
NSY102

Conception de logiciels Intranet

Concurrence

Cnam Paris
jean-michel Douin, douin au cnam point fr
26 Février 2013

Notes de cours

Sommaire

- **Les bases**

- java.lang.Thread
 - start(), run(), join(),...
- java.lang.Object
 - wait(), notify(),...
- java.util.Collection, et java.util.Collections
 - Vector, PriorityQueue, SynchronousQueue ...

- **java.util.concurrent**

- Par petites touches, un autre support

- **3 Patrons**

- **Singleton**, adapté à la programmation concurrente
- Découpler l'acquisition du traitement
 - **Chain of Responsibility** et **Interceptor**

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, <http://g.oswego.edu/dl/>

Mark Grand

http://www.mindspring.com/~mgrand/pattern_synopses.htm#Concurrency%20Patterns

<http://www-128.ibm.com/developerworks/edu/j-dw-java-concur-i.html>

au cnam <http://jfod.cnam.fr/NSY102/lectures/j-concur-ltr.pdf>

<https://developers.sun.com/learning/javaoneonline/2004/corej2se/TS-1358.pdf>

Pré-requis

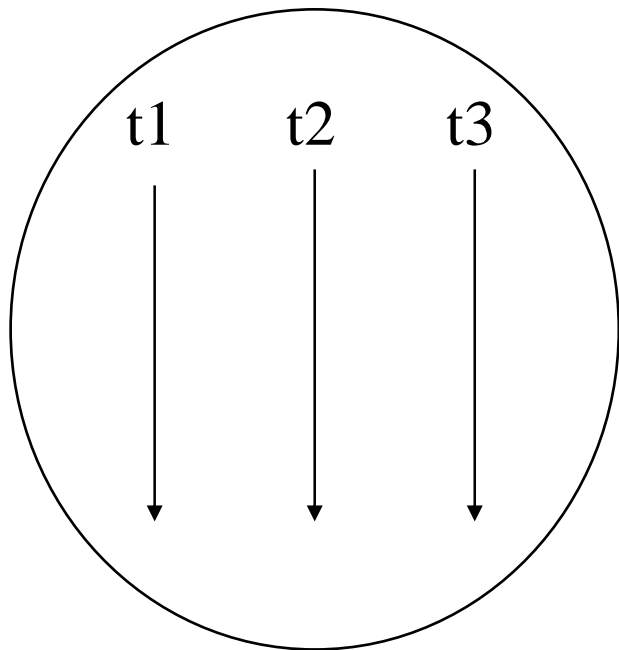
- **Notions de**
 - **Interface**
 - **Abstract superclass**
 - **Patron déléation**

 - **Exclusion mutuelle, interruptions, ...**

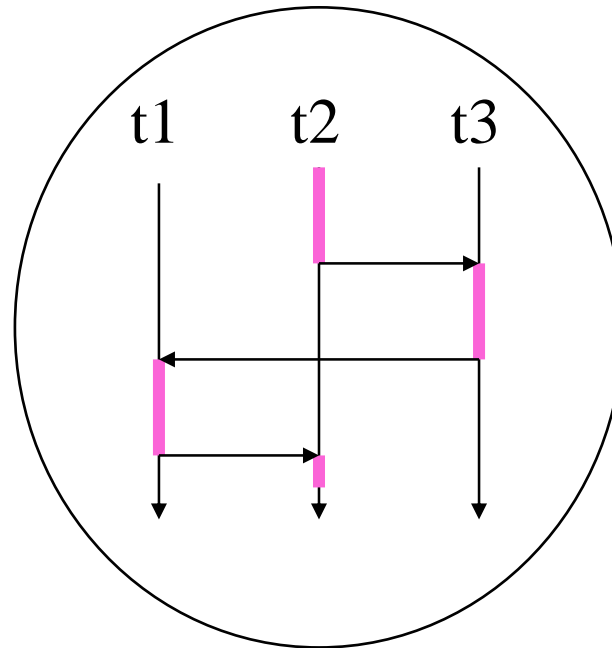
Exécutions concurrentes

- **Les *Threads* Pourquoi ?**
- **Entrées sorties non bloquantes**
 - Lecture/écriture de fichiers, réseaux clients comme serveurs, ...
- **Alarmes, Réveil,**
 - Déclenchement périodique
- **Tâches indépendantes**
- **Algorithmes parallèles**
- ...

Contexte : Quasi-parallèle

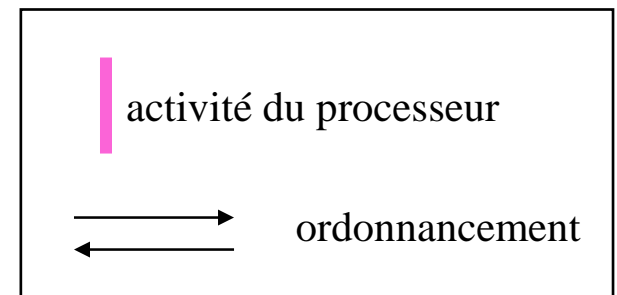


vue logique



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 T (); ...** (*T extends Thread*)
 - *(un(e) Thread pour processus allégé...)*

Puis

1) Éligibilité du processus

- **unThread.start();**

2) Exécution du processus

L'ordonnanceur choisit unThread exécute la méthode run()

- *Soit l'appel par l'ordonnanceur (et uniquement) de la méthode run (ici unThread.run();)*

Un exemple sans fin

```
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");  
        }  
    }  
}
```

Où sont les Threads ?

Un thread peut en créer d'autres

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...)

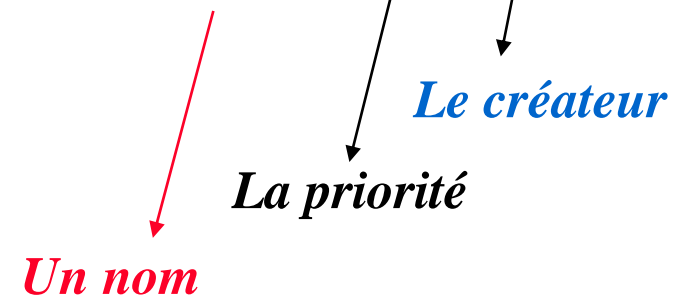
trace d'exécution

```

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
```

toString() :

[Thread-4, 5, main].



- 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**

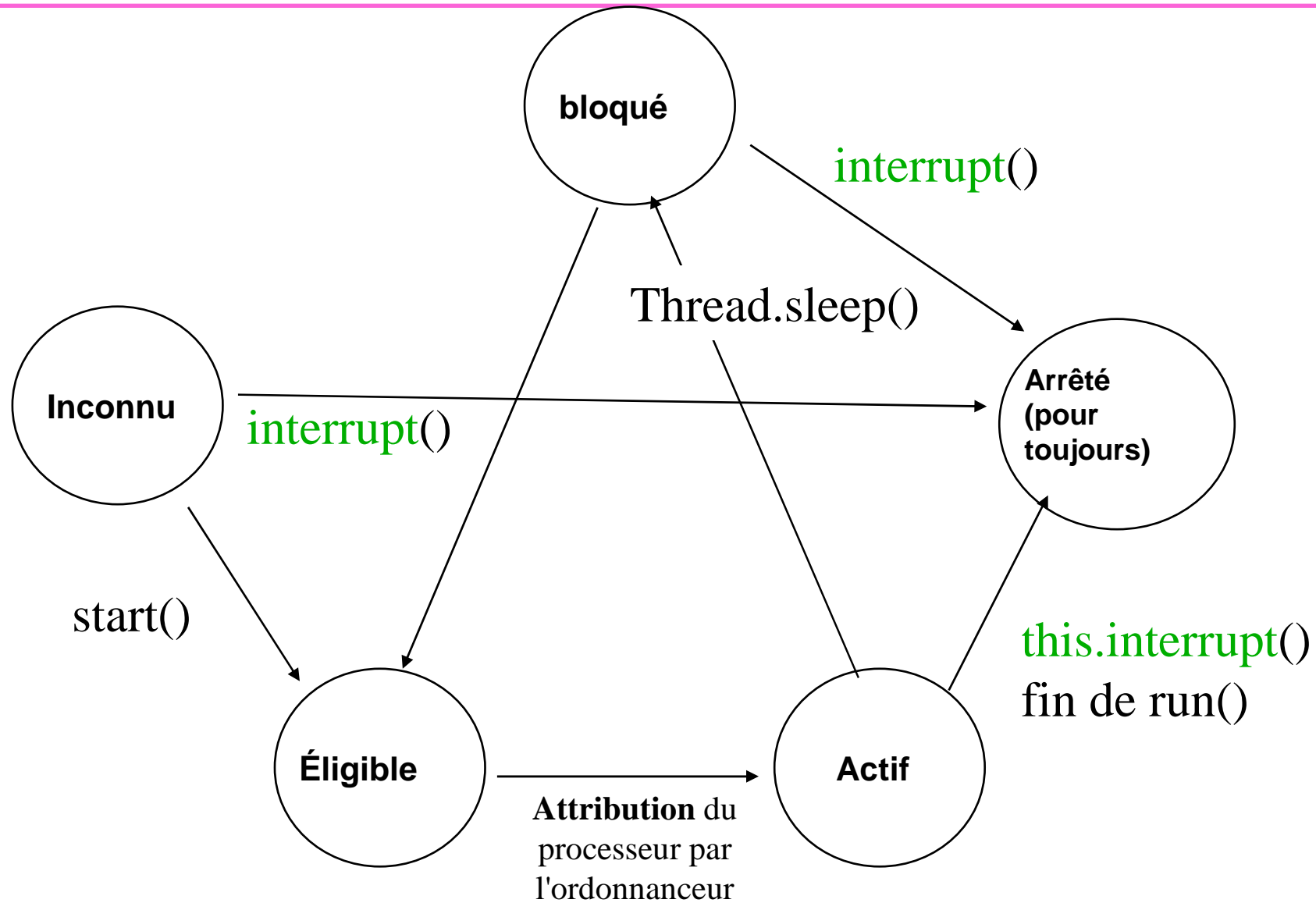
Les constructeurs publics

- ***Thread();*** // création
- ***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: une première 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); // 2 secondes
        t3.interrupt();
    }
}
```

L'exemple initial revisité

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

// test du statut « a-t-il été interrompu » ?

*Note : l'appel de **interrupt** ici se contente de positionner le booléen **interrupted** dans le contexte du thread (lu et remis à false : par l'appel de **interrupted()**, lu seulement : appel de **isInterrupted()**)*

```
public class Exemple {

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

Interrupt mais ne s'arrête pas

```
public class T extends Thread {
    public void run(){
        while(!this.interrupted()){ // test du statut « a-t-il été interrompu » hors sleep ?
            try{
                System.out.println("dans " + this + ".run");
                Thread.sleep(5000);
            }catch(InterruptedException ie){
                return; // sans cet appel le programme ne s'arrête pas (return ou interrupt)
            }
        }
    }
}
```

*Note : l'appel de **interrupt** lève une exception dans le contexte du Thread
Si cela se produit pendant exécution de la méthode **Thread.sleep(5000)**,
À l'extérieur de cette méthode: le transparent précédent*

*Note : l'appel de **interrupt** ici se contente de positionner
le booléen **interrupted** dans le contexte du thread
(lu et remis à false : par l'appel de **interrupted()**,
lu seulement : appel de **isInterrupted()**)*

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

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();*

Le constructeur Thread (Runnable r)

La syntaxe habituelle avec les interfaces

```
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();
```

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


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()* ... voir les remarques faites
- 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)

- **Note**

- *destroy()* et *stop()* ne sont plus utilisés, non sûr
- La JVM ne s'arrête que si tous les Threads créés ont terminé leur exécution

Attendre la fin

- **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);  
    }
```

Critiques

- « jungle » de Thread
- Parfois difficile à mettre en œuvre
 - Création, synchronisation, ressources ... à suivre...
- Très facile d'engendrer des erreurs ...
- Abstraire l'utilisateur des « détails » , ...
 - Éviter l'emploi des méthodes start, interrupt, sleep, etc ...
- 1) Règles d'écriture : *une amélioration syntaxique*
- 2) Usage de `java.util.concurrent`

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 ()) // ← précaution, pourquoi * ?
            while(!local.interrupted()){
                System.out.println("dans " + this + ".run");
            }
    }
}
```

- * Les méthodes des interfaces sont (obligatoirement) publiques...
 - Inhiber l'appel direct de t1.run()

Un style possible d'écriture (2)...

Un paramètre transmis lors de la création de l'instance, *classique*

```
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(){
        while(!local.interrupted()){
            System.out.println("dans " + this.nom + ".run");
        }
    }

    public void stop(){
        local.interrupt();
    }
}
```

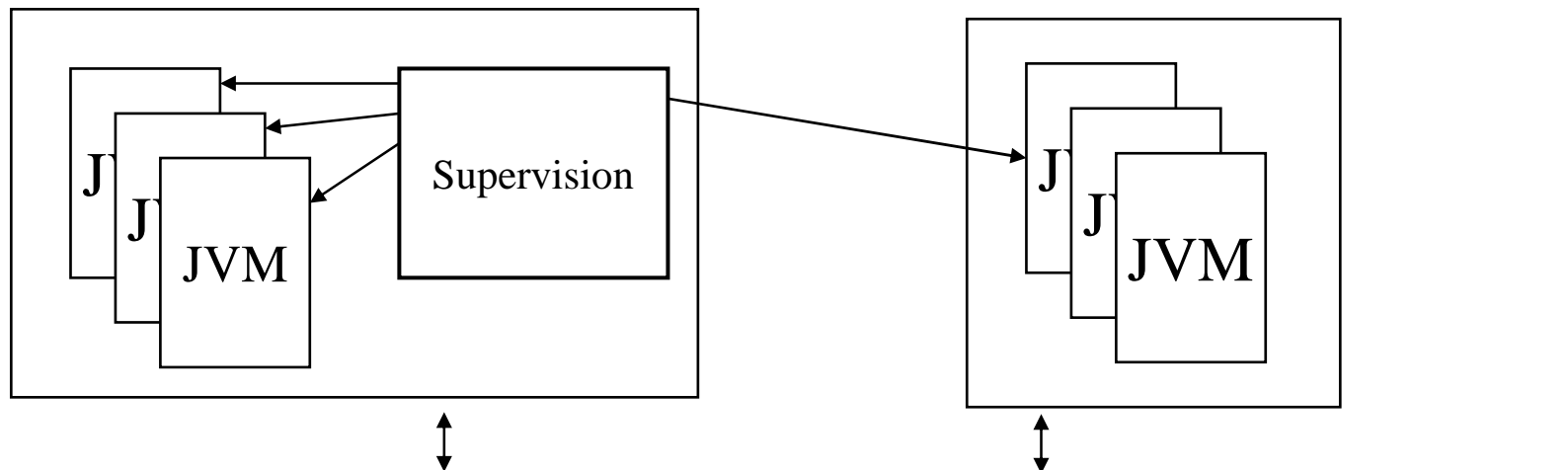
L'interface Executor

- Paquetage `java.util.concurrent`,
- `public interface Executor{ void execute(Runnable command);}`
 - `Executor executor = new ThreadExecutor();`
 - `executor.execute(new T());`
 - `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();  
    }  
}
```

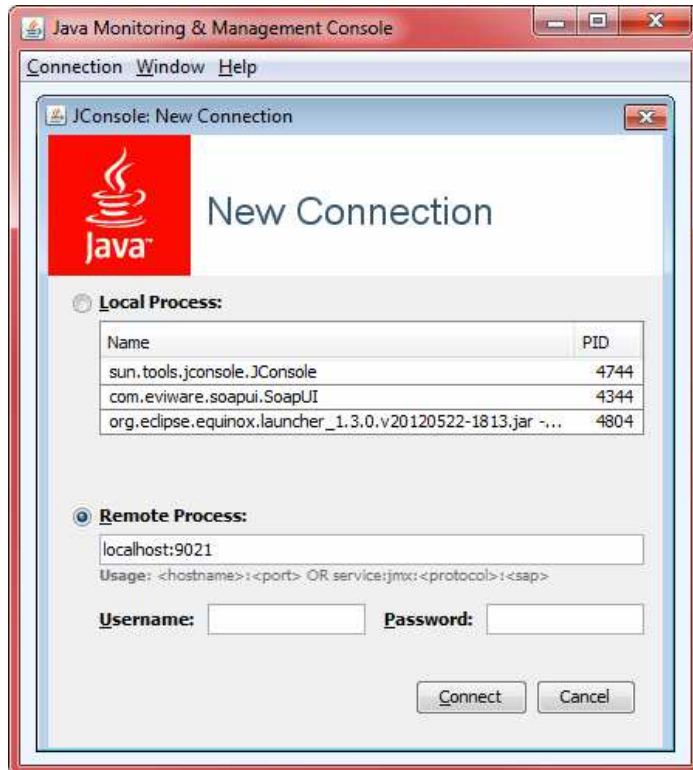
Pause ...

- **Existe-t-il des outils ?**
 - De contrôle, supervision de Thread d'une JVM
 - Une JVM en cours d'exécution ?
 - Quels Threads ?
 - Mémoire utilisée
 - Comment ?
 - Quelles instructions ?, quelles méthodes ...

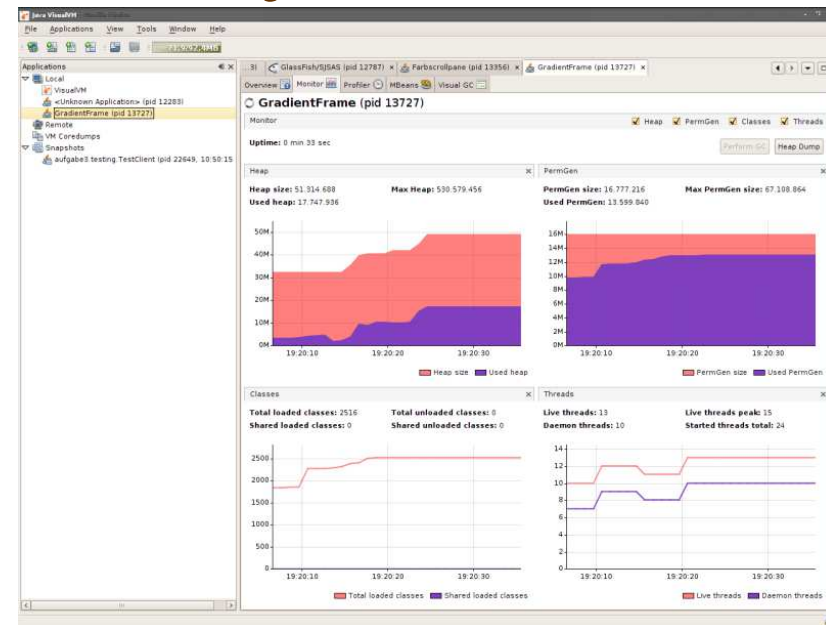


Outils déjà installés !

jconsole



jvisualvm



- Outils présents avec toute installation du jdk
 - jconsole, jvisualvm
 - Supervision, interaction, fuites mémoire, ...
 - Technologie JMX présentée dans cette unité

Pause... suite

- **Interrogation d'une JVM en cours d'exécution ?**

- Diagnostic,
- Configuration,
- Détection d'un Thread qui ne s'arrête pas,
 - *Deadlock...*
- Modification de variables
- Et plus...

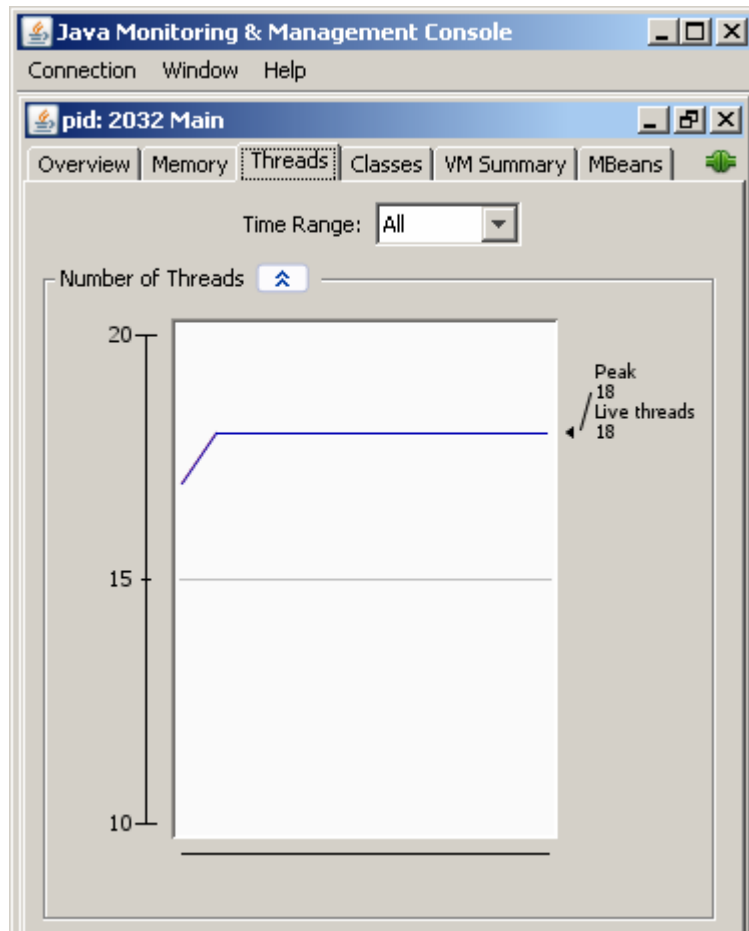
- **JMX : Java Management eXtensions**
 - Outil prédéfini `jconsole`, `jvisualvm`

- **Exemple** « 5 Threads issus de la classe T » :

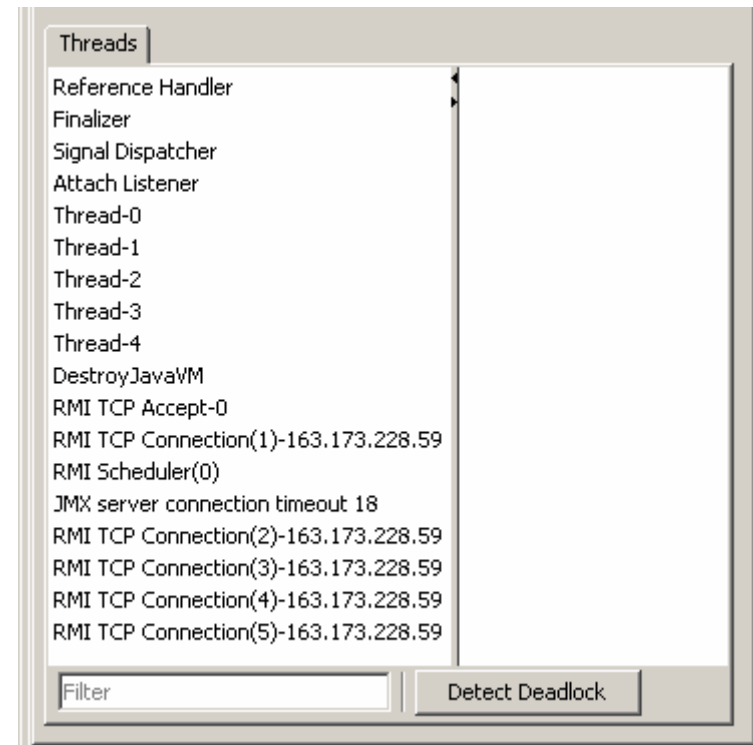
```
public class Main{
    public static void main(String[] args){
        for(int i = 0; i < 5; i++){
            new T(" dans_une_instance_de_T_"+i);
        }
    }
}
```

```
 dans_une_instance_de_T_2.run
 dans_une_instance_de_T_0.run
 dans_une_instance_de_T_3.run
 dans_une_instance_de_T_1.run
 dans_une_instance_de_T_2.run
 dans_une_instance_de_T_4.run
 dans_une_instance_de_T_0.run
 dans_une_instance_de_T_2.run
 dans_une_instance_de_T_3.run
 dans_une_instance_de_T_1.run
 dans_une_instance_de_T_4.run
 dans_une_instance_de_T_0.run
 dans_une_instance_de_T_2.run
 dans_une_instance_de_T_3.run
 dans_une_instance_de_T_1.run
```

Fin de l'intermède JMX



« 5 Threads issus de la classe T »



- **Supervision d'une JVM proche ou distante accessible au programmeur**
- **Lerveur déjà installés sur toute JVM**

Programmation concurrente: suite

- **Notion de priorité**
- **Section critique**
 - Bloc synchronized
- **Synchronisation**
 - wait, notify, méthodes héritées de la classe Object
 - Parfois difficile à mettre en œuvre
- **java.util.concurrent**
 - À la rescousse

Thread: 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,
 - ...`void yield();` // passe « la main » au prochain thread de même priorité

Et le ramasse-miettes ?

La classe java.lang.Thread

- **Quelques méthodes**

Les constructeurs

- *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();*

- *static void yield();* // **passe « la main » au prochain thread de même priorité**
- *setPriority(int priority);*
 - (MIN_PRIORITY..MAX_PRIORITY) (1..10)

Accès aux ressources ?

- **Comment ?**
- **Exclusion mutuelle ?**
- **Synchronisation ?**

Moniteur de Hoare 1974

- *Le moniteur assure un accès en exclusion mutuelle aux données qu 'il encapsule*
- *En Java avec le bloc **synchronized***
- **Synchronisation ?** par les variables conditions :
Abstraction évitant la gestion explicite de files d 'attente de processus bloqués

En Java avec

***wait et notify** dans un bloc **synchronized** (et uniquement !)*

méthodes héritées de la classe Object

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<E> extends Object{
    private E valeur;

    public synchronized E lire(){
        return valeur;
    }

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

Il est garanti qu'un seul Thread accède à une instance de la classe Ressource, ici au champ d'instance valeur

Accès « protégé » aux variables de classe

```
public class LaClasse{  
    private static Double valeur;
```

```
    synchronized( valeur){  
  
    }
```

Et aussi pour les variables de classes

```
    synchronized( LaClasse.class){  
  
    }
```

Synchronisation

Méthodes de la classe `java.lang.Object`

→ toute instance en java

// Attentes

final void wait() throws InterruptedException

final native void wait(long timeout) throws InterruptedException

// Notifications

final native void notify()

final native void notifyAll()

Note : Toujours à l'intérieur d'un bloc `synchronized`

Synchronisation : lire si c'est écrit

```
public class Ressource<E> {  
    private E valeur;  
    private boolean valeurEcrit = false;           // la variable condition  
  
    public synchronized E lire(){  
        while(!valeurEcrit)  
            try{  
                this.wait();                       // attente d'une notification  
            }catch(InterruptedException ie){ ...}  
        valeurEcrit = false;  
        return valeur;  
    }  
  
    public synchronized void ecrire(E elt){  
        valeur = elt; valeurEcrit = true; this.notify(); // notification  
    }  
}
```

Usage de la Ressource

- **Démonstration**
- **Une Ressource**
 - `Ressource<Float> r = new Ressource<Float>();`
- **Un Consommateur**
 - `Consommateur c = new Consommateur(r);`
 - `Float f = r.lire();`
- **Un Producteur**
 - `Producteur p = new Producteur(r);`
 - `p.ecrire(3.0);`

Discussion & questions

- **2 files d'attente à chaque « Object »**
 1. Liée au synchronized
 2. Liée au wait
- **Quel est le Thread bloqué et ensuite sélectionné lors d'un notify ?**
 - Thread de même priorité : aléatoire, celui qui a attendu le plus, le dernier arrivé ?
- **Quelles conséquences si notifyAll ?**
 - test de la variable condition
 - `while(!valeurEcrit) wait()`
- **Encore une fois, serait-ce des mécanismes de « trop » bas-niveau ?**
 - À suivre...

Plus Simple ? : lire avec un délai de garde

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

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

    valeurEcrit = false;
    return valeur;
}
```

Plus simple ?

Synchronisation lire si écrit et écrire si lu

- **Programmation de trop bas-niveau ?**
 - Sources de difficultés ...
 - Interblocage en annexe
 - Développeurs
 - j2se,j2ee -> `java.util.concurrent`

Plus simple ?

- **java.util.concurrent**
 - **SynchronousQueue<E>**
 - Modèle CSP, RdV

```
final BlockingQueue<Integer>  
    queue = new SynchronousQueue<Integer>(true);  
true pour le respect de l'ordre d'arrivée
```

```
queue.put(i) // bloquantes
```

```
i = queue.take()
```

```
queue.offer(i, 1000, TimeUnit.SECONDS)
```

```
i = queue.poll(TimeUnit.SECONDS)
```

Ressource<T>: usage de SynchronousQueue

```
public class Ressource<E> {  
  
    private BlockingQueue<E> queue = new SynchronousQueue<E>();  
  
    public E lire(){  
        return queue.take();  
    }  
  
    public void ecrire(E elt){  
        queue.put(elt);  
    }  
}
```

java.util.concurrent.SynchronousQueue<E>

- java.util.concurrent

- Exemple : une file un **producteur** et deux **consommateurs**

```
final BlockingQueue<Integer>
    queue = new SynchronousQueue<Integer>(true);

Thread producer = new Runnable(){
    public void run(){
        private Integer i = new Integer(1);
        public void run(){
            try{
                queue.put(i);
                i++;
                Thread.sleep(500);
            }catch(InterruptedException ie){}
        }
    });
```

java.util.concurrent.SynchronousQueue<E>

Exemple suite : les deux consommateurs

```
Thread consumer = new Thread(new Runnable(){
    public void run(){
        while(true){
            try{
                System.out.println(queue.take());
            }catch(InterruptedException ie){}
        }
    }
});
```

```
consumer.start();
```

```
Thread idle = new Thread(new Runnable(){
    public void run(){
        while(true){
            try{
                System.out.print(".");
                Thread.sleep(100);
            }catch(InterruptedException ie){}
        }
    }
});
```

```
idle.start();
```

Avec un délai de garde

```
Thread consumer2 =
    new Thread(new Runnable(){
        public void run(){
            while(true){
                try{
                    Integer i = queue.poll(100,TimeUnit.MILLISECONDS);
                    if(i!=null) System.out.println("i = " + i);
                }catch(InterruptedException ie){}
            }
        }
    });
consumer2.start();
```

java.util.concurrent

- `java.util.concurrent.LinkedBlockingQueue<E>`
- `java.util.concurrent.PriorityBlockingQueue<E>`
- *Trop simple? 50 nouvelles classes ...*

java.util.concurrent

- `java.util.concurrent.ThreadPoolExecutor`
 - Une réserve de Threads
 - `java.util.concurrent.ThreadPoolExecutor.AbortPolicy`
 - A handler for rejected tasks that throws a `RejectedExecutionException`.

JavaOne 2004, Un serveur « Web » en une page

```
class WebServer { // 2004 JavaOneSM Conference | Session 1358
    Executor pool = Executors.newFixedThreadPool(7);

    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            pool.execute(r);
        }
    }
}
```


Un serveur Web en quelques lignes

```
public class WebServer implements Runnable{

    public interface Command<T>{
        public void handleRequest(T t) throws Exception;
    }
    private final Executor executor;
    private Thread local;
    private final Command<Socket> command;
    private int port;

    public WebServer(Executor executor,
                    Command<Socket> command, int port) {
        this.executor = executor;
        this.command = command;
        this.port = port;
        this.local = new Thread(this);
        this.local.setDaemon(true);
        this.local.setPriority(Thread.MAX_PRIORITY);
        this.local.start();}
}
```

Serveur web, la suite

```
public void run(){
    try{
        ServerSocket socket = new ServerSocket(port);
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run(){
                    try{
                        command.handleRequest(connection);
                    }catch(Exception e){
                        e.printStackTrace();
                    }
                }
            };
            executor.execute(r);
        }
    }catch(Exception e){e.printStackTrace();}}
```

Un client, une requête en //, attente avec join

```
public class Requête extends Thread {  
    private String url; private String résultat ;
```

```
    public Requête(String url){  
        this.url = url; this.start();  
    }  
}
```

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

```
public void run(){
```

```
    this. résultat = new String();
```

```
    try{
```

```
        URL urlConnection = new URL(url);
```

```
        URLConnection connection = urlConnection.openConnection();
```

```
        BufferedReader in = new BufferedReader( new InputStreamReader(connection.getInputStream()));
```

```
        String inputLine = in.readLine();
```

```
        while(inputLine != null){
```

```
            résultat += inputLine;
```

```
            inputLine = in.readLine();
```

```
        }
```

```
        in.close();
```

```
    }catch(Exception e){ this. résultat = "exception " + e.getMessage();}
```

Une requête/thread + join + résultat

Le Future, Une requête HTTP sans attendre...

- Une requête en //, puis demande du résultat

Avec `java.util.concurrent`

- **Callable** interface au lieu de **Runnable**
 - Soumettre un thread à un **ExecutorService**
 - Méthode *submit()*
 - En retour une instance de la classe `Future<T>`
 - Accès au résultat par la méthode *get()* ou mis en attente

La requête HTTP reste simple, interface Callable<T>

```
public class RequeteHTTP implements Callable<String>{  
    public RequeteHTTP(){...}
```

```
    public String call(){
```

```
        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();
```

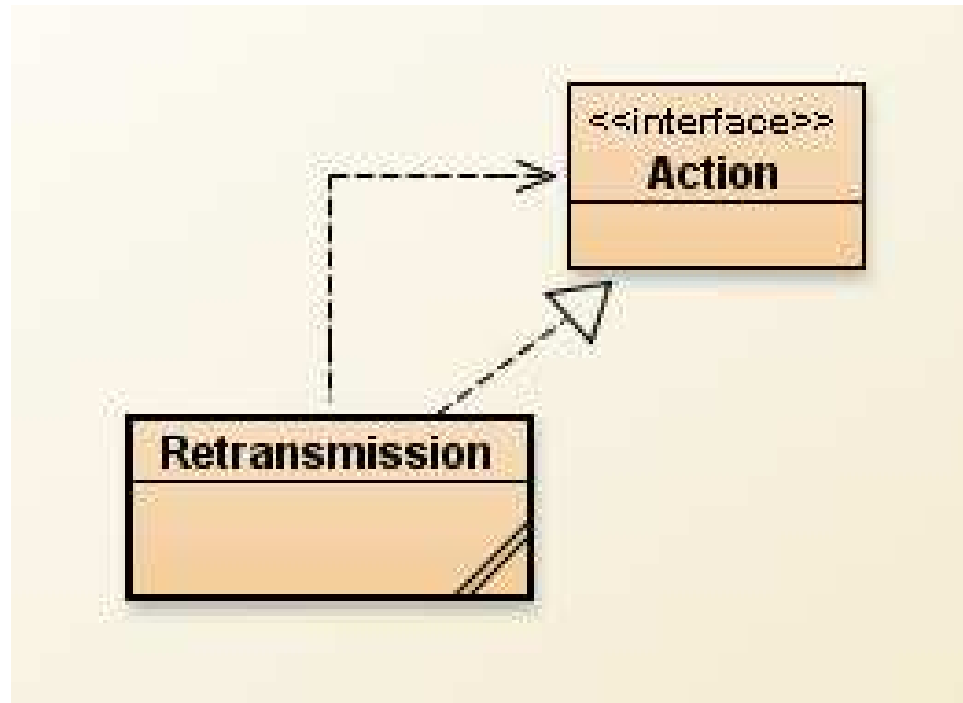
```
        }catch(Exception e){...}}
```

```
        return result.toString()
```

La requête HTTP dans le futur

- `ExecutorService executor = ...`
- `Future<String> future = executor.submit(new RequeteHTTP());`
- ...
- `String s = future.get();` `// get avec un délai de garde`
 `// isDone(), cancel()`

Le patron Retransmission, Mark Grand Vol 3



- **Principe :**

- Effectuer une action (une requête) jusqu'à ce que celle-ci réussisse !
- Apparenté Proxy ?

Le patron Retransmission 1/3

– Une proposition d'implémentation non testée ...

```
public interface Action<E>{  
    public void action(E... e) throws Exception;  
}
```

```
public class Retransmission<E> implements Action<E>{  
    private Action<E> actionInitiale; // à retransmettre, si nécessaire  
    private int period = 10000; // la période entre deux tentatives  
    private int nombre = 10; // le nombre maximal de retransmissions  
  
    public Retransmission(Action<E> actionInitiale, int period, int nombre) {  
        this.actionInitiale = actionInitiale;  
        this.period = period;  
        this.nombre = nombre;  
    }  
  
    public void action(E... e) throws Exception{  
        RetansmissionInterne ri = new RetansmissionInterne(e);  
        if(ri.enEchec()) throw ri.cause();  
    }  
  
    private class RetansmissionInterne extends Thread{  
        // page suivante  
    }  
}
```


Le patron Retransmission 2/3

- *Principe : Effectuer une action (une requête) jusqu'à ce que celle-ci réussisse !*

```
private class RetransmissionInterne extends Thread{
    private E[] e;
    private Exception cause = null;

    public RetransmissionInterne(E e){this.e = e;  this.start();}

    public boolean enEchec() throws Exception{
        try{ this.join();}catch(Exception exc){cause = exc;}
        return cause != null;
    }

    public Exception cause(){return cause;}

    public void run(){
        boolean success = false; int tentatives = 0;
        while(tentatives<=nombre && !success){
            try{
                actionInitiale.action(e);
                success = true;cause = null;
            }catch(Exception e){
                tentatives++;cause = e;
                try{
                    Thread.sleep(period);
                }catch(InterruptedException ie){cause = ie;}
            }
        }
    }
}
```

Le patron Retransmission 3/3

- **Principe :**

- Effectuer une action (une requête) jusqu'à ce que celle-ci réussisse !

- **Pause**

- C'était une proposition d'implémentation non testée ...

- À tester donc ...

Les collections

- **Accès en lecture seule**
- **Accès synchronisé**
 - `static <T> Collection<T> synchronizedCollection(Collection<T> c)`
 - `static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c)`

```
Collection<Integer> c = Collections.synchronizedCollection(  
                                new ArrayList<Integer>());
```

```
Collection<Integer> c1 = Collections.synchronizedCollection(  
                                new ArrayList<Integer>());
```

```
c1.add(3);
```

```
c1 = Collections.unmodifiableCollection(c1);
```

Conclusion intermédiaire

- **Mécanisme de bas niveau, nous avons**
- **java.util.concurrent et quelques patrons, nous préférons**

la roue

- . Whenever you are about to use...
 - . `Object.wait, notify, notifyAll,`
 - . `synchronized,`
 - . `new Thread(aRunnable).start();`

- . Check first if there is a class in `java.util.concurrent` that...
 - Does it already, or
 - Would be a simpler starting point for your own solution

- extrait de <https://developers.sun.com/learning/javaoneonline/2004/corej2se/TS-1358.pdf>

Conclusion première partie

- **java.util.concurrent**
 - À utiliser

- **Quid ? Des Patrons pour la concurrence ?**
 - **Critical Section, Guarded Suspension, Balking, Scheduler, Read/Write Lock, Producer-Consumer, Two-Phase Termination**

 - À suivre ... un autre support

3 patrons

- **Le Singleton** revisité pour l'occasion
 - Garantit une et une seule instance d'une classe

- **Architectures logicielles : un début**
 - Le couple Acquisition/Traitement

 - Le patron **Chaîne de responsabilités**

 - Le patron **Interceptor**

UML et la patron Singleton



- **Une et une seule instance,**
 - même lorsque 2 threads tentent de l'obtenir en « même temps »

Le Pattern Singleton, revisité ?

```
public final class Singleton{
    private static volatile Singleton instance = null;           // volatile ??

    public static Singleton getInstance(){
        synchronized(Singleton.class)                            // synchronized ??
        if (instance==null)
            instance = new Singleton();
        return instance;
    }
}

private Singleton(){
}
}
```

Extrait de Head First Design Pattern,
O'Reilly, <http://oreilly.com/catalog/9780596007126>

Préambule, chain of responsibility

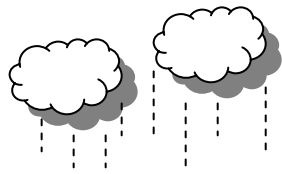
- **Découpler l'acquisition du traitement d'une information**

1) Acquisition

2) Traitement

- Par la transmission de l'information vers une chaîne de traitement
- La chaîne de traitement est constitué d'objets relayant l'information jusqu'au **responsable**

Exemple : capteur d'humidité et traitement



DS2438, capteur d'humidité

Acquisition

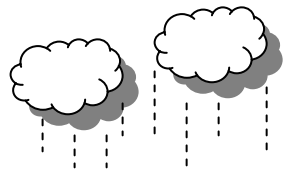


Traitement

- Persistance
- Détection de seuils
- Alarmes
- Histogramme
- ...

- Acquisition / traitement, suite

Acquisition + Chaîne de responsabilités



DS2438, capteur d'humidité

valeur



Persistence

valeur



Détection de seuils

valeur



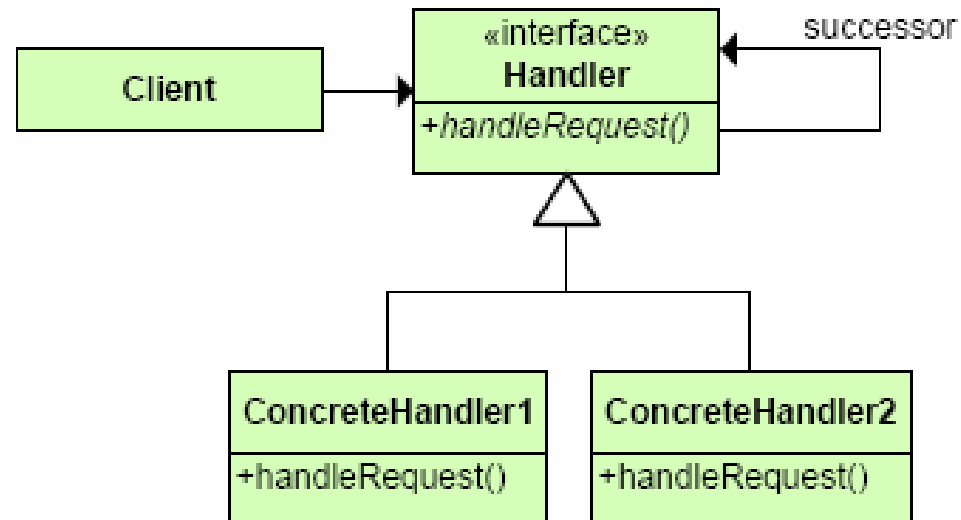
Acquisition

Chaîne de responsabilités

Traitement

Un responsable décide d'arrêter la propagation de la valeur

Le patron Chain of Responsibility



- **Le client ne connaît que le premier maillon de la chaîne**
 - Chaque maillon propage l'information à son successeur,
 - Un maillon peut décider d'arrêter la propagation
 - Ajout/retrait dynamique de maillons dans la liste

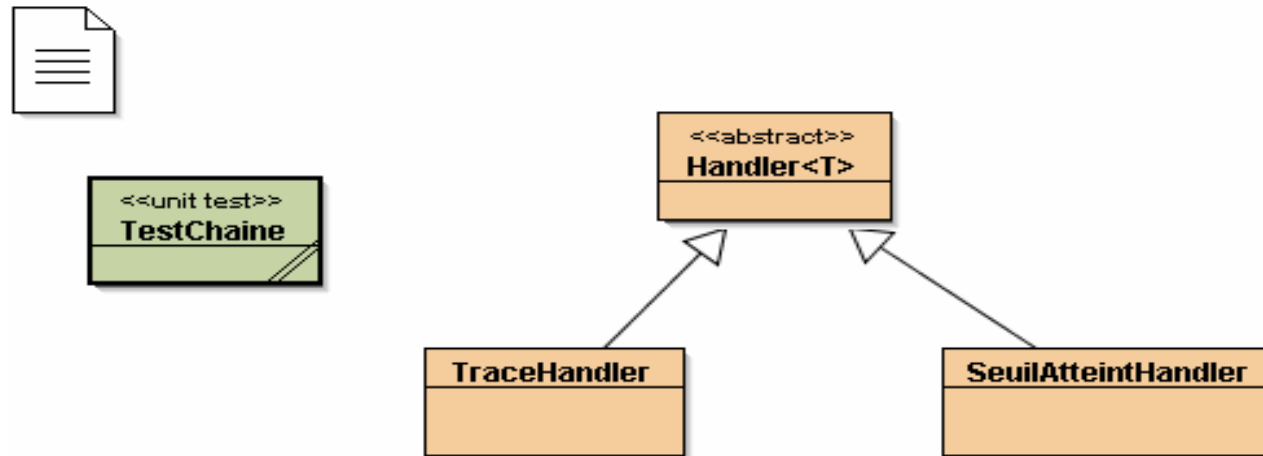
abstract class Handler<V>, V comme valeur

```
public abstract class Handler<V>{           // héritée par tout maillon
    protected Handler<V> successor = null;

    public Handler(){ this.successor = null;}
    public Handler(Handler<V> successor){
        this.successor = successor;
    }
    public void setSuccessor(Handler<V> successor){
        this.successor = successor;
    }
    public Handler<V> getSuccessor(){
        return this.successor;
    }

    public boolean handleRequest(V value){ // propagation
        if ( successor == null ) return false;
        return successor.handleRequest(value);
    }
}
```

2 classes concrètes, pour le moment



Deux maillons : TraceHandler et SeuilAtteintHandler

TraceHandler se contente d'afficher l'information et la propage sans condition

SeuilAtteintHandler interrompt la propagation en delà d'un certain seuil

- Soit la chaîne : TraceHandler → SeuilAtteintHandler

class ConcreteHandler1 : une trace, et il n'est pas responsable

```
public class TraceHandler extends Handler<Integer>{

    public TraceHandler(Handler<Integer> successor){
        super(successor);
    }

    public boolean handleRequest(Integer value){
        System.out.println("received value : " + value);

        // l'information, « value » est propagée sans condition
        return super.handleRequest(value);
    }
}
```


class ConcreteHandler2 : la détection d'un seuil

```
public class SeuilAtteintHandler extends Handler<Integer>{

    private int seuil;

    public SeuilAtteintHandler(int seuil, Handler<Integer> successor){
        super(successor);
        this.seuil = seuil;
    }

    public boolean handleRequest(Integer value){
        if( value > seuil){
            System.out.println(" seuil de " + seuil + " atteint, value = " + value);
            // return true; si le maillon souhaite arrêter la propagation
        }

        // l'information, « value » est propagée
        return super.handleRequest(value);
    }
}
```

Une instance possible, une exécution

la chaîne : `TraceHandler` → `SeuilAtteintHandler(100)`

Extrait de la classe de tests

```
Handler<Integer> chaine =  
    new TraceHandler( new SeuilAtteintHandler(100,null));
```

```
chaine.handleRequest(10);  
chaine.handleRequest(50);  
chaine.handleRequest(150);
```

```
received value : 10  
received value : 50  
received value : 150  
seuil de 100 atteint, value = 150
```

Ajout d'un responsable à la volée

la chaîne : TraceHandler → SeuilAtteintHandler(50) → SeuilAtteintHandler(100)

Détection du seuil de 50

```
Handler<Integer> chaine = new TraceHandler( new SeuilAtteintHandler(100,null));
```

```
chaine.handleRequest(10);  
chaine.handleRequest(50);  
chaine.handleRequest(150);
```

```
Handler<Integer> seuil50 =  
    new SeuilAtteintHandler(50, chaine.getSuccessor());
```

```
chaine.setSuccessor(seuil50);
```

```
chaine.handleRequest(10);  
chaine.handleRequest(50);  
chaine.handleRequest(150);
```

```
received value : 10  
received value : 50  
received value : 150  
    seuil de 100 atteint, value = 150  
received value : 10  
received value : 50  
received value : 150  
    seuil de 50 atteint, value = 150  
    seuil de 100 atteint, value = 150
```

Un responsable ! enfin

```
public class ValeurNulleHandler extends Handler<Integer>{  
  
    public ValeurNulleHandler (Handler<Integer> successor){  
        super(successor);  
    }  
  
    public boolean handleRequest(Integer value){  
        if( value==0) return true;  
        else  
  
        // sinon l'information, « value » est propagée  
        return super.handleRequest(value);  
    }  
}
```

TraceHandler → ValeurNulleHandler → SeuilAtteintHandler(50) → ..

↓
Arrêt de la propagation

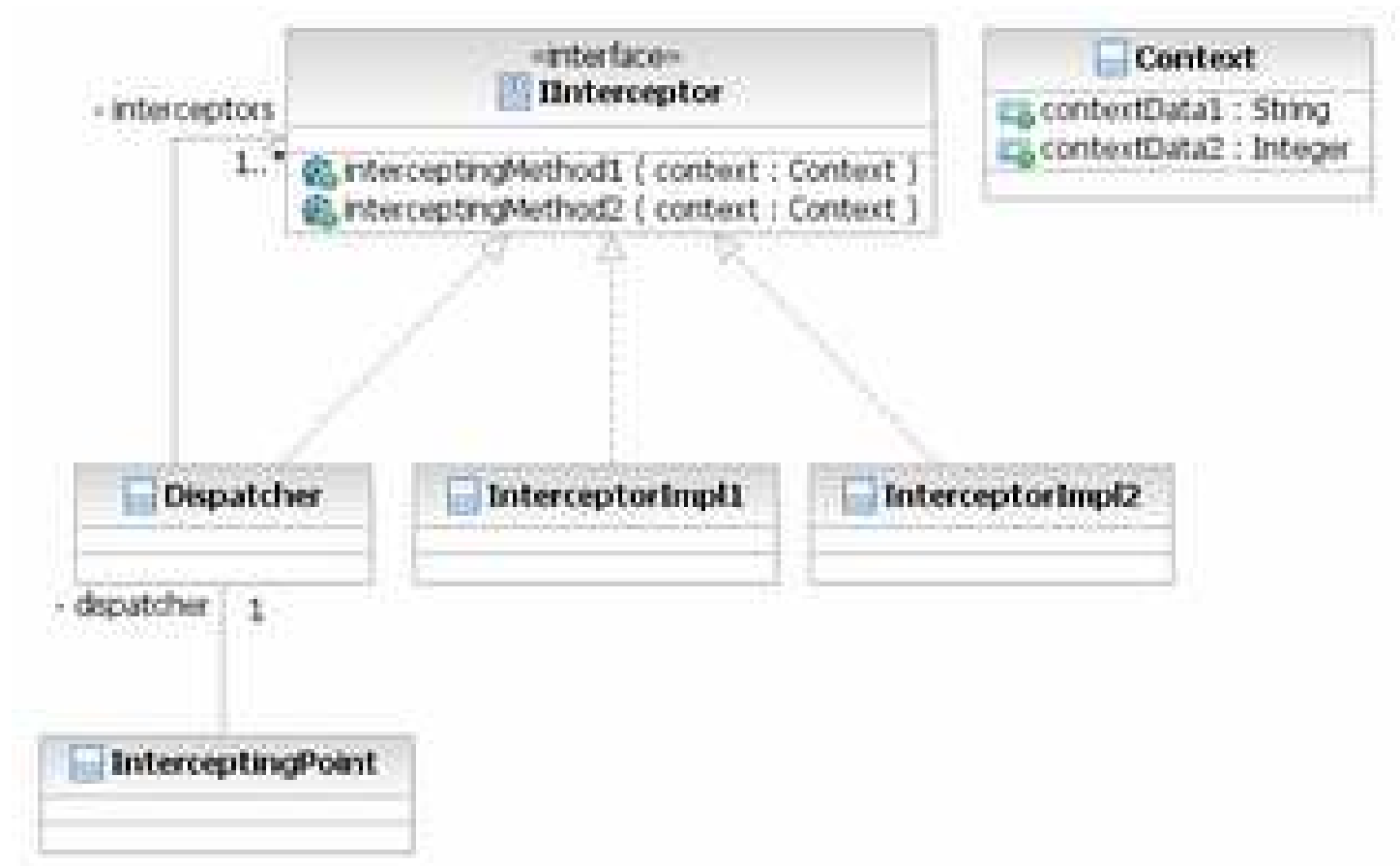
Interceptor : une variante de CoR

- Interception d'appels de méthodes
 - Filtrage de certains contenus, paramètres
 - Trace, journal, temps d'exécution
 - Ajout d'instructions avant et après l'exécution d'une méthode
 - Capture des exceptions
 - Changement des paramètres à la volée
-
- framework
 - @interceptor pour les EJB
 - Configuration en XML
 - <http://longbeach.developpez.com/tutoriels/EJB3/Interceptors/>
 - <http://struts.apache.org/release/2.0.x/docs/interceptors.html>

Framework Interceptors

Interceptor	Name	
Alias Interceptor	alias	Converts similar parar
Chaining Interceptor	chain	Makes the previous Ac type="chain"> (in the
Checkbox Interceptor	checkbox	Adds automatic checkt (usually 'false') value. value is overridable fo
Cookie Interceptor	cookie	Inject cookie with a ce

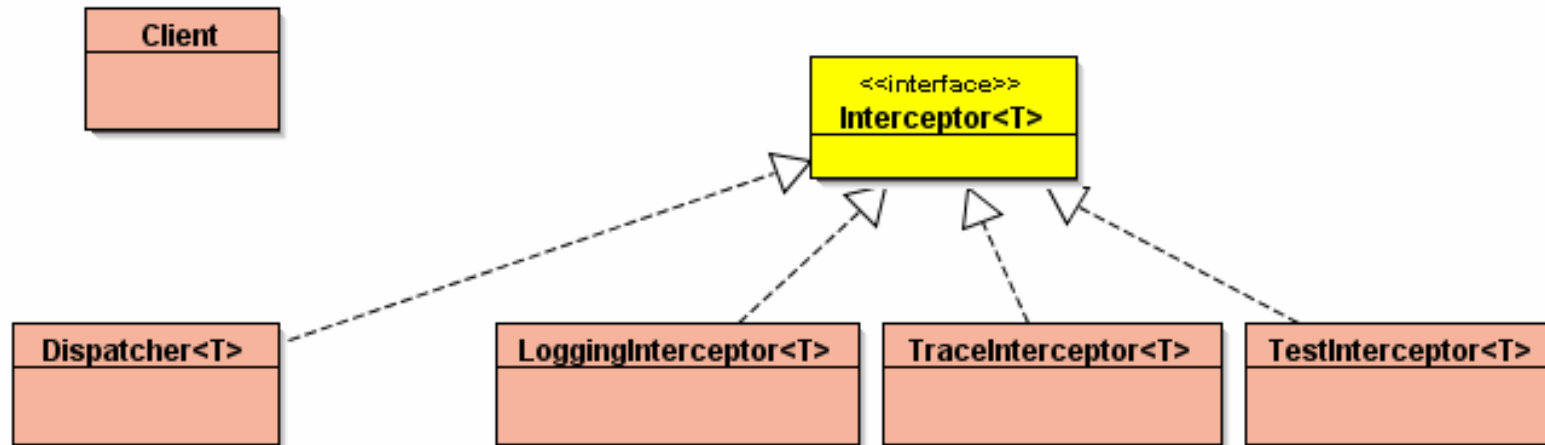
Interceptor : une variante « Chain of Responsibility »



<http://bosy.dailydev.org/2007/04/interceptor-design-pattern.html>

<http://struts.apache.org/release/2.0.x/docs/interceptors.html>

Une implémentation légère* ...



Dispatcher propage l'information le long des *interceptors* ...
Chaque *interceptor* réalise une action

* une implémentation plus conséquente avec introspection cf. le support Proxy

Interceptor<T> et une implémentation

```
public interface Interceptor<T>{  
    public T invoke(T in) throws Exception;  
}  
  
public class TraceInterceptor<T> implements Interceptor<T>{  
    public T invoke(T in) throws Exception{  
        System.out.println(this.getClass().getName() + " : " + in);  
        return in;  
    }  
}  
  
public class LoggingInterceptor<T> idem
```


Dispatcher<T> (une version simple)

```
public class Dispatcher<T> implements Interceptor<T>{
    private Interceptor<T>[] interceptors;

    public Dispatcher(final Interceptor<T>... interceptors){
        this.interceptors = interceptors;
    }

    // Attention public Dispatcher(final Class<? extends Interceptor<T>>... interceptors)
    // serait la « bonne » syntaxe mais Client ne se compile pas
    public Dispatcher(final Class<? extends Interceptor>... interceptors)
        throws InstantiationException, IllegalAccessException{

        int i = 0;
        this.interceptors = new Interceptor[interceptors.length];
        for(Class<? extends Interceptor> interceptor : interceptors){
            this.interceptors[i] = interceptor.newInstance();
            i++;
        }
    }

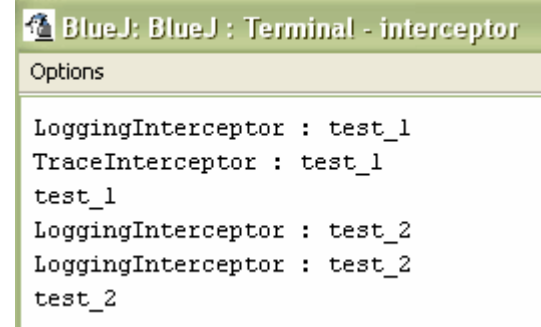
    public T invoke(final T in){ // in1/out1 → in2/out2 → ...
        T out = in;
        for(Interceptor<T> interceptor : interceptors){
            try{
                out = interceptor.invoke(out);
            }catch(Exception e){}
        }
        return out;
    }
}
```

Le Client

```
public class Client{

    public static void main(String[] args) throws Exception{
        Dispatcher<String> dispatcher1 =
            new Dispatcher<String>(new LoggingInterceptor<String>(),
                                   new TraceInterceptor<String>());
        System.out.println(dispatcher1.invoke("test_1"));

        Dispatcher<String> dispatcher2 =
            new Dispatcher<String>(LoggingInterceptor.class,
                                   LoggingInterceptor.class);
        System.out.println(dispatcher2.invoke("test_2"));
    }
}
```



```
BlueJ: BlueJ : Terminal - interceptor
Options
LoggingInterceptor : test_1
TraceInterceptor : test_1
test_1
LoggingInterceptor : test_2
LoggingInterceptor : test_2
test_2
```

Annexe: La classe java.util.Timer

- **Affichage cyclique**

 - "." toutes les secondes

- **Une autre abstraction (réussie)**

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

Annexe: Interblocage : mais où est-il ? discussion

```
class UnExemple{
  protected Object variableCondition;

  public synchronized void aa(){
    ...
    synchronized(variableCondition){
      while (!condition){
        try{
          variableCondition.wait();
        }catch(InterruptedException e){}
      }
    }

  public synchronized void bb(){
    synchronized(variableCondition){
      variableCondition.notifyAll();
    }
  }
}
```