JAVA

Introduction à la programmation objet avec java – 4/4

SOMMAIRE

- Héritage multiple et interface (15mn)
- Les Threads(60mn)

Héritage multiple et interface (1)

- objectif
  - avoir une classe qui hérite de plusieurs classes
  - Complexe et controversé (exemple en C++)
- Réponse avec Java : Les interfaces
  - Une interface correspond à une classe où toutes les méthodes sont abstraites.
- Principe:
  - héritage multiple de classe interdit entre classes
  - Une interface peut hériter (extends) de plusieurs interfaces.
  - Une classe peut implémenter (implements) une ou plusieurs interfaces tout en héritant (extends) d'une classe.
  - Une classe qui implémente une interface doit implémenter toutes les méthodes de l'interface

Héritage multiple et interface (2)

```
abstract class Forme { public abstract double perimetre(); }

interface Affichable { void affiche(); }

class Cercle extends Forme implements Affichable, Serializable {
    public double perimetre() { return 2 * Math.PI * r ; }
    public void affiche() {...}
}

class Rectangle extends Forme implements Affichable, Serializable {
    ...
    public double perimetre() { return 2 * (height + width); }
    public void affiche() {...}
}

...
Affichable[]affichables = {new Cercle(2), new Rectangle(2,3), new Cercle(5)};
for(int i=0; i< affichables.length; i++)
     affichables[i].affiche();
```

Un Thread ?

- Traduction littérale : « tâche » (ou fil d'exécution)
- Thread
  - Unité d'exécution individuelle qui fonctionne en parallèle d'autres Unités d'exécutions
  - Contrairement à un process, un thread partage l'espace mémoire avec le programme principal
  - Parallèle ?
    - Soit plusieurs processeurs et un OS qui gère plusieurs processeurs
    - Dans le cas d'un seul processeur, ceci revient à faire du time sharing, temps partagé
- cette notion n'est pas spécifique à Java!

Les Threads en Java

- La création d'une thread s'effectue via l'instanciation d'une classe `Thread` ou de toute classe qui hérite de `Thread`
- Pour écrire ce que le thread doit faire, il faut créer une méthode `run()` : cette méthode est l'équivalent de main() pour un programme
- La méthode `start()` de la classe `Thread` permet de démarrer le code du thread.
- L'appel de la méthode `start()` retourne immédiatement (détache l'execution du run())
- Une fois démarré, le thread execute le code de `run()` jusqu'à la sortie de la méthode.

Les Threads en Java

```
public class MaClasseThread extends Thread / implements
  Runnable {
   // Attributs
   // Constructeurs

   public MaClasseThread ( [Arguments] ) {
     [Attributs] = [Arguments] ;
   }
   public void run ( ) {
     try {
       // Instructions
     }
     catch (InterruptException ie) {
       // En cas d'interruption
     }
   }
 } // Fin de la classe
```

La classe java.lang.Thread (1)

- Cette classe représente un fil d ’execution
- 2 possibilités : soit hériter de `Thread` soit implémenter `Runnable`

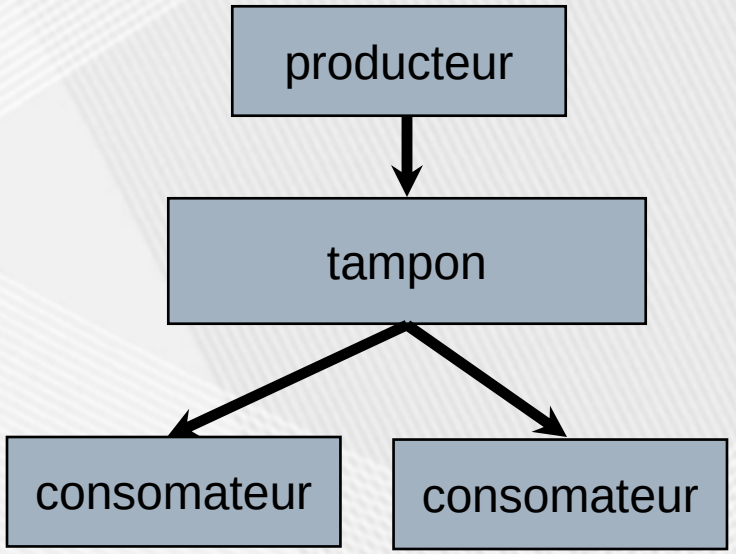
```
class C1 extends Thread
{
     public C1() { this.start(); }
     public void run() {...}
}

class C2 implements Runnable
{
     public C2() {Thread t = new Thread(this); t.start(); }
     public void run() {...}
}
```

La classe java.lang.Thread (2)

- Méthodes :
  - `void start()`
  - `void stop()`
  - `void suspend()`
  - `void resume()`
  - `static void sleep()`
  - `static void yield()`

Interaction entre Threads

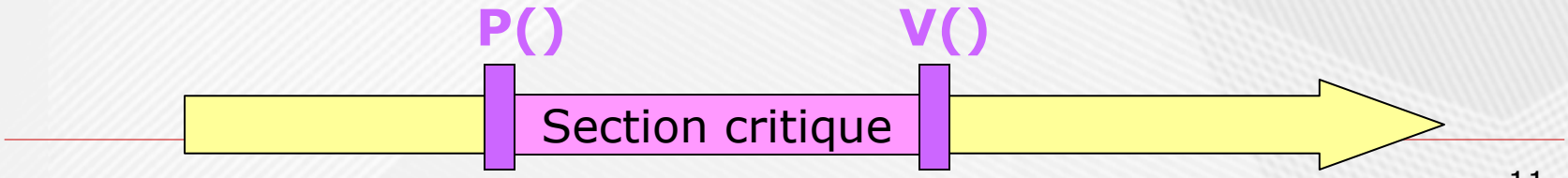
- Le classique producteur/consommateur
  - Comment être sur que 1 consomateur ne peut lire que quand le producteur à fini d ’ecrire ?
  - Comment être sur que 2 consomateur ne lisent la même chose ?
- Solutions:
  - pour “locker” les ressources partagées : **synchronized**
  - sections critiques
  - wait/notify



mot clef synchronised

- synchronized permet de synchroniser l'accès à une partie de code où à une méthode.

```
class Compte {
    synchronized void ajouter(int montant) {...}
    synchronized void retirer(int montant) {...}
}

class Client implements Runnable {
    Banque b;
    public Client(Banque b) {
        this.b = b;
        Thread t = new Thread(this); t.start();
    }
    public void run() { ... b.ajouter(100); ... b.retirer(10); ...}
}

Banque b = new Banque();
Client c1 = new Client (b); Client c2 = new Client(b);
```



wait/notify

- Chaque objet possède les méthode `wait()`, `notify()` et `notifyAll()`
- Dans une une partie `synchronized` d'un objet O :
  - `wait()` relache le verrou et se met en attente.
  - `notify()` reveille un *thread* en attente (fifo)
  - `notifyAll()` reveille tous les *threads* en attente

```
class MyThing {
     synchronized void waiterMethod() {...; wait(); ...}
     synchronized void notifyMethod() {...; notify(); ...}
     synchronized void anOtherMethod() {...}

}
```