Université de Carthage

Institute National des Sciences Appliqués et de Technologie

Année Universitaire 2018-2019

COMPTE RENDU
Applications Réparties
TP3

Réalisée par :

SNOUSSI Anis

BARGHOUDA Mohamed Lamine

LES THREADS

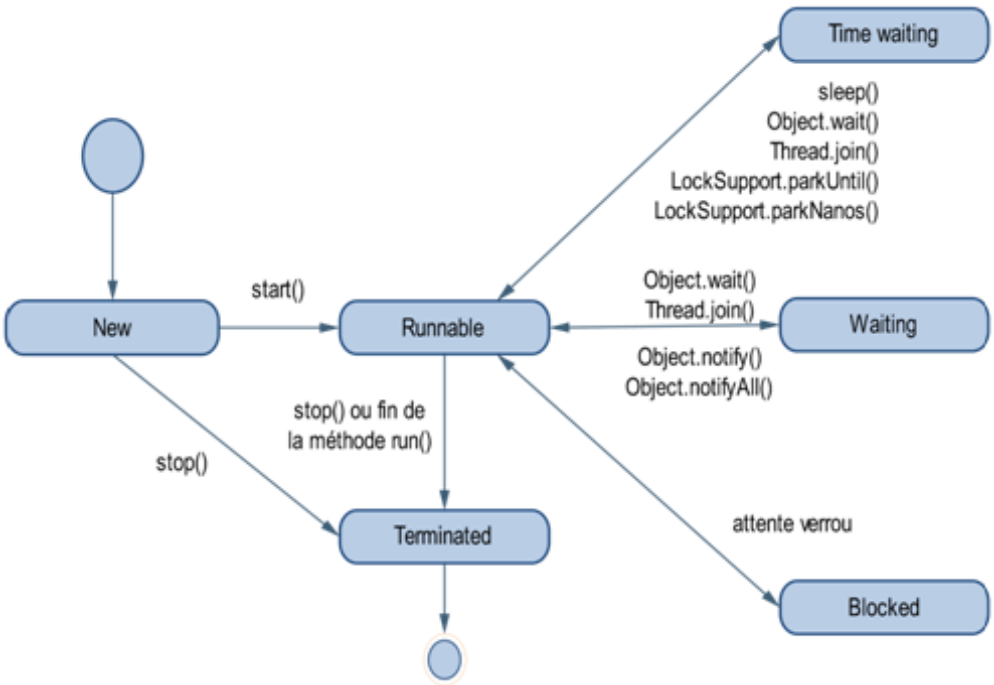
Introduction

Un thread est une unité d'exécution faisant partie d'un programme. Cette unité fonctionne de façon autonome et parallèlement à d'autres threads. Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leurs exécutions "simultanées".

Selon le système d'exploitation et l'implémentation de la JVM, les threads peuvent être gérés de deux manières :

- Correspondre à un thread natif du système
- Correspondre à un thread géré par la machine virtuelle

Dans les deux cas, cela n'a pas d'impact sur le code qui reste le même.



Les threads – LES BASES

Nous souhaitons écrire un programme qui va lancer 10 Threads où chaque thread va annoncer son numéro deux fois d'affilé (deux « println » distincts) sur la sortie standard

On va commencer par créer une classe fille qui hérite de la classe Thread et implémente sa méthode run().Il suffit alors de créer une instance de la classe fille et d'invoquer sa méthode start() pour démarrer le thread

```
public class MyThread extends Thread {
 public int number;
 public MyThread(int n) {
  this.number = n;
 }
 public void run() {
  System.out.print(number + " ");
  System.out.print(number + " ");
 }

public static void main(String[] args) {
  MyThread[] th = new MyThread[10];
  for (int i = 0; i < 10; i++) {
   th[i] = new MyThread(i + 1);
   th[i].start();
  }

 }
}
```

MyThread.java

Résultat de l'exécution :

```
E:\Med Lamine\GL2\2ème semestre\applications repartis\TP3\
7 7 5 5 1 1 3 3 6 6 4 4 2 2 9 9 8 10 8 10
```

PRODUCTEURS/CONSOMATEURS

Dans cet exemple de producteurs/consommateurs partageant un buffer borné à une seule place, il s'agit pour les producteurs et les consommateurs de se synchroniser sur le buffer. Un producteur doit attendre que le buffer soit vide pour mettre la valeur produite dans le buffer. Le producteur attend que le buffer soit plein pour consommer la valeur.

```
public class Producteur extends Thread {
    private Buffer buf;
    private int identite;
    public Producteur(Buffer c, int n) {
        buf = c; this.identite = n;
    }
public void run() {
   for (int i = 0; i < 100; i++) {
       buf.mettre(i);
       System.out.println("Producteur #" + this.identite
                          + " met : " + i);
       try { sleep((int)(Math.random() * 100));}
       catch (InterruptedException e) { }
       }
    }
}
```

Producteur.java

```
public class Consommateur extends Thread {
    private Buffer buf;
    private int identite;
     public Consommateur(Buffer c, int n) {
        buf = c;
        this.identite = n;
    }
    public void run() {
        int val = 0;
        for (int i = 0; i < 10; i++) {
            val = buf.prendre();
            System.out.println("Consommateur #" +
                  this.identite + " prend: " + val);
        }
    }
}
```

Consommateur.java

```
public class Buffer {
	private int valeur;
	private boolean available = false;
	public synchronized int prendre() {
		while (available == false) {
			try {
				wait();
			}
			catch(InterruptedException e) {}
		}
		available = false;
		notifyAll();
		return valeur;
	}
	public synchronized void mettre(int val) {
		while (available == true) {
			try {
				wait();
			} catch(InterruptedException e) {}
		}
		valeur = val;
		available = true;
		notifyAll();
	}
}
```

Buffer.java

```
public class Main {
	public static void main(String[] args) {
		Buffer c = new Buffer();
		Producteur p1 = new Producteur(c, 1);
		Producteur p2 = new Producteur(c, 2);
		Producteur p3 = new Producteur(c, 3);

		Consommateur c1 = new Consommateur(c, 1);
		Consommateur c2 = new Consommateur(c, 2);
		Consommateur c3 = new Consommateur(c, 3);
		p1.start();
		p2.start();
		p3.start();
		c1.start();
		c2.start();
		c3.start();
	}
}
```

Main.java

Résultat de l'exécution :

```
E:\Med Lamine\GL2\2ème semestre\applications repartis\TP3\
Consommateur #3 prend: 0
Producteur #2 met : 0
Consommateur #2 prend: 0
Producteur #1 met : 0
Consommateur #1 prend: 0
Producteur #3 met : 0
Producteur #2 met : 1
Consommateur #3 prend: 1
Producteur #3 met : 1
Consommateur #1 prend: 1
Consommateur #3 prend: 1
Producteur #1 met : 1
```

BARRIERES DE SYNCHRONISATION

Une barrière est un objet synchronisant n threads. Elle dispose d'une fonction attendre qui va être bloquante pour n – 1 appels et qui va débloquer tout le monde à n appels.
On va Ecrire une classe Barrière qui peut être utilisée comme une barrière de synchronisation réutilisable.

- Constructeur Barriere(int nb) : définit une barrière pour nb threads,
- Méthode attendre(void) : bloque jusqu'à ce que nb threads aient appelé cette méthode.

```
public class Barriere{
    private int nombreTotal;
    private int nombreActif;

    public Barriere(int nombreTotal){
        this.nombreTotal=nombreTotal;
    }

    public synchronized void attendre(){
        nombreActif++;
        try
        {
            if(nombreActif==nombreTotal) {
                notifyAll();
                nombreActif=0;
            }
            else wait();
        }
            catch (InterruptedException e) {e.printStackTrace();}
    }
}
```

Barriere.java

```
public class Joueur extends Thread {
        private int id;
        private Barriere b;
        public Joueur(int id, Barriere b) {
                this.id = id;
                this.b = b;
        }
        public void run() {
                //echauffement
                for (int i = 0; i < 1000; i++)
                System.out.println("Barriere Atteinte\n");
                b.attendre();
                //commencer la course
                System.out.println(id + " a termine!");
        }
}
```

Joueur.java

```
public class Main {
        public static void main(String[] args) {
                Barriere b = new Barriere(50);
                for (int i = 0; i < 60; i++) {
                        Joueur j = new Joueur(i, b);
                        j.start();
                }
        }
}
```

Main.java

Résultat de l'exécution :

Début

```
E:\Med Lamine\GL2\2ème semestre\applications repartis\TP3\(
19: 0
19: 1
15: 0
43: 0
3: 0
8: 0
40: 0
44: 0
4: 0
```

FIN

```
46 a termine!
6 a termine!
26 a termine!
30 a termine!
28 a termine!
45 a termine!
42 a termine!
47 a termine!
```

GESTION DE COMPTE BANCAIRE

Rappel : Méthodes de bases utilisées dans JAVA pour créer un thread

- Créer une classe fille qui hérite de la classe Thread. Il suffit alors de créer une instance de la classe fille et d'invoquer sa méthode start() pour démarrer le thread
- Créer une classe qui implémente l'interface Runnable. Pour lancer l'exécution, il faut créer un nouveau Thread en lui passant en paramètre une instance de la classe et invoquer sa méthode start()

Simulation correcte du fonctionnement d'un compte bancaire

On veut maintenant simuler correctement le fonctionnement pour qu'un compte bancaire qui soit accessible par plusieurs personne (Mari et son épouse).
Pour cela on précède par synchroniser tous les méthodes de la classe Compte représentant un risque d'accès concurrent et incohérence de données.
Cela est effectué en ajoutant le mot clé « synchronised » au signature des méthodes concernés mais aussi au sortie de la sortie standard (System.out)

```
public class Compte {
    private int valeur ;
    public Compte (int val ) {
        valeur = val ;
    }
    public synchronized int solde () {
        return valeur ;
    }
    public synchronized void depot (int somme ) {
        if ( somme > 0) valeur += somme ;
    }
    public synchronized boolean retirer (int somme ) throws InterruptedException {
        if ( somme > 0 && somme <= valeur ) {
                Thread.currentThread(). sleep (50) ;
                valeur -= somme ;
                Thread.currentThread(). sleep (50) ;
                return true ;
        }
        return false ;
    }
}
```

Compte.java

```
public class Banque implements Runnable {
 Compte nom;
 Banque(Compte n) {
  nom = n;
 }
 public synchronized void liquide(int montant) throws InterruptedException {
  if (nom.retirer(montant)) {
   Thread.currentThread().sleep(50);
   donne(montant);
   Thread.currentThread().sleep(50);
  } else {
   System.out.println(Thread.currentThread().getName() + ": Tu peux pas prendre " +
montant + " euros .");
  }
  imprimeRecu();
  Thread.currentThread().sleep(50);
 }
 public void donne(int montant) throws InterruptedException {
  System.out.println(Thread.currentThread().getName() + ": Voici vos " + montant + "
euros .");
 }
 public void imprimeRecu() {
  if (nom.solde() > 0)
   System.out.println(Thread.currentThread().getName() + ": Il vous reste " +
nom.solde() + " euros.");
  else
   System.out.println(Thread.currentThread().getName() + ": Vous etes fauches !");
 }
 public void run() {
  try {
   for (int i = 1; i < 10; i++) {
    synchronized(System.out) {
     liquide(100 * i);
     Thread.currentThread().sleep(100 + 10 * i);
    }
   }
  } catch (InterruptedException e) {
   Return e.print;
  }
 }
 public static void main(String[] args) {
  Compte Commun = new Compte(1000);
  Runnable Mari = new Banque(Commun);
  Runnable Femme = new Banque(Commun);
  Thread tMari = new Thread(Mari);  tMari.setName(" Conseiller Mari");
  Thread tFemme = new Thread(Femme); tFemme.setName(" Conseiller Femme ");
  tMari.start();
  tFemme.start();
 }
}
```

Banque.java

Résultat de l'exécution :

```
E:\Med Lamine\GL2\2ème semestre\applications repartis\TP3\
 Conseiller Femme : Voici vos 100 euros .
 Conseiller Femme : Il vous reste 900 euros.
 Conseiller Femme : Voici vos 200 euros .
 Conseiller Femme : Il vous reste 700 euros.
 Conseiller Femme : Voici vos 300 euros .
 Conseiller Femme : Il vous reste 400 euros.
 Conseiller Femme : Voici vos 400 euros .
 Conseiller Femme : Vous etes fauches !
 Conseiller Femme : Tu peux pas prendre 500 euros .
 Conseiller Femme : Vous etes fauches !
 Conseiller Mari: Tu peux pas prendre 100 euros .
 Conseiller Mari: Vous etes fauches !
 Conseiller Femme : Tu peux pas prendre 600 euros .
 Conseiller Femme : Vous etes fauches !
```

QUESTION + : UTILISATION DE LA BARRIERE PREDEFINIE « CyclicBarrier »

Cette fois on va utiliser la barriere prédéfinie « **CyclicBarrier** » pour la synchronisation
En fait, c'est une aide à la synchronisation qui permet à un ensemble de threads de s'attendre mutuellement pour atteindre un point de barrière commun.
CyclicBarriers sont utiles dans les programmes impliquant un groupe de threads de taille fixe qui doivent parfois s'attendre les uns les autres. La barrière est appelée cyclique car elle peut être réutilisée une fois les threads en attente libérés.
CyclicBarrier prend en charge une commande optionnelle Runnable qui est exécutée une fois par point barrière, après l'arrivée du dernier thread de la partie, mais avant la libération des threads. Cette action de barrière est utile pour mettre à jour l'état partagé avant que l'une des parties ne poursuive.

java.util.concurrent

Class CyclicBarrier

java.lang.Object
└ java.util.concurrent.CyclicBarrier

Constructor Summary
CyclicBarrier(int parties) Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and does not perform a predefined action when the barrier is tripped.
CyclicBarrier(int parties, Runnable barrierAction) Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and which will execute the given barrier action when the barrier is tripped, performed by the last thread entering the barrier.

Method Summary	
int	await() Waits until all parties have invoked await on this barrier.
int	await(long timeout, TimeUnit unit) Waits until all parties have invoked await on this barrier, or the specified waiting time elapses.
int	getNumberWaiting() Returns the number of parties currently waiting at the barrier.
int	getParties() Returns the number of parties required to trip this barrier.
boolean	isBroken() Queries if this barrier is in a broken state.
void	reset() Resets the barrier to its initial state.

```
import java.util.concurrent.*;

public class Joueur extends Thread {
        private int id;
        CyclicBarrier b;
        public Joueur(int id, CyclicBarrier b) {
                this.id = id;
                this.b = b;
        }
        public void run() {
                //echauffement
                for (int i = 0; i < 1000; i++){
                  System.out.println(id + ": " + i);
         }
                try {
           b.await();
          } catch (InterruptedException ex) {
            return;
          } catch (BrokenBarrierException ex) {
            return;
          }
          //commencer la course
          System.out.println(id + " a termine!");
        }
}
```

Joueur.java

```
import java.util.concurrent.*;
public class Main {
        public static void main(String[] args) {
                CyclicBarrier b = new CyclicBarrier(50);
                for (int i = 0; i < 50; i++) {
                        Joueur j = new Joueur(i, b);
                        j.start();
                }
        }
}
```

Main.java