

Java Avancé - Cours 4

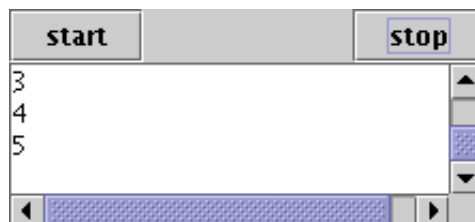
Plan

1 Motivation	1
1.1 Le besoin d'instances d'exécutions	1
1.2 La classe <code>java.lang.Thread</code>	2
1.3 L'interface graphique	2
1.4 Le compteur	3
1.5 Cycle de vie des threads	4
2 Atomicité	4
2.1 Partage d'information	4
2.2 Application bancaire	4
2.3 Explication	5
2.4 Solution : <code>synchronized</code>	5
3 Communication inter threads	6
3.1 Exemple producteur consommateur	6
3.2 L'objet partagé	6
3.3 <code>wait</code> et <code>notifyAll</code>	7
3.4 Priorités	7
3.5 Groupes de threads	8
4 Autres structures de données	8
4.1 <code>BlockingQueue</code>	8
4.2 <code>Semaphore</code>	8
5 Spécificités J2SE 5.0	9
5.1 Types génériques	9
5.2 Autoboxing et Auto-Unboxing	11
5.3 Boucles rapides	11
5.4 Types énumérés	11
5.5 Import statique	12
5.6 Nombre d'arguments variables	12
6 Exercices	12
Le but de ce cours est de détaillé le mécanisme de threads (instance d'exécution) en Java.	

1 Motivation

1.1 Le besoin d'instances d'exécutions

On considère l'application suivante :



Un compteur défile toutes les secondes et peut être arrêté par un bouton stop. Le bouton start permet de redémarrer le compteur à zéro.

Une manière facile de réaliser ce programme est d'utiliser les threads (ou instances d'exécutions) :

- un thread sera en charge de faire les calculs toutes les secondes,
- un autre thread sera en charge de prendre en compte les requêtes utilisateur.

Le code de ce programme est disponible à l'url :

http://www.derepas.com/java/cours4_exemple_1.jar

1.2 La classe `java.lang.Thread`

Pour faire de nouvelles instances d'exécutions il suffit de créer une classe qui hérite de `java.lang.Thread` :

```
class MonThread extends Thread {
    MonThread() {
        //...
    }
    public void run() {
        //...
    }
}
```

Pour démarrer une nouvelle instance d'exécution il suffit alors d'avoir la séquence:

```
MonThread nouveauThread = new MonThread();
nouveauThread.start();
```

L'appel de la méthode `start` sur un `Thread`, permet de déclencher l'exécution en concurrence par l'ordonnanceur de la machine virtuelle Java, de la méthode `run`.

Une autre façon de créer des threads est d'implémenter l'interface `Runnable`, on utilise alors le constructeur `Thread` (`Runnable target`).

1.3 L'interface graphique

On réalise une interface graphique dans la classe `ex1.CounterPanel` de manière complètement standard :

```
package ex1;

import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;

public class CounterPanel
    extends JPanel
    implements ActionListener
{
    Counter counter = null; // Counter est détaillé plus loin
    JTextArea text;        // endroit ou écrire l'avancée du compteur
}
```

Dans le constructeur on initialise les composants graphiques :

```
public CounterPanel() {
    super(new BorderLayout());
    JButton startButton = new JButton("start");
    JButton stopButton = new JButton("stop");
    text = new JTextArea(5,20);
    JScrollPane scrollPane = new JScrollPane(text);
    startButton.setActionCommand("start");
}
```

```

        stopButton.setActionCommand("stop");
        startButton.addActionListener(this);
        stopButton.addActionListener(this);
        add(startButton, BorderLayout.WEST);
        add(stopButton, BorderLayout.EAST);
        add(scrollPane, BorderLayout.SOUTH);
    }

```

La méthode `actionPerformed` va elle prendre en charge les actions à effectuer sur le compteur :

```

public void actionPerformed(java.awt.event.ActionEvent e) {
    if ("start".equals(e.getActionCommand())) {
        counter = new Counter(text);
        counter.start();
    } else if ("stop".equals(e.getActionCommand())) {
        counter.interrupt();
        counter=null;
    }
}

```

On trouve ensuite une méthode statique qui crée un `JFrame` et mets un `CounterPanel` à l'intérieur :

```

public static void main(String [] argv) {
    javax.swing.JFrame frame = new javax.swing.JFrame("Drawing");
    frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
    CounterPanel counter = new CounterPanel();
    frame.setContentPane((javax.swing.JComponent)counter);
    frame.pack();
    frame.setVisible(true);
}

```

1.4 Le compteur

Le compteur va quand à lui hériter de `java.lang.Thread`. Cela spécifie que le compteur va être dans un nouveau thread :

```
package ex1;
```

```

class Counter extends Thread {
    int value=0;
    javax.swing.JTextArea text = null;

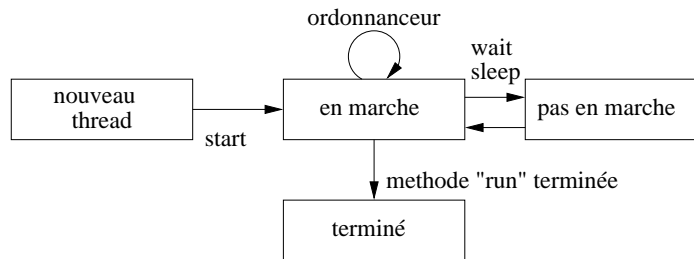
    Counter(javax.swing.JTextArea text) {
        this.text=text;
    }

    public void run() {
        while (true) {
            text.append(new Integer(value).toString()+"\n");
            value=value+1;
            try {
                Thread.sleep(1000);
            } catch (java.lang.InterruptedException e) {
                return;
            }
        }
    }
}

```

1.5 Cycle de vie des threads

Un thread est dans l'état initial tant que la méthode `start` n'a pas été appelée sur lui.



L'appel à la méthode `start` crée une nouvelle instance d'exécution, on arrive dans l'état "en marche" où le fonctionnement du thread est alors soumis à l'ordonnanceur de la machine virtuelle Java.

Une première façon de quitter l'état "en marche" est la terminaison de la méthode `run` qui achève également l'instance d'exécution.

Il existe en revanche deux méthodes pour suspendre le fonctionnement d'un thread :

- l'appel à la méthode `sleep` vue dans le paragraphe précédent où l'exécution du thread est suspendue pour un temps donné.
- l'appel à la méthode `wait` vue plus tard à la section 3 qui permet d'attendre certains événements.

L'appel à la méthode `interrupt()` sur un thread n'est détectée que si dans la méthode `run` du thread un appel à `interrupted()` est effectué pour voir si il y a une interruption.

2 Atomicité

2.1 Partage d'information

Comme dans l'exemple précédent où la variable `text` de type `JTextArea` est partagée entre les deux threads, on peut en fournissant des arguments au constructeur du thread, partager des informations entre threads.

Une autre façon de partager des variables est d'utiliser des variables statiques.

2.2 Application bancaire

On se place dans le cadre d'une application bancaire où deux personnes accèdent simultanément au même compte. Chaque personne est modélisée par un thread. Les personnes se partagent un objet une instance de la classe `Account` définie ci-dessous:

```
package ex2;

class Account {
    int value=0;
    Account(int value) {
        this.value=value;
    }
    public boolean withdrawMoney(int amount) {
        if (amount>value) return false;
        try {
            Thread.sleep(500);
        } catch (java.lang.InterruptedException e) {
            return false;
        }
        value=value-amount;
        System.out.println("il reste "+value);
        return true;
    }
}
```

```

    public boolean addMoney(int amount) {
        value=value+amount;
        return true;
    }
}

```

Le code complet est disponible à l'url :

http://www.derepas.com/java/cours4_exemple_2.jar

On remarque que la méthode `withdrawMoney` (retire argent) vérifie le solde du compte avant de faire le retrait. Cependant on obtient avec un compte initialement à 60, l'exécution suivante :

```

[java] il reste 40
[java] toto: retire 20.
[java] il reste 20
[java] tata: retire 20.
[java] il reste 0
[java] toto: retire 20.
[java] il reste -20
[java] tata: retire 20.
[java] toto: il n'y a plus d'argent!
[java] tata: il n'y a plus d'argent!

```

2.3 Explication

Le thread `tata` appelle `withdrawMoney` avec l'argument 20. Le thread `tata` vérifie alors que la valeur de `amount` est bien inférieur ou égale à celle de `value`.

Le thread `toto` appelle alors `withdrawMoney` avec l'argument 20, avant que `tata` ait effectué le retrait. Le thread `toto` vérifie alors que la valeur de `amount` est bien inférieur ou égale à celle de `value`.

On est alors dans une position où les deux threads peuvent simultanément faire un retrait.

Il est important de comprendre qu'ici on a systématiquement -20 sur le compte à cause de l'appel à la méthode `sleep(500)`. Mais même si on enlevait cet appel, le scénario précédent reste possible même si il n'est pas systématique. On comprend dès lors pourquoi les programmes multi-threadés sont difficiles à déboguer.

2.4 Solution : synchronized

Pour résoudre ce problème il faut que le test `amount > value` et l'instruction `value = value - amount` soient réalisés par le même thread sans être interrompu, autrement dit de manière atomique.

Un mot clé Java permet de faire cela : `synchronized`. Il suffit par exemple de rajouter dans le prototype de la méthode :

```

public synchronized boolean withdrawMoney(int amount) { ... }

```

Et on obtient alors l'exécution suivante :

```

[java] il reste 40
[java] toto: retire 20.
[java] il reste 20
[java] tata: retire 20.
[java] il reste 0
[java] toto: retire 20.
[java] tata: il n'y a plus d'argent!
[java] toto: il n'y a plus d'argent!

```

Le mot clé `synchronized` peut également s'utiliser dans le corps de la méthode `withdrawMoney` :

```

public boolean withdrawMoney(int amount) {
    synchronized(this) {
        if (amount>value) return false;
        try {

```

```

        Thread.sleep(500);
    } catch (java.lang.InterruptedException e) {
        return false;
    }
    value=value-amount;
}
System.out.println("il reste "+value);
return true;
}

```

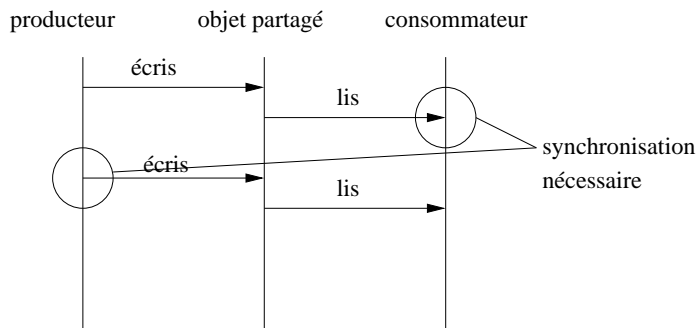
On obtient alors le même résultat que précédemment.

3 Communication inter threads

3.1 Exemple producteur consommateur

On se place dans le cas du programme multithreadé suivant : on dispose de deux threads l'un nommé producteur qui écrit dans un objet, l'autre nommé consommateur qui lit dans le même objet qui est donc partagé avec le producteur.

Si le producteur écrit plusieurs fois de suite sans que le consommateur lise, alors les valeurs sont perdues.



Comme montré à la figure ci-dessus il faut donc que le producteur attende que le consommateur ait lu la donnée avant d'en écrire une nouvelle.

3.2 L'objet partagé

Voici une description possible de l'objet partagé sans synchronisation producteur/consommateur :

```
package ex3;
```

```

class SharedObject {
    int value=0;
    boolean available = false;
    public SharedObject() { }
    public synchronized void write(int i) {
        if (!available) {
            value=i;
            available=true;
            System.out.println("Ecriture de "+i);
        }
    }
    public synchronized int read() {
        if (available) {
            System.out.println("Lecture de "+value);
            available = false;
            return value;
        }
    }
}

```

```

        return -1;
    }
}

```

L'exemple est téléchargeable à l'url :

http://www.derepas.com/java/cours4_exemple_3.jar

L'exécution n'est pas valide car si le résultat n'est pas disponible la méthode `read` renvoie -1, il faut alors systématiquement invoquer `read` via `while (read() == -1) { }` .

Une autre solution qui ne marche pas non plus serait dans chaque méthode d'attendre le changement de valeur du paramètre `available`. En effet les méthode étant `synchronized` elles mettent un verrou sur l'instance de `sharedObject`.

3.3 wait et notifyAll

La manière correcte d'implémenter des notions est d'utiliser les méthodes `wait` et `notifyAll`.

L'appel à la méthode `wait` suspend l'exécution d'un `thread` jusqu'à ce qu'un appel à `notifyAll` soit effectué par un autre `thread`.

Voici alors le code de `SharedObject` :

```

package ex3;

class SharedObject {
    int value=0;
    boolean available = false;
    public SharedObject() { }
    public synchronized void write(int i) throws InterruptedException {
        if (available) {
            System.out.println("Write en attente");
            wait(); // attendre que le consommateur ait lû
        }
        value=i;
        available=true;
        System.out.println("Ecriture de "+i);
        notifyAll(); // notifier le consommateur
    }
    public synchronized int read() throws InterruptedException {
        if (!available) {
            System.out.println("Read en attente");
            wait(); // attendre que le producteur ait écrit
        }
        System.out.println("Lecture de "+value);
        notifyAll(); // notifier le consommateur que
            // la lecture a été effectuée
        available = false;
        return value;
    }
}

```

3.4 Priorités

Les `threads` peuvent avoir des priorités d'exécutions :

```

Thread thread = new Thread( ... );
thread.setPriority(1);

```

La priorité d'un `thread` est l'intervalle : `Thread.MIN_PRIORITY, Thread.MAX_PRIORITY`.

Les priorités sont valables "en moyenne", c'est à dire qu'en moyenne un `thread` avec une priorité plus élevée s'exécutera plus souvent.

3.5 Groupes de threads

Chaque thread appartient à un groupe de thread, qui est une instance de `ThreadGroup`. On obtient le groupe d'un thread par l'appel à la méthode `getThreadGroup()`.

Il existe des constructeurs de `Thread` pour directement créer un thread dans un groupe :

```
public Thread(ThreadGroup group, Runnable runnable)
```

Les groupes permettent de facilement manipuler de manière simultanée tous les éléments du groupe.

4 Autres structures de données

Depuis le J2SE 5.0 le package `java.util.concurrent` contient plusieurs classes souvent utilisées en programmation concurrente. Nous présentons ici seulement l'interface `BlockingQueue` et la classe `Semaphore`.

4.1 BlockingQueue

L'interface `BlockingQueue` définit une file qui est thread-safe, c'est à dire dont l'usage peut être fait par des instances d'exécution concurrentes. Il existe plusieurs implémentations de cette classe : `ArrayBlockingQueue`, `DelayQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, `SynchronousQueue`.

Ainsi on peut re-écrire l'exemple précédent de manière beaucoup plus simple, sans même utiliser le mot clé `synchronized` :

```
package ex4;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.SynchronousQueue;

class SharedObject {
    private final BlockingQueue queue;
    public SharedObject() {
        queue = new SynchronousQueue<Integer>();
    }
    public void write(int i) throws InterruptedException {
        System.out.println("Ecriture de "+i);
        queue.put(i);
    }
    public int read() throws InterruptedException {
        int value = (Integer)queue.take();
        System.out.println("Lecture de "+value);
        return value;
    }
}
```

L'exemple est téléchargeable à l'url :

http://www.derepas.com/java/cours4_exemple_4.jar

4.2 Semaphore

Un sémaphore est un objet qui permet de restreindre le nombre de threads accédant à une ressource. L'appel à la méthode `acquire` est bloquante si le nombre de thread maximum a déjà appelé `acquire` sans avoir encore appelé `release`.

Ce type de structure de donnée permet de restreindre l'usage du mot clé `synchronized` à des portions de code relativement petite.

```
import java.util.concurrent.Semaphore;
class Pool {
    private static final int MAX_AVAILABLE = 100;
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);
```

```

    public Object getItem() throws InterruptedException {
available.acquire();
return getNextAvailableItem();
    }

    public void putItem(Object x) {
        if (markAsUnused(x))
available.release();
    }

    protected Integer[] items = new Integer[MAX_AVAILABLE];
    protected boolean[] used = new boolean[MAX_AVAILABLE];

    protected synchronized Object getNextAvailableItem() {
for (int i = 0; i < MAX_AVAILABLE; ++i) {
    if (!used[i]) {
used[i] = true;
return items[i];
    }
}
return null;
    }

    protected synchronized boolean markAsUnused(Object item) {
for (int i = 0; i < MAX_AVAILABLE; ++i) {
    if (item == items[i]) {
if (used[i]) {
        used[i] = false;
return true;
    } else
        return false;
    }
}
return false;
    }
}

```

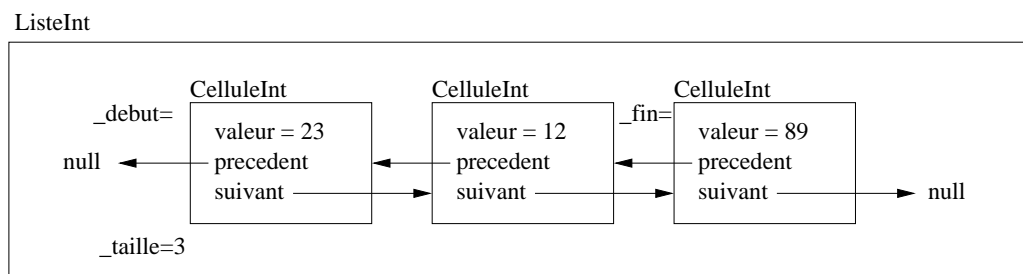
5 Spécificités J2SE 5.0

Cette section détaille certains aspects du langage Java apparus dans le Java 2 Platform Standard Edition (J2SE) à la fin de l'année 2004. Le J2SE version 5.0 a été également parfois nommé version 1.5 car il suivait les versions 1.3 et 1.4.

5.1 Types génériques

L'idée des types génériques est d'éviter d'écrire du code redondant. Prenons un exemple : pour écrire un code permettant de manipuler une liste d'entiers :

Voici un exemple pour la liste (23,12,89), où la liste de type `ListeInt` contient 3 cellules de type `CelluleInt` :



Voici une implémentation possible de la classe `ListeInt` :

```
class ListeInt {
    class CelluleInt {
        int _valeur;
        CelluleInt precedent = null;
        CelluleInt suivant = null;
        CelluleInt(int i) { _valeur=i; }
        public int valeur() { return _valeur;}
    }

    CelluleInt _debut = null;
    CelluleInt _fin = null;
    int _taille=0;
    public ListeInt() { }
    public CelluleInt debut() { return _debut; }
    public CelluleInt fin() { return _fin; }
    public int taille() { return _taille;}
    public void ajoute_debut(int i) {
        CelluleInt nouv_debut = new CelluleInt(i);
        nouv_debut.suivant=_debut;
        _debut.precedent=nouv_debut;
        _debut=nouv_debut;
        _taille=_taille+1;
    }
}
```

Cette classe est relativement inutile car elle est redondant avec `java.util.Vector`, elle est donnée à titre d'exemple. Pour obtenir le premier élément d'une instance `l` d'une liste on peut faire `l.debut.valeur()`. Si l'on voulait faire des listes d'un autre type d'objet que des `int`, il faudrait re-écrire la classe ou bien utiliser des `cast` : `(MonObjet)l.debut.valeur()` (c'est ce qui était fait dans les JDK jusqu'au 1.4).

Les type génériques nous permettent de contourner ceci : pour cela au lieu d'utiliser le type `ListeInt` nous allons utiliser `Liste<T>`, où `T` peut être n'importe quelle classe. Voici l'exemple précédent traduit avec des types génériques pour `Liste` et `Cellule` :

```
class Liste<T> {
    class Cellule<T> {
        T _valeur;
        Cellule precedent = null;
        Cellule suivant = null;
        Cellule(T i) { _valeur=i; }
        public T valeur() { return _valeur;}
    }

    Cellule<T> _debut = null;
    Cellule<T> _fin = null;
    int _taille=0;
    public Liste() { }
    public Cellule<T> debut() { return _debut; }
    public Cellule<T> fin() { return _fin; }
    public int taille() { return _taille;}
    public void ajoute_debut(T i) {
        Cellule<T> nouv_debut = new Cellule<T>(i);
        nouv_debut.suivant=_debut;
        _debut.precedent=nouv_debut;
        _debut=nouv_debut;
        _taille=_taille+1;
    }
}
```

```
}
```

On peut alors instancier des listes d'entiers ou de chaîne de caractères de la manière suivante :

```
Liste<Integer> lInt = new Liste<Integer>();  
lInt.ajoute_debut(345);  
Liste<String> lString = new Liste<String>();  
lString.ajoute_debut("foo");
```

5.2 Autoboxing et Auto-Unboxing

Dans l'exemple précédent nous avons écrit :

```
Liste<Integer> lInt = new Liste<Integer>();  
lInt.ajoute_debut(345);
```

Nous avons en fait fourni un objet de type `int` : 345, qui a été automatiquement converti en `Integer`. Dans les versions précédentes du JDK nous aurions dû écrire : `lInt.ajoute_debut(new Integer(345))` ;

La conversion automatique des types de base (`int`, `boolean`) en des classes équivalentes (`Integer`, `Boolean`) est fait automatiquement.

5.3 Boucles rapides

Voici une façon standard de parcourir les éléments d'une liste avec une boucle `for` :

```
import java.util.*;  
class Boucles {  
    public static void main(String[] args) {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        list.add(100);list.add(102);  
        for (Iterator i = list.iterator(); i.hasNext();) {  
            Integer value=(Integer)i.next();  
            System.out.println(value);  
        }  
    }  
}
```

Ce programme affiche 100 puis 102. On peut également utiliser la syntaxe suivante pour parcourir directement les éléments de la liste :

```
import java.util.*;  
class Boucles {  
    public static void main(String[] args) {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        list.add(100);list.add(102);  
        for (Integer i : list) {  
            System.out.println(i);  
        }  
    }  
}
```

5.4 Types énumérés

Certains champs d'une classe peuvent avoir un nombre déterminé de valeurs. Ainsi supposons que pour une voiture on dispose de trois couleurs : rouge, jaune et noir. Une façon possible de modélisation en java est la suivante :

```
class Voiture {  
    public static final int ROUGE =0;  
    public static final int JAUNE =1;  
    public static final int NOIR =2;
```

```

    int couleur = ROUGE;

    // ...
}

```

Depuis le JDK 5.0 on peut utiliser un type énuméré :

```

class Voiture {
    public enum Couleur { rouge, jaune, noir};
    Couleur couleur = Couleur.rouge;

    // ...
}

```

5.5 Import statique

Depuis le JDK 5.0 on peut importer les symboles statiques d'une classe dans l'espace de nommage courant. Ainsi au lieu à chaque fois d'utiliser la qualification : `BorderLayout.CENTER` on peut faire :

```

import static java.awt.BorderLayout.*;
...
getContentPane().add(new JPanel(), CENTER);

```

5.6 Nombre d'arguments variables

Depuis le JDK 5.0 une fonction peut avoir un nombre variable d'arguments, comme illustré ci-dessous :

```

class Vararg {
    static void argtest(String ... args) {
        for (int i=0;i <args.length; i++) {
            System.out.println(args[i]);
        }
    }
    public static void main(String [] arg) {
        argtest("un argument");
        argtest("un argument","un autre");
        argtest("un argument","un autre","et encore un!");
    }
}

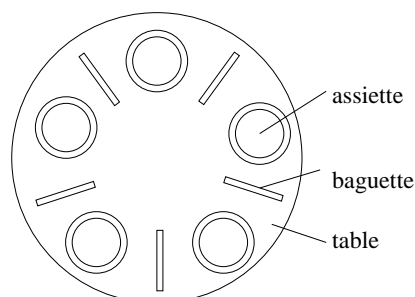
```

6 Exercices

Exercice 4.1

Cet exercice traite le problème classique du dîner des philosophes.

Cinq philosophes sont au restaurant chinois assis à une table circulaire.



Il y a seulement 5 baguettes sur la table, une entre chaque assiette. Or pour manger le riz il faut deux baguettes. Ainsi pour manger un philosophe peut prendre une baguette à sa gauche, une autre à sa droite, manger pendant un certain temps puis les reposer. Un de ses voisins peut alors manger, et ainsi de suite. Un même philosophe mange plusieurs fois.

Q1 Créer une classe `Diner` qui instancie 5 variables de la classe `Philosophes` qui sont des threads modélisant l'énoncé ci-dessus. Remarque : il faut également modéliser les baguettes, qui sont des objets partagés entre les threads.

Si cette question vous semble difficile pour pouvez partir avec une partie de la solution (il suffit alors d'écrire la méthode `run` dans la classe `Philosophe`):

```
http://www.derepas.com/java/cours4_exercice_1_indication.jar
```

Q2 Dans la question précédente un cas de blocage est possible : chaque philosophe prend sa baguette droite puis attend indéfiniment d'avoir la gauche. Mettre en évidence ce cas de blocage (par exemple en rajoutant un délai de 1 seconde entre la prise de la baguette droite et la prise de la gauche).

Q3 Une méthode fréquente pour éviter les cas de blocage est de mettre des priorités. On numérote les baguettes de 1 à 5. Pour manger un philosophe prend alors toujours sa baguette de plus petit nombre plus celle de plus grand nombre. Implémenter cette nouvelle solution.

Exercice 4.2

Q1 Ecrire un thread dont la méthode `run` incrémente un compteur. Lancer le thread puis l'arrêter au bout de 10 secondes à l'aide de la méthode `interrupt`. Afficher la valeur du compteur. Faire de même en lançant deux threads, commentez les résultats.

Q2 Affichez les constantes `MAX_PRIORITY`, `MIN_PRIORITY`, `NORM_PRIORITY`. Vérifiez que pour qu'un thread s'exécute deux fois plus vite sa priorité doit être deux fois plus élevée.