Cet examen à distance s'appuie sur l'archive `TP_E.zip`*, disponible sur AMETICE, qui contient les fichiers et les sources des programmes à compléter. Vous créerez à la fin de l'épreuve une archive ZIP rassemblant ces fichiers et programmes complétés ; l'archive sera désignée selon votre nom suivant le format* `NOM_Prénom.zip`*. Elle devra être déposée aujourd'hui avant 11h sur AMETICE.*

En cas de panne d'AMETICE, cette archive pourra être transmise sous la forme d'un fichier attaché à un email destiné à **`remi.morin@univ-amu.fr`** *avec pour entête «* **`[EXAMEN POC] <votre nom>`***».*

L'archive `TP_E.zip` *comporte trois dossiers correspondants aux trois exercices de cet examen.*

- *Le premier exercice porte sur la corde des babouins du TP D. Vous répondrez aux questions en complétant le fichier* `readme_corde.txt` *du dossier* `1_-_Corde_sans_verrou`*.*
- *Le second exercice consiste à concevoir un moniteur Blanche-Neige. Vous répondrez en complétant le programme* `BlancheNeige.java` *du dossier* `2_-_Association_de_nains` *et le fichier ASCII* `readme_bn.txt`*. Ce dernier illustrera la compilation et l'exécution des programmes dans un terminal. Les programmes Java de ce dossier devront pouvoir être exécutés par les commandes usuelles dans un terminal Unix :* **`javac *.java; java Main`**
- *Le troisième exercice consiste à utiliser un threadpool pour réduire le temps d'exécution d'un calcul sur un tableau. Vous répondrez en complétant le programme* `Inversions.java` *disponible dans le répertoire* `3_-_Inversions`*. Ce programme devra pouvoir être exécuté par les commandes usuelles dans un terminal Unix :* **`javac *.java; java Inversions`***. Enfin, le fichier* `readme_inversions.txt` *illustrera la compilation et l'exécution du programme complété.*

Exercice 1 *(La corde au dessus du canyon — 6 points)*

Nous reprenons à l'identique le problème de la corde et des babouins étudié lors du TP D.

Il y a un canyon très profond quelque part dans le parc national Kruger, en Afrique du Sud, et une corde unique qui enjambe le canyon. Les babouins qui vivent là peuvent traverser le canyon à l'aide de la corde en se balançant d'une main sur l'autre. Cependant, lorsque deux babouins avancent dans des directions opposées, ils finissent par se rencontrer au dessus du canyon, car un babouin ne recule jamais : ils se battent alors et chutent mortellement tous les deux. Par ailleurs, la corde est juste assez solide pour supporter le poids de 5 babouins. S'il y a plus de babouins sur la corde en même temps, elle se rompt et provoque la chute mortelle de tous les babouins en train de traverser.
Nous supposerons dans cet exercice qu'il est possible d'enseigner aux babouins à utiliser la corde sans risque afin que toute traversée entâmée par un babouin se termine sans combat, ni surcharge. Pour cela, les babouins doivent simplement attendre pour s'engager sur la corde que les conditions soient favorables. Ainsi, pour éviter les chutes, l'utilisation de la corde doit satisfaire deux propriétés :

① Il n'y a jamais 2 babouins sur la corde qui avancent en sens opposés : un babouin qui observe sur la corde un congénère avançant vers lui devra donc attendre avant de s'engager.

② Il n'y a jamais plus de 5 babouins sur la corde : un babouin qui observe 5 de ses congénères en train de se balancer sur la corde devra donc attendre avant de s'engager.

Le respect de ces deux conditions garantit que toutes les traversées entamées se terminent avec succès.
Le code du système complet est donné sur la figure 1. Il comporte une énumération qui contient deux objets : EST et OUEST, qui représentent les deux côtés du canyon et un programme principal qui lance une vingtaine de babouins répartis sur les deux côtés du canyon. Il définit également par la méthode **`run()`** de la classe **`Babouin`** le comportement de ces threads et implémente la classe **`Corde`** sous la forme d'un moniteur.

Vous aviez à produire dans le TP D une classe **`Corde`** équivalente à celle définie sur la figure 1, sans utiliser de verrou mais à l'aide d'objets atomiques.

Question 1. Une première proposition erronée consiste à construire une corde munie de deux entiers atomiques, telle que celle décrite sur la figure 2. Expliquez de manière détaillée dans quelle situation cette corde pourrait permettre à *six* babouins de traverser le canyon simultanément dans le même sens.

Question 2. Une seconde proposition erronée consiste à construire une corde munie à nouveau de deux entiers atomiques et implémentée selon le code de la figure 3. Expliquez de manière détaillée dans quelle situation cette corde pourrait permettre à deux babouins de traverser le canyon simultanément en sens opposés.

Question 3. Une troisième proposition erronée consiste à construire une corde consistant en une référence atomique vers un objet immuable encapsulant l'état de la corde (ce qui est une bonne idée) et à coder les méthodes **`saisir()`** et **`lâcher()`** comme indiqué sur la figure 4 (ce qui est *presque* une bonne idée). Expliquez de manière détaillée dans quelle situation cette corde pourrait permettre à deux babouins de traverser le canyon simultanément en sens opposés.

```
enum Côté { EST, OUEST }                    // Le canyon possède un côté EST et un côté OUEST

class Babouin extends Thread{
  private final Côté origine;       // Côté du canyon où apparaît le babouin: EST ou OUEST
  private static final Corde corde = new Corde();  // Corde utilisée par tous les babouins
  Babouin(Côté origine, int i) {                    // Constructeur de la classe Babouin
    this.origine = origine;          // Chaque babouin apparaît d'un côté précis du canyon
    if (origine == Côté.EST) setName("E"+i);
    else setName("O"+i);
  }
  public Côté origine() {
    return origine ;
  }
  public void run() {
    System.out.println("Le babouin " + getName() + " arrive sur le côté " + origine);
    try {
      corde.saisir();                       // Pour traverser, le babouin saisit la corde
    } catch(InterruptedException e){e.printStackTrace();}
    System.out.println("\t Le babouin " + getName() + " commence à traverser.");
    try {
      sleep(5000);                                  // La traversée ne dure que 5 secondes
    } catch(InterruptedException e){e.printStackTrace();}

    System.out.println("\t\t Le babouin " + getName() + " a terminé sa traversée.");
    corde.lâcher();                   // Arrivé de l'autre côté, le babouin lâche la corde
  }
  public static void main(String[] args) {
    for (int i = 1; i < 20; i++){
      try { Thread.sleep(2000); } catch(InterruptedException e){e.printStackTrace();}
      if (Math.random() >= 0.5){
        new Babouin(Côté.EST, i).start();                 // Création d'un babouin à l'est
      } else {
        new Babouin(Côté.OUEST, i).start();              // Création d'un babouin à l'ouest
      }
    }              // Une vingtaine de babouins sont répartis sur les deux côtés du canyon
  }
}

class Corde {
  private int nbEntréesOuest = 0;
  private int nbEntréesEst = 0;
  public synchronized void saisir() throws InterruptedException {
    Babouin courant = (Babouin) Thread.currentThread() ;
    Côté origine = courant.origine() ;
    if (origine == Côté.EST) {                            // Le babouin arrive du côté Est
      while ((nbEntréesOuest > 0) || (nbEntréesEst >= 5)) wait();
      nbEntréesEst++ ;
    } else {                                             // Le babouin arrive du côté Ouest
      while ((nbEntréesEst > 0) || (nbEntréesOuest >= 5)) wait();
      nbEntréesOuest++ ;
    }
  }
  public synchronized void lâcher() {
    Babouin courant = (Babouin) Thread.currentThread() ;
    Côté origine = courant.origine() ;
    if (origine == Côté.EST) {
      nbEntréesEst-- ;
    } else {
      nbEntréesOuest-- ;
    }
    notifyAll();
  }
}
```

FIGURE 1 – Code du programme `Babouin.java` fourni lors du TP D

```
class Corde {
  private final AtomicInteger nbOuest = new AtomicInteger(0) ;
  private final AtomicInteger nbEst = new AtomicInteger(0) ;

  public void saisir() throws InterruptedException {
    Babouin courant = (Babouin) Thread.currentThread() ;
    Côté origine = courant.origine() ;
    if (origine == Côté.EST) {                        // Le babouin arrive du côté Est
      while ((nbOuest.get() > 0) || (nbEst.get() >= 5)) {
        if( Thread.currentThread().interrupted() ) throw new InterruptedException() ;
      } ;
      nbEst.incrementAndGet() ;
    } else {                                          // Le babouin arrive du côté Ouest
      while ((nbEst.get() > 0) || (nbOuest.get() >= 5)) {
        if( Thread.currentThread().interrupted() ) throw new InterruptedException() ;
      } ;
      nbOuest.incrementAndGet() ;
    }
  }
  public void lâcher() {
    Babouin courant = (Babouin) Thread.currentThread() ;
    Côté origine = courant.origine() ;
    if (origine == Côté.EST) nbEst.decrementAndGet() ;
    else nbOuest.decrementAndGet() ;
  }
}

```

FIGURE 2 – Premier code erroné d'une corde sans verrou

```
class Corde {
  private final AtomicInteger nbOuest = new AtomicInteger(0) ;
  private final AtomicInteger nbEst = new AtomicInteger(0) ;

  public void saisir() throws InterruptedException {
    Babouin courant = (Babouin) Thread.currentThread() ;
    Côté origine = courant.origine() ;
    for(;;){
      if( Thread.currentThread().interrupted() ) throw new InterruptedException() ;
      int nbOuestCourantes=nbOuest.get() ;
      int nbEstCourantes=nbEst.get() ;
      if (origine == Côté.EST) {                      // Le babouin arrive du côté Est
        if ((nbOuestCourantes > 0) || (nbEstCourantes >= 5)) continue ;
        if (nbEst.compareAndSet(nbEstCourantes,nbEstCourantes+1)) return ;
      } else {                                        // Le babouin arrive du côté Ouest
        if ((nbEstCourantes > 0) || (nbOuestCourantes >= 5)) continue ;
        if (nbOuest.compareAndSet(nbOuestCourantes,nbOuestCourantes+1)) return ;
      }
    }
  }
  public void lâcher() {
    Babouin courant = (Babouin) Thread.currentThread() ;
    Côté origine = courant.origine() ;
    if (origine == Côté.EST) nbEst.decrementAndGet() ;
    else nbOuest.decrementAndGet() ;
  }
}

```

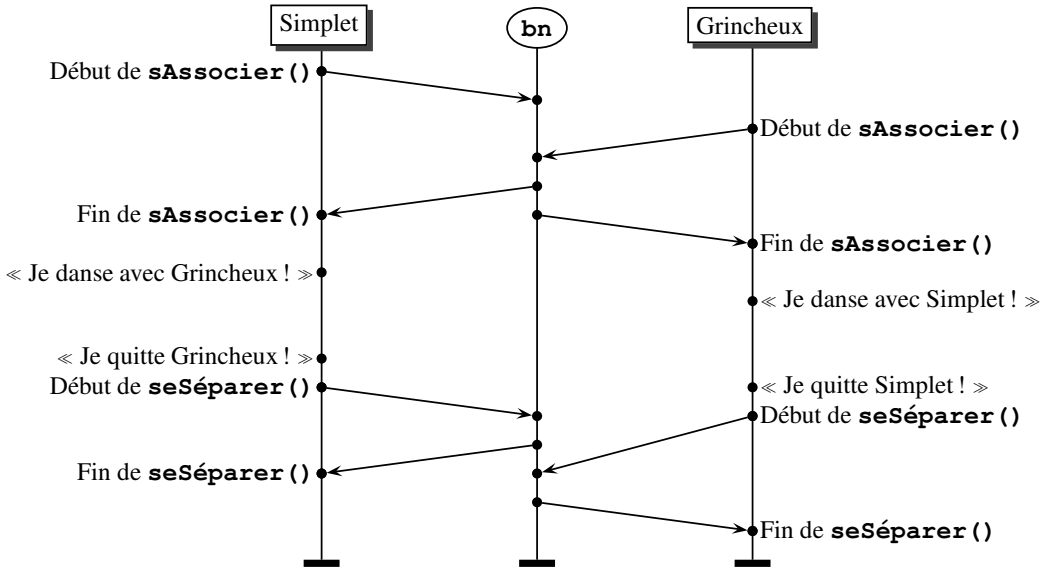
FIGURE 3 – Second code erroné d'une corde sans verrou

```
class CordeImmuable {
  private final int nbEst;            // Le nombre de babouins qui traversent depuis l'Est
  private final int nbOuest;          // et depuis l'Ouest

  public CordeImmuable(int nbEst, int nbOuest){
    this.nbEst = nbEst;
    this.nbOuest = nbOuest;
  }

  public int nbOuest(){
    return nbOuest;
  }

  public int nbEst(){
    return nbEst;
  }
}

class Corde {
  private final AtomicReference<CordeImmuable> corde = new AtomicReference<CordeImmuable>();

  public Corde(){
    this.corde.set(new CordeImmuable(0, 0));  // La corde est initialement libre
  }

  public void saisir() throws InterruptedException {
    Côté origine = ((Babouin) Thread.currentThread()).origine();
    for (;;) {
      if (Thread.currentThread().interrupted()) throw new InterruptedException();
      int nbEst = corde.get().nbEst();
      int nbOuest = corde.get().nbOuest();
      CordeImmuable courante = corde.get();
      if (origine == Côté.EST) {                          // Le babouin arrive du côté Est
        if ((nbOuest > 0) || (nbEst >= 5)) continue;
        nbEst++;
      } else {                                           // Le babouin arrive du côté Ouest
        if ((nbEst > 0) || (nbOuest >= 5)) continue;
        nbOuest++;
      }
      CordeImmuable suivante = new CordeImmuable(nbEst, nbOuest);
      if (corde.compareAndSet(courante, suivante)) return;
    }
  }

  public void lâcher() {
    Côté origine = ((Babouin) Thread.currentThread()).origine();
    for (;;) {
      int nbEst = corde.get().nbEst();
      int nbOuest = corde.get().nbOuest();
      CordeImmuable courante = corde.get();
      if (origine == Côté.EST) nbEst--;
      else nbOuest--;
      CordeImmuable suivante = new CordeImmuable(nbEst, nbOuest);
      if (corde.compareAndSet(courante, suivante)) return;
    }
  }
}
```

FIGURE 4 – Troisième code erroné d'une corde sans verrou

Exercice 2 *(Association de nains en binômes — 8 points)* Nous considérons dans cet exercice un système partiellement décrit par les programmes `Main.java` et `Nain.java` des figures 5 et 6. Ce système est composé de sept threads, les sept nains, qui doivent s'associer en couple pour danser ensemble pendant 3 secondes en compagnie de Blanche-Neige, les uns après les autres.

Pour pouvoir commencer à danser après son réveil, chaque nain demande à l'objet partagé **`bn`** de la classe **`BlancheNeige`** de lui indiquer quel est son partenaire en faisant appel à la méthode **`sAssocier()`**. Si aucun nain ne l'a encore appelée, un appel à cette méthode bloque jusqu'à ce qu'un second nain l'appelle, ce qui permet de finaliser une association. Cette dernière se forme et se dissout conformément au diagramme ci-dessous :



En particulier, chaque nain appelle une seule fois la méthode **`sAssocier()`** puis la méthode **`seSéparer()`**.

L'objectif de cet exercice est de compléter sans les modifier les programmes `Main.java` et `Nain.java` fournis dans l'archive `TP_E.zip` en implémentant la classe **`BlancheNeige`** sous la forme d'un moniteur dans un programme `BlancheNeige.java` afin d'obtenir, une fois l'ensemble de ces trois programmes compilés et le programme `Main` lancé, une exécution produisant un affichage du type suivant (sans les couleurs) :

```
[ 08h 05mn 28,883s ] Simplet veut danser.
[ 08h 05mn 29,097s ] Grincheux veut danser.
[ 08h 05mn 29,098s ] Simplet danse avec Grincheux
[ 08h 05mn 29,098s ] Grincheux danse avec Simplet
[ 08h 05mn 29,471s ] Dormeur veut danser.
[ 08h 05mn 29,527s ] Atchoum veut danser.
[ 08h 05mn 29,616s ] Joyeux veut danser.
[ 08h 05mn 30,572s ] Timide veut danser.
[ 08h 05mn 30,633s ] Prof veut danser.
[ 08h 05mn 32,116s ] Grincheux quitte Simplet
[ 08h 05mn 32,116s ] Simplet quitte Grincheux
[ 08h 05mn 32,117s ] Joyeux danse avec Atchoum
[ 08h 05mn 32,117s ] Atchoum danse avec Joyeux
[ 08h 05mn 35,122s ] Joyeux quitte Atchoum
[ 08h 05mn 35,122s ] Atchoum quitte Joyeux
[ 08h 05mn 35,123s ] Prof danse avec Timide
[ 08h 05mn 35,123s ] Timide danse avec Prof
[ 08h 05mn 38,128s ] Prof quitte Timide
[ 08h 05mn 38,128s ] Timide quitte Prof
[ 08h 05mn 43,132s ] Dormeur ne dansera pas ce soir!
```

Sans surprise, le premier nain qui se réveille doit attendre qu'un second nain se réveille pour être associé avec lui. Par ailleurs, une nouvelle association ne peut être formée sans que la précédente ne soit dissoute, c'est-à-dire que les deux nains aient fini de danser et se soient séparés.

Puisqu'il y a sept nains, qu'ils s'associent en couple avant de danser et qu'ils ne s'associent qu'une seule fois, l'un des sept nains ne pourra pas s'associer : il ne dansera pas. Lorsqu'un nain est réveillé, qu'il appelle la méthode **`sAssocier()`** et qu'il n'obtient pas de partenaire pendant plus de 5 secondes alors qu'aucun nain ne danse, la méthode **`sAssocier()`** lui renvoie **`null`** pour lui signifier qu'il n'obtiendra pas d'association.

Vous compléterez le fichier ASCII `readme_bn.txt` afin qu'il illustre la compilation et l'exécution des programmes via les commandes usuelles (c'est-à-dire **`javac *.java; java Main`**) dans un terminal.

```
public class Main {
  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]);
      nain[i].start();
    }
  }
}
```

FIGURE 5 – Le programme `Main.java`

```
public class Nain extends Thread {
  private static BlancheNeige bn = new BlancheNeige();

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

  private String dateEtNom() {
    final SimpleDateFormat sdf = new SimpleDateFormat("hh'h ''mm'mn ''ss','SSS's'");
    Date maDate = new Date(System.currentTimeMillis());
    return "[ "+sdf.format(maDate)+" ] " + getName();
  }

  public void run(){
    try {
      Random aléa = new Random();
      sleep(1000 + aléa.nextInt(2000));
      System.out.println(dateEtNom() + " veut danser.");
      Nain partenaire = bn.sAssocier();
      if ( partenaire == null ) {
        System.out.println(dateEtNom() + " ne dansera pas ce soir!");
      } else {
        System.out.println(dateEtNom() + " danse avec " + partenaire.getName());
        sleep(3000);
        System.out.println(dateEtNom() + " quitte " + partenaire.getName());
        bn.seSéparer();
      }
    } catch (InterruptedException e) { e.printStackTrace(); }
  }
}
```

FIGURE 6 – Le programme `Nain.java`

```java
public class Inversions {
    static final int taille = 100_000 ;
    static final int [] tableau = new int[taille] ;  // Tableau de la taille voulue
    static final int borne = 10 * taille ;           // Valeur maximale dans le tableau

    private static long nbInversions() {
        long résultat = 0;
        for (int i = 0; i < taille-1; i++) {
            for (int j = i+1; j < taille; j++) {
                if ( tableau[i] > tableau[j] ) résultat++;
            }
        }
        return résultat;
    }

    public static void main(String[] args) {
        Random aléa = new Random() ;
        for (int i=0 ; i<taille ; i++) {              // Remplissage aléatoire du tableau
            tableau[i] = aléa.nextInt(2*borne) - borne ;
        }
        System.out.println("Taille du tableau choisi = " + taille) ;
        long début1 = System.nanoTime() ;
        long résultat1 = nbInversions() ;
        long fin1 = System.nanoTime() ;
        long durée1 = (fin1 - début1) / 1_000_000 ;
        System.out.print("Nb d'inversions = " + résultat1) ;
        System.out.println(" obtenu en " + durée1 + " ms en séquentiel.") ;
    }
}
```

FIGURE 7 – Code du programme `Inversions.java` fourni

Exercice 3 *(Emploi d'un threadpool — 6 points)* Dans un tableau d'entiers, le nombre d'inversions est le nombre de fois où l'on observe deux éléments tels que le premier est strictement supérieur au second, c'est-à-dire que ces deux éléments ne sont pas placés dans l'ordre croissant. Le programme décrit sur la figure 7 permet de calculer le nombre d'inversions dans un tableau choisi aléatoirement. Pour chaque indice i du tableau et pour chaque indice j supérieur à i, il y a une inversion si l'élément à l'indice i est strictement supérieur à l'élément à l'indice j.

Ce programme est fourni dans l'archive `TP_E.zip`. Vous pourrez le compiler et l'exécuter en lançant les commandes usuelles :

```
$ javac Inversions.java
$ java Inversions
Taille du tableau choisi = 100000
Nb d'inversions = 2502243250 obtenu en 5519 ms en séquentiel.
```

L'objectif de cet exercice est de compléter le programme `Inversions.java` de telle sorte qu'après avoir affiché le temps nécessaire pour le calcul du nombre d'inversions de manière séquentielle, le temps observé en utilisant un threadpool formé de 4 threads pour effectuer le même calcul (sur le même tableau) soit également affiché. Le gain, c'est-à-dire le rapport du temps séquentiel sur le temps parallèle, devra également être affiché. Ainsi, votre programme devra produire une exécution produisant un affichage de ce type :

```
$ javac Inversions.java
$ java Inversions
Taille du tableau choisi = 100000
Nb d'inversions = 2505898856 obtenu en 5515 ms en séquentiel.
Nb d'inversions = 2505898856 obtenu en 1842 ms en parallèle avec 4 threads.
Gain: 2,99
```

Vous choisirez en guise de test final conservé dans un fichier `readme_inversions.txt` de lancer votre programme avec une taille de tableau nécessitant approximativement entre 10 et 60 secondes d'exécution en séquentiel.