**Langage Java Chapitre 8 : 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).

Tâche1 Tâche2 Tâche3 Tâche4 Tâche1 Tâche2 Tâche3

Temps d’occupation du microprocesseur

T

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

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.

**Ou attribué par défaut.**

## 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);**

} }

} **Résultat :**

public class TestThread1 { Thread pluie de nom Thread-0 i= 0 priorité = 5

public static void main(String argv[]) { Thread soleil de nom Thread-1 i= 0 priorité = 5 Thread1 th1, th2 ; Thread soleil de nom Thread-1 i= 1 priorité = 10 th1 = new Thread1("pluie"); Thread soleil de nom Thread-1 i= 2 priorité = 10

th2= new Thread1("soleil") ; Thread pluie de nom Thread-0 i= 1 priorité = 1 th1.start() ; Thread pluie de nom Thread-0 i= 2 priorité = 1 th2.start();

}

}

**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

2.3.1 Tester le 1er exemple du chapitre 2.1 page 2.

Ajouter un 3ème thread « neige ».

2.3.2 Tester le 2ème exemple du chapitre 2.2. 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 l'exemple précédent.

# 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.

Sur le serveur, 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 mais le solde n'est pas encore modifié, 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(---) { ---- }  public boolean retirerArgent(int retrait) { if (solde >= retrait ) {  solde = solde – retrait ; return true ;  }  else return false ;  }  } | // Fin du constructeur |

Dans la situation précédente, 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() ;

}

}

**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

Exemples du mélange à l’affichage des 2 tableaux

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 ;

//1

er

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() ;

}

}

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.

**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

**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.

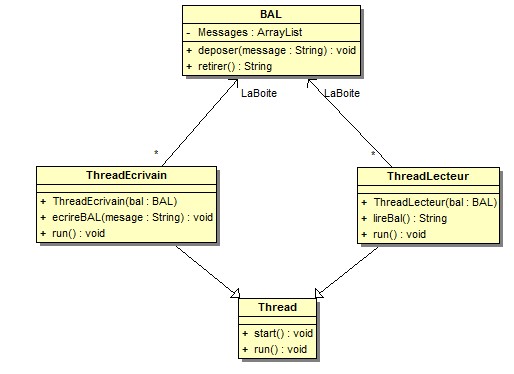
**4.4.3** Exercice de la boîte aux lettres.

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

On demande de créer une classe Boîte aux lettres (BAL) 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 doit être implémenté à l’aide d’un **ArrayList**. 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 soit 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 demande d'implémenter une classe Tampon (qui utilise un **ArrayList**) destinée à recevoir des chaînes de caractères, et comprenant:

* une méthode add() pour ajouter une chaîne et afficher tout le contenu du **ArrayList** dans son état au moment de l'appel.
* une méthode sort() pour trier les éléments du **ArrayList** par ordre alphabétique.

On pourra effectuer un tri itératif simple par recherche et positionnement du dernier élément etc. ( la comparaison de deux chaînes doit 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 ArrayList 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.

Terminer proprement un thread : exemple java 8

private volatile Thread blinker;

public void stop() {// il faut coder la fonction stop pour interrompre l’exécution du thread.

blinker = null;

}

public void run() {

Thread thisThread = Thread.currentThread();

while (blinker == thisThread) {

try {

Thread.sleep(interval);

} catch (InterruptedException e){

}

repaint();// fonction quelconque appelée par le thread à chaque exécution

}

}

**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.

Plan

1. Programmation concurrente versus programmation parallèle
   1. Quelles sont les différences ?

Il existe une grande différence entre une exécution en parallèle et une exécution concurrente.

Dans les deux cas, il s’agit de l’utilisation de plusieurs threads. Cependant, rien n’implique que votre code soit exécuté en parallèle. Cela dépend du système opérationnel et de la machine sur laquelle le code tourne.

Sur une machine ne contenant qu’un seul processeur, il est évident que les threads devront partager cette ressource principale. Dans ce cas, on comprend que le processeur ne puisse gérer qu’un seul thread à chaque instant. Un code en multithreading sera alors moins efficient qu’un code séquentiel. Les différents threads devant attendre la fin du processus précédent avant de pouvoir exécuter leurs propres tâches.

Dans le cas d’une exécution en parallèle les threads ne partagent pas de ressources. Sur une machine comprenant plusieurs processeurs, le processeur 1 s’occupe de réaliser les tâches du thread 1 alors que le processeur 2 s’occupe de réaliser de son côté les tâches du thread 2. Deux tâches sont alors parallélisées et le temps imparti pour les réaliser est divisé par Deux. Vous avez alors un vrai gain de performances.

A travers cet exemple simple, on touche la principale difficulté de la programmation en multithreading qui est l’accès concurrent aux ressources.

* + 1. La notion de ressources partagées.

On a parlé de l’accès concurrent au processeur, cependant, il peut aussi y avoir des accès concurrents à certains espaces mémoires, à des serveurs, à des données partagées. En tant que développeur, cela revient toujours à devoir gérer un accès à une donnée partagée. Pour mieux illustrer ce concept voici l’exemple d’une bibliothèque universitaire.

* + 1. Un exemple pour mieux comprendre.

Un professeur de littérature a donné une liste (une collection) de 7 livres à étudier à un petit groupe de 4 étudiants. Chacun d’entre eux doit lire tous les livres. La difficulté ici réside dans le fait que ces livres très anciens et très précieux n’existent que dans la bibliothèque de l’Université. Pour faire le travail qui leur a été assigné, les étudiants entrent en concurrence pour l’accès à la ressource qui n’est autre que l’accès aux livres. Lorsqu’un étudiant emprunte un livre et le lit les autres doivent attendre que le livre soit à nouveau disponible pour pouvoir l’emprunter et le lire à leur tour. On s’aperçoit rapidement que bien que la situation soit simple, sa résolution et sa programmation le sont moins. On va rapidement devoir faire face à des notions de priorité, d’attente et la gestion d’alertes permettant de signifier que la ressource est disponible ou non. Avant de se lancer dans sa programmation, faisons l’inventaire des notions et des outils qui devraient nous permettre de mieux appréhender le problème.

* 1. Le multithreading
     1. Les notions fondamentales.
        1. DeadLock

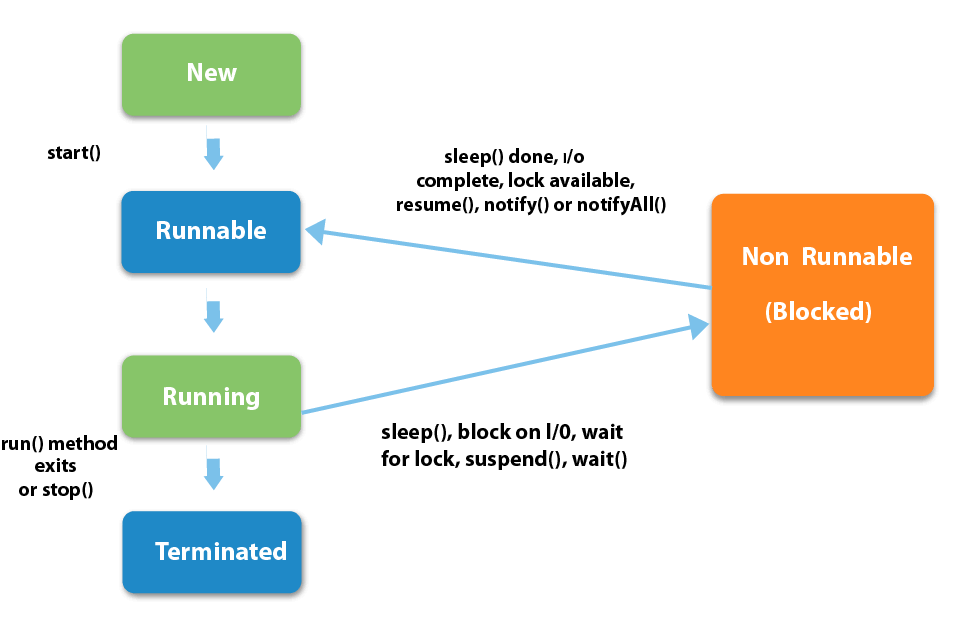
Un deadlock comme un livelock est un runtime bug qui fige l’exécution du programme. Dans le cas d’un deadlock, deux threads suspendent leur activité en attente l’un de l’autre. Ceci arrive lorsque le thread 1 attend que le thread 2 ait fini son action alors que le thread 2 attend que le thread 1 ait fini son activité. Il s’agit d’un cycle où chacun attend l’autre pour s’activer.

Typiquement cela peut arriver, lorsqu’un thread met à jour la ligne 1 d’une base puis la ligne 2 de cette même base alors que le thread 2 met à jour en même temps la ligne 2 puis la ligne 1. Les deux threads sont en attente l’un de l’autre.

* + - 1. LiveLock

Le livelock quant à lui correspond à la situation inverse où deux threads sont trop occupés à interagir ensemble. Ils ne peuvent plus accomplir le reste des tâches qui leur incombe. Cette situation peut arriver lorsque le thread 1 génère une réponse de thread 2 et thread 2 génère une réponse de thread 1. Pour imager un peu, on peut prendre l’exemple de deux personnes essayant de passer dans un corridor. Lorsque l’une fait un pas à droite l’autre fait aussitôt un pas à gauche. Elles s’empêchent mutuellement de passer.

* + - 1. le cycle de vie des threads en java



a. L’état new : lorsque la méthode start () est appelée.

b. L’état runnable est celui dans lequel il est en exécutant sa tâche.

c. L’état waiting : lorsqu’il attend qu’un autre thread termine sa tâche et qu’il lui signale de passer à nouveau dans l’état runnable. Les methodes **wait() et sleep()** plongent les threads dans cet état.

d. L’état dead intervient lorsque le thread a terminé.

Le code ci-dessous illustre l’utilisation des méthodes wait() et notify().

public class Processor {  
  
 public void produce() throws InterruptedException {  
 // synchronize est utilisé afin de verrouiller l’accès à un objet Processor  
 synchronized (this) {  
 System.*out*.println("t1 verrouille la ressource processor et attend en libérant la ressource par un wait(30000)...");  
 wait(30000);// le thread passe en état wait ce qui indique aux autres //threads qu’ils peuvent s’activer pendant ce temps  
 System.*out*.println("Again producer method après 30 secondes ou...");  
 }  
 }  
  
 public void consume() throws InterruptedException {  
  
 Thread.*sleep*(1000);//commence par attendre 1 seconde  
  
 // le premier thread à utiliser cette méthode verrouille l’objet  
 synchronized (this) {  
 System.*out*.println(" T1 a déverrouillé l’objet t2 est en train de consommé... ");  
 //après avoir effectué sa tâche il notifie à tous les threads //dormants qu’ils peuvent se réveiller  
 notifyAll();// commentez et décommentez cette fonction pour constater la différence de comportement  
 System.*out*.println(" t2 réveille t1 par un notifyAll() puis t2 sleep pendant 1 secondes");  
 Thread.*sleep*(1000);  
 System.*out*.println(" t2 s'etait endormi 1 seconde et finit sa tâche et déverrouille l'accès à l'objet" +  
 " donc t1 qui devait attendre 50 secondes peut immédiatement finir sa tâche.");  
 }  
 }  
}

La méthode pour tester le code à appeler dans le main().

public static void testWaitNotify() {  
  
 Processor processor = new Processor();  
  
 Thread t1 = new Thread(new Runnable() {  
 public void run() {  
 try {  
 processor.produce();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 Thread t2 = new Thread(new Runnable() {  
 public void run() {  
 try {  
 processor.consume();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 t1.start();  
 t2.start();  
  
 try {  
 t1.join();  
 t2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

* + - 1. La notion couverte par le mot clé « volatile ».

Le mot clé volatile s’applique à des variables afin de préciser que celles-ci seront stockées dans la mémoire RAM de l’ordinateur durant l’exécution et non dans la mémoire CACHE des unités de calcul du processeur.

Cette notion est importante, car les threads sont exécutés dans des unités de calcul du processeur. Si une variable est stockée à cet endroit, elle ne sera accessible que par le thread qui l’a initiée. Elle sera invisible des autres threads. Donc si cette ressource doit être partagée, il faut s’assurer qu’elle soit placée dans la mémoire RAM, qui elle, est accessible par tous les threads. Il faut cependant noter qu’en termes de performance, il y a un coût induit. L’accès à la mémoire RAM est moins rapide que l’accès à la mémoire CACHE.

* 1. Les nouveaux outils.
     1. Autre alternative de gestion des threads
        1. ReentrantLock

Offre les mêmes possibilités que les fonctionnalités synchronized / wait / notify. Cependant, ReentrantLock offre une granularité plus fine. La responsabilité de réveiller des threads est transmise à un autre thread. Attention, un thread qui aurait pris un verrou peut ainsi se retrouver locké à jamais.

import java.util.concurrent.locks.Condition;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class LockReentrantLock {  
  
 // On crée un nouveau verrou de type ReentrantLock  
 private Lock lock = new ReentrantLock();  
  
 private Condition condition = lock.newCondition();  
  
  
 public void produce() throws InterruptedException {  
 lock.lock();  
 System.*out*.println("Producer method... Attends d'être reactivé par un signal()");  
 condition.await();//comme la fonction wait(), await() fait passer le thread actuel dans un état wait et libère la ressource  
 System.*out*.println("Vient d'être réactivé par un signal provenant d'un autre thread...");  
 }  
  
 public void consume() throws InterruptedException {  
 lock.lock();  
 Thread.*sleep*(2000);// mets le thread en etat wait pendant 2 secondes.  
 System.*out*.println("Le premier thread est en etat wait t2 verrouille à son tour l'objet et sleep pendant 10 secondes ");  
 Thread.*sleep*(10000);  
 condition.signal();//t2 réactive t1  
 System.*out*.println("2 réactive t1 et déverrouille l'objet. ");  
 lock.unlock();//   
 }  
}

La méthode pour tester le code à appeler dans le main().

public static void testreentrantLock() {  
 final LockReentrantLock lockReentrantLock = new LockReentrantLock();  
  
 Thread t1 = new Thread(new Runnable() {  
 public void run() {  
 try {  
 lockReentrantLock.produce();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 Thread t2 = new Thread(new Runnable() {  
 public void run() {  
 try {  
 lockReentrantLock.consume();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 t1.start();  
 t2.start();  
  
}

* + - 1. ExecutorsService

La grande nouveauté vient de cette classe. En effet, elle encapsule complètement l’instanciation de nouveaux threads. Donc plus besoin de déclarer Thread t = new Thread() etc....

Maintenant, il y a trois manières plus efficaces d’instancier des threads.

1. ExecutorService executor = ExecutorService.newCacheThreadPool() ;

Cette classe crée autant de threads qu’il est nécessaire pour effectuer une tâche. La particularité de cette classe est de savoir si un thread est en attente. Si tel est le cas, il réutilise le thread. Par contre, s’il n’y pas de thread en attente d’un job alors cette classe génère un autre thread. C’est la manière la plus efficiente pour adapter la charge de travail en fonction des processeurs et processus disponibles. Il ne sera donc pas possible de créer plus de threads que nécessaire.

1. ExecutorService executor = ExecutorService.newFixedThreadPool(N)

Il est bien entendu possible de créer un nombre fini de threads avec la fonction newFixedThreadPool(N) qui prend un entier N en argument. On peut ainsi décider de créer 10 threads.

ExecutorService exservice = ExecutorService.newFixedThreadPool(10) ; Dans ce cas si tous les threads sont en activité, aucun autre thread supplémentaire ne pourra être crée.

1. ExecutorService executor = ExecutorService.newSingleThreadExecutor() ;

La dernière possibilité est de créer un thread unique.

Une fois l’option de votre choix établie, vous activez les threads soit grâce à la fonction execute() soit par la fonction submit().

La fonction executor.execute(new runnable class())  implémente l’interface runnable que l’on a déjà vue dans la première partie.

La fonction executor.submit(new callable class()) implémente l’interface callable qui renvoie un objet Future<T> générique en fonction des valeurs attendues en retour de l’exécution du thread (voir exemple).

En fin de tâche, on appelle la fonction executor.shutdown()  afin de fermer l’executor.

Voici un exemple de code qui implémente ce service.

public class ExceutorTest implements Runnable {  
  
 @Override  
 public void run()  
 {  
 // la fonction compte jusqu'a neuf mais peut être exécutée en même temps par plusieurs threads  
 for(int i=0;i<10;i++)  
 {  
 System.*out*.println(i);  
 try  
 {  
 Thread.*sleep*(1000);  
 }  
 catch (InterruptedException e)  
 {  
 e.printStackTrace();  
 }  
 }  
 }  
 }

La méthode pour tester le code à appeler dans le main().

public static void testExecutor() {  
 //On crée un pool de 5 threads qui lance 5 fois la fonction run().  
 //Faites varier le nombre de fonctions lancées et le nombres de threads pour constater les différents comportements  
  
 //ExecutorService executorService = Executors.newCachedThreadPool();  
 //ExecutorService executorService = Executors.newFixedThreadPool(7);  
 ExecutorService executorService = Executors.*newSingleThreadScheduledExecutor*();  
 for(int i=0;i<7;i++){  
 executorService.execute(new ExceutorTest());  
 }  
 //Il est indispensable de fermé l'executor sinon le programme ne s'arrête pas  
 executorService.shutdown();  
}

Essayer d’utiliser les trois méthodes et constatez les différences de comportement.

Laquelle préférez-vous ?

* + - 1. Interface Callable / Objet Future <T>

L’interface callable permet le retour de valeurs par l’intermédiaire d’un thread. Vous devez implémenter votre tâche non plus à l’intérieur d’une fonction run() mais dans le corps d’une fonction call() qui renvoie une valeur. Si vous souhaitez par exemple récupérer un String votre fonction call() return String. Ce dernier sera encapsulé dans un objet Future<String> et alors accessible par un get(). L’exemple ci-dessous sera plus parlant.

Implémentation d’un exemple

import java.util.ArrayList;  
import java.util.Date;  
import java.util.List;  
import java.util.concurrent.Callable;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.Future;  
  
// L'interface callable est implémentée avec un Type String car la fonction call retourne un String  
public class CallableTest implements Callable<String>{  
  
 private int id;  
  
 public CallableTest (int id){  
 this.id = id;  
 }  
  
 @Override  
 public String call() throws Exception {  
 Thread.*sleep*(1000);  
 // le retour de la fonction call est bien un String  
 return "Id: "+this.id;  
 }  
}

La méthode pour tester le code à appeler dans le main().

public static void testCallableInterface() {  
  
 ExecutorService executorService = Executors.*newFixedThreadPool*(2);  
 // l'objet List en java est une interface. l'objet ArrayList implémente cette interface  
 List<Future<String>> list = new ArrayList<>();  
  
 for(int i=0;i<5;i++)  
 {  
 //la fonction submit appelle l'objet qui implémente l'interface Callable  
 Future<String> future = executorService.submit(new CallableTest (i+1));  
 list.add(future);  
 }  
// il faut ensuite récupérer le String qui est encapsulé dans l'objet Future  
 for(Future<String> future : list){  
 try  
 {  
 // On utilise la fonction get()  
 System.*out*.println("le resultat est bien sous la forme d'un String : " + future.get());  
 }catch(Exception e){  
 e.printStackTrace();  
 }  
 }  
 //ne pas oublier de fermer l'executor  
 executorService.shutdown();  
 }

* + - 1. Sémaphores

Les Sémaphores peuvent être considérés comme des variables ou des types abstraits qui tiennent un compte précis des ressources disponibles. Si vous devez accéder à un nombre limité de ressources, les sémaphores vous indiquent combien sont disponibles. C’est d’une grande utilité pour la programmation concurrente. Cependant, leur limite réside dans le fait qu’ils n’indiquent pas quelles sont les ressources disponibles (dit combien mais pas lesquelles).

Il existe deux types de Sémaphores.

1. Les Sémaphores de type compteur qui peuvent accepter N valeurs.
2. Les Sémaphores binaires qui ne peuvent prendre comme valeur que 0 et 1.

Les Sémaphores maintiennent un set de « permis/autorisations » et utilisent deux principales méthodes :

- acquire() ; -> si des permis sont disponibles, cette fonction décrémente de 1 le compteur et un verrouille une ressource.

- release() ;-> incrémente le compteur de 1  et libère une ressource.

New Semaphore(int permis, boolean fair) ;

Le paramètre fair mis à true permet de s’assurer qu’aucun thread en attente ne sera oublié car

c’est alors le thread qui a attendu le plus longtemps qui sera activé.

Si, par contre, le paramètre fair est placé à false, il est possible qu’un thread n’effectue jamais sa tâche (il peut être oublié).

* + - 1. Mutex

Les Mutex sont des objets qui ressemblent aux Sémaphores de type binaire. Ils ne peuvent prendre que 0 ou 1 comme valeur. La différence tient au fait que les mutex connaissent le thread auquel ils sont rattachés. Il peuvent augmenter la priorité du thread auquel il sont liés si le besoin est.

Illustration de l’utilisation de Sémaphores.

import java.util.concurrent.Semaphore;  
  
public enum SemaphoreTest {  
  
 *INSTANCE*;  
  
 // On crée sous la forme d'un singleton un semaphore qui obtient 5 permis  
 private Semaphore semaphore = new Semaphore(5, true);  
  
 //la fonction simule un téléchargement  
 public void downloadData() {  
  
 try  
 {  
 semaphore.acquire();  
 download();  
  
 } catch (InterruptedException e)  
 {  
 e.printStackTrace();  
 } finally  
 {// Toujours s'assurer de libérer le semaphore  
 semaphore.release();  
 }  
 }  
  
 private void download()  
 {  
 System.*out*.println("Downloading data from the web...");  
 try  
 {  
 Thread.*sleep*(2000);  
 }  
 catch (InterruptedException e)  
 {  
 e.printStackTrace();  
 }  
 }  
}

La méthode pour tester le code à appeler dans le main().

public static void testSemaphore()  
 {  
//lance 5 threads alors que la fonction doit être appellée 12 fois  
 ExecutorService executorService = Executors.*newFixedThreadPool*(5);  
  
 for(int i=0;i<12;i++)  
 {  
 //on appelle 12 fois la fonction downloadData  
 executorService.execute(new Runnable() {  
 public void run() {  
 SemaphoreTest.*INSTANCE*.downloadData();  
 }  
 });  
 }  
 executorService.shutdown();  
 }

* + 1. Les Collections pour nous aider à gérer le multithreading.
       1. Concurrent maps

Est un dictionnaire optimisé pour le multithreading. Donc nul besoin d’implémenter une stratégie de synchronisation et de verrou.

import java.util.concurrent.ConcurrentMap;  
  
public class FirstWorker3 implements Runnable  
{  
  
 private ConcurrentMap<String, Integer> map;  
  
 public FirstWorker3(ConcurrentMap<String, Integer> map) {  
 this.map = map;  
 }  
  
 @Override  
 public void run() {  
 try {  
 map.put("B",1);  
 map.put("H",2);  
 map.put("F",3);  
 Thread.*sleep*(1000);  
 map.put("A",4);  
 Thread.*sleep*(1000);  
 map.put("E",5);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

import java.util.concurrent.ConcurrentHashMap;  
import java.util.concurrent.ConcurrentMap;  
  
public class SecondWorker3 implements Runnable {  
  
 private ConcurrentMap<String, Integer> map;  
  
 public SecondWorker3(ConcurrentMap<String, Integer> map) {  
 this.map = map;  
 }  
  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(5000);  
 System.*out*.println(map.get("A"));  
 Thread.*sleep*(1000);  
 System.*out*.println(map.get("E"));  
 Thread.*sleep*(1000);  
 System.*out*.println(map.get("H"));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

La fonction à implémenter dans le main().

public static void concurrentMaptest()  
{  
 //aucun besoin de synchronisation ni d'utilisation de verrous. Toutes ces fonctions sont déjà implémentées dans cette collection  
 ConcurrentMap<String, Integer> map = new ConcurrentHashMap<>();  
 FirstWorker3 firstWorker = new FirstWorker3(map);  
 SecondWorker3 secondWorker = new SecondWorker3(map);  
 new Thread(firstWorker).start();  
 new Thread(secondWorker).start();  
}

* + - 1. Blocking queue

Comme son nom l’indique, il s’agit d’une collection de type Queue. Ce conteneur est traditionnellement utilisé dans les stratégies « producteur / consommateur ». Il est optimisé pour attendre que la queue se remplisse avant de pouvoir retirer des éléments.

import java.util.concurrent.BlockingQueue;  
  
public class FirstWorker implements Runnable {  
  
 private BlockingQueue<String> blockingQueue;  
  
 public FirstWorker(BlockingQueue<String> blockingQueue) {  
 this.blockingQueue = blockingQueue;  
 }  
  
 @Override  
 public void run() {  
 try {  
 // Ce thread a pour but d'inserer de maniere concurrente des éléments dans une collection  
 // La fonction put tente d'ajouter un élément.  
 blockingQueue.put("A");  
 Thread.*sleep*(10000);  
 blockingQueue.put("B");  
 Thread.*sleep*(10000);  
 blockingQueue.put("C");  
  
 blockingQueue.put("D");  
 Thread.*sleep*(10000);  
  
 blockingQueue.put("E");  
 blockingQueue.put("F");  
 blockingQueue.put("G");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

import java.util.concurrent.BlockingQueue;  
  
public class SecondWorker implements Runnable {  
//on Instancie une collection de type BlockingQueue  
 private BlockingQueue<String> blockingQueue;  
 public SecondWorker(BlockingQueue<String> blockingQueue) {  
 this.blockingQueue = blockingQueue;  
 }  
  
 @Override  
 public void run() {  
 try {  
 //le thread a pour tâche de retirer de façon concurrente des éléments de la collection  
  
 // La fonction take retire le dernier élément de la queue. Donc celui qui est le plus ancien.  
  
 System.*out*.println(blockingQueue.take());  
 System.*out*.println(blockingQueue.take());  
 System.*out*.println(blockingQueue.take());  
 System.*out*.println(blockingQueue.take());  
 System.*out*.println(blockingQueue.take());  
 System.*out*.println(blockingQueue.take());  
 System.*out*.println(blockingQueue.take());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

Fonction à implémenter dans le main() ;

public static void blockingQueueTest()  
{  
 // est une interface de type blockingQueue . l'ArrayBlockingQueue est la collection qui implémente cette interface  
 BlockingQueue<String> queue = new ArrayBlockingQueue<>(10);  
 FirstWorker firstWorker = new FirstWorker(queue);  
 SecondWorker secondWorker = new SecondWorker(queue);  
 new Thread(firstWorker).start();  
 new Thread(secondWorker).start();  
}

* + 1. S’entrainer : exercice de la Bibliothèque.

Créer une classe de constantes qui contiendra en static le nombre d’étudiants (5), le nombre de livres(7).

import java.util.concurrent.locks.ReentrantLock;  
  
public class Books  
{  
 private int id;  
 private final ReentrantLock lock;  
  
  
 public Books(int id) {  
 // On crée un reentrant lock afin de transférer la responsabilité au thread emprunteur de libérer la ressource  
 this.id = id;  
 this.lock = new ReentrantLock();  
 }  
  
 public void read(Student student) throws InterruptedException  
 {  
 lock.lock();  
 System.*out*.println(student + " lit " + this );  
 Thread.*sleep*(3000);  
 System.*out*.println(student + " rend " + this );  
 lock.unlock();  
 }  
  
 public String toString() {  
 return "Book : " + id;  
 }  
  
}

import java.util.HashMap;  
import java.util.Random;  
import java.util.Vector;  
  
public class Student implements Runnable {  
  
 private int id;  
 //Chaque étudiant possède une liste de livres  
 private HashMap<Integer,Books> livresNonLus;  
 private Books[] listeALire;  
  
 public Student(int id,Books[] seslivresAlire)  
 {  
 this.id = id;  
 this.listeALire = seslivresAlire ;  
 this.livresNonLus = new HashMap<>(this.listeALire.length);  
 this.initLivreNonlus();  
  
 }  
  
 private void initLivreNonlus()  
 {  
 for (int i = 0; i < this.listeALire.length ; i++)  
 {  
 this.livresNonLus.put(i,new Books(i));  
 }  
 }  
  
 @Override  
 public void run() {  
 //On tire au hasard l'id du livre à lire  
 Random random = new Random();  
  
 while (!this.livresNonLus.isEmpty())  
 {  
 int bookId = random.nextInt(Constant.*NUMBER\_OF\_BOOKS*);  
 if(this.livresNonLus.get(bookId)!=null)  
 {  
 try  
 {  
 this.listeALire[bookId].read(this);  
 this.livresNonLus.remove(bookId);  
 System.*out*.print(this.toString() + " doit encore lire : ");  
 for ( Books b : this.livresNonLus.values())  
 {  
 System.*out*.print(b.toString()+" , ");  
 }  
 System.*out*.println();  
 }  
 catch (InterruptedException e)  
 {  
 e.printStackTrace();  
 }  
 }  
 }  
 System.*out*.println();  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"+ this + " A lu tous ses livres");  
 }  
 public String toString() {  
 return "Student " + id;  
 }  
}

Classe de constantes utilisée dans l’exercice.

public class Constant {  
  
 private Constant(){  
 }  
  
 public static final int *NUMBER\_OF\_STUDENTS* = 5;  
 public static final int *NUMBER\_OF\_BOOKS* = 7;  
}

La fonction à utiliser dans le main() ;

public static void bibliotheque() {  
  
 Student[] students=null;  
 Books [] listeLivre=null;  
 ExecutorService executor = Executors.*newFixedThreadPool*(Constant.*NUMBER\_OF\_STUDENTS*);  
  
 try  
 {  
 listeLivre = new Books [Constant.*NUMBER\_OF\_BOOKS*];  
 students = new Student[Constant.*NUMBER\_OF\_STUDENTS*];  
 for(int j= 0; j < Constant.*NUMBER\_OF\_BOOKS*;j++)  
 {  
 listeLivre[j] = new Books(j);  
 }  
 for(int i=0; i< Constant.*NUMBER\_OF\_STUDENTS*; i++)  
 {  
 students[i]=new Student(i,listeLivre);  
 executor.execute(students[i]);  
 }  
 }  
 catch(Exception e)  
 {  
 e.printStackTrace();  
 executor.shutdown();  
 }  
 finally  
 {  
 executor.shutdown();  
 }  
 }

* + 1. Pour aller plus loin : Les Frameworks de parallélisation
       1. Fork-join aide à la construction de code parallelisable sur une machine multicoeurs.