Its constructor receives the counter limit, and a thread identification, and should throw the new (counting) thread. In each count the thread should wait a random time[1] and write the counter information.

Test the class with a program that receives integers as arguments and throws a new `CounterThread` thread to count each one.

The output of the execution `java -ea p21 2 4` should be something like the following:

```
thread #1: counter is 1
thread #2: counter is 1
thread #2: counter is 2
thread #2: counter is 3
thread #1: counter is 2
thread #2: counter is 4
```

Use `join` method to synchronize the end of the program with the end of each thread.

Try changing the threads to daemon threads. Note anything strange?

Exercise 2.2

Based on exercise 2.1, create a `Counter` class to be used and shared by all counter threads (change the name of their classes to `SharedCounterThread`).

Try using this shared counter first with no synchronization, and then declaring the counting method with `synchronized`[2].

Exercise 2.3

Create a program that allows the scheduling and execution of alarms containing text messages. The main program should read lines of text containing an elapsed (alarm) time and

[1]Use the methods `Thread.sleep` and `Math.random`.

[2]To increase the probability of a race condition, insert a random waiting time in the counter incrementing method (between the reading of the previous value and the assignment of the new one).

a message, and throw a thread to handle it (implement termination with an input empty line).

In user interaction, the program should allow the scheduling of a new alarm (always seconds from the time it is created). Whenever an alarm exhausts its waiting time, it should write its message to the standard output.

Exercise 2.4

Create a program to test if a list of numbers – passed as program arguments – are prime.

a. Start by implementing a sequential program to solve the problem.

b. Then modify it to a concurrent version in which a new thread is created to test each number.

c. Try to implement a version in which there is a limit to maximum number of executing threads[3].

You may use `System.nanoTime` to measure execution time of the different versions.

Exercise 2.5

In order to evaluate the gains (or losses) in execution time of the different implementations of exercise 2.4, lets implement comparable benchmark versions of each version.

In order to do that use methods `System.nanoTime` and `Thread.sleep`. The first to measure the elapsed execution time; and the second to force a fixed execution time in the method to determine if a number is prime (the same delay must be applied to all versions in order to make them comparable).

Exercise 2.6

Change program 1.1 to implement animated figures. Consider that the movement of the figures has a linear constant velocity. Whenever a figure collides with the window boundaries, the corresponding velocity component should changes its sign. Ensure also that the initial speed and direction of these animated figures is settable in their construction.

In solving this problem, take into consideration that it is pretended to animate any kind of figure.

Exercise 2.7

Continue to develop the animated figures problem, ensuring that two figures cannot share the same space. So whenever there is a clash, the figures involved whould choose an opposite direction. To simplify the problem consider the space occupied by each figure is defined by a rectangular bounding box.

It is suggested, as a first approximation, the creation of a shared object between all the pictures that records the occupied space, and through which each picture can know whether it collides with another.

Use native Java's synchronization scheme in shared objects.

[3]Set it to the double of `Runtime.getRuntime().availableProcessors()`.