Thread & quelques patrons pour la concurrence

Cnam Paris
jean-michel Douin, douin au cnam point fr
version commencée en Juin 2006

.....

Sommaire

- **Les bases**
  - java.lang.Thread
    - start(), run(), join(),...
  - java.lang.Object
    - wait(), notify(),...
  - le pattern Singleton revisité
  - java.lang.ThreadGroup
  - java.lang.ThreadLocal
  - java.util.Collections

- **les travaux de Doug Léa**
  - Concurrent Programming
  - java.util.concurrent (1.5)

- **Patrons**
  - Critical Section, Guarded Suspension, Balking, Scheduler, Read/Write Lock, Producer-Consumer, Two-Phase Termination

Bibliographie utilisée

- Design Patterns, catalogue de modèles de conception réutilisables
  de Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides [Gof95]
    International thomson publishing France

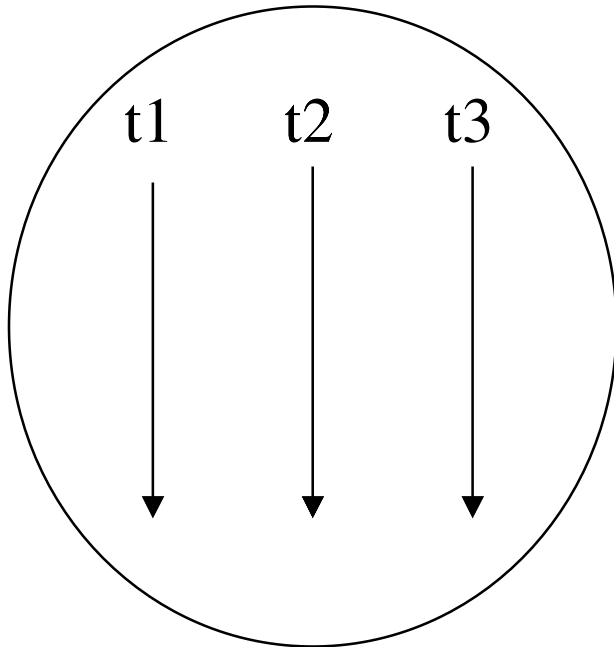
Doug Léa,

Mark Grand

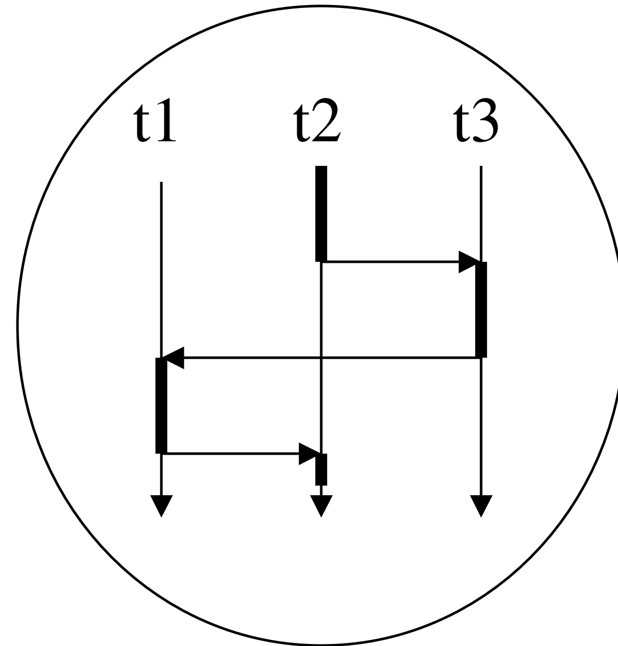
Exécutions concurrentes

- **Les *Threads* Pourquoi ?**
- **Entrées sorties non bloquantes**
- **Alarmes, Réveil, Déclenchement périodique**
- **Tâches indépendantes**
- **Algorithmes parallèles**
- **Modélisation d ’activités parallèles**
- **Méthodologies**
- **…**

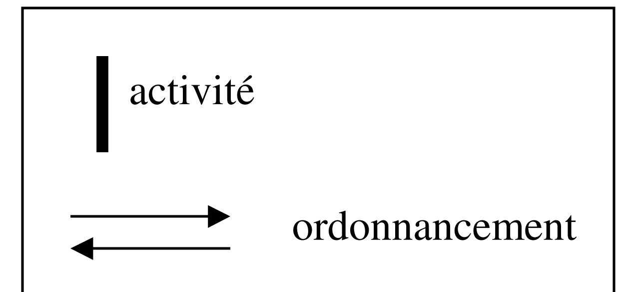
Contexte : Quasi-parallèle



vue logique



vue du processeur



- Plusieurs *Threads* mais un seul processeur abordé dans ce support

La classe Thread

- **La classe Thread est prédéfinie (package java.lang)**

- **Syntaxe : Création d 'une nouvelle instance (comme d 'habitude)**
  - **Thread unThread = *new Thread(); …***
    - ***(un Thread pour processus allégé...)***

- **« Exécution » du processus**
  - **unThread.start();**
    - **éligibilité de UnThread**

ensuite l 'ordonnanceur choisit unThread exécute la méthode run()

  - **unThread.run();**
    - **instructions de unThread**

Exemple

```
public class T extends Thread {
   public void run(){
     while(true){
         System.out.println("dans "  + this + ".run");
       }
   }
}


public class Exemple {

  public static void main(String[] args) {
      T t1 = new T(); T t2 = new T(); T t3 = new T();
      t1.start(); t2.start(); t3.start();
      while(true){
        System.out.println("dans Exemple.main");
      }
  }
}
```

Remarques sur l 'exemple

- Un ***Thread*** est déjà associé à la méthode ***main*** pour une application Java (ou au navigateur dans le cas d 'applettes). (Ce ***Thread*** peut donc en engendrer d 'autres…)

trace d 'exécution

dans Exemple.main
dans Thread[Thread-4,5,main].run
dans Thread[Thread-2,5,main].run
dans Exemple.main
dans Thread[Thread-4,5,main].run
dans Thread[Thread-2,5,main].run
dans Exemple.main
dans Thread[Thread-4,5,main].run
dans Thread[Thread-3,5,main].run
dans Thread[Thread-2,5,main].run
dans Thread[Thread-4,5,main].run
dans Thread[Thread-3,5,main].run
dans Thread[Thread-2,5,main].run
dans Thread[Thread-4,5,main].run
dans Thread[Thread-3,5,main].run
dans Thread[Thread-2,5,main].run
dans Exemple.main
dans Thread[Thread-3,5,main].run

- **un premier constat à l'exécution : *il semble que l'on ait un* Ordonnanceur de type tourniquet, ici sous windows**

La classe java.lang.Thread

- **Quelques méthodes**

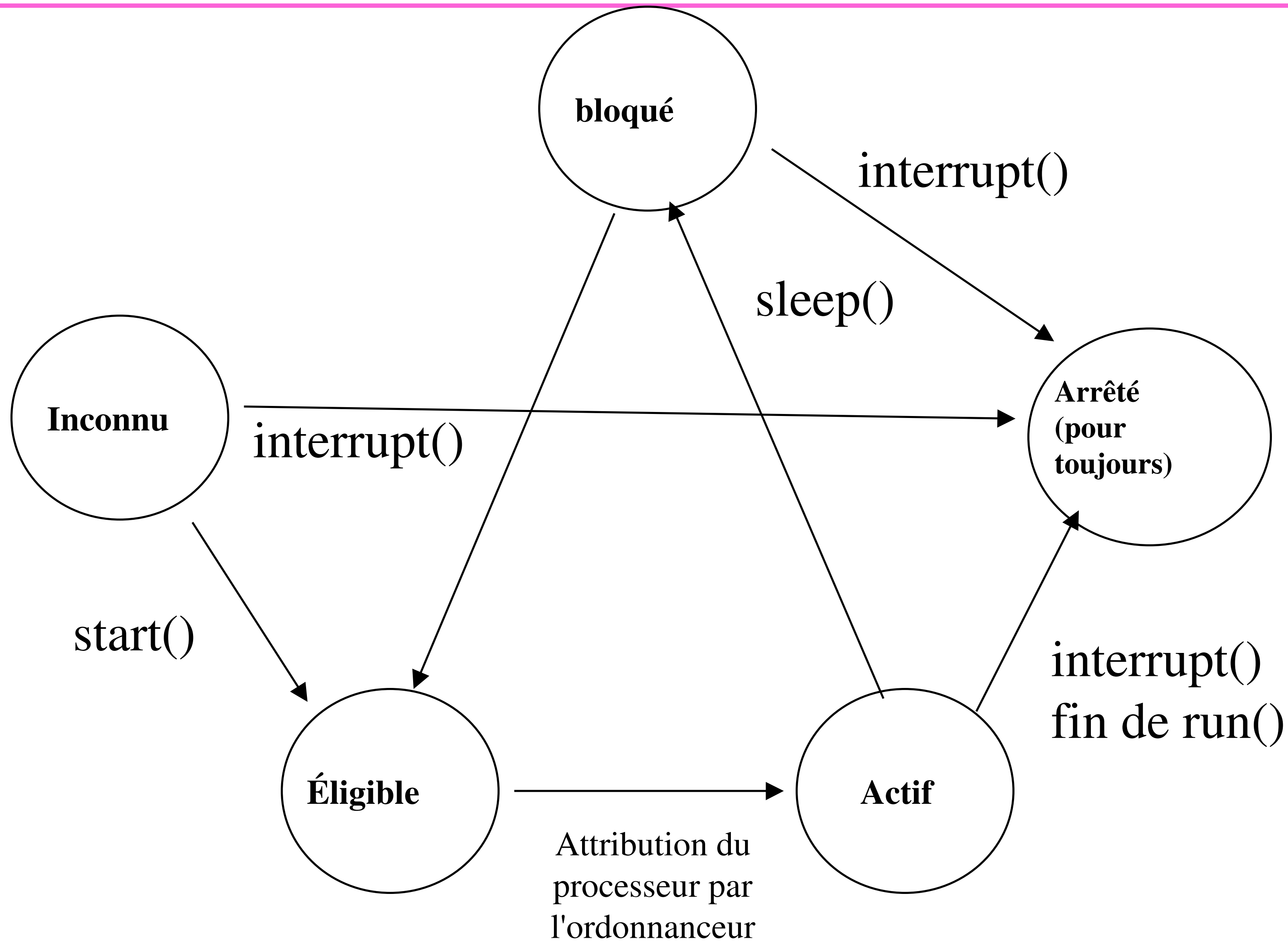
Les constructeurs publics

- ***Thread();***
- ***Thread(Runnable target);***
- ***…***

Les méthodes publiques

- ***void start();*** *// éligibilité*
- ***void run();*** *// instructions du Thread*
- ***void interrupt();*** *// arrêt programmé*
- ***boolean interrupted();***
- *void stop(); // deprecated*
- ***static void sleep(long ms);*** *// arrêt pendant un certain temps*
- ***static native Thread currentThread();***

États d 'un Thread



- **États en 1ère approche**

Exemple initial revisité

```
public class T extends Thread {
   public void run(){
     while(!this.interrupted()){
         System.out.println("dans "  + this + ".run");
       }
   }
}


public class Exemple {

  public static void main(String[] args) throws InterruptedException{
      T t1 = new T(); T t2 = new T(); T t3 = new T();
      t1.interrupt();
      t2.start(); t2.interrupt();
      t3.start();
      System.out.println("dans Exemple.main");
      Thread.sleep(2000);
      t3.interrupt();
  }
}
```

Le constructeur Thread (Runnable r)

La syntaxe habituelle avec les interfaces

```
public interface Runnable{
      public abstract void run();
}
```

```
public class T implements Runnable {

  public void run(){
    ....
  }
}
```

```
public class Exemple{
  public static void main(String[] args){
    Thread t1 = new Thread( new T());
    t1.start();
```

-

Remarques sur l'arrêt d 'un Thread

- Sur le retour de la méthode *run()* le Thread s 'arrête
- Si un autre Thread invoque la méthode *interrupt()* ou this.interrupt()
- Si n'importe quel Thread invoque *System.exit()* ou *Runtime.exit(),* tous les Threads s 'arrêtent
- Si la méthode *run()* lève une exception le Thread se termine (avec libération des ressources)

- *destroy() et stop() ne sont plus utilisés, non sûr*

Une autre méthode de java.lang.Thread

- **attente active de la fin d 'un Thread**
  - **join() et join(délai)**

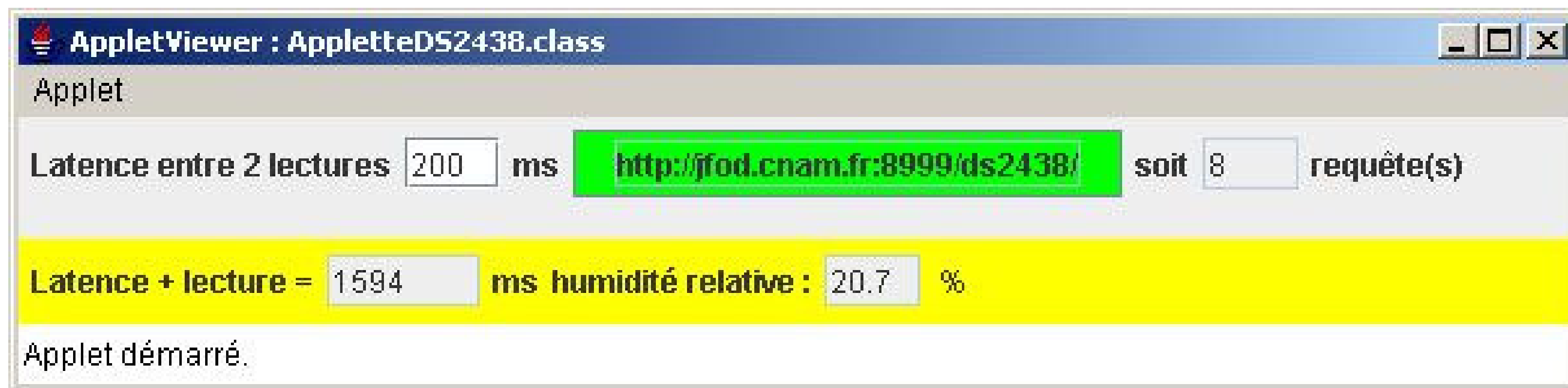
```
public class T implements Runnable {
  private Thread local;
  …

  public void attendreLaFin() throws InterruptedException{
     local.join();
   }

public void attendreLaFin(int délai) throws InterruptedException{
     local.join(délai);
   }
```

Un autre exemple : applette et requête cyclique

```
public class AppletteDS2438 extends Applet
                         implements ActionListener, Runnable {

 public void init(){ local = new Thread(this);t.start();}
 public void stop(){ local.interrupt();}

 public void run() {
  while(!local.interrupted()){
   long top = System.currentTimeMillis();
   try{
    Thread.sleep(dureeDeLatence);

    BufferedReader in = new BufferedReader(
          new InputStreamReader( new URL(urlRequest ).openStream()));
    String str = is.readLine();
    // traiter le retour
    ………
```

L 'applette est en ligne

- **appletviewer http://jfod.cnam.fr:8999/AppletteDS2438.html**



- **le source : http://jfod.cnam.fr:8999/AppletteDS2438.java**
  - **Attention aux droits et permissions depuis un navigateur**

Exemple : Une requête HTTP sans attendre…

```
public class RequeteHTTP implements Runnable{
  public RequeteHTTP(){ this.local = new Thread(this);this.start();…

  public void run(){
   try{
     URL urlConnection = new URL(url);                                    // aller
     URLConnection connection = urlConnection.openConnection();
     …

     BufferedReader in = new BufferedReader(                             // retour
                 new InputStreamReader(connection.getInputStream()));
     String inputLine = in.readLine();
     while(inputLine != null){
       result.append(inputLine);
       inputLine = in.readLine();
     }
     in.close();
   }catch(Exception e){
     …}}}
```

Attendre le résultat

Le résultat est lié à la fin du Thread

```
public String result(){
    try{
      this.local.join();
    }catch(InterruptedException ie){
      ie.printStackTrace();
    }
    return result.toString();
  }
}
```

Critiques

- **Toujours possibles**
  - **« jungle » de Thread**
  - **Parfois difficile à mettre en œuvre**
    - **Création, synchronisation, ressources …**
  - **Très facile d'engendrer des erreurs**
  - **Abstraire l'utilisateur des « détails » , …**
    - **Éviter l'emploi des méthodes start, interrupt, sleep, etc …**

Un style possible d 'écriture...

A chaque nouvelle instance, un Thread est créé

```
public class T implements Runnable {
  private Thread local;
  public T(){
    local = new Thread(this);
    local.start();
  }

  public void run(){
    if(local== Thread.currentThread ())
      while(!local.interrupted()){
        System.out.println("dans " + this + ".run");
      }
  }
}
```

Un style possible d 'écriture (2)...

Un paramètre transmis lors de la création de l 'instance

```
public class T implements Runnable {
  private Thread local; private String nom;
  public T(String nom){
    this.nom = nom;
    this.local = new Thread(this);
    this.local.start();
  }

  public void run(){
    if(local== Thread.currentThread ())
      while(!local.interrupted()){
        System.out.println("dans " + this.nom + ".run");
      } }}
```

L'interface Executor

- **Paquetage java.util.concurrent j2se 1.5, détaillé par la suite**

- **public interface Executor{ void execute(Runnable command);}**

  - Executor executor = new ThreadExecutor();
  - Executor.execute(new Runnable(){ …});
  - Executor.execute(new Runnable(){ …});
  - Executor.execute(new Runnable(){ …});

```
import java.util.concurrent.Executor;
public class ThreadExecutor implements Executor{

  public void execute(Runnable r){
    new Thread(r).start();
  }
}
```

La classe java.util.Timer

- **Une première abstraction (réussie)**

```
Timer timer= new Timer(true);
timer.schedule(new TimerTask(){
                    public void run(){
                      System.out.print(".");
                    }},
                    0L,       // départ imminent
                    1000L);   // période

Thread.sleep(30000L);
}
```

Priorité et ordonnancement

- **Pré-emptif, le processus de plus forte priorité devrait avoir le processeur**

- **Arnold et Gosling96 : *When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to implement mutual exclusion.***

- **Priorité de 1 à 10 (par défaut 5). Un thread adopte la priorité de son processus créateur (*setPriority(int p)* permet de changer celle-ci)**

- **Ordonnancement dépendant des plate-formes (……………...)**
  - **Tourniquet facultatif pour les processus de même priorité,**
  - **Le choix du processus actif parmi les éligibles de même priorité est arbitraire,**
  - **La sémantique de la méthode *yield()* n 'est pas définie, certaines plate-formes peuvent l 'ignorer (en général les plate-formes implantant un tourniquet)**

Et le ramasse-miettes ?

Accès au ressources/synchronisation

Moniteur de Hoare 1974

Moniteur en Java : usage du mot-clé *synchronized*

// extrait de java.lang.Object;

// Attentes

final void wait() throws InterruptedException

final native void wait(long timeout) throws InterruptedException

…

// Notifications

final native void notify()

final native void notifyAll()

Moniteur de Hoare Padiou 1990

- **Un moniteur définit une forme de module partagé dans un environnement parallèle**
- **Il encapsule des données et définit les procédures d 'accès à ces données**
- **Le moniteur assure un accès en exclusion mutuelle aux données qu 'il encapsule**

- **Synchronisation par les variables conditions :**

  Abstraction évitant la gestion explicite de files d 'attente de processus bloqués

- **L 'opération *wait* bloque l 'appelant**
- **L 'opération *notify* débloque éventuellement un processus bloqué à la suite d 'une opération *wait* sur le même moniteur**
  - **Variables conditions de Hoare (le processus signaleur est suspendu au profit du processus signalé); Variables conditions de Mesa(ordonnancement moins stricte)**

Le mot-clé synchronized

- **Construction *synchronized***
- ***synchronized(obj){***
- ***// ici le code atomique sur l 'objet obj***
- ***}***

- ***class C {***
- ***synchronized void p(){ ......}***
- ***}***
- ***////// ou //////***
- ***class C {***
- ***void p(){***
- ***synchronized (this){......}***
- ***}}***

Une ressource en exclusion mutuelle

```
public class Ressource {
   private double valeur;

   public synchronized double lire(){
     return valeur;
   }

  public synchronized void ecrire(double v){
    valeur = v;
   }
}
```

Il est garantit : qu'un seul Thread accède à la ressource, ici un champ d'instance

Accès « protégé » aux variables de classe

- **private static double valeur;**

```
synchronized( valeur){

}
```

Attention aux risques d'interblocage …

Moniteur de Hoare 1974

- *Le moniteur assure un accès en exclusion mutuelle aux données qu 'il encapsule*

- *avec le bloc synchronized*

- **Synchronisation ? par les variables conditions :**

  Abstraction évitant la gestion explicite de files d 'attente de processus bloqués

  wait et notify dans un bloc *synchronized*

Synchronisation : lire si c'est écrit

```
public class Ressource<E> {
   private E valeur;
   private boolean valeurEcrite = false;                    // la variable condition

   public synchronized E  lire(){
     while(!valeurEcrite)
        try{
           this.wait();                                     // attente d'une notification
        }catch(InterruptedException ie){ throw new RuntimeException();}
     valeurEcrite = false;
     return valeur;
   }

  public synchronized void ecrire(E elt){
    valeur = v; valeurEcrite = true;   this.notify();       // notification
  }}
```

Discussion & questions

- **2 files d'attente à chaque « Object »**
  - **Liée au synchronized**
  - **Liée au wait**
- **Pourquoi while(!valeurEcrite) ?**
  - **Au lieu de if((!valeurEcrite) wait()**
- **Quel est le Thread bloqué et ensuite sélectionné lors d'un notify ?**
- **Encore une fois, seraient-ce des mécanismes de « trop » bas-niveau ?**
  - **À suivre…**

lire avec un délai de garde

```
public synchronized E  lire(long délai){
  if(délai <= 0)
    throw new IllegalArgumentException(" le d\u00e9lai doit \u00eatre > 0");

     while(!valeurEcrite)
       try{
         long topDepart = System.currentTimeMillis();
         this.wait(délai);                          // attente d'une notification avec un délai
         long durée = System.currentTimeMillis()-topDepart;
         if(durée>=délai)
           throw new RuntimeException("d\u00e9lai d\u00e9pass\u00e9");
       }catch(InterruptedException ie){ throw new RuntimeException();}

        valeurEcrite = false;
     return valeur;
   }
```