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 :

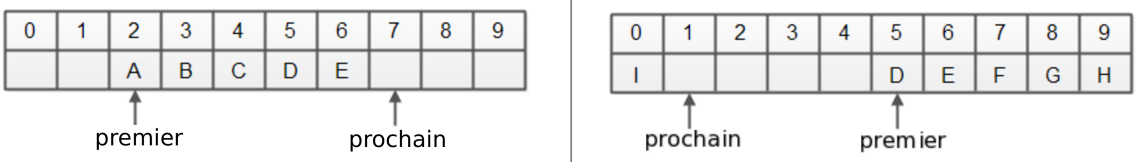
— déposer un nouvel élement (par la méthode **`déposer(item)`**), à condition que le buffer ne soit pas plein ;
— ou retirer l'élement le plus ancien (par la méthode **`retirer()`**), à condition que le buffer ne soit pas vide.

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 de taille 10 dans lequel les lettres de l'alphabet sont déposées une à une sont décrites ci-dessous :



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 11.

Question 1. Complétez les méthodes **`déposer()`** et **`retirer()`** du code donné sur la figure 12.

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

Question 3. Comment modifier le **`main()`** afin qu'il vide le buffer de manière atomique toutes les 5 secondes ?

Exercice III.2 Buffer circulaire concurrent Modifiez le buffer circulaire de l'exercice précédent afin qu'il permette le retrait et l'ajout d'un élement de manière simultanée lorsque qu'il n'est pas plein et qu'un élément est déjà disponible.

```
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 donnée;
  public Consommateur(Buffer buffer){ this.buffer = buffer; }
  public void run(){
    while (true) {
      try { donnée = buffer.retirer(); } catch(InterruptedException e){ break; }
    }
  }
}

class Producteur extends Thread {
  Buffer buffer;
  byte donnée = 0;
  public Producteur(Buffer monBuffer){ this.buffer = buffer; }
  public void run(){
    while (true) {
      donnée = (byte) ThreadLocalRandom.current().nextInt(100);
      try { buffer.déposer(donnée); } catch(InterruptedException e){ break; }
    }
  }
}
```

FIGURE 11 – Exemple typique d'utilisation d'un buffer

```
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 { ... }
  synchronized int charge() { return disponibles; }
}
```

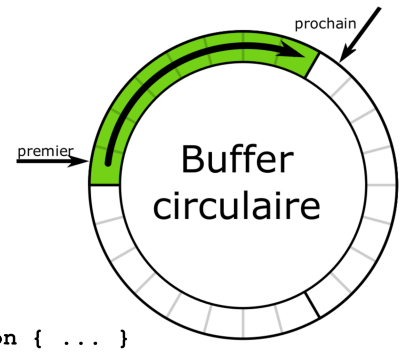


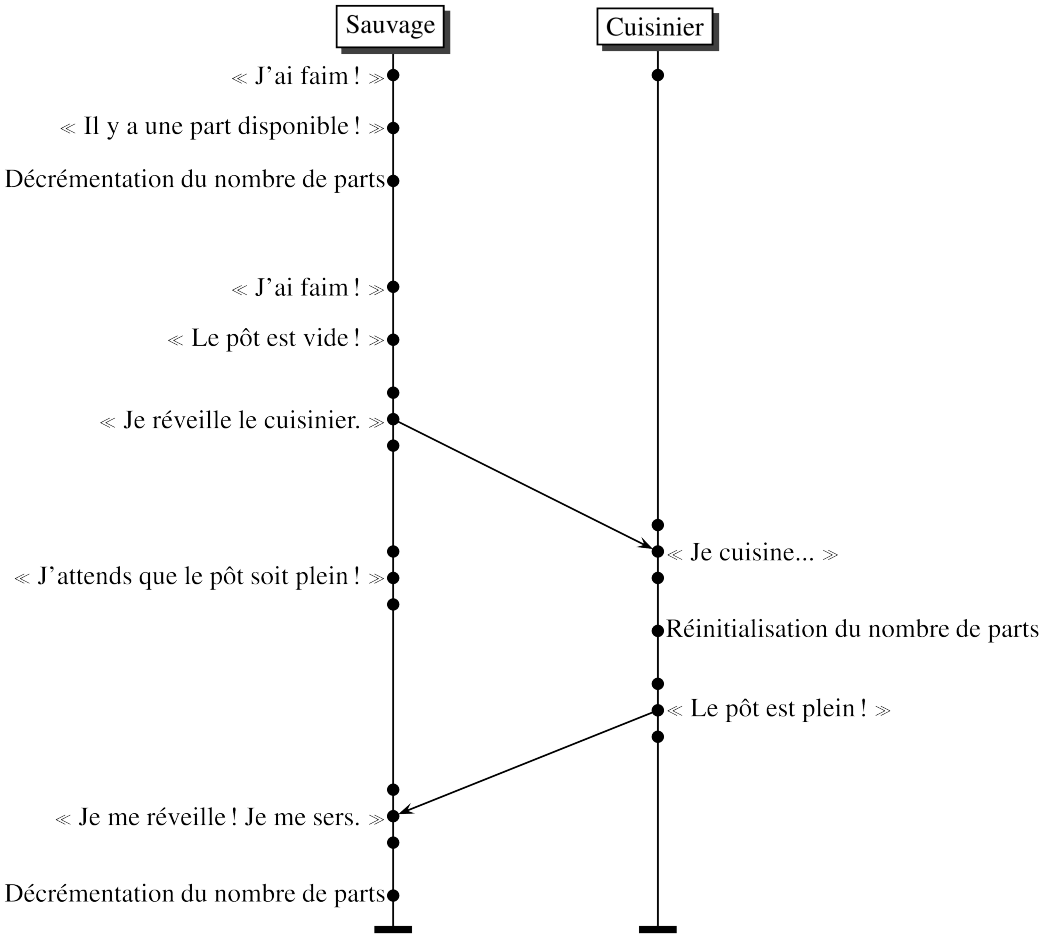
FIGURE 12 – Le buffer circulaire (à compléter)

```
public class Diner {
  public static void main(String args[]) {
    int nbSauvages = 10;                // La tribu comporte 10 sauvages affamés
    int nbPortions = 5;                 // Le pôt contient 5 parts, lorsqu'il est rempli
    System.out.println("Il␣y␣a␣" + nbSauvages + "␣sauvages.");
    System.out.println("Le␣pôt␣peut␣contenir␣"+ nbPortions + "␣portions.");
    Pôt pôt = new Pôt(nbPortions);
    new Cuisinier(pôt).start();
    for (int i = 0; i < nbSauvages; i++) new Sauvage(pôt).start();
  } // CE PROGRAMME N'EST PAS SENSÉ TERMINER !
}

class Sauvage extends Thread {
  public Pôt pôt;
  Random aléa = new Random();
  public Sauvage(Pôt pôt) { this.pôt = pôt ; }
  public void run() {
    while (true) {
      System.out.println(getName() + ":␣J'ai␣faim!");
      try {
        pôt.seServir();
        System.out.println(getName() + ":␣Je␣me␣suis␣servi␣et␣je␣vais␣manger!");
        Thread.sleep(aléa.nextInt(1000));  // Le sauvage mange puis fait la sieste.
      } catch (InterruptedException e) { break; };
    }
  }
}

class Cuisinier extends Thread {
  public Pôt pôt;
  public Cuisinier(Pôt pôt) { this.pôt = pôt ; }
  public void run() {
    while(true){
      try {
        pôt.remplir();
      } catch (InterruptedException e) { break; };
    }
  }
}
class Pôt { ... }
```

FIGURE 13 – Code à compléter de l'exercice III.3

Exercice III.3 Le pôt des sauvages Une tribu de sauvages dîne autour d'un large pôt qui peut contenir jusqu'à M portions de missionnaire cuit à l'étouffée. Tout comme un philosophe de Dijkstra, un sauvage mange régulièrement ; pour cela, il doit aller se servir dans le pôt.

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 correspond à 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 13 présente une partie du code Java de ce système, disponible dans l'archive `TD_III.zip`. Le programme devra au final 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 cuisine.* L'exercice vise à apporter quelques éléments de réflexion avant de compléter le code du pôt.

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

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 ? Où les sauvages attendent-ils que le pôt soit plein ?

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. Vous pourrez distinguer le début et la fin des instructions **`wait()`**.

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

Question 7. Écrire le code de la classe **`Pôt`**. Vous pourrez observer que l'ordre des instructions des méthodes du pôt correspond en fait à l'ordre des évènements placés verticalement sur le diagramme de séquence.

Exercice III.4 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 philosophes ne mangent qu'une fois, que les cinq fourchettes sont placées au centre de la table et que les philosophes peuvent utiliser n'importe quelle fourchette. Il leur en faut 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 14. Notez que ce code compile sans erreur. Quelles remarques préliminaires pourriez-vous faire à propos de ce code lors d'un audit ?

Question 2. Très probablement, une exécution de ce programme permettra d'observer trois philosophes en train de mourir de faim ! Pourquoi ? Comment corriger le code en conséquence ?

Question 3. Cette première correction réalisée, une exécution pourra néanmoins, même en l'absence de signaux intempestifs, permettre d'observer cinq philosophes en train de manger simultanément ! Donnez deux raisons à celà. Comment peut-on corriger le code de la méthode `prendreDeuxFourchettes()` ?

Question 4. Expliquez comment deux fourchettes peuvent disparaître, c'est-à-dire qu'il ne reste que 3 fourchettes disponibles alors que tous les philosophes sont en train de penser. Comment doit-on corriger le code ?

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

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

  public void lâcherDeuxFourchettes() {
    disponibles = disponibles + 2 ;
    synchronized(this){
      notifyAll() ;
    }
  }

  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.") ;
        lâcherDeuxFourchettes() ;              // 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 Philosophe(noms[i]).start() ;
  }
}
```

FIGURE 14 – Philosophes indisciplinés