ESIR SPP – TP3 & 4 (Attention: **TP noté**)

Préparations:

- Assurez vous d'avoir accès au site Moodle du cours sur votre ENT (https://ent.univ-rennes1.fr, puis onglet "La Formation", "Cours en ligne").
- Vérifiez que vous avez bien accès à la page de soumission du TP 3 & 4.

Soumission et notation:

- Il vous est demandé de travailler en **binôme** pour ce TP.
- Vous devez individuellement **soumettre le code source Java des exercices du TP** avant la **date butoir** indiquée sur Moodle. Conseil: N'attendez pas cette date pour soumettre votre solution!
- Veuillez noter dans votre soumission **le binôme** avec lequel vous avez travaillé.
- Le TP **sera noté en classe** durant **la séance de TP 5** à partir du code source que vous aurez soumis.
- Pour recevoir une note pour devez impérativement: (i) **soumettre votre solution** avant la date butoir; et (ii) **être présent lors du TP 5**, et ce pour les deux étudiants de chaque binôme.

Note: Les étudiants d'un même binôme peuvent recevoir des notes différentes en fonction des réponses données lors de la notation.

Exercise 1:

The goal of this exercise is to implement an extended semaphore using **Java monitors**.

Your semaphore should implement the following interface (available in the lab's resources):

```
public interface SemaphoreInterface {
  public void up();
  public void down();
  public int  releaseAll();
} // EndInterface SemaphoreInterface
```

up() and down() correspond to the traditional semaphore operations. `releaseAll()` is an extension that unblocks all threads waiting on the semaphore, and returns the number of threads that have just been unblocked.

In addition to the above interface, your implementation should provide a constructor that takes no parameters, and returns a semaphore that contains zero permits.

Testing your implementation:

To help you test your semaphore, you will find a JUnit test class in the lab's resources `SemaphoreJUnitTest`. You can run this test class in two ways:

- Directly from the command line (the easiest), in the same folder as your compiled classes:

  ```
  java -DSemaphoreImplClass=Foo -cp junit-4.11.jar:.
  org.junit.runner.JUnitCore SemaphoreJUnitTest
  ```

  where Foo is the name of your semaphore class. You will need to copy the `junit-`

`4.11.jar` file accordingly. (Available at http://www.junit.org/, or on Moodle from the labs resources.)

- From within Eclipse, which directly supports JUnit. See for instance the instructions from http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.user/gettingStarted/qs-junit.htm?cp=1_1_0_15 or http://www.vogella.com/articles/JUnit/article.html for more detail.

Note: Passing all the tests does not guarantee that your code is correct, but will provide some level of confidence that it has the expected behaviour.

Marking scheme: Total: 15 points

- You are able to explain how your code solves the problem.	6 points
- Passing all test cases	4 points
- Code correct on manual analysis	5 points

Exercise 2:

The goal of this exercise is to learn to use the rendez-vous mechanism provided by Java.

Rendez-vous synchronisation is provided in Java by the Exchanger<V> class. We will use this class to implement a small ping-pong program in which two threads (called "Alice" and "Bob") repeatedly exchange two string objects ("Ping" and "Pong").

First, read the documentation for this class at http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Exchanger.html.

Then implement a small multithreaded program so that:

- The main program launches two threads called "Alice" and "Bob" (add a field "name" to your Thread or Runnable class to store this name);
- "Alice" and "Bob" both hold a reference to a string. "Alice" starts with a reference to a string containing "Ping", and "Bob" to a string containing "Pong".
- "Alice" and "Bob" execute the following behaviour 3 times:
    - (1) print the number of the current iteration, followed by their name, followed by the content of the string to which they hold a reference;
    - (2) print that they are about to go to sleep;
    - (3) wait a random time between 0 and 5000 ms;
    - (4) print that they are about to use the exchanger;
    - (5) use Java's Exchanger mechanism to exchange the thread's current string with that of the other thread;
    - (6) indicate that the exchange has completed.

The resulting output should look like the following:

```
Iteration: 0 Alice has Ping
Iteration: 0 Alice going to sleep.
Iteration: 0 Bob has Pong
Iteration: 0 Bob going to sleep.
Iteration: 0 Bob ready to exchange
Iteration: 0 Alice ready to exchange
Iteration: 0 Alice exchange completed
Iteration: 1 Alice has Pong
Iteration: 1 Alice going to sleep.
Iteration: 0 Bob exchange completed
Iteration: 1 Bob has Ping
Iteration: 1 Bob going to sleep.
Iteration: 1 Alice ready to exchange
Iteration: 1 Bob ready to exchange
Iteration: 1 Alice exchange completed
```

```
Iteration: 2 Alice has Ping
Iteration: 2 Alice going to sleep.
Iteration: 1 Bob exchange completed
Iteration: 2 Bob has Pong
Iteration: 2 Bob going to sleep.
Iteration: 2 Alice ready to exchange
Iteration: 2 Bob ready to exchange
Iteration: 2 Bob exchange completed
Iteration: 2 Alice exchange completed.
```

Marking scheme: Total: 5 points

- You are able to explain how your code solves the problem. 2 points
- The code works. 3 points
  (0: no attempt; 1: some attempt but does not work; 2: solid effort, but some glitches; 3: works perfectly)