**Opérations atomiques**

Le but principal de cette série d'exercices est d'apprendre à utiliser les variables atomiques en Java afin de gérer les accès concurrents sans utiliser de verrous. Durant cette séance, les threads Java ainsi que les opérations atomiques serons utilisés pour rechercher le plus petit et le plus grand nombre contenus dans un tableau d’entiers.

**Accès concurrents: le mot clé synchronized**

Le mot clé synchronized permet de créer des sections critiques au sein d'une instance. Ce mot clé permet de déclarer une méthode qui n'est accessible par un seul thread à la fois en ajoutant simplement synchronized dans la déclaration de la méthode :

public void synchronized threadSafeMethod( args ) {  
  // le corps de la méthode est accessible par  
  // un seul thread à la fois  
}

Il est aussi possible de créer des blocs synchronized pour définir des sections critiques :

public void threadSafeMethod( args ) {  
  synchronized( this ) {  
    // le corps du bloc est accessible par  
    // un seul thread à la fois  
  }  
}

Il est important de savoir que tous les blocs synchronized et toutes les méthodes synchronized d'une instance de classe sont verrouillés en même temps. C'est-à-dire que dès qu'un thread accède à une méthode synchronisée ou à un bloc synchronisé, tous les autres blocs et méthodes synchronisés de l'instance deviennent inaccessible aux autres threads.

**Opérations atomiques**

Une opération atomique est un ensemble d'opérations pouvant être combinées afin qu'elles apparaissent comme une unique opération au système. Ces opérations ne retournent que deux types de résultats : succès ou échec. En java on utilise le paquet java.util.concurrent.atomic qui définit un ensemble de classes supportant les opérations atomiques sur des variables. Ce paquet étend la définition du modificateur volatile. On rappelle que ce modificateur assure pour une variable :

* La cohérence entre la mémoire de travail et la mémoire principale ;
* La mise à jour forcée afin de prendre en compte les dernières mises à jour ;
* L'atomicité de la lecture ainsi que de l'écriture, mais pas celle des opérations composites (par ex. l'incrémentation).

À titre d'exemple d'extension ce paquet apporte [le compare and swap](https://en.wikipedia.org/wiki/Compare-and-swap) (CAS) qui compare le contenu d'une position en mémoire à une valeur donnée et, si ces deux dernières sont identiques, le contenu en mémoire est remplacé par une nouvelle valeur donnée. Ces opérations se font sans qu'un autre processus ne puisse agir sur le conteneur. De plus amples informations sont disponibles dans la description du paquet [atomic](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html) ainsi que dans ce [tutoriel](http://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html). Dans le cas du problème traité dans cet exercice les classes suivantes seront utiles : [AtomicInteger](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html), [AtomicIntegerArray](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicIntegerArray.html). Il est conseillé de passer un peu de temps à s'informer des méthodes qu'offrent ces classes avant de tenter de résoudre le problème.  
  
**Exercices**

**Exercice 1**

Implémentez une solution permettant de trouver de manière concurrente la valeur minimale et maximale dans un tableau d'entiers. La solution implémentée doit être basée sur le mot clé synchronized et ne doit ni utiliser les verrous ni les Future.

Le principe d'implémentation repose sur la création de N threads/taches. Chacune de ces taches cherchant le min-max du tableau sur la portion qui lui est alloué. Le min et le max retournés sont donnés comme candidat au min et max du tableau.

L’archive ci-jointe donne les principaux éléments. Il vous reste à le compléter.

**Exercice 2**

Implémentez une solution permettant de trouver de manière concurrente la valeur minimale et maximale dans un tableau d'entiers. La solution implémentée ne doit ni utiliser les verrous ni les Future mais uniquement les opérations atomiques. Testez votre solution sur des distributions aléatoires en les comparant au résultat de la version précédente.

L'opération d'assignation concurrente se déroule de la manière suivante :

1. On lit l'extremum global actuel (min ou max).
2. Est-ce que notre candidat est un meilleur extremum ?
3. Non : il n'y a rien à faire.
4. Oui : on tente un compare and swap avec l'extremum local et global.
5. Si le compare and swap échoue c'est que l'extremum global a été assigné de manière concurrente par un autre thread entre les étapes 1 et 2. Dans ce cas on recommence à partir de l'étape 1 dans l'éventualité où l'extremum local est bien l'extremum global.

L’archive ci-jointe donne les principaux éléments. Il vous reste à le compléter.

Un peu de lecture complémentaire en guise de révision:

* Java Memory Model : <http://tutorials.jenkov.com/java-concurrency/java-memory-model.html>
* Réordonnancement d’instructions : <http://tutorials.jenkov.com/java-concurrency/java-happens-before-guarantee.html>
* Le mot-clé volatile : <http://tutorials.jenkov.com/java-concurrency/volatile.html>
* Le block synchronized : <http://tutorials.jenkov.com/java-concurrency/synchronized.html>
* Les verrous : <http://tutorials.jenkov.com/java-concurrency/locks.html>
* Les verrous RW : <http://tutorials.jenkov.com/java-concurrency/read-write-locks.html>
* Les pools de thread : <http://tutorials.jenkov.com/java-concurrency/thread-pools.html>
* Compare and Swap : <http://tutorials.jenkov.com/java-concurrency/compare-and-swap.html>
* Algorithmique non bloquante : http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html