Exercice III.1 Le buffer circulaire Le buffer circulaire est une espèce de file *de taille limitée*, implémentée à l'aide d'un tableau. Deux opérations atomiques peuvent être réalisées sur un buffer circulaire :

— déposer un nouvel élement (par la méthode **`déposer(item)`**), à condition que le buffer ne soit pas plein ;
— ou retirer l'élement le plus ancien (par la méthode **`retirer()`**), à condition que le buffer ne soit pas vide.

Ces deux opérations sont bloquantes si la condition requise pour leur exécution n'est pas remplie.

Le buffer circulaire considéré ici est muni de trois variables entières :

— **`disponibles`**, qui indique le nombre d'éléments effectivement déposés dans le buffer ;
— **`premier`**, qui indique la position dans le tableau du plus ancien élément (s'il y en a un) ;
— **`prochain`**, qui indique la position dans le tableau où sera stocké le prochain élément déposé.

Les deux principales configurations possibles d'un buffer circulaire sont décrites sur la figure 7.

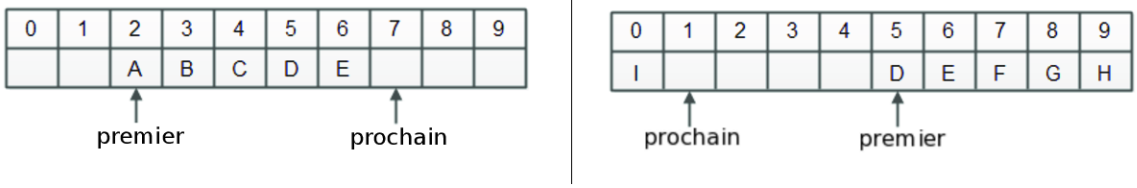


FIGURE 7 – Deux configurations d'un buffer circulaire

Lorsque le buffer est vide ou plein, **`premier`** est égal à **`prochain`** : seule la valeur courante de **`disponibles`** permet donc de distinguer un buffer plein d'un buffer vide. Un exemple typique d'utilisation d'un tel objet, avec deux producteurs et deux consommateurs, est donné sur la figure 8.

Question 1. Complétez sous la forme d'un moniteur le code du buffer donné sur la figure 9 (et disponible dans l'archive `TD_III.zip` sur le site de l'UE).

Question 2. Assurez-vous que chacune des deux opérations atomiques provoque à l'écran l'affichage du nouveau contenu du buffer, du plus ancien au plus récent.

```
public static void main(String[] argv){
  Buffer monBuffer = new Buffer(10);       // Buffer de taille 10
  for(int i=0; i<2; i++) new Producteur(monBuffer).start();
  for(int i=0; i<2; i++) new Consommateur(monBuffer).start();
}
class Consommateur extends Thread {
  Buffer buffer;
  byte donnee;
  public Consommateur(Buffer buffer){ this.buffer = buffer; }
  public void run(){
    while (true) {
      try { donnee = buffer.retirer(); } catch(InterruptedException e){ break; }
    }
  }
}
class Producteur extends Thread {
  Buffer buffer;
  byte donnee = 0;
  public Producteur(Buffer monBuffer){ this.buffer = buffer; }
  public void run(){
    while (true) {
      donnee = (byte) ThreadLocalRandom.current().nextInt(100);
      try { buffer.déposer(donnee); } catch(InterruptedException e){ break; }
    }
  }
}
```

FIGURE 8 – Exemple typique d'utilisation d'un buffer circulaire

```
class Buffer {
  private final int taille;
  private final byte[] buffer;
  private volatile int disponibles = 0;
  private volatile int prochain = 0;
  private volatile int premier = 0;

  Buffer(int taille) {
    this.taille = taille;
    this.buffer = new byte[taille];
  }

  synchronized void déposer(byte b) throws InterruptedException {
    ...
  }

  synchronized byte retirer() throws InterruptedException {
    ...
  }
}
```

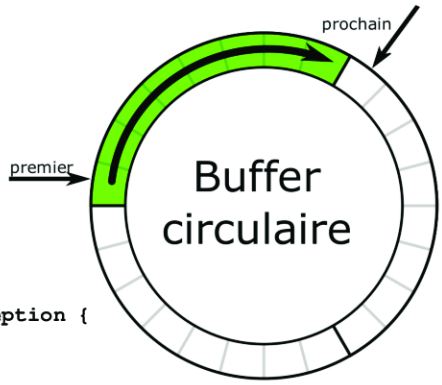


FIGURE 9 – Le buffer circulaire (à compléter)

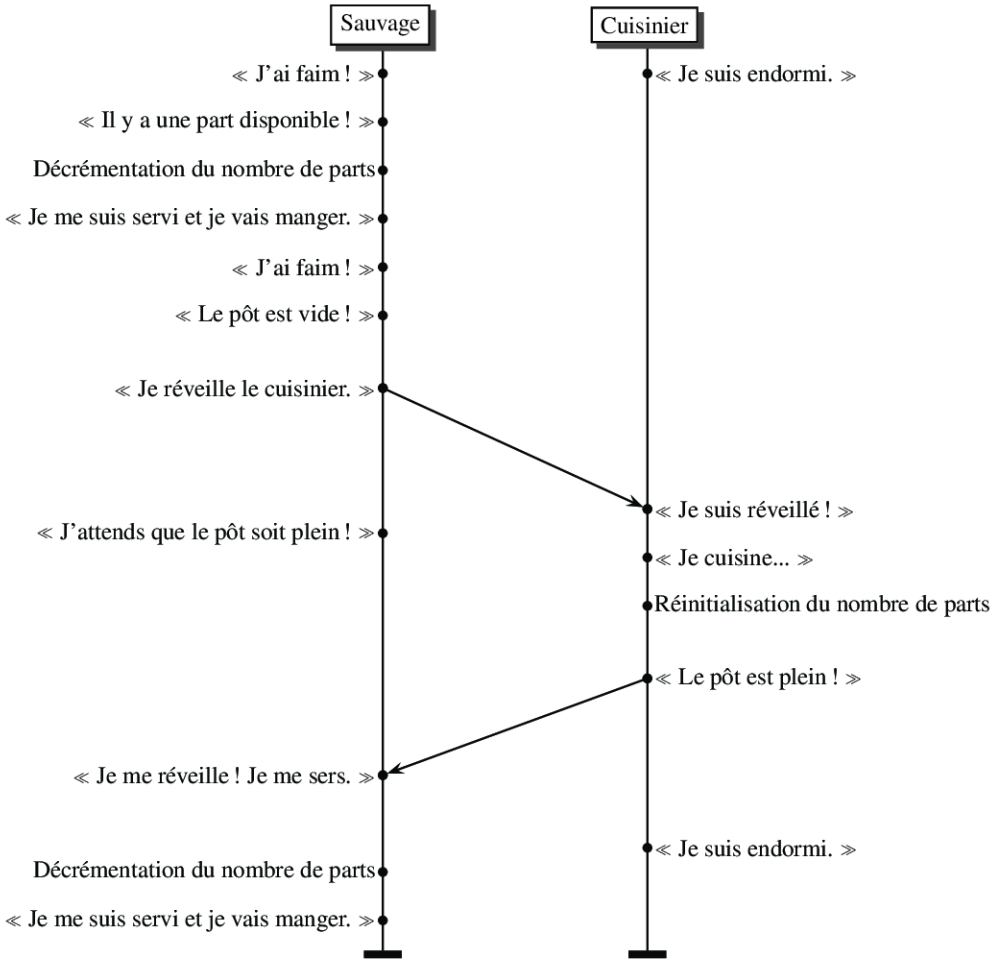
```
public class Diner {
  public static void main(String args[]) {
    int nbSauvages = 100 ;                // La tribu comporte 100 sauvages affamés
    int nbParts = 5 ;                     // Le pôt contient 5 parts, lorsqu'il est rempli
    System.out.println("Il␣y␣a␣" + nbSauvages + "␣sauvages.") ;
    System.out.println("Le␣pôt␣contient␣"+ nbParts + "␣portions.") ;
    Pot pot = new Pot(nbParts) ;
    new Cuisinier(pot).start() ;
    for (int i = 0 ; i < nbSauvages ; i++) { new Sauvage(pot).start(); }
  }
}

class Sauvage extends Thread{
  public Pot pot;
  public Sauvage(Pot pot){ this.pot = pot; }
  public void run(){
    while (true) {
      System.out.println(getName() + ":␣J'ai␣faim!") ;
      pot.seServir() ;
      System.out.println(getName() + ":␣Je␣me␣suis␣servi␣et␣je␣vais␣manger!") ;
    }
  }
}

class Cuisinier extends Thread {
  public Pot pot;
  public Cuisinier(Pot pot){ this.pot = pot; }
  public void run(){
    while (true) {
      System.out.println("Cuisinier:␣Je␣suis␣endormi.");
      pot.remplir() ;
    }
  }
}
class Pot { ... }
```

FIGURE 10 – Code à compléter de l'exercice III.2

Exercice III.2 Le pôt des sauvages Une tribu de sauvages dîne autour d'un large pôt qui peut contenir jusqu'à M portions de missionaire cuit à l'étouffée. Tout comme un philosophe de Dijkstra, un sauvage mange régulièrement ; pour cela, il doit aller se servir dans le pôt.

2012

Lorsqu'un sauvage veut manger, il prend une part, sauf si le pôt est vide. Si le pôt est vide, le sauvage réveille le cuisinier, puis il attend que le pôt soit rempli pour se servir. Ces deux cas de figure sont décrits sur le schéma ci-dessous : chaque sauvage est représenté par un thread, de même que le cuisinier, qui dort lorsqu'il ne cuisine pas (car l'attente active est naturellement proscrite). De même, lorsqu'un sauvage attend que le cuisinier remplisse le pôt, il s'endort et sera réveillé par le cuisinier.



La figure 10 indique une partie du code Java de ce système. L'exercice vise à apporter quelques éléments de réflexion en vue de compléter le code du pôt. Vous devez respecter les messages de sorties écran et l'ordre d'apparition de ces messages indiqués sur le diagramme de séquence ci-dessus. En particulier, *un sauvage doit indiquer à l'écran qu'il réveille le cuisinier* avant *que le cuisinier ne signale qu'il est réveillé.*

Question 1. Quelle instruction doit exécuter le cuisinier afin de s'endormir ? Comment protéger cette instruction contre les *signaux intempestifs* ?

Question 2. Quelle instruction doit exécuter un sauvage pour réveiller le cuisinier ?

Question 3. Quelle instruction devra exécuter un sauvage afin de s'endormir lorsqu'il attend que le pôt soit rempli ? Comment protéger cette instruction contre les *signaux intempestifs* ?

Question 4. Quelle instruction devra exécuter le cuisinier pour réveiller le sauvage après avoir rempli le pôt ?

Question 5. Placer précisément sur le diagramme de séquence ci-dessus les 4 instructions I_1, I_2, I_3 et I_4 déterminées aux questions précédentes.

Question 6. Quelles sont, selon vous, les champs et les méthodes du pôt ?

Question 7. Peut-il y avoir plusieurs sauvages endormis devant le pôt vide ?

Exercice III.3 Étude de philosophes indisciplinés (sans moniteur pour les contrôler) Nous considérons à nouveau le problème classique des philosophes et des spaghettis proposé par Edsger Dijkstra : cinq philosophes se trouvent autour d'une table et chacun a devant lui un plat de spaghetti. Un philosophe n'a que trois états possibles : penser, pendant un temps qui lui est propre ; être affamé ; manger (pendant un temps déterminé). Pour simplifier, nous supposerons aujourd'hui que les cinq fourchettes sont placées au centre de la table et que les philosophes peuvent utiliser n'importe quelle fourchette. Il leur en font cependant deux pour pouvoir manger proprement.

Question 1. Ce système de philosophes est codé en Java à l'aide de 5 threads philosophes par le programme de la figure 11. Notez que ce code compile sans erreur. Quelles remarques préliminaires pourriez-vous faire à propos de ce code ?

Question 2. Expliquez pourquoi trois philosophes risquent très probablement de mourir de faim ! Comment corriger le code en conséquence ?

Question 3. Expliquez comment cinq philosophes peuvent manger simultanément ! Comment peut-on corriger le code ?

Question 4. Expliquez comment trois philosophes peuvent encore manger ensemble simultanément, *même en l'absence de signaux intempestifs* ! Comment doit-on corriger le code ?

Question 5. Expliquez comment deux fourchettes peuvent disparaître, c'est-à-dire qu'il ne reste que 3 fourchettes sur la table alors que tous les philosophes sont en train de penser. Comment doit-on corriger le code ?

```
class PhilosopheIndiscipline extends Thread {
  static volatile int disponibles = 5 ;
  public PhilosopheIndiscipline(String nom) {
    this.setName(nom) ;
  }

  public void prendreDeuxFourchettes() throws InterruptedException {
    if (disponibles < 2) {
      synchronized(this){
        wait() ;
      }
    }
    disponibles = disponibles - 2 ;
  }

  public void lacherDeuxFourchettes() {
    disponibles = disponibles + 2 ;
    synchronized(this){
      notify() ;
    }
  }

  public void run() {
    while (true) {
      try {
        System.out.println("[" + getName() + "] Je pense...") ;
        sleep((int) Math.random()*1000);
        System.out.println("[" + getName() + "] \t Je suis affamé.") ;
        prendreDeuxFourchettes() ;                    // Le philosophe prend deux fourchettes
        System.out.println("[" + getName() + "] \t\t Je mange pendant 3s.") ;
        sleep(3000);
        System.out.println("[" + getName() + "] \t\t\t J'ai bien mangé.") ;
        lacherDeuxFourchettes() ;                     // Le philosophe rend les deux fourchettes
      } catch (InterruptedException e){ break ; }
    }
  }
  public static void main(String[] argv) {
    String noms [] = {"Socrate", "Aristote",  "Epicure", "Descartes", "Nietzsche"};
    for(int i=0; i<5; i++) new PhilosopheIndiscipline(noms[i]).start() ;
  }
}
```

FIGURE 11 – Philosophes indisciplinés