Thread en java

Threads

- threads: plusieurs activités qui coexistent et partagent des données
  - exemples:
    - pendant un chargement long faire autre chose
    - serveur
    - coopérer
    - multicoeur
  - problème de l'accès aux ressources partagées
    - verrous
    - moniteur
    - synchronisation

Note

- A bas niveau
  - processus versus thread
    - processus (typiquement unix): « code » dans son propre espace mémoire, ses propres ressources + IPC pour se synchroniser et échanger avec les autres
    - thread (lightweight processes) « code » avec des resources partagées
    - chaque processus a au moins un thread, les threads partagent les ressources du processus dans lequel elles s 'exécutent
    - chaque application contient au moins un thread et un « main thread »

ProcessBuilder…

```
public class ProcessDemo {
  public static void main(String args[])
      throws InterruptedException,IOException
  {
    ProcessBuilder builder = new ProcessBuilder("ls");
    //l'environnement
    Map<String, String> environ = builder.environment();
    // créer le process
    final Process process = builder.start();
    //récuperer l'inputstream du process crée
    InputStream is = process.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
    System.out.println("Program terminé");
  }
}
```

Principes de base

- extension de la classe *Thread*
  - méthode run est le code qui sera exécuté.
  - la création d'un objet dont la super-classe est `Thread` crée le thread (mais ne la démarre pas)
  - la méthode `start` démarre la thread (et retourne immédiatement)
  - la méthode `join` permet d'attendre la fin du thread
  - les exécutions des threads sont asynchrones et concurrentes

Exemple

```
class ThreadAffiche extends Thread {
    private String mot;
    private int delay;
    public ThreadAffiche(String w, int duree) {
        mot = w;
        delay = duree;
    }
    public void run() {
        try {
            for (;;) {
                System.out.println(
                Thread.sleep(delay);
            }
        } catch (InterruptedException e) {
        }
    }
}
```

```
public static void main(String[] args) {
        new ThreadAffiche("PING",
    1000).start();
        new ThreadAffiche("PONG",
    3000).start();
        new ThreadAffiche("splash!",
    1500).start();
  }
```

Alternative: Runnable

- Une autre solution:
  - créer une classe qui implémente l'interface `Runnable` (cette interface contient la méthode `run`)
  - créer un `Thread` à partir du constructeur `Thread` avec un `Runnable` comme argument.

Exemple

```
class RunnableAffiche implements
    Runnable{
    private String mot;
    private int delay;
    public RunnableAffiche(String
    w,int duree){
        mot=w;
        delay=duree;
    }
    public void run(){
        try{
            for(;;){

    System.out.println(mot)
                Thread.slee
            }
        }catch(InterruptedException
    e){
        }
    }
}
```

```
public static void main(String[] args) {
    Runnable ping=new
    RunnableAffiche("PING", 1000);
     Runnable pong=new
    RunnableAffiche("PONG", 500);
     new Thread(ping).start();
     new Thread(pong).start();
}
```

Thread

- création de l'objet : new ThreadAffiche
- démarrage du thread: méthode start
- un thread est lancé dans une autre thread (main thread) par la méthode start()
- exécution concurrente

attendre la fin du thread
join()

```
Thread t = new Thread() {
    public void run() {
        try {
            Thread.sleep(6000);
        } catch (InterruptedException ex) {
        }
        System.out.println("thread terminée");
    }
};
t.start();
try {
    t.join();
} catch (InterruptedException ex) {
}
System.out.println("FINI");
```

interruption

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class ThreadExemple extends Thread {

    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                Logger.getLogger(ThreadExemple.class.getName()).log(Level.SEVERE, null, ex);
                System.out.println("Thread.sleep interrompue");
            }
            System.out.format("TH nom %s et ID %d i= %d%n", getName(), getId(), i);
        }
    }

    public static void main(String[] args) {
        Thread t1 = new ThreadExemple();
        t1.start();
        Thread t2 = new ThreadExemple();
        t2.start();
        try {
            Thread.sleep(500);
            t1.interrupt();
            Thread.sleep(1000);
            t2.interrupt();

        } catch (InterruptedException ex) {
            Logger.getLogger(ThreadExemple.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Partage

Partage…

- les threads s'exécutent concurremment et peuvent accéder concurremment aux objets dans leur portée: partage
  - quel est l'effet d'une modification d'une variable partagée entre plusieurs thread?

```
public class InVisible {

    public static boolean fait = false;
    public static int n;

    public static class Lecteur extends Thread {

        public void run() {
            while (!fait);
 //                 Thread.yield();
            System.out.println(n);

        }
    }

    public static void main(String[] args) {
        new Lecteur().start();
        n = 150;
        fait = true;

    }
}
```

Ce programme peut donner des résultats différents:

- ne pas terminer
- afficher 150
- afficher 0
- …

Lecture-écriture

- « atomicité »: tout se passe comme si l'opération est d'un seul tenant sans interruption
- lire ou écrire une « valeur simple » est indivisible
- lire ou écrire une valeur correspondant à plusieurs mots mémoires n'est pas forcément indivisible
- optimisation de code: certaines opérations peuvent être inversées (ou supprimées): volatile

problème de la cohérence de la mémoire

En java

- **atomicité des lectures écritures:**
  - lecture et écriture pour les variables des types primitifs sauf long et double
  - lecture et écriture pour les variables déclarées comme volatile

- **happens before** (Spécifié dans le chapitre 17 du Java Language Specification)
  - Each action in a thread happens-before every action in that thread that comes later in the program's order.
  - An unlock (synchronized block or method exit) of a monitor happens-before every subsequent lock (synchronized block or method entry) of that same monitor. And because the happens-before relation is transitive, all actions of a thread prior to unlocking happen-before all actions subsequent to any thread locking that monitor.
  - A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.
  - A call to start on a thread happens-before any action in the started thread.
  - All actions in a thread happen-before any other thread successfully returns from a join on that thread.
- (les objets **immuables** (immutable) -qui, une fois créés ne peuvent être modifiés permettent aussi d'assurer la cohérence mémoire)

Partage…

- atomicité des lectures/écritures sur les variables,
- atomicité des instructions
- il faut contrôler l'accès:
  - thread un lit une variable (R1) puis modifie cette variable (W1)
  - thread deux lit la même variable (R2) puis la modifie (W2)
  - R1-R2-W2-W1
  - R1-W1-R2-W2 donne des résultats différents!

Exemple

```
class X{
    int val;
}
class Concur extends Thread{
    X x;
    int i;
    String nom;
    public Concur(String st, X x){
        nom=st;
        this.x=x;
    }
    public void run(){
        i=x.val;
        System.out.println("thread:"+nom+"
  valeur x="+i);
        try{
             Thread.sleep(10);
        }catch(Exception e){}
        x.val=i+1;
        System.out.println("thread:"+nom+"
  valeur x="+x.val);
    }
}
```

```
public static void main(String[]
    args) {
        X x=new X();
        Thread un=new
    Concur("un",x);
        Thread deux=new
    Concur("deux",x);
        un.start(); deux.start();
        try{
            un.join();
            deux.join();
        }catch (InterruptedException
    e){}
     System.out.println("X="+x.val);
    }
```

thread:deux valeur x=0
thread:un valeur x=0
thread:un valeur x=1
thread:deux valeur x=1
X=1

Deuxième exemple

```
class Y{
    int val=0;
    public int increment(){
        int tmp=val;
        tmp++;
        try{
            Thread.currentThread().sleep(100);
        }catch(Exception e){}
        val=tmp;
        return(tmp);
    }
    int getVal(){return val;}
}
class Concur1 extends Thread{
    Y y;
    String nom;
    public Concur1(String st, Y y){
        nom=st;
        this.y=y;
    }
    public void run(){
        System.out.println("thread:"+nom+"
      valeur="+y.increment());
    }
}
```

```
  public static void
    main(String[] args) {
        Y y=new Y();
        Thread un=new
    Concur1("un",y);
        Thread deux=new
    Concur1("deux",y);
        un.start(); deux.start();
        try{
            un.join();
            deux.join();
        }catch
     (InterruptedException e){}

     System.out.println("Y="+y.ge
     tVal());
    }
-----------
```

- thread:un valeur=1
- thread:deux valeur=1
- Y=1

compteur

```
class Counter1 {
    int n;
    int getAdd() {
        return n++;
    }
}
```

```
class Counter {

    int n;

    int getAdd() {
        int temp = n;
        Thread.yield();
        n = temp + 1;
        return temp;
    }
}
```

```
new Thread() {
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(c.getAdd());
        }
    }
}.start();
new Thread() {
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(c.getAdd());
        }
    }
}.start();
System.out.println(c.getAdd());
```

Verrous

- à chaque objet est associé un verrou (lock)
- synchronized(expr) {instructions}
  - expr doit s'évaluer comme une référence à un objet
  - verrou sur cet objet pour la durée de l'exécution de instructions
- déclarer les méthodes comme synchronized: le thread obtient un verrou sur l'objet et le relâche quand la méthode se termine.
- (les verrous sont ré-entrants)

synchronized

```
class Concur extends Thread{
    X x;
    int i;
    String nom;
    public Concur(String st, X x){
        nom=st;
        this.x=x;
    }
    public void run(){
        synchronized(x){
            i=x.val;
            System.out.println("thread:"+nom+"
     valeur x="+i);
            try{
                Thread.sleep(10);
            }catch(Exception e){}
            x.val=i+1;
            System.out.println("thread:"+nom+"
     valeur x="+x.val);
        }
    }
}
```

```
class X{
    int val;
}
```

```
class Y{
    int val=0;
    public synchronized int
     increment(){
        int tmp=val;
        tmp++;
        try{

     Thread.currentThread().sleep(
     100);
        }catch(Exception e){}
        val=tmp;
        return(tmp);
    }
    int getVal(){return val;}
}
```

Méthode synchronisée

```
class Y{
    int val=0;
    public synchronized int
     increment(){
        int tmp=val;
        tmp++;
        try{

     Thread.currentThread().sleep(100);
        }catch(Exception e){}
        val=tmp;
        return(tmp);
    }
    int getVal(){return val;}
}
```

```
class Counter3 {
    private int n;
    synchronized int getAdd() {
        return n++;
    }
}
....
Counter3 c = new Counter3();
 new Thread() {
          public void run() {
             for (int i = 0; i < 100; i++) {
                System.out.println(c.getAdd());
             }
          }
       }.start();
       new Thread() {
          public void run() {
             for (int i = 0; i < 100; i++) {
                System.out.println(c.getAdd());
             }
          }
       }.start();
       System.out.println(c.getAdd());
```

Mais…

- la synchronisation par des verrous peut entraîner un blocage:
  - la thread un (XA) pose un verrou sur l'objet A et (YB) demande un verrou sur l'objet B
  - la thread deux (XB) pose un verrou sur l'objet B et (YA) demande un verrou sur l'objet A
  - si XA –XB : ni YA ni YB ne peuvent être satisfaites -> blocage
- (pour une méthode synchronisée, le verrou concerne l'objet globalement et pas seulement la méthode)

Exemple

```
class Dead{
    Dead partenaire;
    String nom;
    public Dead(String st){
        nom=st;
    }
    public synchronized void f(){
      try{
            Thread.currentThread().sleep(1000);
       }catch(Exception e){}

     System.out.println(Thread.currentThread().getName()
     +    «  de "+ nom+".f() invoque "+
     partenaire.nom+".g()");
        partenaire.g();    }
    public synchronized void g(){

     System.out.println(Thread.currentThread().getName()
     +
                 " de "+ nom+".g()");
    }
    public void setPartenaire(Dead d){
        partenaire=d;
    }
}
```

```
 final Dead un=new Dead("un");
final Dead deux= new Dead("deux");
un.setPartenaire(deux);
deux.setPartenaire(un);
new Thread(new Runnable(){public void run(){un.f();}
},"T1").start();
new Thread(new Runnable(){public void run(){deux.f();}
},"T2").start();
```

- T1 de un.f() invoque deux.g()
- T2 de deux.f() invoque un.g()

Synchronisation

- attendre qu'une condition soit réalisée pour poursuivre.

```
class Partage{
    private volatile boolean fait=false;
    Integer data=0;
    void ecrire() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {

        }
        data=153; fait=true;
    }
    void lire(){
        while (!fait){}
        System.out.println(data);
    }
}
```

```
final Partage partage = new Partage();

new Thread(new Runnable(){
    public void run(){partage.ecrire();
    }
 }).start();
 new Thread(new Runnable(){
    public void run(){partage.lire();
    }
}).start();
```

attendre une condition

- lecteur/écrivain: pour lire le lecteur doit attendre qu'il y ait quelque chose à lire
- (l'écrivain doit aussi attendre que le lecteur ait lu la précédente valeur)
- pourquoi la solution précédente n'est pas bonne?

Synchronisation…

- wait, notifyAll notify
  - attendre une condition / notifier le changement de condition:

```
synchronized void fairesurcondition(){
      while(!condition){
           wait();}
      faire ce qu'il faut quand la condition est vraie


--------------------
synchronized void changercondition(){
 … changer quelque chose concernant la condition
 notifyAll(); // ou notify()
}
```

Class Object

void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
void	**wait**() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

```
class Buffer {
    private String message;
    private boolean empty = true;
    public synchronized String get() {
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        empty = true;
        notifyAll();
        return message;
    }

    public synchronized void put(String message) {
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        empty = false;
        this.message = message;
        notifyAll();
    }
}
```

Le buffer

producteur

```
class Producer implements Runnable {
    private Buffer buf;
    public Producer(Buffer buf) {
        this.buf = buf;
    }

    public void run() {
        String data[] = {
            "..."
        };
        Random random = new Random();

        for (int i = 0;
             i < data.length;
             i++) {
            buf.put(data[i]);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
        buf.put("FINI");
    }
}
```

consommateur

```
class Consumer implements Runnable {

    private Buffer buf;
    private String nom;

    public Consumer(Buffer buf, String n) {
        this.buf = buf;
        this.nom = n;
    }

    public void run() {
        Random random = new Random();
        for (String message = buf.get();
                !message.equals("FINI");
                message = buf.get()) {
            System.out.format(
"%s MESSAGE RECU : %s%n",nom, message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Exemple (file: Cellule)

```
public class Cellule<E>{
    private Cellule<E> suivant;
    private E element;
    public Cellule(E val) {
        this.element=val;
    }
    public Cellule(E val, Cellule suivant){
        this.element=val;
        this.suivant=suivant;
}
public E getElement(){
    return element;
}
public void setElement(E v){
    element=v;
}
public Cellule<E> getSuivant(){
    return suivant;
}
public void setSuivant(Cellule<E> s){
    this.suivant=s;
}
```

File synchronisées

```
class File<E>{
    protected Cellule<E> tete, queue;
    private int taille=0;

    public synchronized void enfiler(E item){
        Cellule<E> c=new Cellule<E>(item);
        if (queue==null)
            tete=c;
        else{
            queue.setSuivant(c);
        }
        c.setSuivant(null);
        queue = c;
        notifyAll();
}
```

File (suite)

```
public synchronized E defiler() throws InterruptedException{
        while (tete == null)
            wait();
        Cellule<E> tmp=tete;
        tete=tete.getSuivant();
        if (tete == null) queue=null;
        return tmp.getElement();
}
```

mémoire locale des threads…

```
1  public class ThreadID {
2    private static volatile int nextID = 0;
3    private static class ThreadLocalID extends ThreadLocal<Integer> {
4      protected synchronized Integer initialValue() {
5        return nextID++;
6      }
7    }
8    private static ThreadLocalID threadID = new ThreadLocalID();
9    public static int get() {
10     return threadID.get();
11   }
12   public static void set(int index) {
13     threadID.set(index);
14 }
```

ThreadLocal

Pour finir

```
class WorkerThread implements Runnable {
    private String command;
    public WorkerThread(String s){
        this.command=s;
    }
     public void run() {
        System.out.println(Thread.currentThread().getName()+" Start. Command = "+command);
        processCommand();
        System.out.println(Thread.currentThread().getName()+" End.");
    }
    private void processCommand() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public String toString(){
        return this.command;
    }
}
class SimpleThreadPool {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i);
            executor.execute(worker);
          }
        executor.shutdown();
        while (!executor.isTerminated()) {
        }
        System.out.println("Finished all threads");
    }
}
```

```
pool-1-thread-3 Start. Command = 2
pool-1-thread-1 Start. Command = 0
pool-1-thread-2 Start. Command = 1
pool-1-thread-4 Start. Command = 3
pool-1-thread-5 Start. Command = 4
pool-1-thread-3 End.
pool-1-thread-1 End.
pool-1-thread-2 End.
pool-1-thread-2 Start. Command = 5
pool-1-thread-1 Start. Command = 6
pool-1-thread-4 End.
pool-1-thread-4 Start. Command = 8
pool-1-thread-5 End.
pool-1-thread-5 Start. Command = 9
pool-1-thread-3 Start. Command = 7
pool-1-thread-1 End.
pool-1-thread-4 End.
pool-1-thread-3 End.
pool-1-thread-5 End.
pool-1-thread-2 End.
Finished all threads
```

```
import java.util.Arrays;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
class Main {
  public static void main(String[] args) {
    ForkJoinPool fjPool = new ForkJoinPool();
    int[] a = new int[3333344];
    for (int i = 0; i < a.length; i++) {
      int k = (int) (Math.random() * 22222);
      a[i] = k;
    }
    ForkJoinQuicksortTask forkJoinQuicksortTask =
            new ForkJoinQuicksortTask(a, 0, a.length - 1);
    long start = System.nanoTime();
    fjPool.invoke(forkJoinQuicksortTask);
    System.out.println("Time: " + (System.nanoTime() - start));
  }
}
class ForkJoinQuicksortTask extends RecursiveAction {
  int[] a;
  int left;
  int right;
  public ForkJoinQuicksortTask(int[] a) {
    this(a, 0, a.length - 1);
  }
  public ForkJoinQuicksortTask(int[] a, int left, int right) {
    this.a = a;
    this.left = left;
    this.right = right;
  }
void swap(int[] a, int p, int r) {
    int t = a[p];
    a[p] = a[r];
    a[r] = t;
  }
```

```
@Override
  protected void compute() {
    if (serialThresholdMet()) {
      Arrays.sort(a, left, right + 1);
    } else {
      int pivotIndex = partition(a, left, right);
      ForkJoinQuicksortTask t1 = new ForkJoinQuicksortTask(a,
left,
          pivotIndex - 1);
      ForkJoinQuicksortTask t2 = new ForkJoinQuicksortTask(a,
pivotIndex + 1,
          right);
      t1.fork();
      t2.compute();
      t1.join();
    }
  }

  int partition(int[] a, int p, int r) {
    int i = p - 1;
    int x = a[r];
    for (int j = p; j < r; j++) {
      if (a[j] < x) {
        i++;
        swap(a, i, j);
      }
    }
    i++;
    swap(a, i, r);
  return i;
  }

  private boolean serialThresholdMet() {
    return right - left < 100000;
  }
}
```