Java Avance

Gestion de processus

Emmanuel ADAM

Université Polytechnique des Hauts-De-France

UPHF/INSA HdF

Processus

Présentation

- Un processus est un code qui s'exécute "en parallèle" c'est-à-dire dont l'exécution n'est pas bloquante pour le reste du programme
- Exemple d'utilisation :
  - Longs calculs lancés en tâche de fond, permettant de lancer d'autres calculs
  - L'interaction avec l'utilisateurs (les fenêtres sont des processus)
  - Le garbage collector (ramasse miettes) est un processus

Création de processus

classe Thread

- Etendre la classe **Thread** permet de créer un processus
- Le comportement principal du processus est à définir dans la méthode `public void run()`
- La méthode **run()** est constituée généralement d'une boucle longue
  - longs calculs
  - attente de données (de l'interface graphique, du réseau, ...)
- Elle est appelée par la méthode **start()**

Exemple de Thread (1/2)

Le programme principal est le processus principal

- Exemple de création d'une classe de processus qui affiche son nom en boucle.

```
class UneTache extends Thread
{
  int nbRuns=0;
  UneTache(String nom) { super(nom); }
 // remarque : appel au constructeur de Thread
  public void run()
  {
    while (true)
    {
     System.out.println("Pour la "+ (++nbRuns) +"e fois , mon
         nom est : " + getName());
      Thread.yield(); // petite pause
} } }
```

Exemple de Thread (2/2)

Le programme principal est le processus principal

- Partage des ressources machine entre les processus et le processus principal

```
public class MaTache
{
  public static void main(String args[])
  {
    UneTache tache1 = new UneTache("tache 1");
    UneTache tache2 = new UneTache("tache 2");
    tache1.start(); tache2.start();
    int i = 0;
    int fin = 100;
    while (i<fin)
    {
      System.out.println("Ici tache principale , i=" + i);
      i++;
      Thread.yield();
    }
    System.exit(0);
  }
}
```

Exemple de Thread resultat

Résultat de l'exécution

```
Pour la 1e fois, mon nom est : tache 1
Pour la 2e fois, mon nom est : tache 1
Je suis la tache principale, i=0
Pour la 1e fois, mon nom est : tache 2
Pour la 3e fois, mon nom est : tache 1
Pour la 2e fois, mon nom est : tache 2
Je suis la tache principale, i=1
Pour la 4e fois, mon nom est : tache 1
Pour la 3e fois, mon nom est : tache 2
Je suis la tache principale, i=2
....
```

Transformation en processus

interface Runnable

- Si une classe étend une autre classe, elle ne peut étendre Thread (pas d'héritage multiple)
- La solution :
  - implémenter l'interface **Runnable**
  - définir la méthode **run()**
  - pour lancer un élément 'runnable', créer un Thread prenant en paramètre cet élément
  - et démarrer ce processus

Exemple d'utilisation de Runnable (1/2)

Le programme principal est le processus principal

- Exemple de création d'une classe qui compte sans arrêt en implémentant l'interface Runnable.

```
class ObjetComptant implements Runnable
{
  String name;
  ObjetComptant(String nom) { name = nom; }

  public void run()
  {
    int i=0;
    while (true)
    {
     System.out.println("moi " + name + " j'en suis a " + (i
         ++));
     Thread.yield();
} } }
```

Exemple d'utilisation de Runnable (2/2)

Le programme principal est le processus principal

- Créer 2 processus à partir d'objets 'Runable'

```
public class MonRunnable
{
  public static void main(String args[])
  {
    new Thread(new ObjetComptant("c1")).start();
    new Thread(new ObjetComptant("c2")).start();
    int i = 0;
    while (i<100)
    {
      System.out.println("Je suis la tache principale , i=" +
          i);
      i++;
      Thread.yield();
    }
    System.exit(0);
  }
}
```

Exemple d'utilisation de Runnable resultat

Résultat de l'exécution

```
moi c1 j'en suis à 0
moi c2 j'en suis à 0
Je suis la tache principale, i=0
moi c1 j'en suis à 1
moi c2 j'en suis à 1
Je suis la tache principale, i=1
moi c1 j'en suis à 2
moi c2 j'en suis à 2
Je suis la tache principale, i=2
moi c1 j'en suis à 3
moi c2 j'en suis à 3
Je suis la tache principale, i=3
....
```

Création de processus en ligne par notation lambda

Le programme principal est le processus principal

- Création par notation lambda de deux processus, l'un comptant de 0 à 5, l'autre de 100 à 105
- Attente tant qu'un des deux est encore en vie

```
public class MonRunnable {
 public static void main(String args[]) {
 Thread p1 = new Thread(()->{for(int i=0; i++<5; ) {System.
     out.println("p1->"+i);Thread.yield();}});
 Thread p2 = new Thread(()->{for(int i=100; i++<105; ) {
     System.out.println("p2->"+i);Thread.yield();}});
  p1.start(); p2.start();
  while(p1.isAlive() || p2.isAlive()) Thread.yield();
  System.exit(0);
 }
}
```

Création de processus en ligne par notation lambda resultat

Résultat de l'exécution

```
p1->0
p2->100
p1->1
p2->101
p2->102
p2->103
p2->104
p1->2
p1->3
p1->4
Ils ont fini !!!
```

Groupement de processus 1/2

Grouper les processus pour mieux les gérer

- Possibilité d'associer des processus à des groupes (`ThreadGroup`) ; possibilité d'associer des groupes à des groupes
- quelques méthodes :
  - activeCount() : nombre de processus actifs dans le groupe
  - enumerate(Thread[] tab) :place dans tab la liste des processus actifs
  - interrupt() : interrompt tous les processus du groupe
  - join() : appel bloquant, attend que le processus meure
  - join(long delai) : attend au plus delai ms que le processus meure

```
ThreadGroup groupe = new ThreadGroup("mon groupe");
Thread p1 = new Thread(groupe, ()->{for(int i=0; i<5; i++){
    System.out.println("p1->" + i);Thread.yield();}});
Thread p2 = new Thread(groupe, ()->{for(int i=100; i<105; i
    ++){System.out.println("p2->" + i);Thread.yield();}});
p1.start(); p2.start();
while(groupe.activeCount()!=0 ) Thread.yield();
System.err.println("Ils ont fini !!!");
```

Groupement de processus 2/2

Relation processus-groupe

- Création de processus
  - Thread(Runnable cible) : crée un processus sur l'objet cible
  - Thread(Runnable cible, String nom) : crée un processus sur l'objet cible et donne un nom
  - Thread(ThreadGroup groupe, Runnable cible, String nom) : crée un processus sur l'objet cible, lui donne un nom et l'affecte à un groupe
  - Thread(ThreadGroup groupe, Runnable cible) : crée un processus sur l'objet cible et l'affecte à un groupe, le nom est donné automatiquement
- retrouver le groupe, le processus
  - getThreadGroup() : dans la classe `Thread` retourne le groupe auquel appartient le processus
  - Thread.currentThread() : retourne le processus exécutant la methode

Planifier l'exécution de processus 1/3

Tâche planifiée

- Une tâche planifiée est de type **TimerTask**, elle possède les méthodes
  - cancel() : annule la tâche
  - run() : action exécutée par la tâche
  - scheduledExecutionTime() : retourne la prochaine date en ms à laquelle run() va être exécutée

```
//EXEMPLE DE TACHE QUI AFFICHE L'HEURE
import java.time.LocalTime;
import java.util.TimerTask;
import java.time.format.DateTimeFormatter;

class MaTachePlanifiee extends TimerTask
{
  public void run() {
    System.out.println(LocalTime.now().format(
        DateTimeFormatter.ofPattern("hh:mm:ss")) + " -> "
        Execution de ma tache");
  }
```

Planifier l'exécution de processus 2/3

Planification par Timer

- Un **Timer** (`java.util.Timer`) est un objet qui peut temporiser l'exécution de tâche de time `java.util.TimerTask`. La classe Timer propose les méthodes
  - schedule(TimerTask task, long delay) : planifie l'exécution de la tâche après un delai
  - schedule(TimerTask task, long delay, long period) : planifie l'exécution de la tâche après un delai, et cylciquement selon une période
  - schedule(TimerTask task, Date moment) : planifie l'exécution de la tâche à une date donnée
  - ...

Planifier l'exécution de processus 3/3

```
Timer timer = new Timer();
//lancement de la tache dans 1 sec, et la repeter toutes les
     5 sec.
timer.schedule(new MaTachePlanifiee(), 1000, 5000);
// attendre 12 sec.
try { Thread.sleep(12000); } catch (InterruptedException e)
    {}
//annuler la planification
timer.cancel();
System.out.println(LocalTime.now().format(DateTimeFormatter.
    ofPattern("hh:mm:ss")) + " -> FIN !!");
```

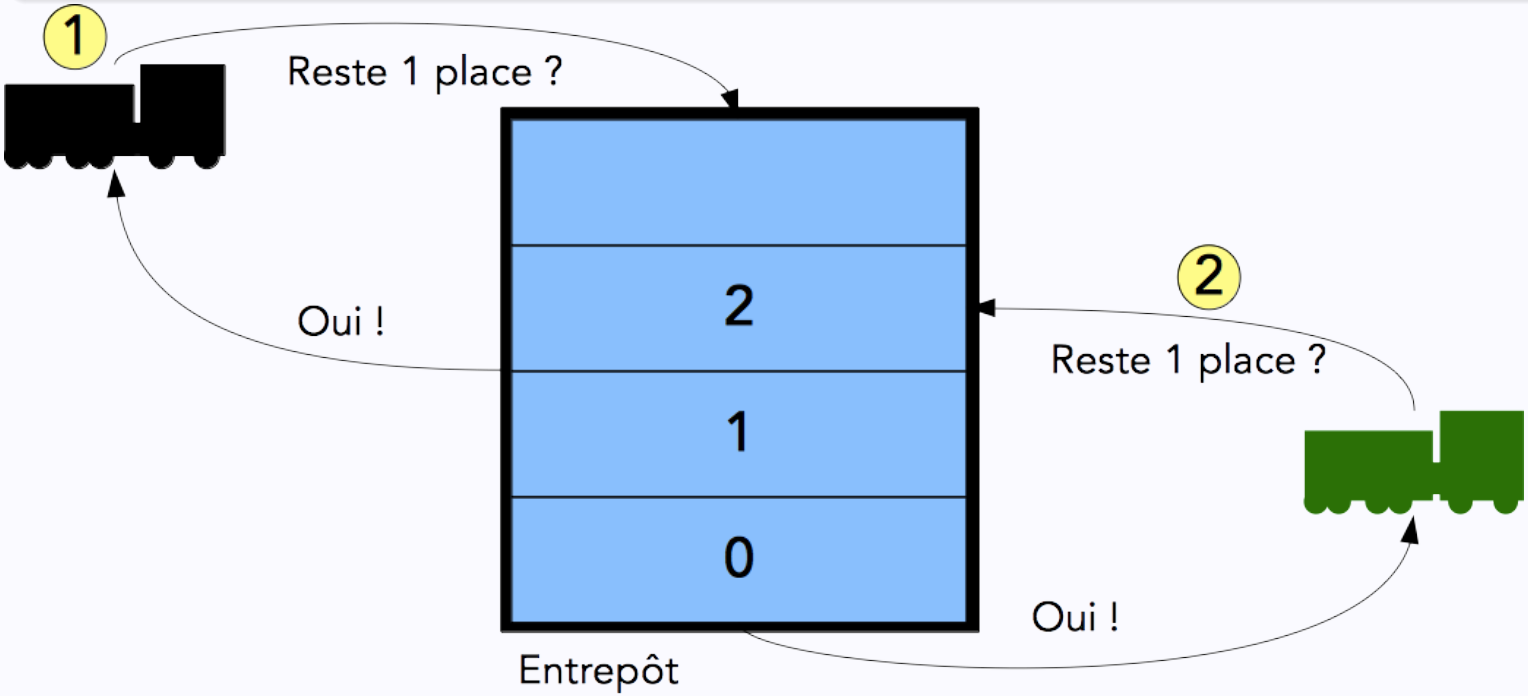
Résultat de l'exécution

```
10:50:21 -> Execution de ma tache
10:50:26 -> Execution de ma tache
10:50:31 -> Execution de ma tache
10:50:32 -> FIN !!
```

Principes de la synchronisation

Problème d'accès mutuel à une même ressource

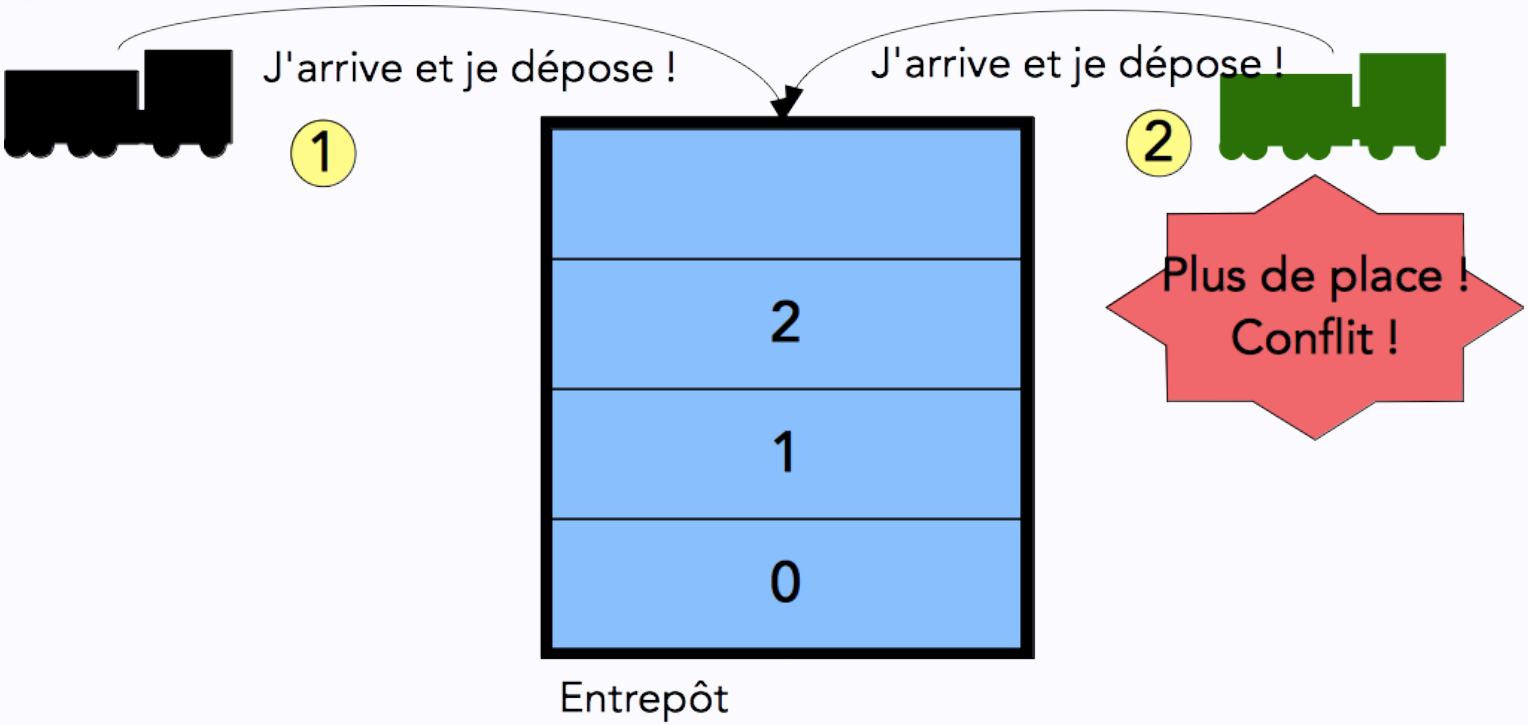
- Supposons deux transporteurs voulant stocker une palette dans un entrepôt
- Un transporteur vérifie la capacité à distance par une appli sur smartphone et perçoit qu'il reste une place
- Un autre transporteur fait de même



Principes de la synchronisation

Problème d'accès mutuel à une même ressource

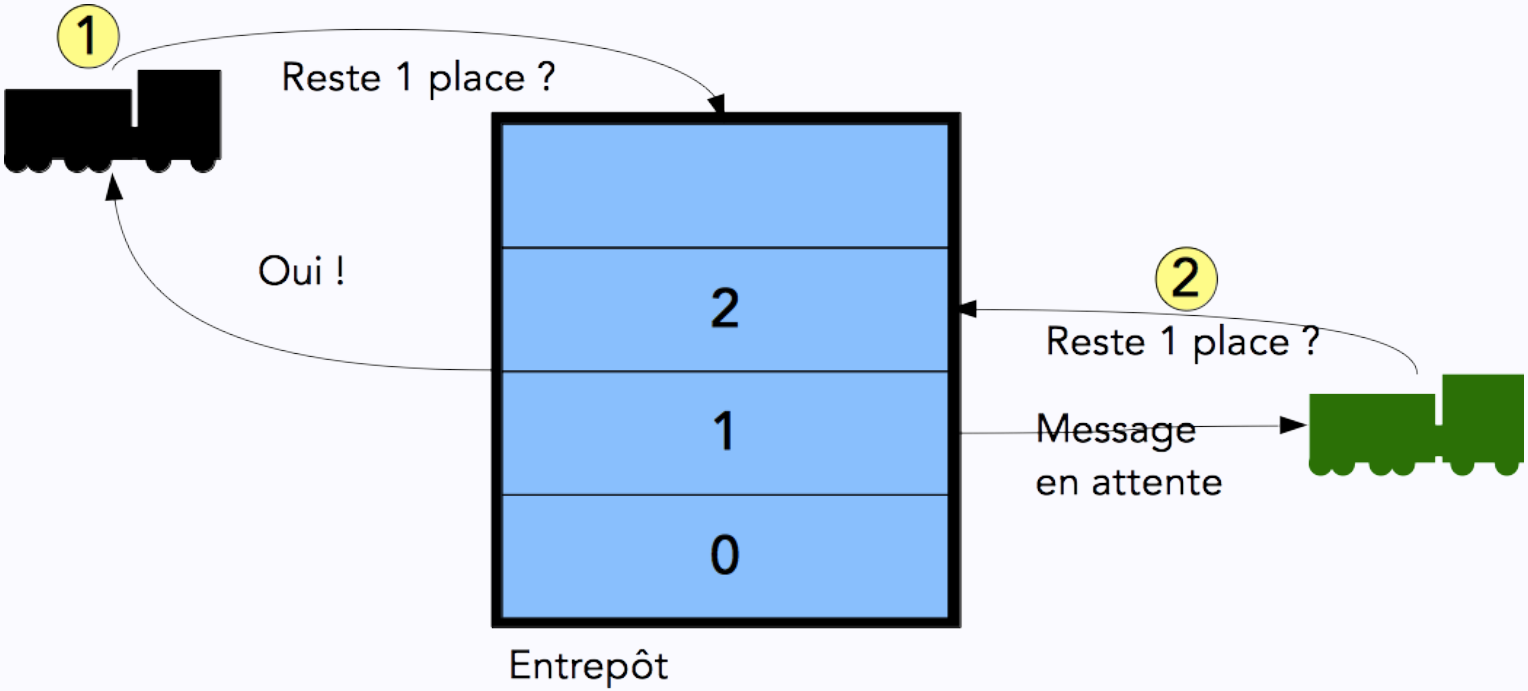
- Les deux transporteurs arrivent pour stocker leur palette
- Un conflit survient !



Principes de la synchronisation

Solution à l'accès mutuel à une même ressource

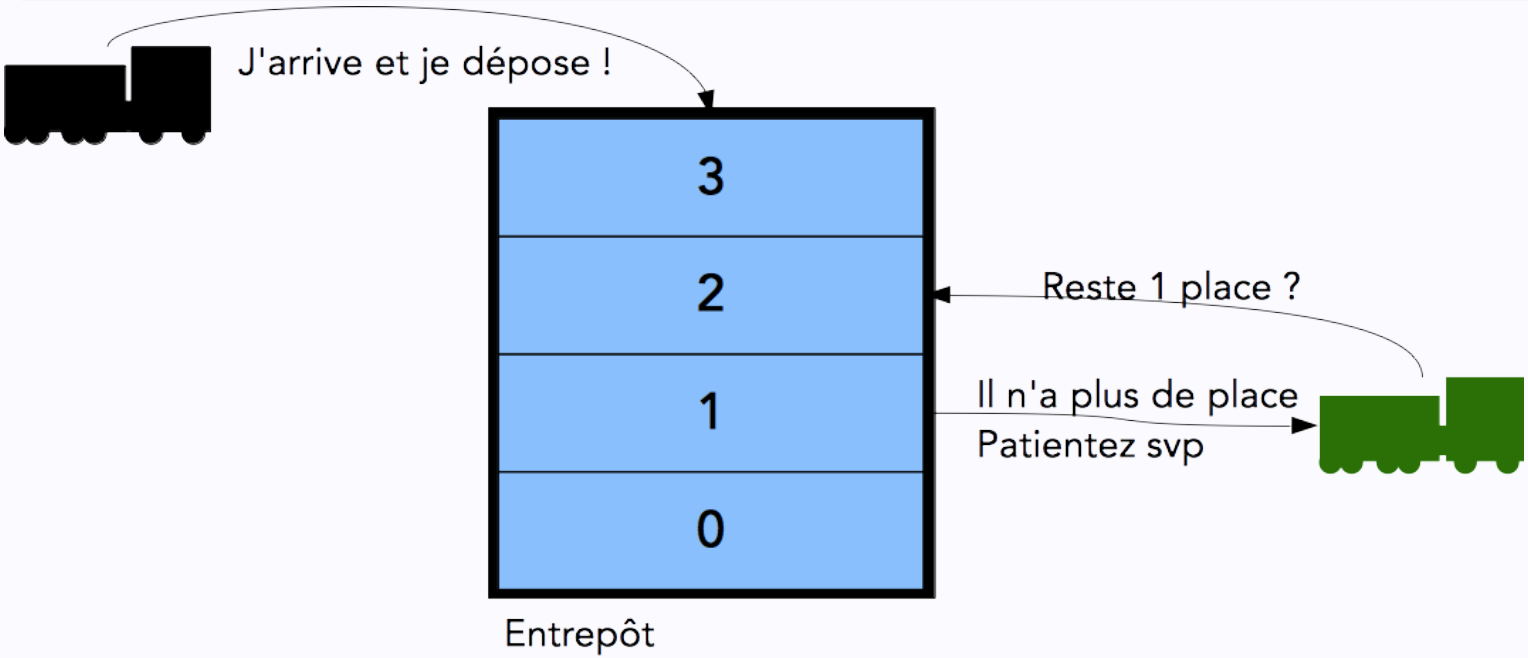
- L'entrepôt doit gérer les conflits
- Dès qu'un transporteur fait une demande, les autres sont mis en attente tant que la demande n'est pas traitée



Principes de la synchronisation

Solution à l'accès mutuel à une même ressource

- Une réponse est ensuite envoyée :
  Soit une autorisation d'accès, soit mise en attente



Synchronisation de méthode

Eviter l'accès mutuel à une méthode

- Pour bloquer l'accès mutuel à une fonction,
  celle-ci doit être précédée du mot clé **synchronized**
- Si un processus appelle une fonction synchronisée dans lequel se trouve déjà un processus, il est bloqué au portes de la fonction.
  Il entrera dans la fonction si :
  - le processus l'utilisant en sort
    et qu'aucun autre processus n'était mis en attente avant lui
  - le processus l'utilisant est mis en attente (wait())
    et qu'aucun autre processus n'était mis en attente avant lui

Synchronisation de méthode : Exemple (1/3)

```
/**classe simple possedant une fonction synchonisee*/
public class ClasseSynchro {
ClasseSynchro(){}
/** fonction synchonisee :
bloque les processus appelants si le parametre est impair
les reveille si un processus arrive avec un parametre pair*/
public synchronized void  testPair(int i)  {
 System.out.println(Thread.currentThread().getName() + ",␣je
     ␣suis␣entre");
 try { Thread.sleep(2000); } catch (Exception e1) { }
 if ((i % 2) != 0)
 {
  System.out.println("le␣nb␣n'est␣pas␣pair,␣je␣mets␣en␣
      attente");
  try { wait(); } catch (InterruptedException e) {  }
 }
 else System.out.println("le␣nb␣est␣pair");

 System.out.println(Thread.currentThread().getName() + "␣
     sort␣de␣la␣fonction");
  }
}
```

Synchronisation de méthode : Exemple (2/3)

```
ClasseSynchro classeSynchro = new ClasseSynchro();
Thread p1 = new Thread(()->classeSynchro.testPair(3), "p1");
Thread p2 = new Thread(()->classeSynchro.testPair(5), "p2");
Thread p3 = new Thread(()->classeSynchro.testPair(4), "p3");
p1.start();
p2.start();
try { Thread.sleep(5000); } catch (InterruptedException e) {
    e.printStackTrace();}
p3.start();
```

Résultat de l'exécution

```
p1, je suis entré
le nb n'est pas pair, je mets en attente
p2, je suis entré
le nb n'est pas pair, je mets en attente
p3, je suis entré
le nb est pair
p3 sort de la fonction
```

Synchronisation de méthode : Exemple (3/3)

Résultat de l'exécution

- p1 et p2 ne **sortent jamais de la méthode run()** !
- ils restent endormis
- $\rightarrow$ modification pour qu'un processus ayant donné un nombre pair réveille tous les processus mis en attente
  ajout de la fonction **notify()** (réveille un processus)
  ou de **notifyAll()** (réveille tous les processus)

Réveil de processus : Exemple (1/2)

```
/**classe simple possedant une fonction synchonisee*/
public class ClasseSynchro {
ClasseSynchro(){}
/** fonction synchonisee :
bloque les processus appelants si le parametre est impair
les reveille si un processus arrive avec un parametre pair*/
public synchronized void  testPair(int i)  {
 System.out.println(Thread.currentThread().getName() + ",␣je
     ␣suis␣entre");
 try { Thread.sleep(2000); } catch (Exception e1) { }
 if ((i % 2) != 0)
 {
  System.out.println("le␣nb␣n'est␣pas␣pair,␣je␣mets␣en␣
      attente");
  try { wait(); } catch (InterruptedException e) {  }
 }
 else {
 System.out.println("le␣nb␣est␣pair");
 notifyAll();

 System.out.println(Thread.currentThread().getName() + "␣
     sort␣de␣la␣fonction");
  }
}
```

Réveil de processus : Exemple (2/2)

Résultat de l'exécution

Cette fois à l'exécution, on obtient :

```
p1, je suis entré
le nb n'est pas pair, je mets en attente
p2, je suis entré
le nb n'est pas pair, je mets en attente
p3, je suis entré
le nb est pair
p3 sort de la fonction
p1 sort de la fonction
p2 sort de la fonction
```

- p1 et p3 **sortent de la méthode run()** !

Mise en veille et Réveil de processus

wait() and notify()

- Placée dans une fonction d'un objet, l'appel à la fonction **wait()** met en veille le processus appelant
- Placée dans une fonction d'un objet, **notify()** réveille un processus mis en veille dans une des procédures de l'objet
- Placée dans une fonction d'un objet, **notifyAll()** réveille tous les processus mis en veille dans une des procédures de l'objet

Exemple type : les producteur et les consommateurs

Producteurs et Consommateurs : Enoncé

- Il existe un Entrepôt de taille limitée (n)
- np producteurs produisent régulièrement des produits qu'ils cherchent à stocker dans l'entrepôt
- nc consommateurs tentent régulièrement de retirer des produits de l'entrepôt
- Les producteurs et consommateurs sont des processus partageant le même entreprôt
- le dépôt d'objets et le retrait d'objet doit donc être synchronisé

Producteurs et consommateurs : définition de l'entrepôt I

```
class Entrepot {
 private int[] stockage;
 private int taille, nbDeposes, nbPris, nbObjetsCourants;

 public Entrepot(int _taille) {
  stockage = new int[_taille];  taille = _taille;
 }

 public synchronized void depose(int obj) {
  // tq que le tableau est plein, mettre en pause
  while (nbObjetsCourants == (taille - 1)) {
  // affiche le nom du processus appelant et mis en pause
   System.out.println(Thread.currentThread().getName() + "--
       pause dans le depot d'objet");
   try { wait(); } catch (InterruptedException e) { }
  }
  stockage[nbObjetsCourants] = obj;
  nbObjetsCourants++;  nbDeposes++;
  //reveiller des autres processus au cas ou
```

Producteurs et consommateurs : définition de l'entrepôt II

```
    notifyAll();
  }

  public synchronized int preleve() {
    // tq que le tableau est vide, mettre en pause
    while (nbObjetsCourants == 0) {
      // affiche le nom du processus appelant et mis en pause
      System.out.println(Thread.currentThread().getName() + "--
          pause dans le retrait d'objet");
      try { wait(); } catch (InterruptedException e) { }
    }

    int obj = stockage[nbObjetsCourants - 1];
    nbObjetsCourants--;  nbPris++;
    //reveiller des autres processus au cas ou
    notifyAll();
    return obj;
  }
}
```

Producteurs et consommateurs : définition du Producteur I

```
class Producteur extends Thread{
 /**lien vers l'entrepot*/
 Entrepot entrepot;

 /**indique si le processus doit s'arreter*/
 private boolean arret = false;

 /**total des objets produits*/
 private static int nbObjets = 0;
 /**no du produit a deposer*/
 private int noObjet = 0;

 public Producteur(String _nom, Entrepot _entrepot)
  { super(_nom); entrepot = _entrepot;
  noObjet = nbObjets++;}
```

Producteurs et consommateurs : définition du Producteur II

```
public void run()
{
//tant que l'arret n'est pas demande
 while (!arret) {
  noObjet = nbObjets++;
  //tente de deposer
  entrepot.depose(noObjet);
  System.out.println(getName() + " : j'ai depose l'objet "+
      noObjet);
  // petite pause de 100ms maxi
  try { Thread.sleep((int)(Math.random()*100));}
      catch (InterruptedException e) {}
 }
}
/**permet de demander l'arret du processus*/
public void halte() { arret = true; }
}
```

Producteurs et consommateurs : définition du Consommateur

```
class Consommateur extends Thread{
 /**lien vers l'entrepot*/
 Entrepot entrepot;
 boolean arret = false;

 public Consommateur(String _nom, Entrepot _ent)
 { super(_nom); entrepot = _ent; }

 public void run() {
 //tant que l'arret n'est pas demande
  while (!arret) {
   int obj = entrepot.preleve();
   System.out.println(getName() + " : j'ai  l'objet " + obj);
   // petite pause de 200ms maxi
   try  { Thread.sleep((int)(Math.random()*200)); }
        catch (InterruptedException e) {}
 } }

 public void halte() { arret = true; }     }
```

Producteurs et consommateurs : Lancer la Simu

```
class ProductConsom {
 public static void main(String args[]) {
 //creer un entrepot limite a 10 places
 Entrepot tab = new Entrepot(10);
 Producteur prod1 = new Producteur("product1", tab);
 Producteur prod2 = new Producteur("product2", tab);
 Consommateur cons1 = new Consommateur("consom 1", tab);
 Consommateur cons2 = new Consommateur("consom 2", tab);
 Consommateur cons3 = new Consommateur("consom 3", tab);

 Producteur[] tabProd = { prod1, prod2};
 Consommateur[] tabCons = {cons1, cons2, cons3};
 for (int i = 0; i<tabProd.length; i++) tabProd[i].start();
 for (int i = 0; i<tabCons.length; i++) tabCons[i].start();
 //faire tourner les processus 4 secondes
 try {Thread.sleep(4000);} catch(InterruptedException e){}

 for (int i = 0; i<tabProd.length; i++) tabProd[i].halte();
 for (int i = 0; i<tabCons.length; i++) tabCons[i].halte();

 System.out.println(tab); }}
```

Volatilité : pour forcer l'écriture en mémoire

Mémoire centrale et mémoire cache

- La JVM de Java est composée d'une mémoire centrale et d'une mémoire cache
- Un changement de valeur d'une variable s'effectue en mémoire cache
- Si un processus souhaite accéder à la variable, il reçoit la valeur de la mémoire centrale
- Certains attributs doivent donc être rafraîchis en permanence
- $\rightarrow$ Utilisation du mot clé **volatile**

Volatilité : pour forcer l'écriture en mémoire

Mot clé volatile

```
int volatile valeur = 0;
```

- Les attributs volatiles :
  - sont chargés de la mémoire centrale avant chaque utilisation.
  - sont stockés en mémoire centrale après chaque accès/écriture.
- A utiliser si une variable est partagée, hors d'un bloc synchronisé

Créer un bloc synchronisé

Possibilité de synchroniser un bloc d'un code

```
...
synchronized (compteur) {compteur.add(); compteur.mult(); }
...
```

- → synchronise l'accès à l'objet compteur.
- Lorsqu'une méthode synchronisée d'un objet est appelée, le verrou est mis, aucune autre méthode synchronisée de cet objet peut être exécutée.
  - tant que le processus n'est pas sorti.
  - sauf s'il a été mis en attente (wait)
- Acquérir le verrou d'un objet est forcément coûteux.
  Il faut donc savoir gérer et diminuer au maximum les sections critiques.

Communication entre processus

Tube de communication

- Si un processus veut envoyer une donnée à un autre processus, il peut appeler une fonction, ou **communiquer par envoi de message**
- Pour cela, il faut créer un **tube de communication**
  - L'émetteur reçoit l'entrée du tube,
  - Le destinataire reçoit la sortie du tube.

```
PipedWriter out = new PipedWriter();
PipedReader in=null;
try { in = new PipedReader(out); }
catch (IOException e) {e.printStackTrace(); }
```

Exemple de communication de chaînes de caractères I

```
PipedWriter out = new PipedWriter();
PipedReader in=null;
try { in = new PipedReader(out); }
catch (IOException e) {e.printStackTrace();  }

final PrintWriter strOut = new PrintWriter(out);
final BufferedReader strIn = new BufferedReader(in);

Thread sender=new Thread( ()-> {
  Scanner sc = new Scanner(System.in);
  String ligne = "";
  while (!ligne.equals("end")) {
    System.out.println("sender->entrez une chaine : ");
    ligne = sc.nextLine();
    try { strOut.println(ligne); } catch(Exception e){}
  }} );
```

Exemple de communication de chaînes de caractères II

```
Thread receiver=new Thread( ()-> {
  String ligne = "";
  while (!ligne.equals("end")) {
    try{ ligne=strIn.readLine();} catch(IOException e){}
    System.out.println("receiver -> recu " + ligne);
  }} );

sender.start();
receiver.start();
```