Les deux premiers exercices sont inspirés de l'article « CHECK-THEN-ACT Misuse of Java Concurrent Collections » *publié par Yu Lin et Danny Dig, de l'université de l'Illinois, à la sixième conférence ICST en 2013.*

Exercice IV.1 Problème d'atomicité avec une file concurrente Le bout de code donné sur la figure 18 s'appuie sur une file `maFile` rassemblant des éléments qui comportent un attribut public `m`. La méthode `poll()` retourne le premier élément de `maFile`, en le supprimant de la collection. S'il n'y a plus d'éléments dans la collection lors de l'appel à cette méthode, `poll()` retournera `null` [5].

Question 1. Indiquez en français ce que semble devoir faire ce bout de code.

Question 2. Quelle erreur peut se produire lors de l'exécution de ce bout de code ?

Question 3. Nous supposons ici que `maFile` est une collection *synchronisée*. Comment peut-on corriger ce code afin de garantir cette boucle d'affichages et de retraits s'effectue de manière atomique ?

Question 4. Nous supposons à présent que `maFile` est une collection *concurrente*. Comment peut-on corriger le code de la figure 18 afin d'empêcher l'exception d'être levée ?

```
while(! maFile.isEmpty()){
  Element e = maFile.poll() ;
  System.out.println(e.m) ;
}
```

FIGURE 18 – Mauvaise utilisation d'une file concurrente

Exercice IV.2 Problème d'atomicité avec une table de hâchage Le bout de code donné sur la figure 19 utilise une table de hâchage `maTable` qui rassemble une collection de paires, formées chacune d'une clef et d'une valeur non nulle. Par principe, il ne peut pas y avoir deux fois la même clef dans une table de hachâge [6]. L'avantage de ce type de collections est qu'elles permettent de retrouver rapidement la valeur associée à une clef donnée : elles sont donc, en pratique, fréquemment utilisées pour gérer des annuaires, des dictionnaires, etc. On rappelle que la méthode `get()` retourne la valeur associée à la clef donnée s'il y en a une, et `null` sinon, alors que la méthode `put()` insère la nouvelle paire donnée dans la table et renvoie la valeur précédemment associée à la clef s'il y en avait une, et `null` sinon.

Question 1. Indiquez en français ce que semble devoir faire ce bout de code.

Question 2. Quelle erreur peut se produire lors de l'exécution de ce bout de code ?

Question 3. On suppose à présent que la collection `maTable` appartient à la classe `ConcurrentHashMap`. Celle-ci propose la méthode `putIfAbsent(clef, v)` qui insère *de manière atomique* la paire (`clef`,`v`) dans la table, à condition que la clef n'y soit pas déjà associée à une valeur. Un peu comme `put()`, cette méthode renvoie la valeur associée à la clef s'il y en a déjà une avant l'appel (c'est-à-dire que l'insertion a échoué), et `null` sinon (c'est-à-dire que la nouvelle paire a bien été insérée). Comment peut-on corriger ce code à l'aide de `putIfAbsent()` afin de garantir qu'à la fin de ce bout de code la valeur `v` est *non nulle* et associée à la clef `clef` dans la table ?

```
Valeur v = maTable.get(clef);
if ( v == null ) {
  v = calc() ;          // renvoie toujours une valeur différente de null
  maTable.put(clef, v) ;
}
... // Utilisation de v ici
```

FIGURE 19 – Mauvaise utilisation d'une table de hachage

5. alors que la méthode `remove()` renverra une exception de type `NoSuchElementException`.
6. Par contre, une même valeur peut être associée à deux clefs différentes.

```
public class MonSémaphore {
  private volatile int nbTickets;     // Nombre de tickets disponibles

  public MonSémaphore(int n) {        // Constructeur
    if ( n<0 ) throw new IllegalArgumentException("Valeur initiale < 0");
    this.nbTickets = n;
  }

  synchronized public void P() {      // Pour prendre un ticket
    while (nbTickets < 1){            // Les conditions ne sont pas favorables
      try { wait(); }                 // Il en faut un pour pouvoir un prendre un
      catch(InterruptedException e){e.printStackTrace();}
    }
    nbTickets--;
  }

  synchronized public void V() {      // Pour donner un ticket
    nbTickets++;
    notifyAll();                      // Réveil de quiconque en attente d'un ticket
  }
}
```

FIGURE 20 – Le code d'un sémaphore sous la forme d'un moniteur

```
public class MonSémaphore {
  private AtomicInteger nbTickets = new AtomicInteger(0);

  public MonSemaphore(int n){
    if ( n<0 ) throw new IllegalArgumentException("Valeur initiale < 0");
    nbTickets.set(n);
  }

  public void P() {
    while (nbTickets.get() < 1) ;     // Attente active d'un ticket
    nbTickets.addAndGet(-1);
  }

  public void V() {
    nbTickets.addAndGet(1);
  }
}
```

FIGURE 21 – Le code erroné d'un sémaphore implémenté sans verrou à l'aide d'un AtomicInteger

```
class MonVerrou {
  private Thread propriétaire = null ;
  private int nbPrises = 0 ;

  public synchronized void verrouiller() throws InterruptedException {
    while(propriétaire != null && propriétaire != Thread.currentThread()) wait();
    propriétaire = Thread.currentThread() ;
    nbPrises++ ;
  }

  public synchronized void déverrouiller() {
    if (nbPrises > 0 && propriétaire == Thread.currentThread()) {
      nbPrises-- ;
      if ( nbPrises == 0 ) propriétaire = null ;
      notifyAll();
    }
  }
}
```

FIGURE 22 – Verrou réentrant sous la forme d'un moniteur

Exercice IV.3 Construction d'un sémaphore sans verrou La figure 20 présente une implémentation simple en Java du concept de sémaphore, sous la forme d'un moniteur. Un sémaphore est essentiellement un entier *positif* qui peut être modifié par deux opérations atomiques : l'incrémentation, notée traditionnellement `V()` et la décrémentation, notée `P()`, qui ne peut avoir lieu que si l'entier est supérieur à 1 : lorsque la valeur du sémaphore vaut 0, le thread qui demande à appliquer `P()` attend jusqu'à ce qu'il observe une autre valeur et puisse effectuer la décrémentation requise.

Dans cet exercice, vous devez proposer un code alternatif pour la classe `MonSémaphore` qui n'utilise aucun verrou d'aucune sorte, ni, bien entendu, la classe `Semaphore` disponible dans Java ; en particulier, les mots-clefs `synchronized`, `wait()` et `lock()` sont proscrits. En revanche, sans surprise, l'*attente active*, les *objets atomiques* et l'instruction `compareAndSet()` doivent être employés.

2015

Question 1. La figure 21 propose un codage sans verrou à l'aide d'un entier atomique. Ce codage est erroné. Indiquez un scenario problématique qui conduit à une valeur *strictement négative* du sémaphore, sous la forme d'un diagramme de séquence.

Question 2. Écrire un code correct en suivant le patron usuel pour la modification d'un objet atomique :

(a) obtenir une copie locale de la valeur courante de l'objet atomique ;

(b) préparer une modification de l'objet *à partir de la copie obtenue* ;

(c) appliquer une mise-à-jour atomique de l'objet conformément à l'étape 2, si sa valeur courante correspond encore à celle copiée ; et sinon, retourner en (a).

Question 3. Expliquez pourquoi les méthodes `P()` et `V()` sont correctes.

Exercice IV.4 Construction d'un verrou réentrant sans utiliser de verrou La figure 22 décrit un verrou réentrant implémenté sous la forme d'un moniteur. Comme le verrou intrinsèque des objets Java, un verrou réentrant peut être pris plusieurs fois par le même thread ; il doit alors être relâché autant de fois afin d'être libéré. Deux attributs sont naturellement utilisés : une référence du thread propriétaire, qui est `null` lorsque le verrou est libre ; le nombre de prises, qui est égal à zéro lorsque le verrou est libre.

Il s'agit dans cet exercice d'écrire une classe équivalente *sans utiliser le mot-clef* ***synchronized****, ni aucun type de verrou*, mais avec des objets atomiques. Les verrous réentrants possédant deux attributs, la technique générale illustrée ici dans cet exercice consiste, avant toute chose, à se ramener à des objets possédant *un seul attribut*, dont le rôle est d'encapsuler l'ensemble des champs utiles. Nous procéderons donc en deux temps.

Question 1. Modifier le moniteur de la figure 22 en remplaçant les deux attributs de cette classe par un seul, à savoir un objet de la classe `VerrouImmuable` décrite la figure 23. Notez que les objets de cette classe sont immuables et encapsulent les deux champs de la classe `MonVerrou` donnée. La nouvelle classe `MonVerrou` doit encore être un moniteur : toutes les méthodes seront donc déclarées `synchronized`.

Question 2. Écrire ensuite une classe `MonVerrou` équivalente à celle de la figure 22 *sans utiliser le mot-clef* ***synchronized****, ni aucun type de verrou*. Cette classe s'inspirera de votre réponse à la question précédente et disposera d'un seul attribut : une référence *atomique* vers un `VerrouImmuable`.

```
class VerrouImmuable {      // Les objets de cette classe sont manifestement immuables
  private final Thread propriétaire ;
  private final int nbPrises ;

  public VerrouImmuable (Thread propriétaire, int nbPrises) {
    this.propriétaire = propriétaire ;
    this.nbPrises = nbPrises ;
  }
  public int nbPrises() {
    return nbPrises ;
  }
  public Thread propriétaire() {
    return propriétaire ;
  }
}
```

FIGURE 23 – Objets immuables encapsulant les champs utiles d'un verrou réentrant