Exercice IV.1 Problème d'atomicité avec une file concurrente Le bout de code donné sur la figure 14 s'appuie sur une file `maFile` rassemblant des éléments qui comportent un attribut `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` [4].

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 14 afin d'empêcher l'exception d'être levée ?

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

FIGURE 14 – 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 15 utilise une table de hâchage `maTable` qui rassemble une collection de paires, formées chacune d'une clef et d'une valeur. Par principe, il ne peut pas y avoir deux fois la même clef dans une table de hachâge [5]. 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 ? Imaginez un contexte concret dans lequel cette erreur serait critique.

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()` ?

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

FIGURE 15 – Mauvaise utilisation d'une table de hachage

Exercice IV.3 Blanche-Neige équitable sans verrou Écrire une classe équivalente à celle décrite sur la figure 16, c'est-à-dire une Blanche-Neige équitable sous la forme d'un moniteur, *sans utiliser de verrou*. L'emploi de `wait()` étant de fait proscrit, l'attente du privilège d'accès s'opérera via une boucle d'attente active.

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

```
class BlancheNeige {
  private ArrayList<Thread> file = new ArrayList<Thread>();

  public synchronized void requerir() {
    file.add(Thread.currentThread());
  }

  public synchronized void acceder() throws InterruptedException {
    while( file.get(0) != Thread.currentThread() ) { wait(); }
  }

  public synchronized void relacher() {
    file.remove(0);
    notifyAll();
  }
}
```

FIGURE 16 – Blanche-Neige équitable sous la forme d'un moniteur (solution de l'exercice B.1)

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

  public MonSemaphore(int n) {        // Constructeur
    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 17 – Le code d'un sémaphore sous la forme d'un moniteur

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

  public MonSemaphore(int n){
    nbTickets.set(n);
  }

  public void P() {
    while (nbTickets.get() < 1) ;
    nbTickets.decrementAndGet();
  }

  public void V() {
    nbTickets.incrementAndGet();
  }
}
```

FIGURE 18 – Le code erroné d'un sémaphore sans verrou

Exercice IV.4 Construction d'un sémaphore sans verrou La figure 17 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.

Dans cet exercice, vous devez proposer un code alternatif pour la classe `MonSemaphore` 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, l'*attente active* et les *objets atomiques* doivent être employés.

2015

Question 1. La figure 18 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 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).

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

  public synchronized void verrouiller() throws InterruptedException {
    while( nbPrises > 0 && propriétaire != Thread.currentThread() ) { wait(); }
    propriétaire = Thread.currentThread() ;
    nbPrises++ ;
  }

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

FIGURE 19 – Verrou réentrant sous la forme d'un moniteur

Exercice IV.5 Construction d'un verrou réentrant sans utiliser de verrou Il s'agit dans cet exercice d'écrire une classe équivalente à celle décrite sur la figure 19 *sans utiliser le mot-clef synchronized, ni aucun type de verrou*. Pour celà, l'approche classique consiste consiste à écrire une version intermédiaire *avec un seul attribut* : une référence vers un objet immuable ; dans un deuxième temps, à supprimer l'emploi du verrou intrinsèque en utilisant une *référence atomique* en guise de seul attribut. Vous pourrez donc vous appuyer sur les objets immuables de la classe ci-dessous :

```
class MonVerrouImmuable {
  private final int nbPrises ;
  private final Thread propriétaire ;
  public MonVerrouImmuable (int nbPrises, Thread propriétaire) {
    this.nbPrises = nbPrises ;
    this.propriétaire = propriétaire ;
  }
  public int nbPrises() {
    return nbPrises ;
  }
  public Thread propriétaire() {
    return propriétaire ;
  }
}
```