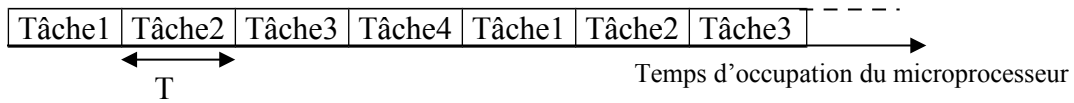
# Langage Java Chapitre 7 : les threads

## 1. INTRODUCTION

### LES CONCEPTS DE BASE: LE MULTITACHE ET LES THREADS

Une tâche ou processus est un programme en cours d'exécution à un instant donné. Les systèmes d'exploitation actuels sont de type multitâche, c'est à dire qu'ils donnent l'impression que plusieurs tâches s'exécutent simultanément.
En réalité, le temps est découpé en intervalles réguliers : le système d'exploitation alloue le microprocesseur a une tâche donnée pendant un intervalle de temps, suspend cette tâche quand son temps est terminé puis alloue le microprocesseur à autre tâche pendant l'intervalle de temps suivant, et ainsi de suite.
Le système d'exploitation assure ainsi une allocation périodique du microprocesseur aux diverses tâches permettant ainsi l'exécution de plusieurs tâches par le microprocesseur. Cette rotation rapide donne l'apparence de la simultanéité de l'exécution des tâches. Le module logiciel du système d'exploitation chargé de la commutation des tâches est appelé l'ordonnanceur (scheduler).




L'exécution d'un programme Java où n'est pas défini explicitement de thread donne naissance à l'exécution d'un processus initié par la méthode main(). On peut dire alors que ce processus ne contient qu'un seul thread, le thread peut ainsi être défini comme étant l'activité du processus.
Un processus peut avoir plusieurs activités et donc donner naissance à l'exécution de plusieurs threads.
Par exemple, un navigateur peut simultanément télécharger un fichier et explorer un site.

Le système de commutation précédent sur les tâches s'applique pour les threads : la machine virtuelle Java s'occupe de faire commuter les threads d'un processus Java, ces threads donnent alors l'impression d'être exécutés simultanément.

### TECHNIQUES POUR CREER DES THREADS EN JAVA

Il existe 2 techniques pour créer des threads :

- créer une classe dérivée de la classe **Thread** (java.lang.Thread),
- créer une classe qui implémente l'interface **Runnable** (java.lang.Runnable).

Dans les 2 cas :
- il faut implémenter la méthode **run()** pour définir le comportement du thread (c'est à dire les instructions qu'il exécute),
- l'objet thread est démarré par exécution de sa méthode **start()** qui entraîne automatiquement l'exécution de la méthode **run()**,
- le thread se termine quand sa méthode run() finit.

# 2. DERIVATION DE LA CLASSE Thread

## 2.1 1er exemple : création de 2 threads identiques

```
class Thread1 extends Thread{
   String nom ;
   public Thread1(String s) {
   nom = s ;
   }
   public void run() {
     String ch= getName() ;
     for (int i=0; i<3; i++) {
        System.out.println("Thread "+nom+ " de nom "+ch+ " i= "+i);
        }
   }
}

public class TestThread1 {
  public static void main(String argv[]) {
   Thread1 th1, th2 ;
   th1 = new Thread1("pluie");
   th2= new Thread1("soleil") ;
   th1.start() ;
   th2.start();
   }
}
```

**Résultat :**

```
Thread pluie de nom Thread-0 i= 0
Thread pluie de nom Thread-0 i= 1
Thread pluie de nom Thread-0 i= 2
Thread soleil de nom Thread-1 i= 0
Thread soleil de nom Thread-1 i= 1
Thread soleil de nom Thread-1 i= 2
```

Les threads sont créés lors de l'appel du constructeur.

**`void run()`** : cette méthode est définie dans la classe `java.lang.thread`, où elle ne fait rien ; il faut la redéfinir en y donnant le code correspondant à ce qu'on attend du thread correspondant. La méthode `run` doit être appelée par l'intermédiaire de la méthode `start`.

**`void start()`** : la méthode `start` est définie dans la classe `java.lang.thread`. Elle appelle la méthode `run` pour démarrer le thread, et ce qui est très particulier est qu'elle "retourne" immédiatement, sans attendre que la méthode `run` ait terminée son travail.

**String getName()** : méthode de la classe Thread qui retourne le nom attribué par Java au thread.

## 2.2 2ème exemple : priorité des threads

```
class Thread1 extends Thread{
   String nom ;
   public Thread1(String s) {
   nom = s ;
   }
   public void run() {
     String ch= this.getName() ;
     for (int i=0; i<3; i++) {
        int priorite = getPriority();
```

```
        System.out.println("Thread "+nom+ " de nom "+ch+ " i= "+i+ " priorité = "+ priorite);
        if ((nom.compareTo("pluie"))==0) setPriority (MIN_PRIORITY);
        else setPriority (MAX_PRIORITY);
        }
    }
}
public class TestThread1 {
  public static void main(String argv[]) {
    Thread1 th1, th2 ;
    th1 = new Thread1("pluie");
    th2= new Thread1("soleil") ;
    th1.start() ;
    th2.start();
    }
}
```

**Résultat :**

```
Thread pluie de nom Thread-0 i= 0 priorité = 5
Thread soleil de nom Thread-1 i= 0 priorité = 5
Thread soleil de nom Thread-1 i= 1 priorité = 10
Thread soleil de nom Thread-1 i= 2 priorité = 10
Thread pluie de nom Thread-0 i= 1  priorité = 1
Thread pluie de nom Thread-0 i= 2  priorité = 1
```

**`getPriority()`** : la méthode `getPriority` de la classe `java.lang.Thread` `retourne` le niveau de priorité du thread courant.
**`setPriority()`** : la méthode `setPriority` permet de fixe le niveau de priorité d'un thread.
**`MIN_PRIORITY`** : priorité minimale pour un thread.
**`MAX_PRIORITY`** : priorité maximale pour un thread.

La priorité initiale (par défaut) des threads est de 5.
La priorité du thread « pluie » est abaissée à la valeur minimale (1) lors du 1er passage de la boucle alors que celle de « soleil » est mise à sa valeur maximale (10).
L'influence de la priorité est bien montrée dans l'affichage résultant.

### 2.3 Travail demandé

2.31 Tester le 1er exemple du chapitre 2 page 2.
Ajouter un 3ème thread « neige ».

2.32 Tester le 2ème exemple du chapitre 2. Ajouter un 3ème thread « neige », afficher son nom et sa priorité, faire quelques essais de modification des priorités, observer les résultats.

## 3. IMPLEMENTATION DE L'INTERFACE Runnable

### 3.1 Exemple

```
class Thread2 implements Runnable
{
 String chaine;
 Thread2(String chaine) {
    this.chaine=chaine;
  }

 public void run()  {
    for (int i=0;i<2;i++)  {
         System.out.println(chaine +" "+Thread.currentThread() );
        }
```

```
    }
}

public class TestThread2
{
  public static void main(String[] argv)
    {
     Thread2 th1 =  new Thread2("soleil") ;
     Thread2 th2 =  new Thread2("pluie") ;
     Thread2 th3 =  new Thread2("neige") ;
     new Thread(th1).start();
     new Thread(th2).start();
     new Thread(th3).start();
    }
}
```

**Résultats :**

```
soleil Thread[Thread-0,5,main]
soleil Thread[Thread-0,5,main]
pluie Thread[Thread-1,5,main]
pluie Thread[Thread-1,5,main]
neige Thread[Thread-2,5,main]
neige Thread[Thread-2,5,main]
```

L'interface **java.lang.Runnable** offre la possibilité de créer une classe avec 2 comportements : un comportement « classique » mais également un comportement de type thread. Lorsque l'objet est créé, on peut démarrer le thread quand c'est nécessaire, la fin du thread n'entraînant pas la fin de l'objet.

La méthode **run()** est la seule méthode à redéfinir.

L'application doit instancier un objet **cible** implémentant **Runnable** :

```
class Thread2 implements Runnable {...}
Thread2 th1 =  new Thread2("soleil") ;
```

puis créer un thread en utilisant le constructeur **Thread(Runnable cible)** et le démarrer par :

```
new Thread(th1).start() ;
```

ou

```
Thread  t = new Thread(th1) ;
t.start();
```

Le nom du thread est obtenu par la méthode **Thread.currentThread()** : Thread-n est le nom du thread, 5 sa priorité et main le groupe de thread auquel il appartient par défaut.

### 3.2 Travail demandé

Tester cet exemple.

# 4. LA SYNCHONISATION AVEC synchronized

## ACCES D'UNE RESSOURCE COMMUNE PAR PLUSIEURS THREADS

Lorsqu'on développe une application en utilisant des threads, l'accès à une ressource commune par plusieurs threads est une situation très fréquemment rencontrée.
L'exemple suivant illustre une situation où plusieurs threads accèdent en même temps à la ressource commune.
Prenons le cas d'une méthode activée par un thread dont le but est d'autoriser un retrait d'argent sur un compte, le solde est ici la ressource commune.

```
public boolean  retirerArgent(int retrait) {
        if  (solde >= retrait ) {
                        solde = solde – retrait ;
                        return true ;
```

```
                    }
        else return false ;
        }
```

Imaginons la situation extrême où 2 personnes autorisées, par exemple les 2 conjoints, retirent de l'argent sur un même compte et, fâcheuse coïncidence, en même temps. Ces 2 personnes veulent retirer chacune une somme d'argent inférieure au solde du compte.
Cette méthode est alors activée simultanément 2 fois. Du fait de la commutation des threads, il est possible que le 1er thread n'exécute que l'instruction **if (solde >= retrait )**, le 1er retrait est donc autorisé, puis le 2ème thread exécute à son tour **if (solde >= retrait )** et le 2nd retrait est à son tour autorisé. Le solde final peut ainsi être négatif si la somme des 2 retraits est supérieure au solde. Cet exemple illustre un dysfonctionnement consécutif à l'accès simultané de plusieurs threads à une ressource commune.

Quelle est ici la ressource commune ?
Le solde est la ressource commune, mais en programmation Java il faut aller plus loin et créer des objets.

On peut définir une *classe Compte*, dont *solde* est un *attribut* et *retirerArgent()* une *méthode*.

```
class Compte {
        private solde ;
        public Compte(---) {  ---- }                    // Fin du constructeur
public boolean  retirerArgent(int retrait) {
        if  (solde >= retrait ) {
                        solde = solde – retrait ;
                        return true ;
                        }
        else return false ;
        }
}
```

Lors du retrait, un objet Compte est créé, la méthode retirerArgent() est exécutée 2 fois simultanément (chacune par un thread) lors des retraits d'argent simultanés du mari et de sa femme.
Il faut imposer que sur l'objet commun Compte créé, il soit impossible d'exécuter simultanément plusieurs fois la méthode retirerArgent() : il faut pour cela déclarer la méthode **synchronized**.

La méthode **retirerArgent()** est déclarée **synchronized** de la manière suivante :

```
public synchronized boolean  retirerArgent(int retrait) {
        if  (solde >= retrait ) {
                        solde = solde – retrait ;
                        return true ;
                        }
        else return false ;
        }
```

En Java, chaque objet **class** dispose d'un verrou, ce **verrou** peut être utilisé en déclarant dans la classe des méthodes **synchronized**.
Le verrou est pris quand une méthode **synchronized** s'exécute, le verrou est libéré dès que la méthode **synchronized** se termine. On réalise ainsi un mécanisme d'exclusion mutuelle.

Une méthode **synchronized** est dite **atomique** parce que la JVM ne l'exécutera jamais plusieurs fois « simultanément » : une méthode synchronized d'un objet ne peut interrompre une méthode synchronized du même objet qui est en cours d'exécution.

**EXEMPLE AVEC UNE METHODE synchronized**

Un classe Boite contient un tableau de 30 entiers dont les valeurs vont de 0 à 29. On crée 2 threads, chaque thread doit afficher le tableau en entier.

```
class Boite {
        int [] Tableau ;
        public Boite() {
                        Tableau = new int[30] ;
                         for (int i=0; i<30; i++) Tableau[i] = i ;
        }
        public  /*synchronized */ void afficher() {  //méthode à tester synchronisée ou non
                for (int i=0; i<30 ; i++)                    // balaye le tableau pour l'afficher
                        {
                        System.out.print(Tableau[i]+" ");
                        }
                System.out.println("");
     }
}
class Thread6 extends Thread{
        String nom ;
        public Boite tableau ;
        public Thread6(String nom) {
                        this.nom = nom ;
        }
        public  void run() {
                        tableau.afficher() ;
        }
}
public class TestThread6 {
  public static void main(String argv[]) {
  Boite b = new Boite() ;
  Thread6 t1, t2 ;
  t1 = new Thread6("liretableau1") ;
  t2 = new Thread6("liretableau2");
  t1.tableau = b ;
  t2.tableau = b ;
  t1.start() ;
  t2.start() ;
  }
}
```

Exemples du mélange à l'affichage des 2 tableaux

**Résultats**

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 0 20 1 21 2 22 3 4 5 6 7 8 9 10 11 12 13 14 15 16 23 24 25 26 27 28 29
17 18 19 20 21 22 23 24 25 26 27 28 29

On constate que l'affichage des 2 tableaux est mélangé.

Il faut enlever les commentaires autour du mot **synchronized** de la ligne **public /*synchronized */ void afficher(),** compiler et exécuter. L'affichage résultant suivant convient maintenant.

**Résultats**
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

## UN BLOC D'INSTRUCTIONS synchronized

L'exemple suivant présente toujours le problème de l'accès par 2 threads à une ressource commune. Chaque thread a son propre tableau de nombres, on veut qu'un 1er thread affiche entièrement son tableau puis que le 2nd affiche le sien et sans que les affichages se mélangent.

```
class Thread3 extends Thread{
  String nom ;
  int debut ;
  int fin ;
  public Thread3(String nom ,int d, int f) {
    this.nom = nom ;
    debut = d ;                                  //1er nombre début du tableau
    fin = f ;                                    // dernier fin du tableau
  }

  public void run() {
          for (int i=debut; i<fin ; i++)
                  {
                  System.out.println(nom+" "+i+" ");
                  }
  }
}

public class TestThread3 {
  public static void main(String argv[]) {
  Thread3 t1, t2 ;
  t1 = new Thread3("soleil",10,30) ;
  t2 = new Thread3("pluie",50,70);
  t1.start() ;
  t2.start() ;
  }
}
```

**Résultats** :
soleil 10
soleil 11
soleil 12
soleil 13
soleil 14
soleil 15
soleil 16
soleil 17
soleil 18
pluie 50
soleil 19
pluie 51
soleil 20
pluie 52
soleil 21
pluie 53
pluie 54
pluie 55
soleil 22
soleil 23

**Résultats suite**
soleil 24
soleil 25
soleil 26
soleil 27
soleil 28
soleil 29
pluie 56
pluie 57
pluie 58
pluie 59
pluie 60
pluie 61
pluie 62
pluie 63
pluie 64
pluie 65
pluie 66
pluie 67
pluie 68
pluie 69

On constate que les affichages sont mélangés.

Chaque thread est interrompu par la JVM pour laisser l'exécution à l'autre alors qu'il n'a pas terminé d'exécuter la boucle d'affichage.

**Les 2 threads partagent la ressource commune – le même objet commun - System.out pour afficher.**

Il faut indiquer que cette boucle d'affichage ne doit pas être interrompue. On effectue cela en posant un verrou sur l'objet commun **System.out** à l'aide du mot réservé **synchronized** de la manière suivante :

```
synchronized (System.out) { bloc d'instructions qui ne doit pas être interrompu }
```

On dit que le thread a verrouillé l'objet.
Il faut alors modifier la méthode run() de la manière suivante :

```
public void run() {
  synchronized (System.out) {
        for (int i=debut; i<fin ; i++)      System.out.println(nom+" "+i+" ");
       }
}
```

**En conclusion** :
Un verrou peut être mis sur un objet :

- soit par une méthode d'instance affectée du modificateur `synchronized`, le verrou est alors mis sur l'instance sur laquelle est invoquée la méthode.
- soit par l'instruction `synchronized` suivi du nom de l'objet indiqué entre parenthèses : `synchronized(unObjet)` met un verrou sur l'objet `unObjet`.

Ce mécanisme implémente l'exclusion mutuelle, c'est à dire que l'accès d'un thread à la ressource commune par une fonction « synchonized » de la ressource ou dans un bloc d'instructions « synchonized » sur la ressource interdit aux autres threads d'accéder à cette ressource par des fonctions « synchonized » sur la ressource ou dans leurs propres blocs d'instructions « synchonized » sur la ressource.

Attention : Une méthode ou un bloc « synchronized » pourrait être interrompu par un autre thread qui ferait des accès à la ressource sans avoir « synchonized » les instructions correspondantes.

**Travail demandé**

**4.4.1** Tester l'exemple du chapitre 4.2 page 6.

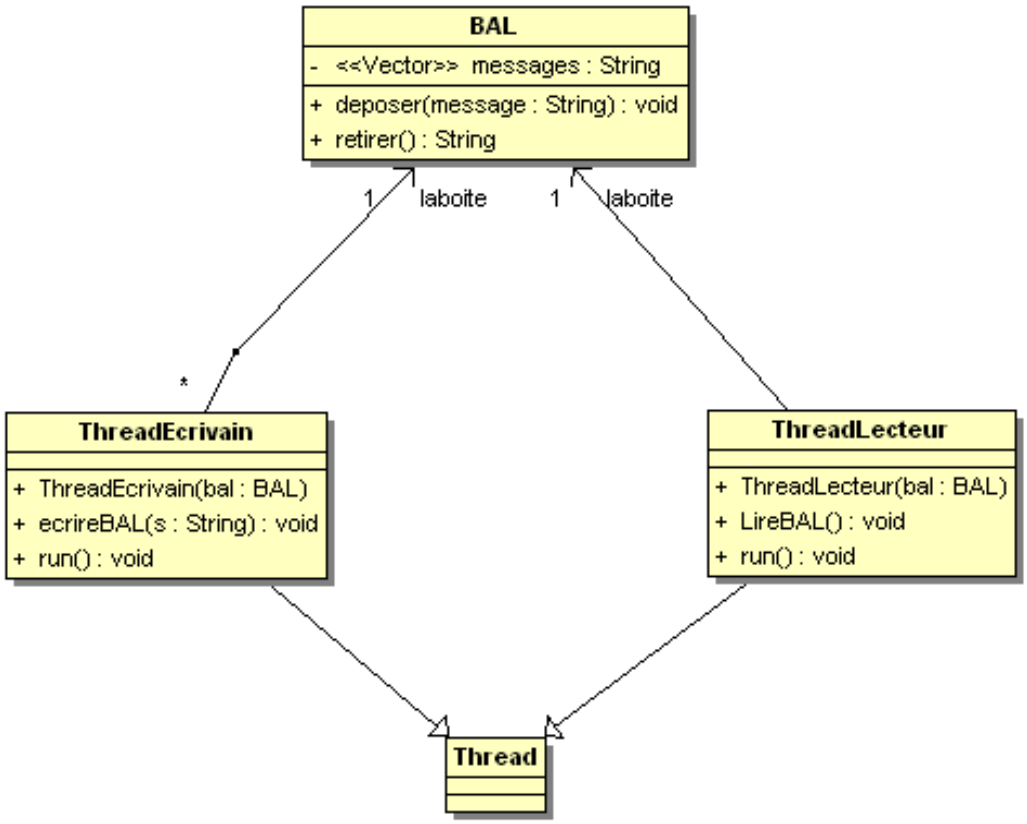
**4.4.2** Tester l'exemple du chapitre 4.3 page 7.

**4.4.3** Exercice de la boîte aux lettres.

Le diagramme UML suivant propose les classes nécessaires au fonctionnement décrit en suivant.

On demande de créer une classe Boîte aux lettres pour stocker des messages simples de type String, le stockage dans cette boîte fonctionne sur le principe du 1er message entré – 1er message sorti (FIFO). Ce stockage peut être implémenté à l'aide d'un Vecteur. La boîte aux lettres est accessible à tous les threads utilisateurs à l'aide de 2 méthodes **deposer()** et **retirer()**. Il ne faut pas oublier de gérer la tentative de retrait de message si la boîte est vide. On mettra en place un mécanisme d'exclusion (synchronisation) des méthodes concernées. Il est ici préférable de gérer la synchronisation sur l'objet commun BAL. La méthode run() de chaque Thread appelle l'exécution de ecrireBAL() ou lireBAL() suivant que le Thread est écrivain ou lecteur.

On peut prendre l'exemple d'une boîte mail accessible à une personne en lecture (un seul thread lecteur) et à plusieurs personnes en écriture.
Ecrire un programme qui teste le fonctionnement avec de nombreux threads Ecrivain et un seul thread Lecteur de la boîte aux lettres.




**4.4.4** On doit implémenter une classe Vect (qui utilise un Vector) destinée à recevoir des chaînes de caractères, et comprenant:

- une méthode add() pour ajouter une chaîne et afficher tout le contenu du Vector dans son état au moment de l'appel.
- une méthode sort() pour trier les éléments du Vector par ordre alphabétique.
  On pourra effectuer un tri itératif simple par recherche et positionnement du dernier élément etc. ( la comparaison de deux chaînes peut s'effectuer avec la méthode String.compareTo( String ) qui renvoie une valeur < 0 , 0 ou > 0 selon la position relative des chaînes ).

L'ajout d'éléments se fera uniquement à partir du « thread » main(), tandis que le tri se fera uniquement à partir d'un Thread Trieuse crée pour cela, et comprenant:

- la méthode run() qui effectuera automatiquement le tri du Vector toutes les 10 secondes, donc de façon indépendante de la saisie.
- une méthode end() qui sera appelée en fin de programme pour mettre fin au Thread en utilisant un flag booléen.

Afin que les procédures de saisie et de tri ne se télescopent pas, on mettra en place un mécanisme d'exclusion (synchronisation) des méthodes concernées.
On affichera également des messages circonstanciés lors du démarrage et de l'arrêt du Thread afin d'en vérifier le bon fonctionnement.

# 6. QUELQUES COMPLEMENTS SUR LES THREADS

### a. LA METHODE sleep()

La méthode **sleep(long milliseconde)** de la classe Thread suspend l'exécution du thread pendant un temps donné en milliseconde, elle doit être placée dans un bloc try --- catch comme indiqué :

```
try {  sleep(duree) ; }
catch (InterruptedException  e) { --- } ;
```

### b. LA METHODE join()

Exemple d'utilisation avec le 1er exemple du chapitre 2 :

```
static void main(String argv[]) {
            Thread1 th1, th2 ;
             th1 = new Thread1("pluie");
             th2= new Thread1("soleil") ;
             th1.start() ;
            th2.start();
            try {    th1.join() ;
                     th2.join() ;
            }
      catch(InterruptedException e) { } ;
   System.out.println("Threads terminés) ;
   }
```

L'instruction th1.join(); suspend la méthode appelante jusqu'à ce que le thread th1 se termine.
L'instruction th2.join(); suspend la méthode appelante jusqu'à ce que le thread th2 se termine.

### c. TERMINER UN THREAD

Un thread se termine lorsque la méthode run() finit.
Soit la méthode run() parcourt un nombre fini de fois un certain nombre d'instructions, et elle se termine normalement.
Soit la boucle de la méthode run() teste un booléen à chaque entrée ou sortie de la boucle pour se terminer. Ce booléen doit alors être positionné par un autre thread.

# 7. LES THREADS user et daemon

Tous les threads précédents sont des threads de type *user*. La fin de la méthode **main** n'impose pas la fin de ces threads.

Un thread *daemon* se termine obligatoirement quand la main méthode main se termine.
La méthode **setDaemon(boolean on)** de la classe **Thread** passe en *daemon* le thread concerné.

```
class ThreadDaemon extends Thread{
   String nom ;
   public ThreadDaemon(String s) {
   nom = s ;
```

```
   }
   public void run() {
     String ch= getName() ;
     for (int i=0; i<30; i++) {
        System.out.println("Thread "+nom+ " de nom "+ch+ " i= "+i);
        }
   }
}

public class TestDaemon {
  public static void main(String argv[]) {
          ThreadDaemon th1 = new ThreadDaemon("pluie");
          ThreadDaemon th2= new ThreadDaemon("soleil") ;
          //th1.setDaemon(true);                                    /*ligne1*/
          //th2.setDaemon(true);                                    /*ligne2*/
          th1.start() ;
          th2.start();
          try {
                  Thread.sleep(1);
          } catch (InterruptedException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
          }
   }
}
```

Tester et observer le fonctionnement de cette application.

Dé commenter les lignes 1 et 2. A quoi servent-elles ?
Tester, observer et justifier le fonctionnement de cette application.

## Remarque

Ce cours a présenté les techniques pour créer des threads en Java et gérer le partage d'une ressource commune à plusieurs threads en implantant le mécanisme de l'exclusion mutuelle avec **synchronized**.
Les techniques pour gérer l'ordre d'accès des threads à une ressource commune n'ont pas été étudiées.