
Concurrence en java Patrons Singleton (revisit ) Cha ne de responsabilit s

jean-michel Douin, douin au cnam point fr
version : 29 Septembre 2009

Notes de cours

ESIEE

1

Sommaire pour les Patrons

- **Classification habituelle**

- **Cr ateurs**

- **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 ...

- **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 visitée
- **Composite, Interpreter, Visitor, Decorator, ...**

ESIEE

3

Sommaire

- **Les bases, une introduction**
 - java.lang.Thread
 - start(), run(), join(),...
 - java.lang.Object
 - wait(), notify(),...
 - le pattern Singleton revisité
 - java.util.Collections
- **les travaux de Doug Lea**
 - Devenus java.util.concurrent.*
- **Deux patrons**
 - Singleton revisité, Chain of responsibility
- **Patrons pour la concurrence (*une autre unité*)**
 - Critical Section, Guarded Suspension, Balking, Scheduler, Read/Write Lock, Producer-Consumer, Two-Phase Termination

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 Lea, <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>

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

ESIEE

5

Exécutions concurrentes

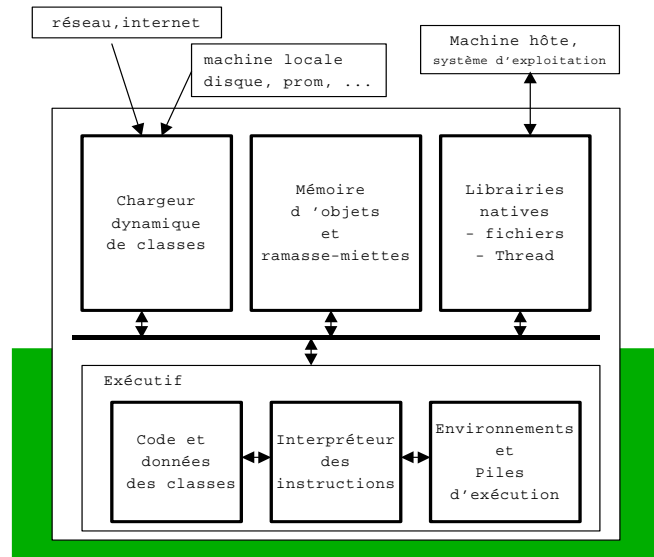
- **Les *Threads* Pourquoi faire ?**
- **Entrées sorties non bloquantes,**
- **Alarmes, Réveil, Déclenchement périodique,**
- **Tâches indépendantes,**
- **Algorithmes parallèles,**
- **Modélisation d'activités (intrinsèquement) parallèles**
- **Méthodologies**
- ...

ESIEE

6

Prémisse, Une JVM : un exécutif, un processeur

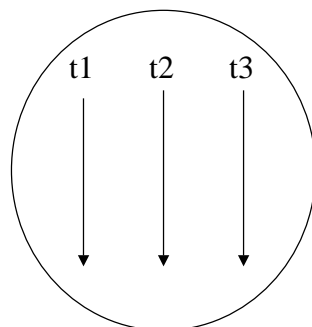
Java
Virtual
Machine



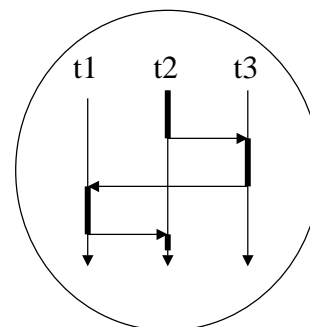
ESIEE

7

Contexte : Quasi-parallèle

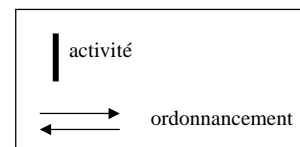


vue logique



vue du processeur

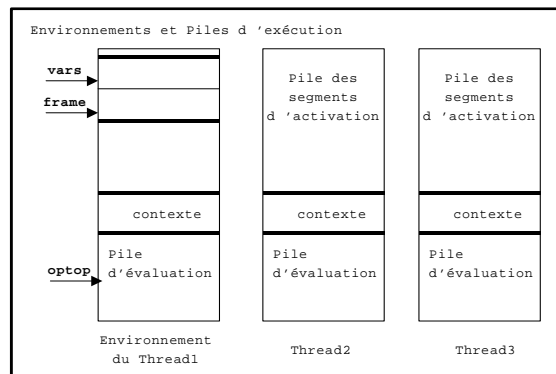
- Plusieurs *Threads*
- Un seul processeur abordé dans ce support



ESIEE

8

Soit Thread et « JVM virtuelles »



- Une pile par Thread... ou processus léger

ESIEE

9

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...)
- **Comportement : « Exécution » du processus**
 - `unThread.start();`
 - éligibilité de UnThread
 - ensuite l'ordonnanceur choisit unThread et exécute la méthode run()
 - `unThread.run();`
 - instructions associée à unThread

ESIEE

10

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

11

Remarques sur l'exemple

Note : 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 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
```

- premier constat à la visualisation de cette trace :
 - il semble que l'on ait un Ordonnanceur de type tourniquet, ici sous windows

ESIEE

12

La classe java.lang.Thread

- Quelques méthodes

Les constructeurs publics

- `Thread();`
- `Thread(Runnable target);`

...

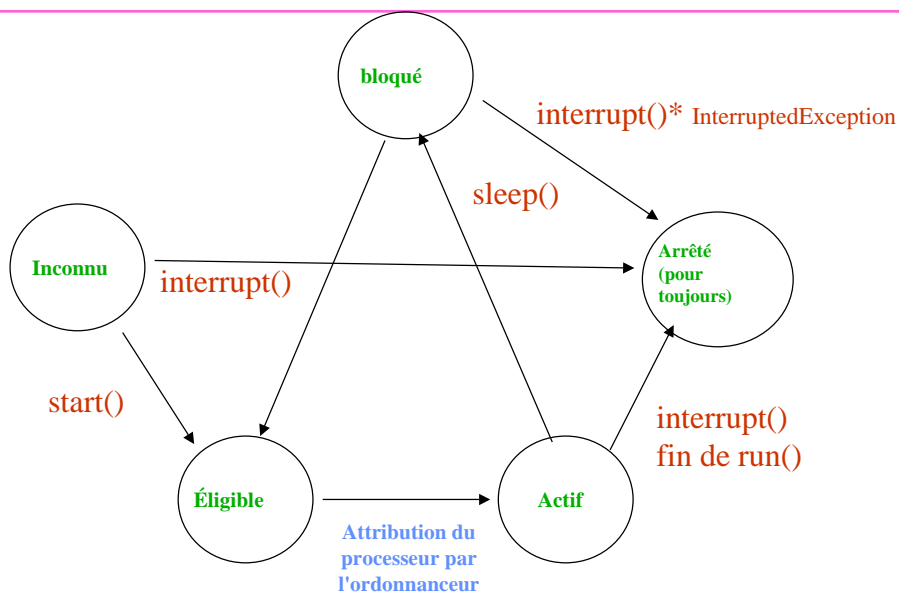
Les méthodes publiques

- `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();` // celui qui a le processeur

ESIEE

13

États d'un Thread



- États d'un Thread une 1ère approche

ESIEE

14

L'exemple initial revisité

```
public class T extends Thread {
    public void run(){
        while(!this.interrupted()){           // test du statut « a-t-il été interrompu » ?
            System.out.println("dans " + this + ".run");
        }
    }
}

public class Exemple {

    public static void main(String[] args) throws InterruptedException{
        T t3 = new T();
        t1.interrupt();
        t2.start(); t2.interrupt();           // t2.interrompu est vrai

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

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

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)
            }
        }
    }
}

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();
    }
}
```

*Note : L'appel de **interrupt** lève une exception dans le contexte du Thread Si cela se produit pendant l'exécution du wait, en dehors idem transparent précédent*

ESIEE

16

Le constructeur Thread (Runnable r)

La syntaxe habituelle avec les interfaces

```
public class T implements Runnable {
```

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

```
        ....
```

```
    }
```

```
}
```

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

```
public class Exemple{
```

```
    public static void main(String[] args){
```

```
        Thread t1 = new Thread( new T());
```

```
        t1.start();
```

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()*
- 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, car non sûr

ESIEE

18

Arrêt mais en attendant 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

- **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 ...
- 1) Règles, style d'écriture
- 2) `java.util.concurrent` à la rescousse
 - Bien meilleur

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

ESIEE

21

Un style possible d'écriture (2)...

Avec 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");
            }
    }
}
```

ESIEE

22

L'interface Executor, la classe ThreadExecutor

- 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();
    }
}
```

ESIEE

23

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);
}
```

ESIEE

24

Intermède

- **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 :**

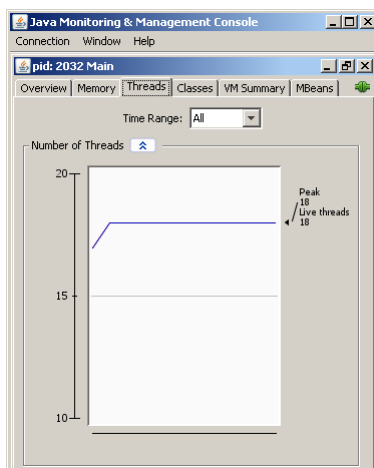
```
public class Main{
    public static void main(String[] args){
        for(int i = 0; i < 5; i++){
            new T(" dans_une_instance_de_T_"+i);
        }
    }
}
```

```
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_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

25

Fin de l'intermède JMX



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

ESIEE

26

Accès au ressources/synchronisation

Moniteur de Hoare 1974

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

// méthodes extraites 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()

ESIEE

27

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

28

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 instance de la classe Ressource

ESIEE

29

Accès « protégé » aux variables de classe

- **private static double valeur;**

```
synchronized( valeur){

}

```

Attention aux risques d'interblocage ...

ESIEE

30

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* et uniquement !

ESIEE

31

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){ throw new RuntimeException();}
        valeurEcrit = false;
        return valeur;
    }

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

Plus simple ?

ESIEE

32

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(!valeurEcrit)
        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();}

        valeurEcrit = false;
    return valeur;
}
```

Plus simple ? Sûrement !

ESIEE

33

Synchronisation lire si écrit et écrire si lu

- Facile ? ...
-
- À faire sur papier ... en fin de cours ...
- Programmation de trop bas-niveau ?

ESIEE

34

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

35

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 ?**
- **Quelles conséquences si notifyAll ?**
 - test de la variable condition
 - `while(!valeurEcrit) wait() ?` Au lieu de `if(!valeurEcrit) wait()`
- **Encore une fois, seraient-ce des mécanismes de « trop » bas-niveau ?**
 - À suivre...
- **Plus simple ...?**

ESIEE

36

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

37

java.util.concurrent.SynchronousQueue<E>

- **java.util.concurrent**

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

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

```
Timer producer = new Timer(true);
producer.schedule(new TimerTask(){
    private Integer i = new Integer(1);
    public void run(){
        try{
            queue.put(i);
            i++;
        }catch(InterruptedException ie){}
    }},
    0L,
    500L);
```

ESIEE

38

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();
```

ESIEE

39

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

40

java.util.concurrent

- `java.util.concurrent.LinkedBlockingQueue<E>`
- `java.util.concurrent.PriorityBlockingQueue<E>`
- *Trop simple !*

ESIEE

41

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`.

ESIEE

42

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

43

Le Future, Une requête HTTP sans attendre...

- **Au début de ce support**
 - Une requête + join + résultat

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

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

44

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

45

ExecutorService, submit

```
public static void main(String[] args) throws Exception{
    ExecutorService executor =
        Executors.newSingleThreadExecutor();
    Future<String> res =
        executor.submit(new RequeteHTTPWithFuture(args[0]));

    // instructions ici

    try{
        System.out.println(res.get());
    }catch(InterruptedException ie){
        ie.printStackTrace();
    }catch(ExecutionException ee){
        ee.printStackTrace();
    }
}
```

ESIEE

... Simple ...

46

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

47

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

48

## Conclusion intermédiaire

---

- **Mécanisme de bas niveau,**
  - Thread, wait, notify, ...

Nous avons

- **java.util.concurrent et quelques patrons,**

Nous préférons

ESIEE

49

## java.util.concurrent : il était temps

---

- . 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/javaonline/2004/corej2se/TS-1358.pdf>

ESIEE

50

## Conclusion

---

- **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... un autre cours...
- **Il nous reste 2 patrons pour un dénouement heureux**

ESIEE

51

## 2 patrons

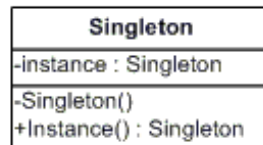
---

- **Le Singleton** revisité pour l'occasion
  - Garantit une et une seule instance d'une classe
- **Architectures logicielles : un bon début**
  - Le couple Acquisition/Traitement
- **Le patron Chaîne de responsabilités**

ESIEE

52

## UML et la patron Singleton



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

ESIEE

53

## Le Pattern Singleton, revisité

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

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

private Singleton(){
}
}
```

ESIEE

54

## Préambule, chain of responsibility

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

### 1) Acquisition

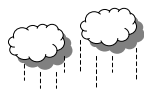
### 2) Traitement

- 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**

ESIEE

55

## Exemple : capteur d'humidité et traitement



DS2438, capteur d'humidité

Acquisition



Traitement :

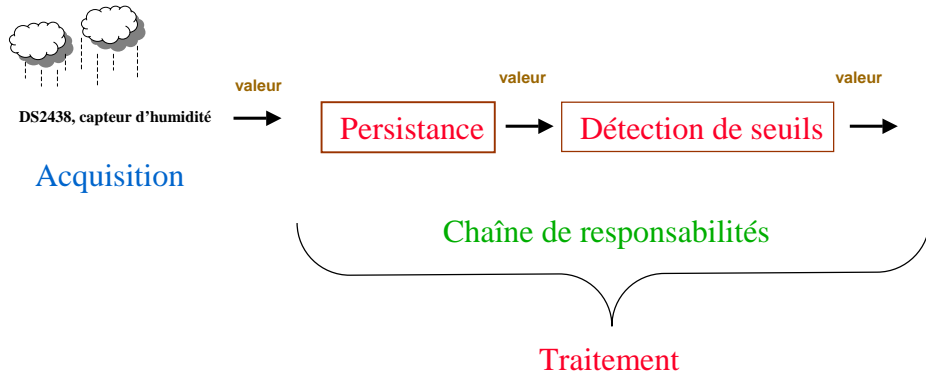
- Persistance
- Détection de seuils
- Alarmes
- Histogramme
- ...

- **Acquisition / traitement, suite**

ESIEE

56

## Acquisition + Chaîne de responsabilités

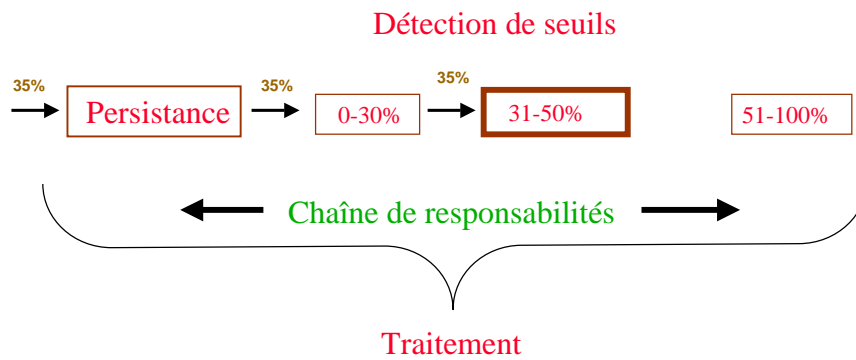


**Un responsable décide d'arrêter la propagation de la valeur**

ESIEE

57

## Chaîne de responsables



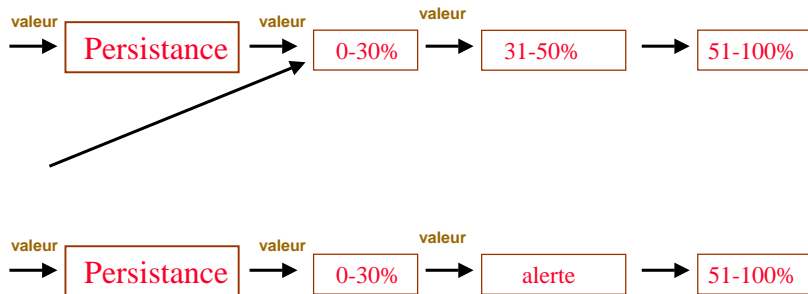
- Un responsable enfin !
- Arrêt de la propagation de la valeur

ESIEE

58

## Chaîne dynamique

### Plusieurs entrées

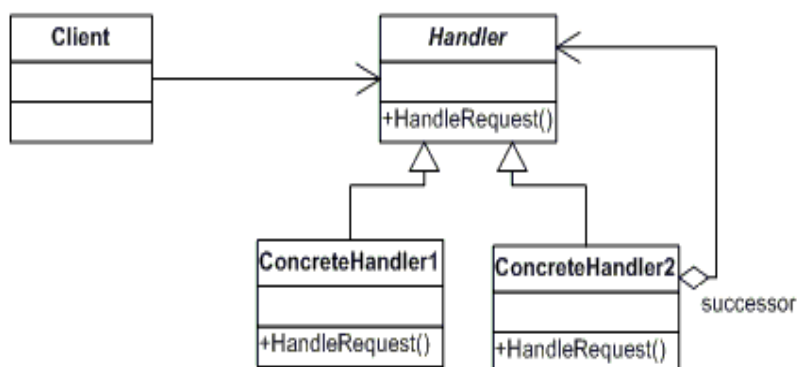


- Ajout, retrait de responsables en cours d'exécution
- Plusieurs entrées, plusieurs niveaux ...

ESIEE

59

## 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

60

## 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;
 }

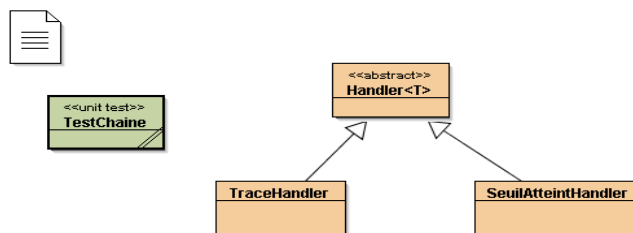
 // gestion de la liste chaînée
 public void setSuccessor(Handler<V> successor){this.successor=successor;}
 public Handler<V> getSuccessor(){return this.successor;}

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

ESIEE

61

## 2 classes concrètes, pour le moment



TraceHandler  
+ SeuilAtteintHandler

- Soit la (petite) chaîne : TraceHandler → SeuilAtteintHandler

ESIEE

62

## class TraceHandler : 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 ici propagée
 // quoiqu'il arrive ...
 return super.handleRequest(value);
 }
}
```

ESIEE

63

## class SeuilAtteintHandler : 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; // arrêt de la propagation, je suis responsable
 }
 // sinon l'information, « value » est propagée
 return super.handleRequest(value);
 }
}
```

ESIEE

64

## 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

65

## Démonstration avec Bluej

---

ESIEE

66

## 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

67

## Un autre responsable

```
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);
 }
}
```

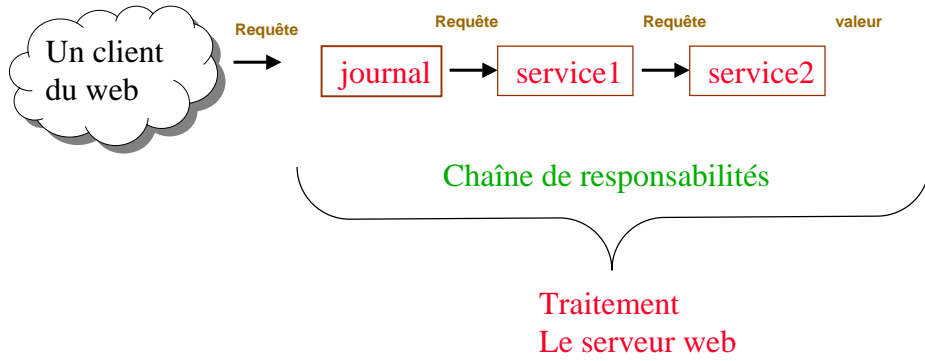
TraceHandler → ValeurNulleHandler → SeuilAtteintHandler(50) → ..

↓ Arrêt de la propagation

ESIEE

68

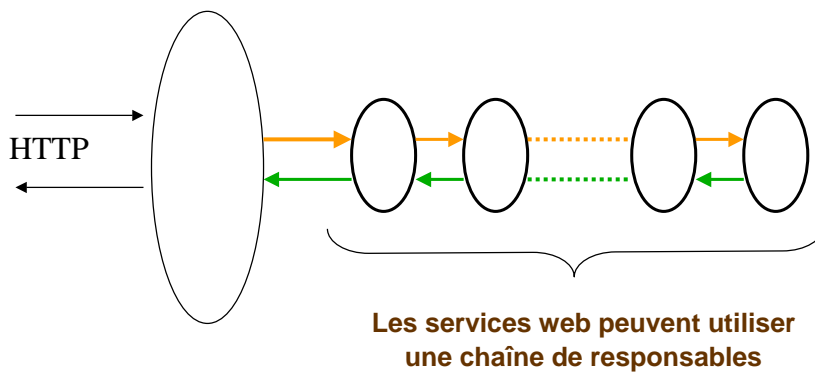
## Retour sur le serveur Web en une page



ESIEE

69

## Retour sur le serveur Web 2ème épisode



ESIEE

70

## 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

71

## Annexe

- Un SerialExecutor ... sources extraits de Sun

ESIEE

72

## L'interface Executor, le retour

- **SerialExecutor un décorateur d'executor...**
  - sérialise l'exécution vers un second Executor

```
public class SerialExecutor implements Executor {
 private final Queue<Runnable> tasks = new
 LinkedBlockingQueue<Runnable>();

 private Runnable active;
 final Executor executor;

 public SerialExecutor(Executor executor) {
 this.executor = executor;
 }
}
```

ESIEE

73

## SerialExecutor

```
public synchronized void execute(final Runnable r) {
 tasks.offer(new Runnable() {
 public void run() {
 try { r.run();
 } finally { scheduleNext();}
 }
 });
 if (active == null) {
 scheduleNext();
 }
}

protected synchronized void scheduleNext() {
 if ((active = tasks.poll()) != null) {
 executor.execute(active);
 }
}
}}
```

ESIEE

74