Exercice IV.1

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

On est parfois bien embarassé pour dire ce que semble devoir faire un code lorsque l'on ne l'a pas soi-même écrit. Néanmoins ce bout de code est assez clair. Il s'agit de vider la file en affichant l'attribut `m` de chaque élément de la collection.

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

Dans un contexte multithread, `poll()` pourra retourner `null` si un autre thread vide la pile en parallèle alors que le tour de boucle est déjà engagé. Il y aura alors une `NullPointerException` lors de l'appel à `e.m`. Par ailleurs, certaines collections soulèveront une exception lors du `poll()` si la collection est vide.

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 ?*

La collection étant synchronisée, toutes les méthodes associées sont synchronisées : elles requièrent le verrou intrinsèque de l'objet avant de s'exécuter. On écrira donc :

```
synchronized( maFile ) {      // Plus aucun accès concurrent n'est possible
  while(! maFile.isEmpty()){
    Element e = maFile.poll() ;
    System.out.println(e.m) ;
  }
}
```

La file ne peut plus être modifiée par le reste du programme pendant l'exécution de ce bout de code, car le verrou est pris pendant toute la durée de l'itération.

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 ?*

Il serait contradictoire d'imposer un verrou sur chaque usage de la file après avoir opté pour une collection concurrente. Au contraire, le choix d'une collection concurrente indique que l'on souhaite autoriser les accès concurrents à la collection y compris pendant les itérations. Une solution simple :

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

Exercice IV.2

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

Ce bout de code manipule deux variables : une clef `clef` et une valeur `v`. Une chose est sûre : à l'issue de ce bout de code, `v` n'est pas `null`, puisque `calc()` ne renverra jamais `null`.

Une seconde chose est presque sûre : à l'issue de ce bout de code, `v` sera associée à la clef `clef` dans la table. En effet, de deux choses l'une : soit `v` est déjà associée à la clef `clef` dans la table, soit la méthode `get()` renvoie `null`, et dans ce cas une paire associant `v` à la clef `clef` est insérée dans la table. Seule une action simultanée du reste du programme pourra faire disparaître cette paire de la table.

Par ailleurs, ce bout de code semble prendre des précautions pour n'insérer une nouvelle paire (`clef`, `v`) dans la table que si `clef` n'est associée à aucune valeur. En particulier, ces quelques lignes ne doivent, semble-t-il, jamais effacer une paire déjà présente dans la table. En effet, l'instruction `put()` n'est réalisée qu'après une vérification de l'absence de valeur associée à la clef.

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

Dans un contexte multithread, une paire associée à la clef pourra être insérée juste après le test effectué par ce bout de code. Cette paire sera alors effacée de la table lors du `put()` réalisé par ce bout de code.

Autre scenario, ce bout de code peut éventuellement être exécuté simultanément par deux threads : le second `put()` réalisé effacera le premier et la valeur de `v` ne sera pas associée à la clef `clef` à l'issue du 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 ?*

Il s'agit de s'assurer que la valeur de la variable locale **`v`** est bien associée à **`clef`** à l'issue du bout de code, en ne supprimant aucune paire de la table. Voici un code possible :

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

De deux choses l'une :

— ou bien la valeur de **`v`** obtenue à la première ligne est non nulle, et on la garde (Le ≪if≫ ne fait rien) ; Dans ce cas, **`v`** est bien associée à **`clef`** et aucune paire n'est supprimée de la table.

— ou bien la valeur de **`v`** obtenue à la première ligne est nulle, car **`clef`** n'est associée à aucune valeur. Dans ce cas, le code tente, avec **`putIfAbsent()`**, d'insérer la paire (**`clef,v`**) dans la table. *Cette tentative peut échouer !* Néanmoins :

  — ou bien l'insertion a lieu lors de **`putIfAbsent()`** : (**`clef,v`**) est dans la table et aucune paire n'a donc été supprimée ;

  — ou bien l'insertion échoue lors de **`putIfAbsent()`** : Alors **`tmpV`** est la valeur associée à **`clef`** et cette valeur est ensuite recopiée dans **`v`**. Là encore, aucune paire n'a été supprimée de la table.

  Par conséquent, à la fin de ce code **`v`** est associée à **`clef`** et aucune paire n'a été supprimée de la table.

Le concept de sémaphore est très populaire et souvent utilisé en systèmes d'exploitation ou dans la programmation concurrente de bas niveau. Dans ce cours néanmoins, le conception de moniteur est privilégiée à l'utilisation de sémaphores.

Dans cet exercice, il s'agit de voir comment un sémaphore peut être construit non pas sous la forme d'un moniteur, mais sans verrou à l'aide d'un entier atomique. Cela permet de s'exercer sur un cas simple à la technique de programmation sans verrou vue en cours.

Pour illustrer l'usage des sémaphores, nous pourrons utiliser le programme `SaturdayNightFever` dans lequel les sept nains dansent avec Blanche-Neige, mais jamais plus de deux à la fois : il y a donc deux tickets disponibles et il faut disposer d'un ticket pour danser. Ce programme est indiqué sur la Table 23 et disponible dans l'archive `TD_IV.zip` déposée sur AMETICE. La méthode **`P()`** permet de prendre un ticket disponible alors que la méthode **`V()`** permet de rendre ce ticket. Une exécution de ce système construit à l'aide du sémaphore de la Figure 20 est présentée sur la Table 24.

```
public class SaturdayNightFever {

  public static void main(String[] args) throws InterruptedException {
    int nbNains = 7;
    String nom [] = { "Simplet", "Dormeur",  "Atchoum", "Joyeux",
      "Grincheux", "Prof", "Timide" };
    Nain nain [] = new Nain [nbNains];
    for(int i = 0; i < nbNains; i++) nain[i] = new Nain(nom[i]);
    for(int i = 0; i < nbNains; i++) nain[i].start();
  }
}

class Nain extends Thread {
  static final MonSémaphore blancheNeige = new MonSémaphore(2);
  public Nain(String nom) {
    this.setName(nom);
  }
  public void run() {
    Random aléa = new Random() ;
    while(true) {
      try {
        sleep(aléa.nextInt(1000));
      } catch (InterruptedException e) {e.printStackTrace();}
      System.out.println(getName() + "␣souhaite␣danser␣avec␣Blanche-Neige.");
      blancheNeige.P();         // Puis-je danser avec Blanche-Neige?
      System.out.println(getName() + "␣danse␣avec␣Blanche-Neige.");
      try {
        sleep(1000);
      } catch (InterruptedException e) {e.printStackTrace();}
      System.out.println(getName() + "␣va␣se␣reposer␣un␣peu.");
      blancheNeige.V();         // Vas-y, à ton tour!
    }
  }
}
```

TABLE 23 – Programme illustrant l'usage d'un sémaphore

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.*

La Table 25 illustre une exécution du programme des nains qui dansent lorsque l'on s'appuie sur le code de la Figure 21 en guise de sémaphore. On observe que lorsqu'il ne reste qu'un ticket, deux nains peuvent le prendre simultanément, car l'opération **`P()`** n'est pas atomique.

Plus simplement, si on définit dans cette classe un objet **`s`** initialisé à 1 et si on demande ensuite à deux threads d'effectuer **`s.P()`**, ces deux threads pourront simultanément constater qu'il y a un ticket disponible puis, à la ligne suivante, décrémenter le compteur de tickets de manière atomique. Ce scenario est décrit sur le diagramme de la Table 26.

```
$ java SaturdayNightFever
Grincheux souhaite danser avec Blanche-Neige.
Grincheux danse avec Blanche-Neige.
Dormeur souhaite danser avec Blanche-Neige.
Dormeur danse avec Blanche-Neige.
Joyeux souhaite danser avec Blanche-Neige.
Atchoum souhaite danser avec Blanche-Neige.
Prof souhaite danser avec Blanche-Neige.
Timide souhaite danser avec Blanche-Neige.
Simplet souhaite danser avec Blanche-Neige.
Grincheux va se reposer un peu.
Joyeux danse avec Blanche-Neige.
Dormeur va se reposer un peu.
Simplet danse avec Blanche-Neige.
Grincheux souhaite danser avec Blanche-Neige.
Dormeur souhaite danser avec Blanche-Neige.
Joyeux va se reposer un peu.
Atchoum danse avec Blanche-Neige.
^C
$
```

TABLE 24 – Exécution illustrant l'usage d'un sémaphore correctement implémenté

```
$ java SaturdayNightFever
Grincheux souhaite danser avec Blanche-Neige.
Grincheux danse avec Blanche-Neige.
Simplet souhaite danser avec Blanche-Neige.
Simplet danse avec Blanche-Neige.
Dormeur souhaite danser avec Blanche-Neige.
Joyeux souhaite danser avec Blanche-Neige.
Timide souhaite danser avec Blanche-Neige.
Prof souhaite danser avec Blanche-Neige.
Atchoum souhaite danser avec Blanche-Neige.
Grincheux va se reposer un peu.
Joyeux danse avec Blanche-Neige.
Dormeur danse avec Blanche-Neige.
Grincheux souhaite danser avec Blanche-Neige.
Simplet va se reposer un peu.
^C
$
```

TABLE 25 – Exécution illustrant l'erreur du code incorrect de la Figure 21

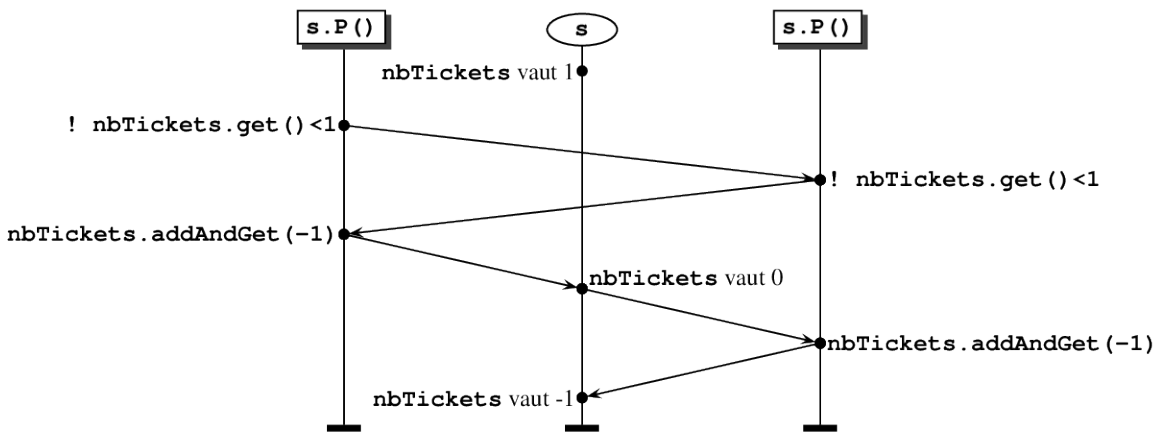

TABLE 26 – Diagramme expliquant l'erreur du code de la Figure 21

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).

De multiples codages respectant les principes de la recette du cours peuvent être proposés. En voici quelques uns.

(a) Tout d'abord, il semble naturel de boucler deux fois : d'abord pour attendre de manière active que les conditions soient favorables ; puis, au-dessus, pour s'assurer que l'on réussit au final l'écriture atomique souhaitée, en recommençant si nécessaire.

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

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

  public void P() {
    int nb;
    do {
      do {
        nb = nbTickets.get(); // On récupère une copie de la donnée jusqu'à
      } while (nb < 1) ;      // ce que les conditions soient favorables!
    } while (! nbTickets.compareAndSet(nb, nb-1)) ;
  }

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

Pour coder l'état d'un sémaphore à l'aide d'un objet atomique, il suffit d'utiliser un entier atomique. Cependant les **`AtomicInteger`** ne disposent pas de méthode qui effectue la décrémentation conditionnelle et bloquante que doit réaliser **`P()`** : on applique donc la recette du cours pour « enrichir » en quelque sorte l'objet **`AtomicInteger`** utilisé. Dans ce cas précis, la préparation de la nouvelle valeur potentielle est excessivement simple : c'est **`nb - 1`**.

En revanche, la méthode **`incrementAndGet()`** fait exactement ce qui est attendu de **`V()`** : inutile d'appliquer la recette. Notez que **`getAndIncrement()`** fait tout aussi bien.

(b) On peut simplifier le code de la méthode **`P()`** avec une seule boucle.

```
public void P() {
  for(;;) {
    int nb = nbTickets.get();   // Copie locale de la valeur de l'atomique
    if (nb > 0) {
      if (nbTickets.compareAndSet(nb, nb-1)) return ;
    }
  }
}
```

Le premier **`if`** assure l'attente active tant que les conditions sont défavorables ; dans le cas contraire, on tente la mise-à-jour voulue. Le second **`if`** assure que l'on recommencera une tentative si la modification de l'objet atomique échoue, c'est-à-dire si l'instruction **`compareAndSet()`** renvoie **`false`** car la copie réalisée n'est plus égale à la valeur courante de l'objet atomique.

Une variante, toujours avec deux **`if`**, mais non imbriqués, à l'aide de l'instruction **`continue`** :

```
public void P() {
  for(;;) {
    int nb = nbTickets.get();   // Copie locale de la valeur de l'atomique
    if (nb < 1) continue ;      // renvoie au for et donc au get().
    if (nbTickets.compareAndSet(nb, nb-1)) return ;
  }
}
```

(c) On peut encore simplifier le code de la méthode **`P()`** en condensant les deux **`if`** en un seul.

```
public void P() {
  for(;;) {
    int nb = nbTickets.get(); // On récupère une copie de la donnée
    if ( (nb > 0) && (nbTickets.compareAndSet(nb, nb-1)) ) return ;
  }
}
```

Si **`nb>0`** n'est pas vrai, c'est-à-dire que les conditions ne sont pas favorables, l'instruction **`compareAndSet()`** ne sera pas tentée [13] : c'est la raison pour laquelle nous employons ici **`&&`** et non **`&`** dans ce **`if`**. *Donc ce code est strictement équivalent aux deux précédents.*

On peut si on le souhaite remplacer l'instruction **`return`** par **`break`**.

(d) Ultime alternative : on peut, si on le souhaite, revenir à une boucle du type **`do ... while`**.

```
public void P() {
  int nb;
  do {
    nb = nbTickets.get(); // On récupère une copie de la donnée
  } while ( (nb < 1) || ! (nbTickets.compareAndSet(nb, nb-1)) )
}
```

Après la lecture de la valeur courante,
— si les conditions ne sont pas favorables
— ou si les conditions sont favorables, mais que la tentative d'écriture échoue
il faudra recommencer.

Ici encore, tant que **`nb<1`**, l'instruction **`compareAndSet()`** ne sera pas exécutée.

En revanche le code ci-dessous est erroné :

```
public void P() {
  int nb;
  do {
    nb = nbTickets.get(); // On récupère une copie de la donnée
    if ( nb < 1 ) continue ;
  } while ( ! (nbTickets.compareAndSet(nb, nb-1)) )
}
```

car **`continue`** est sans effet, puisqu'il conduira à exécuter l'instruction **`compareAndSet()`**.

13. C'est vrai en Java comme en C.

Pour situer le contexte et tester éventuellement le code chez soi, on peut modifier le programme des sept nains souhaitant un accès exclusif à Blanche-Neige en remplaçant le moniteur utilisé précédemment par un verrou [14]. Ce dernier sera pris trois fois pour s'assurer qu'il est bien *réentrant*. Ce programme est disponible dans l'archive `TD_IV.zip`.

```
public class SeptNains {
  public static void main(String[] args) {
    int nbNains = 7;
    String nom [] = {"Simplet", "Dormeur",  "Atchoum", "Joyeux",
                     "Grincheux", "Prof", "Timide"};
    Nain nain [] = new Nain [nbNains];
    for(int i = 0; i < nbNains; i++) nain[i] = new Nain(nom[i]);
    for(int i = 0; i < nbNains; i++) nain[i].start();
  }
}

class Nain extends Thread {
  private static MonVerrou verrou = new MonVerrou();

  public Nain(String nom) { this.setName(nom); }

  public void run() {
    while(true) {
      try {
        verrou.verrouiller();
        verrou.verrouiller();
        verrou.verrouiller();
        System.out.println(getName() + "␣accède␣à␣Blanche-Neige.");
        sleep(1000);
        System.out.println(getName() + "␣quitte␣Blanche-Neige.");
        verrou.déverrouiller();
        verrou.déverrouiller();
        verrou.déverrouiller();
        sleep(100);
      } catch (InterruptedException e) { break ;}
    }
  }
}
```

Il s'agit dans cet exercice de remplacer le moniteur de la classe **`MonVerrou`** de la Figure 22 par une classe équivalente qui ne s'appuie sur aucun verrou, mais qui emploie des objets atomiques. Suivant une approche classique, le nouvel objet ne va comporter qu'*un seul attribut : une référence atomique vers un objet qui encapsule les données.* Ce dernier sera immuable, pour faciliter l'analyse du fonctionnement (c'est-à-dire la preuve de sa correction). Pour illustrer cette approche, nous procédons 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`***.

Un moniteur équivalent reposant uniquement sur un **`VerrouImmuable`** est décrit sur la Table 27. Notez que les objets de la classe **`VerrouImmuable`** sont immuables : pour changer l'état du verrou, il faut changer d'objet !

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`***.

Un code sans verrou reposant uniquement sur une référence atomique vers un **`VerrouImmuable`** est décrit sur la table 28. Notez que pour construire un verrou, il faut d'abord construire un verrou immuable libre, puis créer une référence atomique vers ce verrou. Par ailleurs, pour être parfaitement équivalent au moniteur fourni, il faut gérer les interruptions en renvoyant le cas échéant une **`InterruptedException`**.

14. car Blanche-Neige n'était dans cet exemple rien d'autre qu'un verrou.

```
class MonVerrou {
  private VerrouImmuable étatDuVerrou = new VerrouImmuable(null, 0);
  public synchronized void verrouiller() throws InterruptedException {
    while (  étatDuVerrou.propriétaire() != null
             && étatDuVerrou.propriétaire() != Thread.currentThread() ) wait() ;
    int nb = étatDuVerrou.nbPrises() + 1 ;
    étatDuVerrou = new VerrouImmuable(Thread.currentThread(), nb) ;
  }
  public synchronized void déverrouiller() {
    Thread propriétaire = étatDuVerrou.propriétaire() ;
    int nb = étatDuVerrou.nbPrises() ;
    if ( nb > 0 && propriétaire == Thread.currentThread() ) {
      nb-- ;
      if ( nb == 0 ) propriétaire = null ;
      étatDuVerrou = new VerrouImmuable(propriétaire, nb) ;
      notifyAll();
    }
  }
}
```

TABLE 27 – Première étape : un seul attribut

```
class MonVerrou {
  private final AtomicReference<VerrouImmuable> verrou ;
  public MonVerrou() {
    VerrouImmuable verrouLibre = new VerrouImmuable(null, 0);
    this.verrou = new AtomicReference<VerrouImmuable>() ;
    this.verrou.set(verrouLibre);
  }
  public void verrouiller() throws InterruptedException {
    VerrouImmuable courant ;
    VerrouImmuable suivant ;
    for (;;) {
      if ( Thread.currentThread().interrupted() ) {
        throw new InterruptedException("Il y a eu une interruption!");
      }
      courant = verrou.get() ;
      int nbPrises = courant.nbPrises() ;
      Thread propriétaire = courant.propriétaire();
      if (  propriétaire != null && propriétaire != Thread.currentThread() )
        continue ; // Les conditions sont défavorables: il faut attendre!
      suivant = new VerrouImmuable(Thread.currentThread(), nbPrises+1) ;
      if ( verrou.compareAndSet(courant, suivant) ) return ;
    }
  }
  public void déverrouiller() {
    VerrouImmuable courant ;
    VerrouImmuable suivant ;
    for (;;) {
      courant = verrou.get() ;
      int nbPrises = courant.nbPrises() ;
      Thread propriétaire = courant.propriétaire() ;
      if ( ! (nbPrises > 0 && propriétaire == Thread.currentThread()) )
        break ; // Les prérequis ne sont pas remplis : il faut quitter!
      nbPrises-- ;
      if ( nbPrises == 0 ) propriétaire = null ;
      suivant = new VerrouImmuable(propriétaire, nbPrises);
      if ( verrou.compareAndSet(courant, suivant) ) return ;
    }
  }
}
```

TABLE 28 – Seconde étape : une référence atomique