




**COURS PARALLELISME**

Méthodes de programmation parallèle en Java

*un zeste de génie logiciel pour le parallélisme*

Jean-Louis PAZAT  
IRISA / INSA

Institut National des Sciences Appliquées  
École nationale d'ingénieurs

---

---

---

---

---

---

---

Plan

- But de ce cours
- Activités
  - créer une activité
  - passer des paramètres à une activité
  - terminer une activité
- Concevoir un objet dans un cadre parallèle
  - sûreté et vivacité
  - immutability (stateless /immutable objects)
  - verrouillage - découpage - confinement
  - prise en compte de l'état
- Conclusion

---

---

---

---

---

---

---

But de ce cours

- Notions « système »
  - « mécanique » : comment ça marche dedans
- Notions « langage »
  - « utilisation » : comment ça s'écrit, où sont les boutons
- Notions « construction de logiciels »
  - « bons usages » : comment « bien » écrire
    - vers des Design Patterns

---

---

---

---

---

---

---

### Activités :

#### Créer une activité (Thread /runnable)

- Hériter de Thread :

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Hello World");  
    }  
}  
MyThread mythread = new MyThread();  
mythread.start();
```

---

---

---

---

---

---

---

#### Créer une activité (Thread /runnable)

- Utiliser un "Runnable" :

- l'interface runnable n'a qu'une méthode (run)

```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello World");  
    }  
    ... //autres méthodes  
}  
// dans le main()  
Runnable myrun = new MyRunnable();  
Thread t1 = new Thread(myrun);  
t1.start();
```

---

---

---

---

---

---

---

#### Créer une activité (Thread /runnable)

- Hériter de Thread ou implémenter Runnable ?

- Thread : abstraction d'un "acteur"
- Runnable : abstraction d'un "travail"
- nouvelle partie de travail : runnable
- nouveau type d'action : Thread

---

---

---

---

---

---

---

Créer une activité à démarrage automatique

```
Class AutoRun implements runnable {
    public AutoRun(){
        new Thread(this).start() // danger !
    }
    public void run() {
        System.out.println("Démarrage
        Auto");
    }
}
// dans le main()
AutoRun t1 = new AutoRun();
```

---

---

---

---

---

---

---

Créer une activité à démarrage automatique

- Héritage ?
  - Impose que les sous classes soient aussi à démarrage automatique
  - si pas d'héritage prévu spécifier final
  - comment permettre aux sous classes de choisir démarrage automatique ou non ?

---

---

---

---

---

---

---

Créer une activité à démarrage automatique

```
Class AutoOrManRun implements runnable {
    protected Thread _me;
    protected AutoOrManRun(boolean auto) {
        _me = new Thread(this);
        if (auto) _me.start();
    }
    public void run() {
        System.out.println("running");
    }
    public AutoOrManRun() {
        this(true);
    }
}
Class ManRun extends AutoOrManRun {
    public ManRun() { super(false); }
    public getThreadRef() { return _me; }
```

---

---

---

---

---

---

---

Créer une activité à démarrage automatique

- Protéger run
  - run est obligatoirement public
  - n'importe qui peut l'invoquer directement !
- Comment ?
  - Utiliser le Thread id
  - Encapsuler les runnables

---

---

---

---

---

---

---

Créer une activité à démarrage automatique  
utilisation du Thread id

```
Class AutoRun implements runnable {
    private Thread _me;
    public AutoRun(){
        _me = new Thread(this);
        _me.start();
    }
    public void run() {
        if (_me == Thread.currentThread())
            System.out.println("Autorunning");
    }
}
// dans main()
Autorun t1 = new AutoRun(); // affiche
Autorunning
t1.run(); // n'affiche rien
```

---

---

---

---

---

---

---

Créer une activité à démarrage automatique  
encapsuler les runnables

```
class AutoRun2 {
    class InnerRunnable implements runnable {
        public void run() {
            System.out.println("Autorunning");
        }
    }
    public AutoRun2(){
        new Thread(new InnerRunnable()).start();
    }
}
// dans main()
Autorun2 t1 = new AutoRun2();
t1.run(); // n'est pas possible !
```

---

---

---

---

---

---

---

### passer des paramètres à une activité

- Utile
- Contrairement a `main(String[]args)` `run()` ne prend pas de paramètres
- solutions
  - passer les paramètres à la création du runnable
  - ou
  - utiliser une “innerclass” pour lancer les activités

---

---

---

---

---

---

---

### passer des paramètres à une activité

```
class Worker implements Runnable {
    private int _param1, _param2;
    private Thread _me;
    public Worker(int p1, int p2) {
        _param1 = p1; _param2 = p2;
        _me = new Thread(this);
        _me.start();
    }
    public void run() {
        System.out.println("p1 =" + _param1);
        System.out.println("p2 =" + _param2);
    }
}

class Server {
    public void addJob(int p1, int p2){
        new Worker(p1,p2)
    }
}

//dans main
Server myServer = new Server();
myServer.addJob(10, 20);
```

---

---

---

---

---

---

---

### passer des paramètres à une activité

```
class Server {
    public void addJob(final int p1, final int p2){
        Runnable r = new Runnable() {
            public void run() {
                System.out.println("p1 =" + p1);
                System.out.println("p2 =" + p2);
            }
        }
        new Thread(r).start();
    }
}

(final est nécessaire pour que les "innerclass" puissent y accéder)
```

---

---

---

---

---

---

---

Terminer une activité (Thread)

- Ce n'est pas une bonne idée !
  - Asynchronisme
    - on ne sait pas dans quel état est le thread
  - “Libre arbitre”
    - un thread peut ignorer une demande de terminaison
- Elle a été abandonnée ...
  - `stop()`, `suspend()`, `resume()` abandonnés
  - `destroy()` jamais implémenté

---

---

---

---

---

---

---

---

Interrompre une activité (Thread)

- Juste un changement d'état
  - `void interrupt()`
    - fait passer l'état du Thread a `interrupted`
    - test : `isInterrupted()`
    - test-and-reset: `interrupted()`
    - permettent de décider de lever l'interruption `InterruptedException`
    - levée par `sleep()`, `join()`, `wait()`

---

---

---

---

---

---

---

---

Concevoir un objet dans un cadre parallèle

- Ce qui est important pour le parallélisme
  - c'est bien de pouvoir faire plusieurs choses à la fois
  - c'est difficile de bien faire plusieurs choses à la fois
    - Il faut garantir qu'il ne se passe rien de mauvais (Sûreté *Safety*)
    - Il faut garantir qu'il se passe quelque chose (Vivacité *Liveness*)

---

---

---

---

---

---

---

---

Garantir la sûreté (Safety)

- Sûreté :
  - puis je faire cette action maintenant
  - la JVM garanti au niveau bas :
    - pas de conflit d'accès mémoire
    - atomicité des actions sur les types de base 32bits
  - A vous de garantir :
    - les objets ne sont accessibles que lorsque leur état est cohérent (*consistent*)

synchronisation

---

---

---

---

---

---

---

Garantir la vivacité (Liveness)

- Vivacité :
  - vérifier la disponibilité des services
    - toute méthode appelée doit pouvoir être exécutée au bout d'un temps fini (*eventually executes*)
  - Assurer la progression des activités
    - attention aux deadlocks, blocages infinis (lockout)
    - assurer l'équité (fairness)
    - assurer la tolérance aux fautes
    - se prémunir des interférences extérieures

communication

---

---

---

---

---

---

---

Concevoir un objet dans un cadre parallèle

- Immutability (stateless /immutable objects)
- Locking (monitors)
- Splitting objects
- Containment
- State dependence

---

---

---

---

---

---

---

#### Stateless : objets sans état interne

```
class StatelessAdder {  
    int addOneTo(int i){return i+1;}  
    int addTwoto(int i){return i+2;} }  
}
```

- aucun souci avec le parallélisme !
  - Pas d'état interne
    - pas besoin d'invariants : pas de pb de sûreté
  - Exécutions concurrentes possibles
    - sans limitation : pas de pb de vivacité
  - Objet passif
    - ne nécessite pas de création de threads
    - pas besoin de définir de règle d'utilisation

---

---

---

---

---

---

---

#### Immutable : objets sans état interne modifiable

```
class ImmutableAdder {  
    private final int _offset;  
    public ImmutableAdder(int offset){  
        _offset = offset;  
    }  
    int add(int i){return i+_offset;}  
}
```

- aucun souci avec le parallélisme !
  - état interne fixé à l'initialisation
    - pas besoin d'invariants : pas de pb de sûreté
  - correspond bien aux TAD "fermés"
    - java.lang.String, java.lang.Integer, ...

---

---

---

---

---

---

---

#### Moniteurs simples (sans wait/notify)

- Objet
  - dont aucun attribut n'est visible
  - dont toutes les méthodes sont `synchronized`
- Toujours sûr (safety)
  - un seul Thread actif à la fois !
- Pb : assurer la vivacité du système !

---

---

---

---

---

---

---



Un cas d'école : coder une localisation

- Spec. Fonctionnelle
  - coder une localisation par ses coordonnées x,y
  - permettre son déplacement
  - permettre de voir les coordonnées x,y
- Garanties de sûreté
  - la localisation doit être toujours valide
  - un déplacement doit être entièrement exécuté avant qu'un nouveau déplacement soit effectué
  - les coordonnées (x,y) vues doivent toujours être valides

---

---

---

---

---

---

---

---

Un cas d'école : coder une localisation  
solution 1 : classe modifiable/moniteur

```

class LocationV1 {
    private long _x, _y; // représentation état
    public LocationV1(long x, long y){_x =x; _y =y;}
    synchronized long x(){return _x;}
    synchronized long y(){return _y;}
    synchronized void moveBy(long dx, long dy){
        _x = _x +dx; _y = _y+dy;}
    }

```

- OK ?
  - Représentation de l'état protégée
  - moveBy atomique
  - x() et y() rendent toujours des valeurs valides ...

---

---

---

---

---

---

---

---

Un cas d'école : coder une localisation  
solution 1 : classe modifiable/moniteur

- Mais ...
 

```

x = loc.x(); .....; y = loc.y()
||
..... ;loc.moveBy(1,2); .....

```

 donne un résultat incorrect
- POURQUOI ?
  - x() et y() ne sont pas une paire (x,y)
    - il faut séparer le changement de représentation de la classe Location (splitting)

---

---

---

---

---

---

---

---

Un cas d'école : coder une localisation  
solution 1 : classe modifiable/moniteur

```
class coord{//immutable
    private final long _x, _y;
    public coord(long x, long y){_x = x; _y = y}
    public long x(){return _x;}
    public long y(){return _y;}
}

class LocationV2 {
    private coord _xy; // représentation état
    public LocationV2(long x, long y){_xy = new coord(x,y);}
    synchronized coord xy(){return _xy;}
    synchronized void moveBy(long dx, long dy){
        _xy = new coord(_xy.x() +dx, _xy.y() +dy);}
}

• OK !
```

---

---

---

---

---

---

---

---

Autres utilisation du *splitting*

- Idée commune
  - n'exposer que des états cohérents
- Utilisations
  - réduire la contention sur les locks
    - associer un sous ensemble de l'état à un lock
  - réduire la contention sur les wait()
    - idem
  - permettre de revenir en arrière (rollback)
    - isoler différents états dans des objets séparés

---

---

---

---

---

---

---

---

Un cas d'école : coder une localisation  
solution 3 : confinement (*containment*)

```
import java.awt.Point;

class LocationV3 {
    private final Point _xy;
    public LocationV3(int x, int y){_xy = new Point(x,y);}
    synchronized Point xy(){
        return new Point(_xy); // copie !
    }
    synchronized moveBy(int dx, int dy){
        _xy.translate(dx,dy);
    }
}

• les champs x,y de la classe Point sont public et modifiables
    – il faut les protéger !
    – On rend des copies
```

---

---

---

---

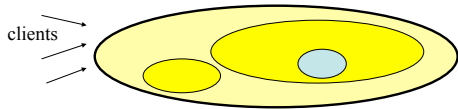
---

---

---

---

### Technique du confinement



- Confinement strict == îlots d'objets
  - utilisé dans les composants (EJB)
  - mais peut aussi être récursif
- Encapsulation de code natif possible
- Impose
  - pas de comm entre les objets internes et l'extérieur
  - pas de lien depuis l'extérieur vers des objets internes
- difficile à assurer et à vérifier

---

---

---

---

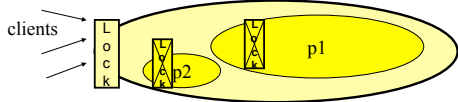
---

---

---

---

### Technique du confinement : verrouillage



- Verrouillage local inadapté :
  - utilisation de p1 et appel vers p2
  - ||
  - utilisation de p2 et appel vers p1
  - le client n'a pas connaissance des appels internes
  - la technique des ressources ordonnées n'est pas tj applicable
- Verrouillage hiérarchique :
  - toute partie visible utilise le verrou du container global
  - verrouillage interne ou externe possibles

---

---

---

---

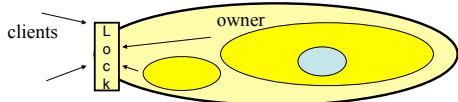
---

---

---

---

### Technique du confinement : verrouillage



- Verrouillage interne
  - les objets utilisent le verrou du container
  - évite les deadlocks entre appels de différentes sous parties

```

class Part {
    protected Container _owner;
    public Container owner() {return _owner;}
    void unprotectedAction() { . . . }
    public void protectedAction() {
        synchronized(owner()) {unprotectedAction();}
    }
}
  
```

---

---

---

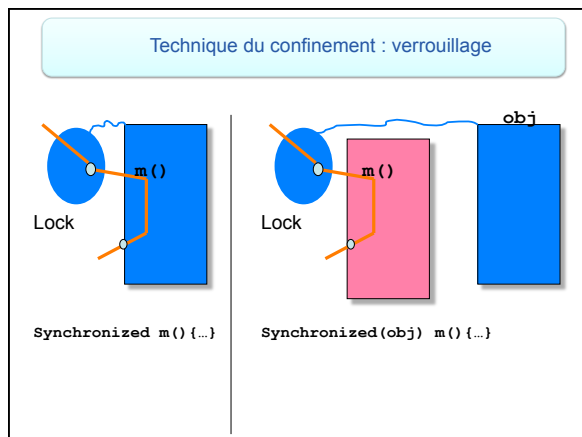
---

---

---

---

---




---

---

---

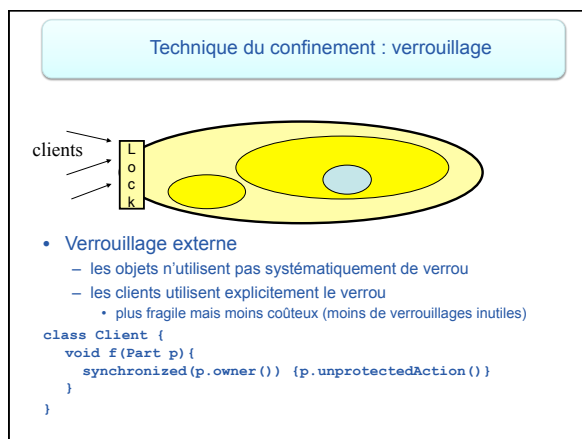
---

---

---

---

---




---

---

---

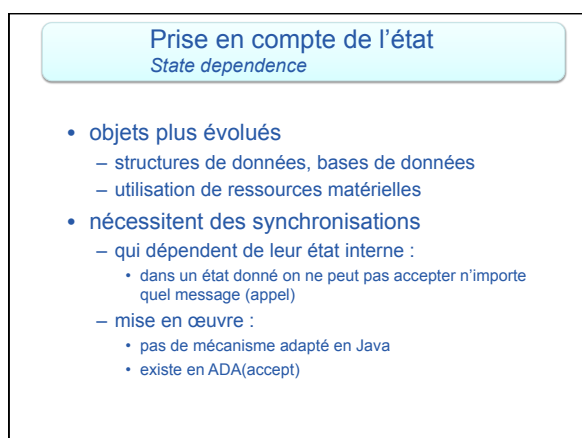
---

---

---

---

---




---

---

---

---

---

---

---

---

State dependence

- Etapes de spécification
  - choisir une politique
    - qui assure que les actions sont réalisées seulement si l'état interne le permet
  - Définir les interfaces et les protocoles
    - qui reflètent cette politique
  - S'assurer que les objets
    - peuvent évaluer l'état interne pour mettre en œuvre la politique

---

---

---

---

---

---

---

State dependence : politiques

Blind action :	sans garantie ( <i>on verra bien</i> )
Inaction :	ignorer la requête dans certains états
<b>Balking</b> :	rapporter l'échec de l'invocation (refuser)
<b>Guarding</b> :	bloquer jusqu'au changement d'état
<b>Trying</b> :	Essayer, si ça a marché ok sinon échec
Retrying :	Réessayer jusqu'à ce que ça marche
Timing out :	Attendre ou réessayer pendant un certain temps
Planning :	Faire en sorte que l'objet parvienne dans l'état désiré pour effectuer l'action

---

---

---

---

---

---

---

State dependence : Accès à l'état interne

- L'état interne
  - doit être explicitement représenté
  - doit être protégé
    - voire encapsulé
  - ne doit pas changer pendant l'exécution d'une méthode
  - test et modification de l'état
    - nécessitent des synchronisations

---

---

---

---

---

---

---

State dependence : interfaces

```

public interface IBoundedBuffer{//encore lui !
    int capacity();           //Inv: capacity() >0
    int count();              //Inv: 0<=count()<=capacity()
                                //Init: count()== 0
    void put(Object x);       //Pre: count() < capacity()
    Object get();             //Pre: count() > 0

```

- Insuffisant pour imposer la politique
  - peut la suggérer
    - exceptions (balking) : si une methode lève une exception/laquelle
    - avoir des methodes différentes pour avoir des plitiques différentes
    - annotations (Invariants, preconditions)

---

---

---

---

---

---

---

---

State dependence : balking (refuser)

- Vérifier l'état à l'entrée de la méthode
  - échouer immédiatement si on n'est pas dans un état compatible avec l'action
- C'est au client de gérer l'échec
- non spécifique au parallélisme
  - applicable aux structures de données vues en 3e année en séquentiel

---

---

---

---

---

---

---

---

State dependence : balking (refuser)

```

Public class BalkingBoundedBuffer
implements IBoundedBuffer {
    protected int _count;
    protected final int CAPACITY;
    protected final Object[] _data;
    protected int _head, _tail;
    public BalkingBoundedBuffer(int capacity){
        CAPACITY = capacity;
        _data = new Object[CAPACITY];
        _head = _tail = _count = 0;
    }

```

---

---

---

---

---

---

---

---

```

public int count() {return _count;}
public int capacity(){return CAPACITY;}
public synchronized void put(Object x)
    throws FailureException{
    if (_count == CAPACITY)
        throw new FailureException("full");
    else {
        _count++; _data[_tail] = x;
        _tail = (_tail +1) % CAPACITY;
    }
}
public synchronized Object get()
    throws FailureException{
    if (_count == 0)
        throw new FailureException("empty");
    else {
        Object x = _data[_head];
        _count--; _head = (_head +1) % CAPACITY;
        return x ;
    }
}
}

```

---

---

---

---

---

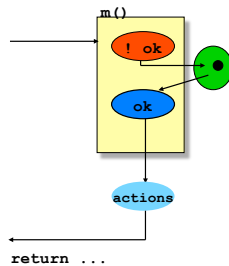
---

---

---

#### State dependence : guarding (attendre)

- Vérifier l'état à l'entrée de la méthode
  - attendre si on n'est pas dans un état compatible avec l'action
- Il faut veiller à la vivacité du système
  - un autre Thread devra faire une action pour changer l'état et vous réveiller
- spécifique au parallélisme
  - n'a pas de sens en séquentiel




---

---

---

---

---

---

---

---

#### State dependence : guarding (attendre)

- Mécanismes d'attente :
  - attente active
    - while (!condition);
    - nécessite du partage de temps ou un multiprocesseur
      - aucun moyen de vérifier ça en Java
    - utiliser plutôt while (!condition) sleep(n);
  - blocage/réveil
    - voir cours précédents

---

---

---

---

---

---

---

---

#### Politiques optimistes : Trying (transactionnel optimiste)

- Sauvegarder les états (versions)
- Regrouper les changements d'états en opérations atomiques
- A l'entrée de la méthode :
  - sauvegarder l'état courant
  - effectuer l'action sur une copie de l'état courant
- commit (rendre l'action effective)
  - si l'action réussit et si l'état n'a pas été modifié

---

---

---

---

---

---

---

#### Politiques optimistes : Trying (transactionnel optimiste)

- Si commit pas possible :
  - échouer ou recommencer
    - échec "propre" : l'état précédent est restitué
    - recommencer = attente active
- Applicable si
  - pas d'entrées/sorties
    - sauf si elles peuvent être annulées de manière sûre
  - actions réversibles
    - toutes les actions doivent pouvoir être défaites (undo)
- intéressant si
  - pas trop souvent de conflits

---

---

---

---

---

---

---

#### Conclusion

- Pas de méthode miracle
  - utiliser des objets dans un cadre parallèle
    - pas de pb si *stateless*, sinon il faut implémenter une politique de contrôle
  - choisir entre
    - pessimiste: protéger / optimiste: être capable de défaire une action
  - dépend fortement du contexte applicatif

---

---

---

---

---

---

---



## Bibliographie

- Bibliographie :
  - David Holmes, Doug Lea  
Tutorial M2 + M5  
TOOLS EUROPE 2000
  - Doug Lea  
Concurrent Programming in Java, 2nd edition  
The Java Series ... from the source

---

---

---

---

---

---

---