TD 1 - Java concurrency: synchronizers

To set up for the practical exercises go to CELENE and download `code.zip`. Create a Java project in your favorite IDE (e.g., Eclipse, Netbeans) and import the contents of `code.zip` into your project. You should now have 6 packages, one per exercise, called `polytech.tours.di.parallel.td1.exo#` where `#` is the number of the exercise. You're now good to go.

1 Thread interference

The objective of this first exercise is to see thread interference in action. Study the three classes in package `polytech.tours.di.parallel.td1.exo1`, namely, `Counter`, `ParrallelCounting`, and `Tester`. Run the `main()` method of class `Tester` several times. Do you observe any abnormal behavior? if so, can you explain it?

2 Synchronize methods and sections

Refactor the code in package `polytech.tours.di.parallel.td1.exo2` so the thread interference is avoided.

Hint: remember the synchronize methods and synchronize sections we discussed in class.

3 Explicit locks

We saw in class that `java.util.concurrent.ReentrantLock` provides a ready-to-go implementation of a re-entrant lock for thread synchronization. For learning purposes we will reinvent the wheel and code our own implementation (but please in real applications use the one provided by Java).

Code a class called `polytech.tours.di.parallel.td1.exo3.Lock` implementing two methods `lock()` and `unlock()`. The `lock()` method locks the `Lock` instance so that all threads calling `lock()` are blocked until `unlock()` is executed.

Hint: remember the `wait()` and `notify()` methods we studied in class.

To test your `Lock`, refactor class `polytech.tours.di.parallel.td1.exo3.Counter` so it uses an instance of your `Lock` to prevent memory inconsistency errors and thread interference. You can use `polytech.tours.di.parallel.td1.exo3.Tester` to conduct the experiments.

Hint: we saw an example of these guarded blocks in class.

4 Is your `Lock` re-entrant?

Study class the `polytech.tours.di.parallel.td1.exo4.ReentrantTask`. Implement a class called `polytech.tours.di.parallel.td1.exo4.Tester` with a `main()` method that launches the execution of an instance of `ReentrantTask` in a `Thread`. What happens? why?

5 Making our Lock re-entrant

Implement a class called `polytech.tours.di.parallel.td1.exo5.ReentrantLock` that solves the reentrance problem. Use `polytech.tours.di.parallel.td1.exo5.Tester` to test your solution.

Hint: remember that a thread may try to obtain the same lock more than twice.
Hint: remember than a thread executing a task is nothing but an instance of class `Thread`.

6 Implementing a cyclic barrier

A cyclic barrier is a synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. Cyclic barriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released (for further details check page 72 of the class slides for chapter 3). Package `java.util.concurrent` in the Java high level concurrency API contains a class implementing a cyclic barrier (`java.util.concurrent.CyclicBarrier`). The objective of this exercise is to implement our own (reduced) version of that class.

1. Study the sample usage for a cyclic barrier reported in the javadoc for class `java.util.concurrent.CyclicBarrier` (available at: `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html`)
2. Study interface `CyclicBarrier` in package `polytech.tours.di.parallel.td1.exo6`
3. Develop a class called `polytech.tours.di.parallel.td1.exo6.MyCyclicBarrier` providing a concrete implementation of the interface
4. Test your cyclic barrier using class `polytech.tours.di.parallel.td1.exo6.Tester`

POLYTECH TOURS
Département Informatique

TD 2 - Java concurrency: solving liveness problems

To set up for the practical exercises go to CELENE and download `code.zip`. Create a Java project in your favorite IDE (e.g., Eclipse, Netbeans) and import the contents of `code.zip` into your project. You should now have 3 packages, one per exercise, called `polytech.tours.di.parallel.td2.exo#` where `#` is the number of the exercise. You are now good to go.

1 Experimenting a deadlock

The objective of this exercise is to analyze code leading to a deadlock. Study classes `polytech.tours.di.parallel.tp2.exo1.ParallelTask` and `polytech.tours.di.parallel.tp2.exo1.Tester`. Run the `exo1.Tester.#main` method. What do you observe? Can you explain the situation?

To help analyzing the problem you can take a look at the java thread dump of the application. Launch the Java Visual Virtual Machine (`jvisualvm`) while running your code and execute a dump thread. If you do not know how to lauch `jvsualvm`, check the documentation provided by Oracle at `https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/intro.html`.

2 Solving deadlocks

We are now going to try to avoid the deadlock situation using different strategies.

2.1 Lock ordering

If you did your homework (of course you did), you read about lock ordering as one of the most common strategies to avoid deadlocks. Refactor class `polytech.tours.di.parallel.tp2.exo2_1.Tester` so it makes sure that the three threads always access the locks in the same order. Run some experiments with your refactored class. What do you observe?

2.2 Avoiding nested locks

Nested locks is probably the most common reason for deadlocks. Therefore, avoiding locking one resource if you already hold one is always a good idea. Refactor class `polytech.tours.di.parallel.tp2.exo2_2.ParallelTask` to avoid nested locks. Run method `exo2_2.Tester.#main`. What do you observe?

2.3 Lock timeout

Another deadlock prevention mechanism is to put a timeout on lock attempts. Under this mechanism a thread trying to obtain a lock will only try for a given (and usually pre-defined) time before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, then retry or abandon its task.

Refactor class `polytech.tours.di.parallel.tp2.exo2_3.ParallelTask` so the threads wait for 5 seconds on the locks. If a thread cannot obtain a lock, it should free all acquired locks and abort its execution. Run method `polytech.tours.di.parallel.tp2.exo2_3.Tester` to test your implementation.

Hint: class `ReentrantLock` provides a method signed `public boolean tryLock(long timeout, TimeUnit unit)` that allows a thread to acquire the lock if it is not held by another thread within the given waiting time and the calling thread has not been interrupted. Check the Javadoc.

3 Fairness

Study the classes in package `polytech.tours.di.parallel.tp2.exo3_1`. Run several times method `polytech.tours.di.parallel.tp2.exo3_1.Tester#main` with different values for T and D. What do you observe?

3.1 Implementing fairness 1

Class `ReentrantLock` includes a constructor that accepts a fairness parameter. When set true, under contention, the method favors granting access to the longest-waiting thread. Study the Javadoc for this constructor and refactor class `polytech.tours.di.parallel.tp2.exo3_1.Testers#main` so it implements fairness. Repeat your experiments. What do you observe?

3.2 Implementing fairness 2

For teaching purposes, and also to have some fun, we are going to implement our own version of `ReentrantLock` providing fairness. Implement methods `lock()` and `unlock()` in class `polytech.tours.di.parallel.tp2.exo3_2.FairLock`. Run method `polytech.tours.di.parallel.tp2.exo3_2.Tester#main`. Do you obtain results that are similar to those obtained in exercise **??**?.

Hint: Use objects of class `polytech.tours.di.parallel.tp2.exo3_2.QueueObect` to build a queue of objects on which calling threads can wait on.

TD 3 - Java concurrency: parallel algorithms

To set up for the practical exercises open your favorite IDE (e.g., Eclipse, Netbeans) and create a Java project. You're now good to go.

1 Estimating the value of π

The value of π can be calculated in a number of ways. Consider the approach below:

- Inscribe a circle in a 1x1 square
- Randomly generate points in the square
- Determine the number of points in the square that are also in the circle
- Let p be the number of points in the circle divided by the number of points in the square, then $\pi \approx 4 \times p$

Figure **??** illustrates the approach.Note that the quality of the approximation increases with the number of generated points.

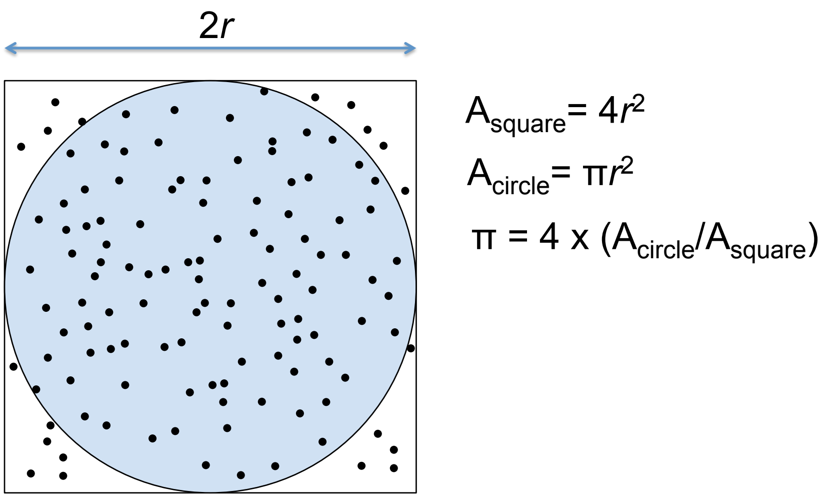

Figure 1: Approximating π

It is possible to solve this problem using what is known as an embarrassingly parallel solution; that is a solution which is computationally intensive and has minimal communication and minimal I/O. The objective of this exercise is to come up with such solution.

1. Propose a parallel algorithm for this problem
   - What decomposition strategy better fits the problem?
   - What is the task dependency graph of your algorithm?
   - What is the task interaction graph of your algorithm?
2. Propose a Java implementation for your algorithm