JAVA - Compléments du cours 9 -Correction

Nicolas Baudru

Année 2008-2009

Exercice 1 (Création de threads)

Pourquoi existe-t-il deux méthodes ? Quels sont les différences entre ces deux méthodes ?

Une classe ne peut étendre qu'une seule autre classe. La première méthode est donc très contraignante. De plus, d'un point de vue conception, il est plus propre de définir des tâches, puis des threads exécutant ces tâches. La deuxième méthode est donc celle à utiliser.

Exercice 2 (Threads et JVM)

Ecrivez en java un programme qui utilise deux threads en parallèle :
- le premier affichera les 26 lettres de l'alphabet ;
- le second affichera les nombres de 1 à 26.

```
class Thread1 implements Runnable{
  public void run(){
    try {for(int i = 1; i<=26;i++) System.out.println(i +"␣");}
    catch (InterruptedException e){ return; }
  }
}

class Thread2 implements Runnable{
  public void run(){
    try {for(int i = 'a'; i<='z'; i++) System.out.println(i +"␣");}
    catch (InterruptedException e){ return; }
  }
}

class MonProgramme {
  public static void main(String[] args) {
    Runnable t1 = new Thread1();
    Runnable t2 = new Thread2();
    new Thread(t1).start();
    new Thread(t2).start();
  }
}
```

Déterminez le résultat de ce programme lors de son exécution ?
Impossible, les exécutions sont entrelacées et on ne peut pas prédire l'entrelacement...

A votre avis, par qui et comment les threads sont gérés dans la machine ?
L'ordonnanceur du SE gère le processus de la JVM. L'ordonnanceur de la JVM gère les threads prêts à s'exécuter (ainsi que le ramasse miettes). Il n'y a aucun moyen pour l'utilisateur d'agir sur l'ordonnanceur de la JVM...

Exercice 3 (Méthodes de la classe Thread)

Information sur les threads - A quoi sert les méthodes suivantes :
- static int activeCount() : renvoie le nombre de threads actuellement exécutés
- static int enumerate(Thread[] tarray) : stocke l'ensemble des Threads du même groupe dans le tableau et renvoie le nombre de threads.
- static Thread currentThread() : renvoie le Thread correspondant au thread en train d'être exécuté.

Ordonnancement sur les threads - A quoi sert les méthodes suivantes :
- void setPriority(int) : fixe la priorité du thread
- int getPriority() : renvoie la priorité du thread
- static void yield() : provoque une "pause" du thread en cours et un réordonnancement

Attention l'ordonnanceur de la JVM peut ne pas tenir compte des priorités.

Manipulation sur les threads - Quelles méthodes provoquent :
- la mise en attente : void sleep(long millis) et void sleep(long millis, int nanos)
- l'attente de l'arrêt d'un thread donné : void join() et void join(long millis) avec délai ou void join(long millis, int nanos) avec délai
- l'interruption d'un thread : void interrupt() ou un thread "interromp" (envoie une InterruptedException) un autre.

Ecrivez un programme dont le thread principal lance et nomme trois nouveaux threads. Chaque thread ainsi créé doit effectuer 10 fois les actions suivantes :
- attendre un temps aléatoire compris entre 0 et 200 ms,
- puis afficher son nom

Le thread principal devra attendre la fin de l'exécution des trois threads qu'il a créés avant de terminer son exécution.

```
class MonThread extends Thread {
  public void run() {
    for (int i = 0; i < 10; i++) {
      // Le thread en cours d'execution attend entre 0 et 200 ms
      try {
        Thread.currentThread().sleep((int) Math.random()*200);}
      catch (InterruptedException e) {return ;}
      // Puis ecrit son nom
      System.out.println("Je␣suis␣" + Thread.currentThread().getName()
      + "␣et␣je␣m'execute");
    }
  }
}
```

```
class Tache implements Runnable {

  public void run() {
    for (int i = 0; i < 10; i++) {
      // Le thread en cours d'execution attend entre 0 et 200 ms
      try {
        Thread.currentThread().sleep((int) Math.random()*200);}
      catch (InterruptedException e) {return ;}
      // Puis ecrit son nom
      System.out.println("Je␣suis␣" + Thread.currentThread().getName()
      + "␣et␣je␣m'execute");
    }
  }
}

class TestThread {

  public static void main (String[] args) {
    // Declare trois threads
    Thread t1;
    Thread t2;
    Thread t3;

    // ====== METHODE 1 : en heritant de la class Thread ========

    System.out.println("--------␣Avec␣la␣methode␣1␣-------");
    // Cree trois thread suivant le modele defini dans MonThread.
    // Leur comportement est celui du run de la class MonThread.
    t1 = new MonThread();
    t2 = new MonThread();
    t3 = new MonThread();
    // Donne un nom a chaque thread
    t1.setName("Tom");
    t2.setName("Pierre");
    t3.setName("Lucie");
    // Lance les threads
    t1.start();
    t2.start();
    t3.start();
    try {
      // le thread main attend la fin d'execution des threads t1 et t2
      t1.join();
      t2.join();
      t3.join();
    }
    catch (InterruptedException e) {return ;};
    System.out.println();
    System.out.println();
```

```
      // ===== METHODE 2 : en implementant l'interface Runnable =====

      System.out.println("--------␣Avec␣la␣methode␣2␣-------");
      // cree une tache a executer pour les threads
      Tache job = new Tache();
      //Crée trois thread avec le comportement defini par l'objet job
      // Leur comportement est donc celui du run de la class Tache.
      t1 = new Thread(job);
      t2 = new Thread(job);
      t3 = new Thread(job);
      // Donne un nom a chaque thread
      t1.setName("Jerry");
      t2.setName("Luc");
      t3.setName("Namie");
      // Lance les threads
      t1.start();
      t2.start();
      t3.start();
      try {
        // le thread main attend la fin d'execution des threads t1 et t2
        t1.join();
        t2.join();
        t3.join();
      }
      catch (InterruptedException e) {return ;};
    }
}
```

Exercice 4 (Threads et concurrence)

L'exercice suivant montre ce qui peut se passer quand deux threads (ici Sylvie et Bruno) partagent un même objet (ici un compte en banque commun). Pour cela vous allez écrire un programme qui comprendra deux classes, Compte et JobSylvieEtBruno.

La classe Compte est très simple. Elle comprend :

- un attribut privé solde initialisé à 100 représentant le solde courant du compte
- une méthode retirer(int montant) permettant de retirer un certain montant du compte. La voici en détaille :

```
class Compte {
  private int solde = 100;

  public int getSolde() {
    return solde;
  }
  public void retirer(int montant){
    solde = solde - montant;
  }
}// fin class Compte
```

La classe JobSylvieEtBruno implémente Runnable et représente le comportement que Sylvie et Bruno ont tous les deux. Leur comportement est assez particulier puisque Sylvie et Bruno s'endorment très souvent, et en particulier pendant qu'ils effectuent un retrait . Cette classe contient :

- un attribut compte de type Compte représentant le compte en banque de Sylvie et Bruno.
- une méthode effectuerRetrait(int montant) permettant à Sylvie ou Bruno d'effectuer un retrait sur leur compte en banque. Le comportement de cette méthode est le suivant : la personne voulant effectuer le retrait vérifie le solde, puis s'endort 500 ms, puis à son réveil (au bout des 500 ms) effectue le retrait. Le nom de la personne effectuant le retrait doit être signalé.
- une méthode run() décrivant le comportement de Sylvie et Bruno. Il s'agit d'effectuer en boucle (par exemple 10) des retraits de 10 euros.
- enfin la méthode main(String[] args) crée deux threads (Sylvie et Bruno) avec le même Runnable (ici de type JobSylvieEtBruno), les nomme Sylvie et Bruno, puis les lance.

Après avoir écrit la classe JobSylvieEtBruno, examinez le comportement de ce programme.

```
public class JobSylvieEtBruno implements Runnable {

  private Compte compte = new Compte();

  public static void main(String[] args){
    JobSylvieEtBruno tache = new JobSylvieEtBruno();
    Thread t1 = new Thread(tache);
    Thread t2 = new Thread(tache);
    t1.setName("Bruno");
    t2.setName("Sylvie");
    t1.start();
    t2.start();
  } // fin methode main

  public void run(){
    for (int x = 0; x < 10; x++) {
      effectuerRetrait(10);
      if (compte.getSolde() < 0) {
        System.out.println("decouvert !");
      }
    }
  } // fin methode run

  private void effectuerRetrait(int montant){
    String nom = Thread.currentThread().getName();
    if(compte.getSolde() >= montant) {
      System.out.println(nom + "va retirer");
      try {
        System.out.println(nom + "va dormir");
        Thread.sleep(500);
      } catch (InterruptedException e) {}
      System.out.println(nom + "se reiveille");
      compte.retirer(montant);
      System.out.println(nom + "retrait fait");
```

```
      }
      else {
        System.out.println(nom + "pas assez");
      }
    }// fin methode effectuerRetrait

}// fin class JobSylvieEtBruno
```

Exercice 5 (Threads et attributs partagés)

Ecrivez un (ou des) petit(s) programme(s) mettant en évidence si deux threads d'une classe A implémentant Runnable partagent les attributs static, public, private,... de cette classe A.

Rappels

Le mot clé "synchronized" et l'exclusion mutuelle

Chaque objet Java possède un verrou dont la clé est gérée par la JVM. Ce verrou, qui est en général ouvert, entre en scène lorsque l'une des méthodes de l'objet est synchronisée (mot clé synchronized). En effet, lorsqu'un thread souhaite accéder à une méthode synchronisée d'un objet, il demande la clé de cet objet à la JVM, entre dans la méthode, puis ferme le verrou à clé. De cette façon, aucun autre thread ne peut accéder aux méthodes synchronisées de cet objet. Lorsque le thread sort de la méthode synchronisée, il ouvre de nouveau le verrou et rend la clé à la JVM. Un autre thread peut alors accéder aux méthodes synchronisées de l'objet. Le verrouillage des méthodes synchronisées permet indirectement de protéger des données d'un objet : il suffit de synchroniser toutes les méthodes manipulant ces données.

Remarques :

- en Java on ne protège pas directement les données d'un objet, mais plutôt la manière d'accéder à ces données.
- il n'y a pas 1 verrou par méthode synchronisée d'un objet, mais bien 1 verrou pour toutes les méthodes synchronisées de cet objet.
- la gestion des clés et des verrous est réalisée par la JVM.

Un premier exemple simple d'utilisation :

```
class Solde {
  private int solde = 0;

  public Solde() { }
  public synchronized void incrementer() {
    int n = solde;
    try { Thread.currentThread.sleep(200);}
    catch (InterruptedException e) {}
    solde = n + 1;
  }
}

class MaTache implements Runnable {
```

```
  private Solde sld;

  public MaTache(Solde s) { sld = s;  }
  public void run() {  sld.incrementer(); }

  public static void main (String[] args) {
    Solde s = new Solde(s);
    Thread t1 = new Thread(new MaTache(s));
    Thread t2 = new Thread(new MaTache(s));
    t1.setName("t1");
    t2.setName("t2");
    t1.start();
    t2.start();
  }
}
```

Les méthodes de synchronisation inter-processus

- void unObjet.wait() : Le thread appelant cette méthode rend la clé du vérrou et attend l'arrivée d'un signal sur l'objet unObjet. Cette méthode doit être utilisée à l'intérieur d'une méthode ou d'un bloc synchronized.
- void unObjet.notify() : indique à un thread en attente d'un signal sur unObjet de l'arrivée de celui-ci. A utiliser dans une méthode ou un bloc synchronized.
- void unObjet.notifyAll() : indique à tous les threads en attente d'un signal sur unObjet de l'arrivée de celui-ci. A utiliser dans une méthode ou un bloc synchronized.

Remarque : Si plusieurs threads exécutent unObjet.wait(), chaque unObjet.notify() débloquera un thread bloqué, dans un ordre indéterminé.

Exemple d'utilisation : On peut grâce à ces méthodes programmer une classe Semaphore :

```
public class Semaphore {
  int n;
  String name;

  public Semaphore (int max, String S) {
    n = max;
    name = S;
  }

  public synchronized void P() {
    if (n == 0) {
      try { wait(); }
      catch (InterruptedException e) {}
    }
    n--;
  }

  public synchronized vois V() {
```

```
      n++;
      notify();
    }
  }
```

Exercice 6 (Le producteur-consommateur)

Pour une introduction au problème du producteur et du consommateur, vous pouvez consulter le lien suivant, qui pointe vers un aperçu sur google book du livre « Programmation concurrente » de André Schiper.

`http://books.google.fr/books?id=Pst_B3DugJkC&pg=PP1`

Le chapitre 5 parle de la coopération entre processus (ou threads) et la partie 5.5 traite le problème du producteur-consommateur en particulier.

Le programme suivant est une ébauche d'une implémentation du producteur-consommateur avec buffer circulaire. Complétez ce programme.

Réponse :

```
// implementation du producteur consommateur avec un buffer circulaire
class BufferCirc {

  private Object[] tampon;
  private int taille;
  private int prem, der, nbObj;

  public BufferCirc (int t) {
    taille = t;
    tampon = new Object[taille];
    prem = 0;
    der = 0;
    nbObj = 0;
  }

  public synchronized void depose(Object o) {
    while (nbObj == taille) {
      try { wait();}
      catch (InterruptedException e) {}
    }
    tampon[der] = o;
    der = (der + 1) % taille;
    nbObj = nbObj + 1;
    notify();
  }

  public synchronized Object preleve() {
    while (nbObj == 0) {
      try { wait();}
```

```
      catch (InterruptedException e) {}
    }
    Object o = tampon[prem];
    tampon[prem] = null;
    prem = (prem + 1) % taille;
    nbObj = nbObj - 1;
    notify();
    return (o);
  }
} // fin class BufferCirc


class Producteur implements Runnable {

  private BufferCirc buffer;
  private int val;

  public Producteur(BufferCirc b) {
    buffer = b;
  }

  public void run() {
    while (true) {
      buffer.depose(new Integer(val));
      System.out.println (Thread.currentThread().getName() +
      "␣a␣depose␣" + val);
      val++;
      try {
        Thread.sleep((int)(Math.random()*100));
      }
      catch (InterruptedException e) {}
    }
  }
} // fin classe Producteur


class Consommateur implements Runnable {

  private BufferCirc buffer;

  public Consommateur(BufferCirc b) {
    buffer = b;
  }

  public void run() {
    Integer val;
    while (true) {
      val = (Integer) buffer.preleve();
      System.out.println (Thread.currentThread().getName() +
      "␣a␣preleve␣" + val);
```

```
        try {
          Thread.sleep((int)(Math.random()*300));
        }
        catch (InterruptedException e) {}
      }
    }
  } // fin classe Consommateur


  class Principal {
    public static void main (String[] args) {
      BufferCirc b = new BufferCirc(20);
      Producteur p = new Producteur(b);
      Consommateur c = new Consommateur(b);
      Thread P1 = new Thread(p);
      P1.setName("P1");
      Thread C1 = new Thread(c);
      C1.setName("C1");
      P1.start();
      C1.start();
    }
  }
```

Exercice 7 (Attention aux "verrous mortels")

Le programme suivant illustre l'un des dangers inhérent à l'utilisation de méthodes synchronisées : les interblocages. Il met en jeu deux processus t1 et t2 tentant d'accéder de manière concurrente à deux objets o1 et o2. Les deux processus t1 et t2 sont issus de la classe MonThread. Les deux objets o1 et o2 sont deux instances de la classe MonObjet qui contient deux méthodes synchronisées action1 et action2.

Malheureusement, lors d'une erreur de manipulation, les instructions des classes MonThread et MonObjet ont été mélangées. Seule la classe Deadlock contenant la méthode main() est restée intégre. Seriez-vous capable de reconstituer le programme ?

```
1 - try{Thread.currentThread().sleep(200);}
2 - public synchronized void action1 (MonObjet o) {
3 - o.action1(this);  } }
4 - obj2 = o2; }
5 - catch (InterruptedException ex) { return ; }
6 - o.action2(this); }
7 - try{Thread.currentThread().sleep(200);}
8 - public synchronized void action2 (MonObjet o) {
9 - public MonThread(MonObjet o1, MonObjet o2) {
10 - private MonObjet obj1, obj2;
11 - obj1.action1(obj2); } }
12 - catch (InterruptedException ex) { return ; }
13 - class MonObjet {
14 - class MonThread extends Thread {
15 - obj1 = o1;
16 - public MonObjet() {}
```

```
17 - public void run() {


class Deadlock {

        public static void main (String[] args) {
                MonObjet o1 = new MonObjet();
                MonObjet o2 = new MonObjet();
                MonThread t1 = new MonThread(o1,o2);
                t1.setName("t1");
                MonThread t2 = new MonThread(o2,o1);
                t2.setName("t2");
                t1.start();
                t2.start();
        }
}
```

Réponse : petit programme générant un interblocage des threads

```
  13 - class MonObjet {

    16 - public MonObjet() {}

    2 - public synchronized void action1 (MonObjet o) {
      1 - try{Thread.currentThread().sleep(200);}
      5 - catch (InterruptedException ex) { return ; }
      6 - o.action2(this);     }

    8 - public synchronized void action2 (MonObjet o) {
      7 - try{Thread.currentThread().sleep(200);}
      12 - catch (InterruptedException ex) { return ; }
      3 - o.action1(this);   }  }


  14 - class MonThread extends Thread {
    10 -   private MonObjet obj1, obj2;

    9 -    public MonThread(MonObjet o1, MonObjet o2) {
      15 - obj1 = o1;
      4 - obj2 = o2; }

    17 -  public void run() {
      11 - obj1.action1(obj2); }  }
```

Exercice 8 (Synchronisation de méthode static)

A votre avis, peut-on synchroniser une méthode static ? Si oui, qu'est-ce que cela signifie ?

Un programme presentant ce qui arrive si on synchronise une methode static.

```
class MonObjet {
  static private int i;
  private String name;

  public MonObjet(String n, int x) {
    i = x;
    name = n;
  }

  public void action(int x) {
    System.out.println(Thread.currentThread().getName()
          + "␣au␣porte␣de␣" + name);
    ajoute(x, name);
    System.out.println(Thread.currentThread().getName()
          + "␣est␣sorti␣de␣" + name);
  }

  public static synchronized void ajoute(int x, String n) {
    System.out.println(Thread.currentThread().getName()
          + "␣entre␣dans␣" + n);
    try {
      System.out.println(Thread.currentThread().getName()
            + "␣attend␣");
      Thread.currentThread().sleep(200);
    }
    catch (InterruptedException ex) {  return ; }
    System.out.println(Thread.currentThread().getName()
          + "␣ajoute␣dans␣" + n);
    System.out.println("i␣=␣" + i);
    i = i + x;
    System.out.println("␣:␣i␣=␣" + i);
    System.out.println(Thread.currentThread().getName()
          + "␣sort␣de␣" + n);
  }
} // fin classe MonObjet


class MonThread extends Thread {

  private MonObjet Obj;
  private int a;

  public MonThread (MonObjet o, int x) {
    Obj = o;
    a = x;
  }

  public void run() {
    Obj.action(a);
  }
```

```
} // fin classe MonThread


class Principale {

  public static void main (String[] args) {
    MonObjet o1 = new MonObjet("o1", 1);
    MonObjet o2 = new MonObjet("o2", 2);
    MonThread t1 = new MonThread (o1, 3);
    MonThread t2 = new MonThread (o2, 5);
    t1.setName("t1");
    t2.setName("t2");
    t1.start();
    t2.start();
  }
} // fin classe Principale
```