

# Programmation concurrente

Cnam Paris  
jean-michel Douin, douin au cnam point fr  
9 Octobre 2013

Notes de cours

ESIEE

1

## Sommaire pour les Patrons

- **Classification habituelle**

- **Créateurs**

- Abstract Factory, Builder, Factory Method, Prototype, Singleton

- **Structurels**

- Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy,

- **Comportementaux**

- Chain of Responsibility, Command, Interpreter, Iterator,  
Mediator, Memento, Observer, State,  
Strategy, Template Method, Visitor

ESIEE

2

## Les patrons déjà vus en quelques lignes ...

- **Adapter**
  - Adapte l'interface d'une classe conforme aux souhaits du client
- **Proxy**
  - Fournit un mandataire au client afin de contrôler/vérifier ses accès
- **Observer**
  - Notification d'un changement d'état d'une instance aux observateurs inscrits
- **Template Method**
  - Laisse aux sous-classes une bonne part des responsabilités
- **Iterator**
  - Parcours d'une structure sans se soucier de la structure interne choisie
- **Composite**
  - Définition d'une structure données récursives
- **Visitor**
  - Instruction en fonction du noeud traversé

ESIEE

3

## Sommaire

- **Les bases**
  - java.lang.Thread
    - start(), run(), join(),...
  - java.lang.Object
    - wait(), notify(),...
  - java.util.Collections
- **Patrons**
  - Singleton
    - En accès concurrent
  - Chaîne de responsabilités
    - Acquisition/traitement

ESIEE

4

## 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.mindspring.com/~mgrand/pattern_synopses.htm#Concurrency%20Patterns)

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

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

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

ESIEE

5

## Pré-requis

- **Notions de**
  - Interface
  - Abstract superclass
  - Patron délégation
  - Patron Décorateur

ESIEE

6

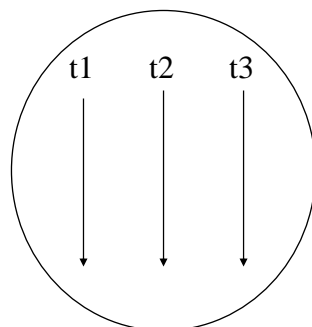
## Exécutions concurrentes

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

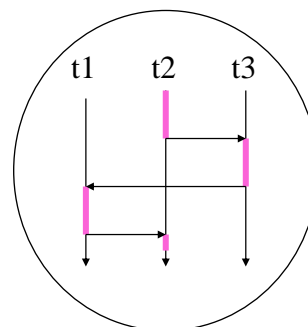
ESIEE

7

## Contexte : Quasi-parallèle

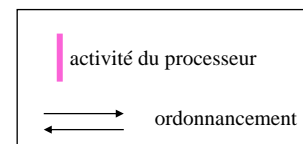


vue logique



vue du processeur

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



ESIEE

8

## 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(e) Thread pour processus allégé...)*
- « Exécution » du processus
  - `unThread.start();`
    - éligibilité de UnThread

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

  - *Soit l'appel par l'ordonnanceur de la méthode run*  
`unThread.run();`
    - Les instructions de `unThread`

ESIEE

9

## 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");
        }
    }
}
```

ESIEE

10

## 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*

ESIEE

11

## La classe java.lang.Thread

### • Quelques méthodes

#### Les constructeurs publics

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

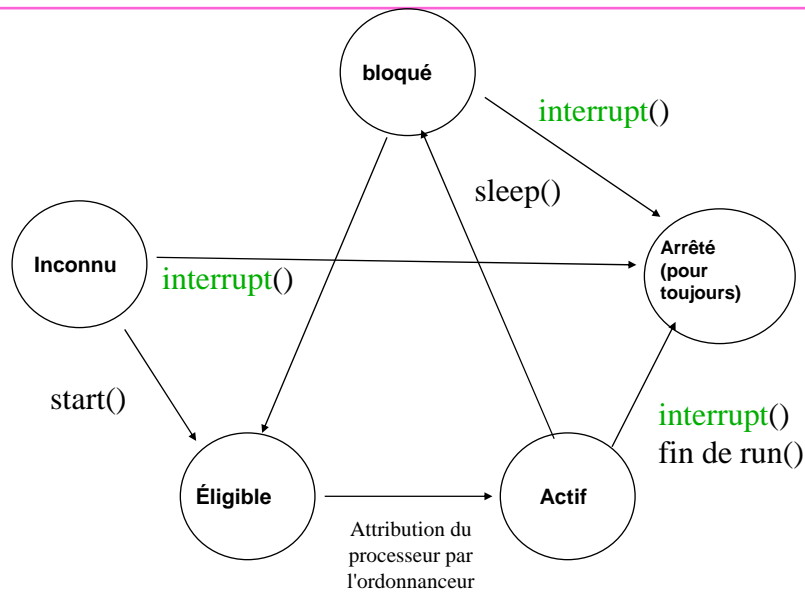
#### Les méthodes publiques

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

ESIEE

12

## États d'un Thread



- États en 1ère approche

ESIEE

13

## 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();
        t3.start();
        System.out.println("dans Exemple.main");
        t2.interrupt();
        Thread.sleep(2000); // 2 secondes
        t3.interrupt();
    }
}
  
```

ESIEE

14

## 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();
        t1.interrupt();
        t2.start();
        t3.start();
        System.out.println("dans Exemple.main");
        t2.interrupt();
        Thread.sleep(2000);
        t3.interrupt();
    }
}
```

ESIEE

15

## 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 du wait, en dehors idem transparent précédent

```
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();
    }
}
```

ESIEE

16



## 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();  
}
```

ESIEE

17

## 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

ESIEE

18

## 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);  
    }  
}
```

ESIEE

19

## 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) Les patrons à la rescousse
- 3) `java.util.concurrent` **une présentation**, nécessite un autre support

ESIEE

20

## 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...

ESIEE

21

## 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");
        }
    }
}
```

ESIEE

22

## Abstraction possible: 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();  
    }  
}
```

ESIEE

23

## Intermède, pause, détente...

- Interrogation d'une JVM en cours d'exécution ?
  - Diagnostic,
  - Configuration,
  - Détection d'un Thread qui ne s'arrête pas,
  - Et bien plus encore...
- JMX : Java Management eXtensions
  - Outil prédéfini jconsole

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

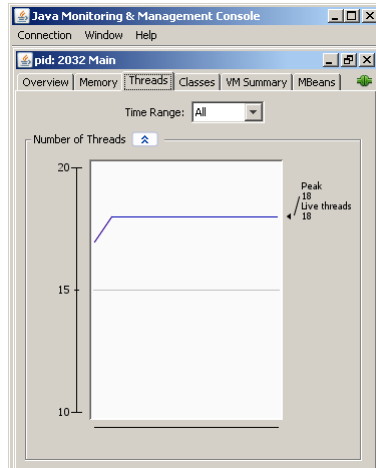
```
public class Main{  
    public static void main(String[] args){  
        for(int i = 0; i < 5; i++){  
            new T(" dans_une_instance_de_T_"+i);  
        }  
    }  
}
```

```
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
```

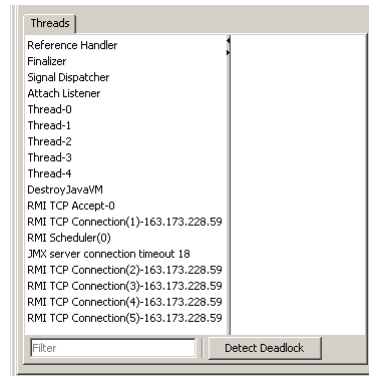
ESIEE

24

## Fin de l'intermède JMX



« 5 Threads issus de la classe T »



- Supervision d'une JVM proche ou distante accessible au programmeur

ESIEE

25

## 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 ?

ESIEE

26

## Accès aux ressources ?

- Comment ?
- Exclusion mutuelle ?
- Synchronisation ?

ESIEE

27

## 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 !)*

ESIEE

28

## 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){.....}  
    }}
```

ESIEE

29

## Une ressource en exclusion mutuelle

```
public class Ressource extends Object{  
    private double valeur;  
  
    public synchronized double lire(){  
        return valeur;  
    }  
  
    public synchronized void ecrire(double v){  
        valeur = v;  
    }  
}
```

**Il est garanti** qu'un seul Thread accède à une ressource, ici au champ d'instance

ESIEE

30

## 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){  
  
    }
```

ESIEE

31

## 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***

ESIEE

32



## 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){ Thread.currentThread().interrupt();}  
        valeurEcrit = true;  
        return valeur;  
    }  
  
    public synchronized void écrire(E elt){  
        valeur = elt; valeurEcrit = true; this.notify(); // notification  
    }  
}
```

ESIEE

33

## 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...

ESIEE

34

## 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 ?

ESIEE

35

## Synchronisation lire si écrit et écrire si lu

- À faire sur papier ... en fin de cours ...
- Programmation de trop bas-niveau ?
  - Sources de difficultés ...
    - Interblocage, transparent suivant

ESIEE

36

## 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();
            }
        }
    }
}
```

ESIEE

37

## Plus simple ?

- **java.util.concurrent**
  - **SynchronousQueue<E>**

- **FIFO** / producteur/consommateur

```
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)
```

ESIEE

38

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

ESIEE

39

## java.util.concurrent.SynchronousQueue<E>

### Exemple suite : les deux **consommateurs**, + **Idle**

```
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();
```

ESIEE

40

## 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();
```

ESIEE

41

## java.util.concurrent

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

ESIEE

42

## 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`.
  - `AtomicInteger`
    - Accès en exclusion mutuelle prêt à l'emploi

ESIEE

43

## 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);
        }
    }
}
```

ESIEE

44

## 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();}
}
```

ESIEE

45

## 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();}}
```

ESIEE

46

## 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

ESIEE

47

## 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

ESIEE

48



## 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()
    }
}
```

ESIEE

49

## 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);
```

ESIEE

50

## Conclusion intermédiaire

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

ESIEE

51

## 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>

ESIEE

52

## 2 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**

ESIEE

53

## UML et la patron Singleton

| Singleton               |
|-------------------------|
| -instance : Singleton   |
| -Singleton()            |
| +Instance() : Singleton |

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

ESIEE

54

## 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>

ESIEE

55

## Préambule, Chain of Responsibility (CoR)

- Découpler l'acquisition du traitement d'une information

### 1) Acquisition

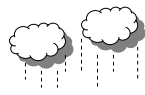
### 2) Traitement

- 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**

ESIEE

56

## Exemple : capteur d'humidité et traitement



DS2438, capteur d'humidité

Acquisition

Traitement

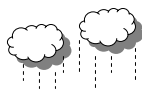
- Persistance
- Détection de seuils
- Alarmes
- Histogramme
- ...

- Acquisition / traitement, suite

ESIEE

57

## Acquisition + Chaîne de responsabilités



DS2438, capteur d'humidité

Acquisition

valeur

Persistance

valeur

Détection de seuils

valeur

Chaîne de responsabilités

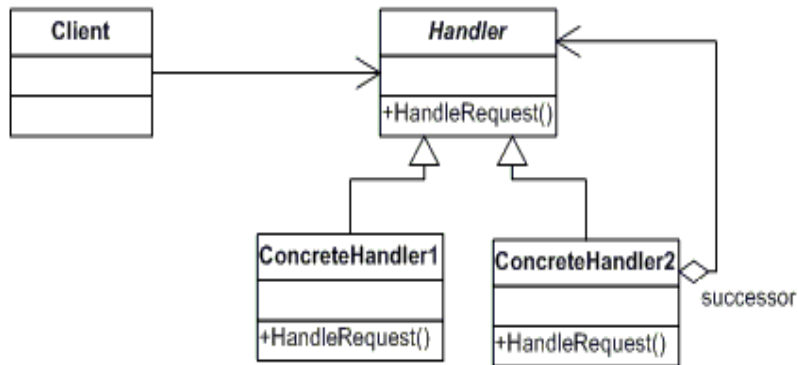
Traitement

**Un responsable décide d'arrêter la propagation de la valeur**

ESIEE

58

## Le patron Chain of Responsibility



- Le client ne connaît que le premier maillon de la chaîne
  - La recherche du responsable est à la charge de chaque maillon
  - Ajout/retrait dynamique de responsables ( de maillons)

ESIEE

59

## abstract class Handler<V>, V comme valeur

```

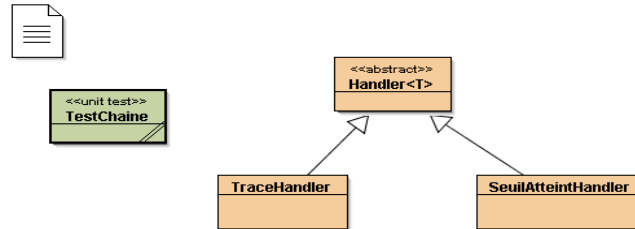
public abstract class Handler<V>{           // héritée par tout maillon
    protected Handler<V> successor = null;

    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){
        if ( successor == null ) return false;
        return successor.handleRequest(value);
    }
}
    
```

ESIEE

60

## 2 classes concrètes, pour le moment



**TraceHandler**  
+ **SeuilAtteintHandler**

- Soit la chaîne : TraceHandler → SeuilAtteintHandler

ESIEE

61

## 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
        return super.handleRequest(value);
    }
}
```

ESIEE

62

## 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);
    }
}
```

ESIEE

63

## 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
```

ESIEE

64



## 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
```

ESIEE

65

## 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);  
    }  
}
```

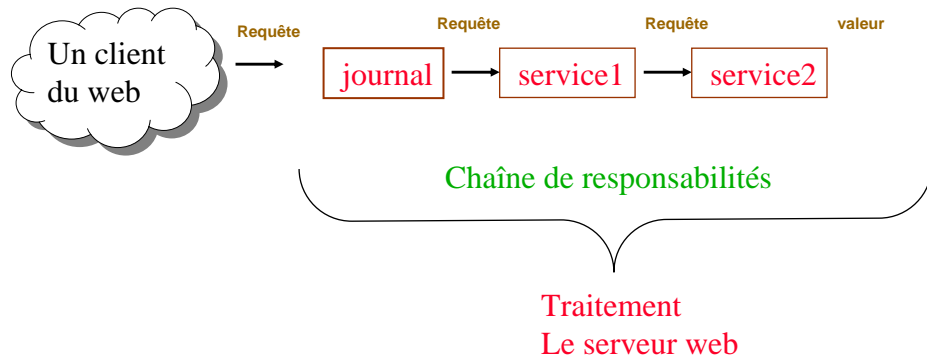
`TraceHandler` → `ValeurNulleHandler` → `SeuilAtteintHandler(50)` → ..

↓ Arrêt de la propagation

ESIEE

66

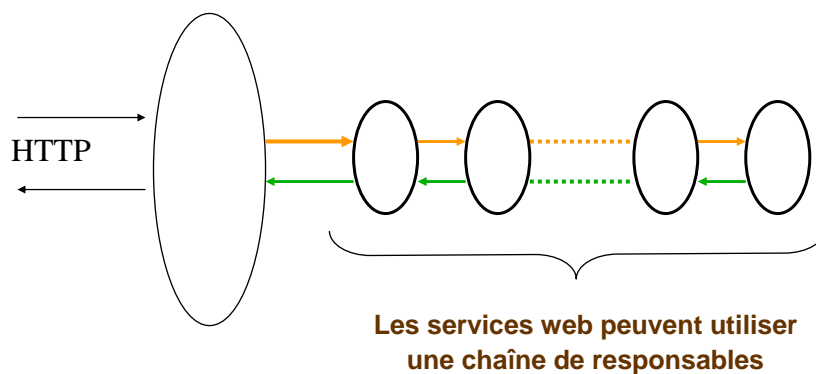
## Retour sur le serveur Web



ESIEE

67

## Retour sur le serveur Web 2ème épisode



ESIEE

68

## Acquisition/traitement

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

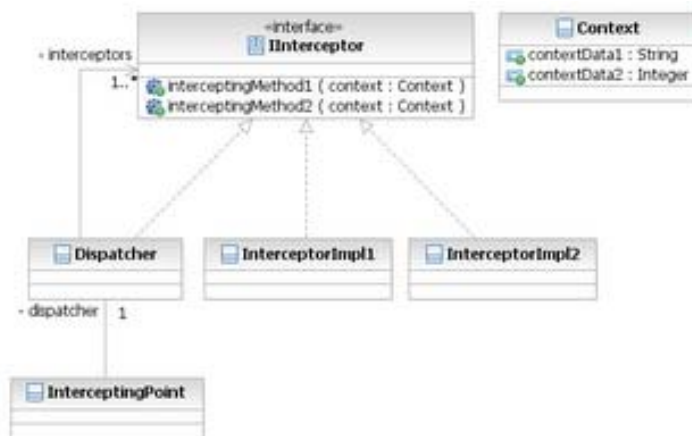
    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {
                    handleRequest(connection);
    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



ESIEE

69

## Interceptor : une variante « Chain of Responsibility »



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

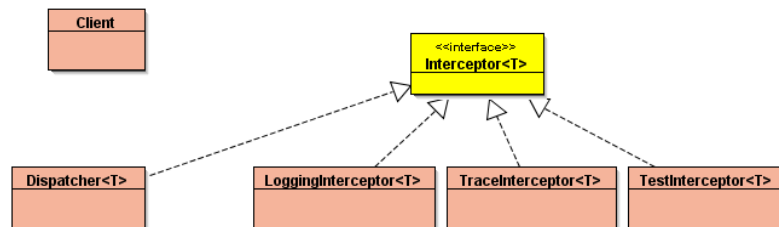
<http://longbeach.developpez.com/tutoriels/EJB3/Interceptors/>

<http://homepage.mac.com/cewcew/edu/cmsi688/InterceptorFilters.ppt>

ESIEE

70

## Une implémentation légère



ESIEE

71

## 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;
    }
}
```

ESIEE

72

## Dispatcher<T>

```
public class Dispatcher<T> implements Interceptor<T>{
    private Interceptor<T> interceptors[];
    public Dispatcher(final Interceptor<T>... interceptors){
        this.interceptors = interceptors;
    }

    // public Dispatcher(final Class<? extends Interceptor<T>>... interceptors) throws
    // InstantiationException, IllegalAccessException{ syntaxe préférée 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){
            T out = in;
            for(Interceptor<T> interceptor : interceptors){
                try{
                    out = interceptor.invoke(out);
                }catch(Exception e){}
            }
            return out;
        }
    }
}
```

ESIEE

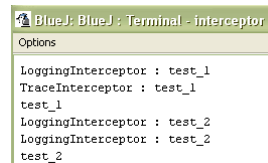
73

## 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
```

ESIEE

74