

Exercice D.1 Problème de vue Le code du programme `Visible.java` ci-dessous est disponible dans le sous-dossier A du dossier `1_-_Visible` de l'archive `TP_D.zip` déposée sur AMETICE.



```
public class Visible {
    public static void main(String[] args) throws Exception {
        A a = new A() ;           // Création d'un objet "a" de la classe A
        a.start() ;               // Lancement du thread "a"
        Thread.sleep(1000) ;
        a.valeur = 1 ;            // Modification de l'attribut valeur de "a"
        a.fin = true ;            // Modification de l'attribut fin de "a"
        System.out.println("Le_main_a_terminé.") ;
    }
}

class A extends Thread {
    public int valeur = 0 ;
    public boolean fin = false ;

    public void run() {
        while(! fin) {} ;        // Boucle d'attente active
        System.out.println(valeur) ;
    }
}
```



Ce programme peut être testé facilement en lançant la commande « `java Visible.java` » dans un terminal Unix ou une console PowerShell sous Windows. Il lance un thread `a` muni de deux attributs, `valeur` et `fin`, et qui attend que `fin` soit vrai pour afficher la valeur de `valeur`.

- Question 1. Vérifiez tout d'abord que ce programme ne termine pas sur votre machine (car le thread `a` n'affiche rien : il reste bloqué dans la boucle d'attente). Sinon, changez de machine et prenez par exemple un ordinateur des salles de TP.
- Question 2. Que faut-il ajouter, en principe et d'après le premier cours, à ce programme pour qu'il termine à coup sûr, c'est-à-dire que le thread `a` affiche la valeur de `valeur` ? Testez cette solution en lançant la version « B » du programme `Visible.java`, disponible dans le sous-dossier B.
- Question 3. Complétez le tableau ci-dessous en notant la valeur affichée par le thread `a` dans chacune des versions C à H du programme `Visible.java`.

Version	Rien	0	1
A	X		
B			X
C			
D			
E			
F			
G			
H			

Question 4. Parmi les affirmations ci-dessous, cochez celles qui vous semblent correctes au vu des observations précédentes :


- ☐ L'observation par le thread `a` de la nouvelle valeur de `fin` induit en général l'observation de la nouvelle valeur de `valeur`, modifiée préalablement.
- ☐ Les valeurs des objets atomiques ont un caractère volatile.
- ☐ Une instruction `System.out.print("")` ; placée à un endroit adéquat du programme peut pallier à l'omission du mot-clef `volatile`.
- ☐ L'emploi d'un verrou peut garantir la visibilité des modifications de données non volatiles.
- ☐ Une instruction `yield()` peut pallier à l'omission du mot-clef `volatile`.
- ☐ Une instruction `sleep()` peut pallier à l'omission du mot-clef `volatile`.
- ☐ Chaque interruption est nécessairement perçue par le thread interrompu.

```

1 class Buffer {
2     private final int taille;
3     private final byte[] buffer;
4     private volatile int disponibles = 0;           // Le buffer est vide
5     private volatile int prochain = 0;             // Position du prochain dépôt
6     private volatile int premier = 0;              // Position du prochain retrait
7     Buffer(int taille) {
8         this.taille = taille;
9         this.buffer = new byte[taille];
10    }
11    synchronized void déposer(byte b) throws InterruptedException {
12        while (disponibles == taille) wait();       // Il n'y a plus de place dans le buffer!
13        buffer[prochain] = b;
14        prochain = (prochain + 1) % taille;         // Nouvelle position du prochain dépôt
15        disponibles++;
16        afficher();
17        notifyAll();
18    }
19    synchronized byte retirer() throws InterruptedException {
20        while (disponibles == 0) wait();             // Il n'y a rien à retirer!
21        byte élément = buffer[premier];
22        premier = (premier + 1) % taille;            // Nouvelle position du prochain retrait
23        disponibles--;
24        afficher();
25        notifyAll();
26        return élément;
27    }
28    synchronized public void afficher() {
29        StringBuffer sb = new StringBuffer();
30        sb.append (" [ ");
31        for (int i=0; i<disponibles; i++) sb.append(buffer[(premier + i) % taille] + " ");
32        sb.append(" ] ");
33        System.out.println(sb.toString());
34    }
35
36    public static void main(String[] argv){
37        Buffer monBuffer = new Buffer(1);            // Buffer de taille 1
38        for(int i=0; i<2; i++) new Producteur(monBuffer).start();
39        for(int i=0; i<2; i++) new Consommateur(monBuffer).start();
40    }
41
42    static class Consommateur extends Thread {
43        Buffer monBuffer;
44        public Consommateur(Buffer buffer){ this.monBuffer = buffer; }
45        public void run(){
46            while (true) {
47                try { Byte donnée = monBuffer.retirer(); } catch(InterruptedException e){ break; }
48            }
49        }
50    }
51
52    static class Producteur extends Thread {
53        Buffer monBuffer;
54        public Producteur(Buffer buffer){ this.monBuffer = buffer; }
55        public void run(){
56            byte donnée = 0;
57            while (true) {
58                donnée = (byte) ThreadLocalRandom.current().nextInt(100);
59                try { monBuffer.déposer(donnée); } catch(InterruptedException e){ break; }
60            }
61        }
62    }
63 }

```

FIGURE 24 – Système de deux producteurs et deux consommateurs sur un buffer de taille 1


Exercice D.2 Le risque de l'emploi de `notify()` à la place de `notifyAll()` Dans cet exercice, nous considérons un buffer circulaire synchronisé de taille 1 servant deux consommateurs et deux producteurs. Le système est décrit sur la figure 24 et disponible dans l'archive `TP_D.zip`. Il s'agit d'expérimenter le risque encouru si l'on remplace les deux instructions `notifyAll()` par `notify()`. 

Question 1. Remplacez chaque `notifyAll()` par `notify()` et testez le système. Observez-vous un blocage ?

Question 2. Le programme modifié peut conduire à une exécution qui s'affichera sur l'écran comme ceci :

```
$ java Buffer.java
[ 34 ]
[ ]
```


bloqué pendant 5 secondes et donc manifestement bloqué. Comment expliquer ce bug ? 


Exercice D.3 Ajout d'une louche au pôt des sauvages Nous poursuivons l'étude du code du pôt des sauvages obtenu à l'exercice III.3, rappelé sur les figures 25 et 26 et disponible dans l'archive `TP_D.zip`. 

Question 1. Observez sur une exécution que plusieurs sauvages peuvent attendre simultanément que le pôt soit plein. Dans quel état sont ces threads ? Observez également que le premier sauvage ayant réveillé le cuisinier n'est pas toujours le premier à se servir⁷.


Question 2. Il s'agit dans cet exercice de garantir que le sauvage qui réveille le cuisinier en premier soit toujours le premier à se servir lorsque le pôt est plein. Pour cela, l'idée proposée est d'ajouter une louche au pôt et d'exiger qu'un sauvage prenne la louche pour se servir et la garde pour lui lorsqu'il réveille le cuisinier. Intuitivement, il faut la louche pour se servir mais aussi pour réveiller le cuisinier ! Décommentez les lignes proposées dans le code fourni afin d'ajouter l'emploi de la louche (vue comme un verrou) dans le comportement des sauvages.

Question 3. Observez que le système peut bloquer, comme si le cuisinier ne parvenait pas à se réveiller malgré la sollicitation d'un sauvage devant le pôt. Cette situation d'interblocage est par exemple illustrée par l'exécution reproduite sur la figure 27.

Question 4. De manière abstraite, un interblocage est caractérisé par un ensemble de processus dans lequel chaque processus attend un événement que seul un autre processus de cet ensemble peut provoquer⁸. Pouvez-vous décrire l'interblocage observé à la question précédente ? Combien de sauvages sont impliqués dans cet interblocage ? Dans quels états sont le cuisinier et le sauvage qui tient la louche ? 

Question 5. Corrigez le programme faisant de l'attribut `louche` un simple booléen et en adaptant le reste du code du pôt afin d'utiliser correctement cette louche (c'est-à-dire en supprimant l'interblocage). Il s'agit ici de s'assurer qu'un sauvage qui arrive devant le pôt vide alors que la louche est déjà prise n'empêche pas le cuisinier de remplir le pôt. 


Question 6. Testez votre programme afin de vérifier que le sauvage qui réveille le cuisinier est désormais bien toujours le premier à se servir une fois le pôt rempli.

Exercice D.4 Implémentation de la corde des babouins sans verrou Nous reprenons à présent l'exercice II.3 des babouins qui se balancent sur une corde reliant les côtés d'un canyon. Pour éviter les chutes, l'utilisation de cette corde doit satisfaire les deux propriétés suivantes : 

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.

Une corde codée sous la forme d'un moniteur est fournie dans le programme `Babouin.java` de l'archive `TP_D.zip`. Il s'agit dans cet exercice de proposer une alternative pour la construction de la corde sous la forme d'un code non-bloquant, c'est-à-dire sans utiliser de verrou. Autrement dit, vous devez dans cet exercice modifier le code de la corde afin d'implémenter une corde alternative qui repose uniquement sur l'emploi d'objets atomiques.

Sur le modèle de l'exercice IV.4, vous pourrez procéder en deux temps : commencer par réduire le moniteur muni d'un seul attribut, en gardant `synchronized`; puis remplacer cet attribut unique par un objet atomique en supprimant le verrou. L'attribut de la corde modifiée pourra être une référence vers une corde immuable ou plus simplement un `integer` codant l'état courant de la corde. 

7. En fait, il n'est même pas certain de pouvoir se servir une fois le pôt rempli s'il y a trop de sauvages autour du pôt vide.

8. Cette définition est plus simple à utiliser que celle proposée par Coffman et al. dans leur article fondateur intitulé « *System Deadlocks* » (Computing Surveys, Vol. 3, 1971, p. 67).

```

public class Diner {
    public static void main(String args[]) {
        int nbSauvages = 10;           // La tribu comporte 10 sauvages affamés
        int nbPortions = 3;            // Le pôt contient 3 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, i).start();
    } // CE PROGRAMME N'EST PAS SENSÉ TERMINER !
}

class Sauvage extends Thread {
    public Pôt pôt;
    Random aléa = new Random();
    public Sauvage(Pôt pôt, int numéro) {
        this.pôt = pôt ;
        this.setName("S" + numéro) ;
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep(aléa.nextInt(5_000));
                System.out.println(getName() + ":_J'ai_faim!");
                pôt.seServir();
            } catch (InterruptedException e) { break; };
        }
    }
}

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

```

FIGURE 25 – Le système des sauvages et du cuisinier autour du pôt

```

1 class Pôt {
2     private volatile int nbPortions;
3     private final int volume;
4     // private ReentrantLock louche = new ReentrantLock(true);
5     // La louche est libre au départ
6
7     public Pôt(int nbPortions) {
8         this.volume = nbPortions;
9         this.nbPortions = nbPortions;
10    }
11
12    synchronized public boolean estVide() {
13        return (nbPortions == 0);
14    }
15
16    synchronized public void remplir() throws InterruptedException {
17        while( ! estVide() ) { // Tant que le pôt n'est pas vide, je ne le remplis pas.
18            wait();
19        }
20        System.out.println("Cuisinier: Je cuisine...");
21        Thread.sleep(2000);
22        nbPortions = volume;
23        System.out.println("Cuisinier: Le pôt est plein!");
24        notifyAll();
25    }
26
27    synchronized public void seServir() throws InterruptedException {
28        // louche.lock();
29        // System.out.println("\t"+Thread.currentThread().getName()
30        //         + ": Je prends la louche.");
31        if ( ! estVide() ) {
32            System.out.println("\t" + Thread.currentThread().getName()
33                               + ": Il y a une part disponible ! ");
34        } else { // Le pôt est vide: on réveille le cuisinier
35            System.out.println("\t" + Thread.currentThread().getName()
36                               + ": Le pôt est vide!");
37            System.out.println("\t\t" + Thread.currentThread().getName()
38                               + ": Je réveille le cuisinier.");
39            notifyAll();
40            System.out.println("\t\t" + Thread.currentThread().getName()
41                               + ": J'attends que le pôt soit plein!");
42            while ( estVide() ) { // Tant que le pôt est vide, je ne me sers pas.
43                wait();
44            }
45            System.out.println("\t\t" + Thread.currentThread().getName()
46                               + ": Je me réveille! Je me sers.");
47        }
48        nbPortions--;
49        // System.out.println("\t" + Thread.currentThread().getName()
50        //         + ": Je pose la louche.");
51        // louche.unlock();
52    }
53 }

```

FIGURE 26 – Pôt des sauvages sous la forme d'un moniteur fourni dans TP_D.zip.

```

$ java Diner
Il y a 10 sauvages.
Le pôt peut contenir 3 portions.
S7: J'ai faim!
S0: J'ai faim!
    S7: Je prends la louche.
    S7: Il y a une part disponible !
    S7: Je pose la louche.
    S0: Je prends la louche.
    S0: Il y a une part disponible !
    S0: Je pose la louche.
S2: J'ai faim!
    S2: Je prends la louche.
    S2: Il y a une part disponible !
    S2: Je pose la louche.
S0: J'ai faim!
    S0: Je prends la louche.
    S0: Le pôt est vide!
        S0: Je réveille le cuisinier.
        S0: J'attends que le pôt soit plein!
Cuisinier: Je cuisine...
S8: J'ai faim!
S1: J'ai faim!
S2: J'ai faim!
S9: J'ai faim!
S3: J'ai faim!
S6: J'ai faim!
S7: J'ai faim!
S4: J'ai faim!
S5: J'ai faim!
Cuisinier: Le pôt est plein!
    S0: Je me réveille! Je me sers.
    S0: Je pose la louche.
    S5: Je prends la louche.
    S5: Il y a une part disponible !
    S5: Je pose la louche.
    S4: Je prends la louche.
    S4: Il y a une part disponible !
    S4: Je pose la louche.
    S7: Je prends la louche.
    S7: Le pôt est vide!
        S7: Je réveille le cuisinier.
        S7: J'attends que le pôt soit plein!
S4: J'ai faim!
S0: J'ai faim!
S5: J'ai faim!
(bloqué)
^C$

```

FIGURE 27 – Illustration de l'interblocage du pôt avec la louche

```

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
            System.out.println("\tLe_babouin_" + getName() + "_commence_à_traverser.");
            sleep(5000); // La traversée ne dure que 5 secondes
            System.out.println("\t\tLe_babouin_" + getName() + "_a_terminé_sa_traversée.");
            corde.lâcher(); // Arrivé de l'autre côté, le babouin lâche la corde
        } catch (InterruptedException e) {e.printStackTrace();}
    }

    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 nbBabouins = 0; // C'est le nombre de babouins sur la corde
    private Côté sens = null; // C'est le côté d'origine des babouins sur la corde

    public synchronized void saisir(){
        Babouin courant = (Babouin) Thread.currentThread() ;
        Côté origine = courant.origine() ;
        while (nbBabouins >= 5 || ( nbBabouins > 0 && sens != origine)) {
            try { wait(); } catch (InterruptedException e) {e.printStackTrace();}
        }
        nbBabouins += 1;
        sens = origine;
    }

    public synchronized void lâcher(){
        nbBabouins -= 1 ;
        notifyAll();
    }
}

```

FIGURE 28 – La classe **Corde** sous la forme d'un moniteur