Les documents, les calculatrices et les téléphones sont interdits.

L'évaluation des copies prendra en compte la clarté et la précision des réponses produites.

Exercice 1 *(Les petit-déjeuners des Hobbits : 8 points)* Le programme `PetitDejeuner.java` de la Figure 1 lance 3 threads Hobbits qui doivent, autour d'une table commune, prendre *ensemble* deux petits-déjeuners. Chaque repas d'un Hobbit dure au maximum cinq secondes. Pour prendre un repas, un Hobbit doit d'abord s'asseoir à table. Il se lève lorsqu'il a fini. Les deux petits-déjeuners d'un Hobbit sont séparés par une période de digestion d'une durée aléatoire (entre 0 et 3 secondes). Pour être correcte, une exécution doit produire une sortie écran dans laquelle les trois Hobbits prennent ensemble chacun de leurs petits-déjeuners : en particulier, *aucun Hobbit ne commence un petit-déjeuner avant que les deux autres aient faim.* Une exécution correcte et une exécution incorrecte sont données sur les Figures 3 et 4.

Question 1. Complétez le code de la Figure 1 en proposant une implémentation de la classe **`Table`**, sous la forme d'un moniteur, qui garantit la synchronisation des petits-déjeuners des Hobbits. Ni le **`main`** ni la classe **`Hobbit`** ne doivent être modifiés.

Question 2. Expliquez précisément pourquoi la table implémentée à la question 1 garantit qu'aucun Hobbit ne peut commencer son second petit déjeuner, avant que les deux autres aient fini leur premier petit-déjeuner.

```
1  public class PetitDejeuner {
2    public static void main(String[] args) {
3      final int nbHobbits = 3 ;
4      final String nom [] = {"Bilbo", "Frodo", "Peregrin"} ;
5      Table table = new Table(nbHobbits) ;
6      for(int i = 0; i < nbHobbits; i++) new Hobbit(nom[i], table).start();
7    }
8  }
9
10 class Table {...}
11
12 class Hobbit extends Thread {
13   Table table ;
14   Random aléa = new Random() ;
15   public Hobbit(String nom, Table table) {
16     this.setName(nom) ;
17     this.table = table ;
18   }
19   private void affiche(String message) {
20     SimpleDateFormat sdf = new SimpleDateFormat("'['hh'h 'mm'mn 'ss','SSS's] '");
21     System.out.println(sdf.format(new Date(System.currentTimeMillis()))
22                        + Thread.currentThread().getName() + " " + message);
23   }
24   private void petitDejeuner(String repas) {
25     affiche("commence son "+repas+" petit-déjeuner.");
26     try { sleep(aléa.nextInt(5000)); } catch(InterruptedException e){ e.printStackTrace(); }
27     affiche("termine son "+repas+" petit-déjeuner.");
28   }
29   public void run() {
30     try { sleep(aléa.nextInt(3000)); } catch(InterruptedException e){ e.printStackTrace(); }
31     affiche("est réveillé. Il a faim!");
32     table.sAsseoir() ;
33     petitDejeuner("1er") ;
34     table.seLever() ;
35     try { sleep(aléa.nextInt(3000)); } catch(InterruptedException e){ e.printStackTrace(); }
36     affiche("a faim de nouveau!");
37     table.sAsseoir() ;
38     petitDejeuner("2nd") ;
39     table.seLever() ;
40   }
41 }
```

FIGURE 1 – Code à compléter de l'exercice 1

Exercice 2 *(Ping-pong en double : 6 points)* Le programme `PingPong.java` de la Figure 2 lance 2 threads Pongistes positionnés de chaque côté d'une table (à droite et à gauche). Ces quatre Pongistes doivent renvoyer la balle partagée trois fois chacun. Naturellement, ils attendent que la balle soit passée de leur côté avant de la renvoyer.

Une exécution de ce programme est donnée sur la Figure 5. L'alternance des affichages des « PING » et des « PONG » confirme le blocage des Pongistes tant que la balle n'est pas de leur côté de la table.

Question 1. Peut-on remplacer l'instruction **`notifyAll()`** par l'instruction **`notify()`** dans le code de la méthode **`renvoyer()`** de la classe **`Balle`** ? Pourquoi ?

Question 2. Écrire une classe équivalente à la classe **`Balle`** sans utiliser de verrou, à l'aide d'objets atomiques. En particulier, les affichages des « PING » et des « PONG » doivent être conservés et illustrer l'alternance des renvois. Ni le **`main`** ni la classe **`Pongiste`** ne doivent être modifiés.

```
1  public class PingPong {
2    public static void main(String[] args) {
3      Balle balle = new Balle(Côté.DROITE) ;
4      new Pongiste(balle, Côté.DROITE, "D1").start();  // Le pongiste D1 est à droite
5      new Pongiste(balle, Côté.DROITE, "D2").start();  // Le pongiste D2 est à droite
6      new Pongiste(balle, Côté.GAUCHE, "G1").start();  // Le pongiste G1 est à gauche
7      new Pongiste(balle, Côté.GAUCHE, "G2").start();  // Le pongiste G2 est à gauche
8    }
9  }
10
11 enum Côté {DROITE, GAUCHE}                  // La balle passe des deux côtés de la table
12
13 class Balle {
14   Côté côté = null ;
15   public Balle(Côté côté) {
16     this.côté = côté ;
17   }
18   synchronized public void renvoyer(Côté position) {
19     while ( côté != position ) {       // Si la balle n'est pas de mon côté, j'attends
20       try{ wait() ; } catch (InterruptedException e) {e.printStackTrace() ; }
21     }
22     if (position == Côté.DROITE) {     // Si je suis à droite,
23       côté = Côté.GAUCHE ;             // je renvoie la balle à gauche
24       System.out.println(Thread.currentThread().getName()+" fait PING");
25     }
26     else {                             // Si je suis à gauche,
27       côté = Côté.DROITE;              // je renvoie la balle à droite
28       System.out.println(Thread.currentThread().getName()+" fait PONG");
29     }
30     notifyAll() ;
31   }
32 }
33
34 class Pongiste extends Thread {
35   final long nbRenvois = 3 ;
36   final Balle balle ;
37   final Côté position ;
38   public Pongiste(Balle balle, Côté position, String nom) {
39     this.balle = balle ;
40     this.position = position ;
41     this.setName(nom) ;
42   }
43   public void run() {
44     for (int i = 0 ; i < nbRenvois ; i++) balle.renvoyer(position) ;
45   }
46 }
```

FIGURE 2 – Code de l'exercice 2

```
% java PetitDejeuner.java
[09h 33mn 32,670s] Peregrin est réveillé. Il a faim!
[09h 33mn 33,271s] Bilbo est réveillé. Il a faim!
[09h 33mn 33,848s] Frodo est réveillé. Il a faim!
[09h 33mn 33,854s] Bilbo commence son 1er petit-déjeuner.
[09h 33mn 33,854s] Peregrin commence son 1er petit-déjeuner.
[09h 33mn 33,854s] Frodo commence son 1er petit-déjeuner.
[09h 33mn 36,185s] Peregrin termine son 1er petit-déjeuner.
[09h 33mn 36,187s] Peregrin a faim de nouveau!
[09h 33mn 37,090s] Frodo termine son 1er petit-déjeuner.
[09h 33mn 37,093s] Frodo a faim de nouveau!
[09h 33mn 38,492s] Bilbo termine son 1er petit-déjeuner.
[09h 33mn 38,494s] Bilbo a faim de nouveau!
[09h 33mn 38,495s] Frodo commence son 2nd petit-déjeuner.
[09h 33mn 38,495s] Bilbo commence son 2nd petit-déjeuner.
[09h 33mn 38,495s] Peregrin commence son 2nd petit-déjeuner.
[09h 33mn 41,780s] Bilbo termine son 2nd petit-déjeuner.
[09h 33mn 41,806s] Frodo termine son 2nd petit-déjeuner.
[09h 33mn 42,467s] Peregrin termine son 2nd petit-déjeuner.
```

FIGURE 3 – Une exécution correcte

```
% java PetitDejeuner.java
[09h 33mn 32,670s] Peregrin est réveillé. Il a faim!
[09h 33mn 33,271s] Bilbo est réveillé. Il a faim!
[09h 33mn 33,848s] Frodo est réveillé. Il a faim!
[09h 33mn 33,854s] Bilbo commence son 1er petit-déjeuner.
[09h 33mn 33,854s] Peregrin commence son 1er petit-déjeuner.
[09h 33mn 33,854s] Frodo commence son 1er petit-déjeuner.
[09h 33mn 36,185s] Peregrin termine son 1er petit-déjeuner.
[09h 33mn 36,187s] Peregrin a faim de nouveau!
[09h 33mn 37,090s] Frodo termine son 1er petit-déjeuner.
[09h 33mn 38,093s] Frodo a faim de nouveau!
[09h 33mn 38,095s] Frodo commence son 2nd petit-déjeuner.
[09h 33mn 38,492s] Bilbo termine son 1er petit-déjeuner.
[09h 33mn 38,494s] Bilbo a faim de nouveau!
[09h 33mn 38,495s] Bilbo commence son 2nd petit-déjeuner.
[09h 33mn 38,495s] Peregrin commence son 2nd petit-déjeuner.
[09h 33mn 41,780s] Bilbo termine son 2nd petit-déjeuner.
[09h 33mn 41,806s] Frodo termine son 2nd petit-déjeuner.
[09h 33mn 42,467s] Peregrin termine son 2nd petit-déjeuner.
```

FIGURE 4 – Une exécution incorrecte

```
% java PingPong.java
D1 fait PING
G2 fait PONG
D2 fait PING
G1 fait PONG
D2 fait PING
G1 fait PONG
D2 fait PING
G1 fait PONG
D1 fait PING
G2 fait PONG
D1 fait PING
G2 fait PONG
```

FIGURE 5 – Une exécution du programme Ping-Pong

Exercice 3 *(Réservoir de threads : 3 points)* Dans un système client-serveur, le rôle du serveur est d'accepter chaque requête soumise et d'y répondre. Le code simplifié d'un serveur est indiqué sur la Figure 6 : chaque appel à la méthode **`accepterUneRequête()`** renvoie une nouvelle requête acceptée qui est modélisée par un objet de la classe **`Requête`** et à laquelle il est répondu lors de l'application de la méthode **`traiterUneRequête()`** de la classe **`Requête`** sur cet objet. Le traitement d'une requête est indépendant du traitement des autres requêtes soumises : ce traitement peut donc être réalisé en parallèle. Écrire un code alternatif à celui de la Figure 6 qui utilise un réservoir de 4 threads afin de permettre le traitement de plusieurs requêtes en parallèle.

```
while ( ! arrêtDuServeur ) {
  Requête r = accepterUneRequête();
  r.traiterUneRequête();
}
```

FIGURE 6 – Code simplifié d'un serveur

Exercice 4 *(Choses à savoir : 3 points)*

Question 1. En entreprise, vous croiserez peut-être des collègues vous invitant à saupoudrer un code Java multi-thread d'instructions du type **`System.out.print("")`** afin de faire disparaître certaines erreurs à l'exécution d'un programme. Quel problème sous-jacent ce type d'astuce curieuse peut-il résoudre ?

Question 2. Du point de vue de la programmation multithread, quelles sont les trois catégories de collections disponibles en Java ? Pourquoi est-il important de les distinguer ?

Question 3. Les verrous de lecture/écriture autorisent les lectures en parallèle des données en assurant que les écritures s'effectuent en exclusion mutuelle. Parfois les performances constatées avec l'emploi de ces verrous sont décevantes. Quel verrou alternatif peut on envisager d'utiliser dans ces conditions ?