



Université Mohamed V - Rabat
Ecole Nationale Supérieure d'Informatique
et d'Analyse des Systèmes

Programmation Objet Avancée

PROF. GUERMAH HATIM
EMAIL: GUERMAH.ENSIAS@GMAIL.COM

Chapitres

- Multithreading Java
- Sockets Java
- RMI et les objets distribués

Rappel : Concepts fondamentaux de l'orienté Objet

- Classe
- Objet
- Héritage
- Encapsulation
- Polymorphisme

Multithreading Java

Plan du cours

- Introduction :
- Processus/ Multitâches et Parallélismes / Multithreading
- Définition et Création d'un Thread Java
- Cycle de Vie
- Concurrency d'accès

Introduction

- Chaque application exécutée sur un ordinateur lui est associée un processus représentant ses activités.
- Chaque processus est associé un ensemble de ressources propres à lui comme l'espace mémoire, le temps CPU etc.



Introduction

Exemple d'application :

- l'interface graphique peut lancer un processus pour charger une image pendant qu'elle continue de traiter les événements générés par des actions de l'utilisateur
- une application serveur qui attend les demandes de connexions venant des clients peut lancer un processus pour traiter les demandes de plusieurs clients simultanément
- la multiplication de 2 matrices (m,p) et (p,n) peut être effectuée en parallèle par $m * n$ processus

Ecrire des programmes permettant d'effectuer plusieurs traitements, spécifiés distinctement les uns des autres, en même temps.

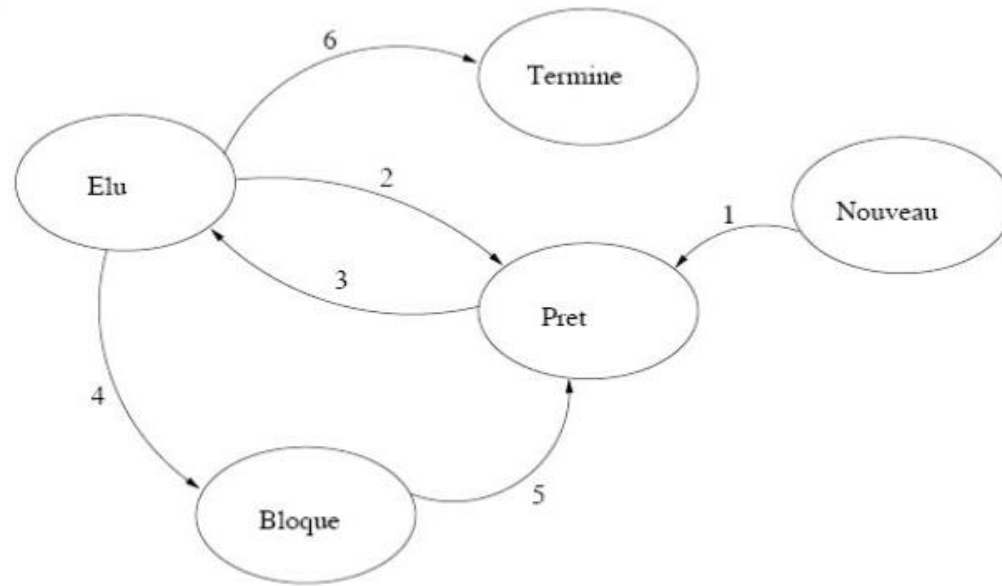
Multitâche et Parallélisme

- Multi-tâches : Exécution de plusieurs processus simultanément sans conflit.
- Parallélisme : pouvoir faire exécuter plusieurs tâches par un ordinateur avec plusieurs processeurs. Si on possède moins de processeurs que de processus à exécuter :
 - division du temps d'utilisation du processeur en tranches de temps (Time Slice)
 - attribution des tranches de temps à chacune des tâches de façon telle qu'on ait l'impression que les tâches se déroulent en parallèle : pseudo-parallélisme
- Utilité du Multitâche/ Parallélisme :
 - Amélioration des performances en répartissant les différentes tâches sur différents processeurs
 - profiter des temps de pause d'une tâche (attente d'entrées/sorties ou d'une action utilisateur) pour faire autre chose
 - réagir plus vite aux actions de l'utilisateur en rejetant une tâche longue et non-interactive dans un autre processus .

Processus

Un processus : une instance de programme qui est en train d'être exécutée sur un ou plusieurs processeurs sous le contrôle d'un système d'exploitation.

Un processus comprend un ensemble d'instructions + des données stockées en mémoire et un contexte d'exécution



Etat d'un Processus

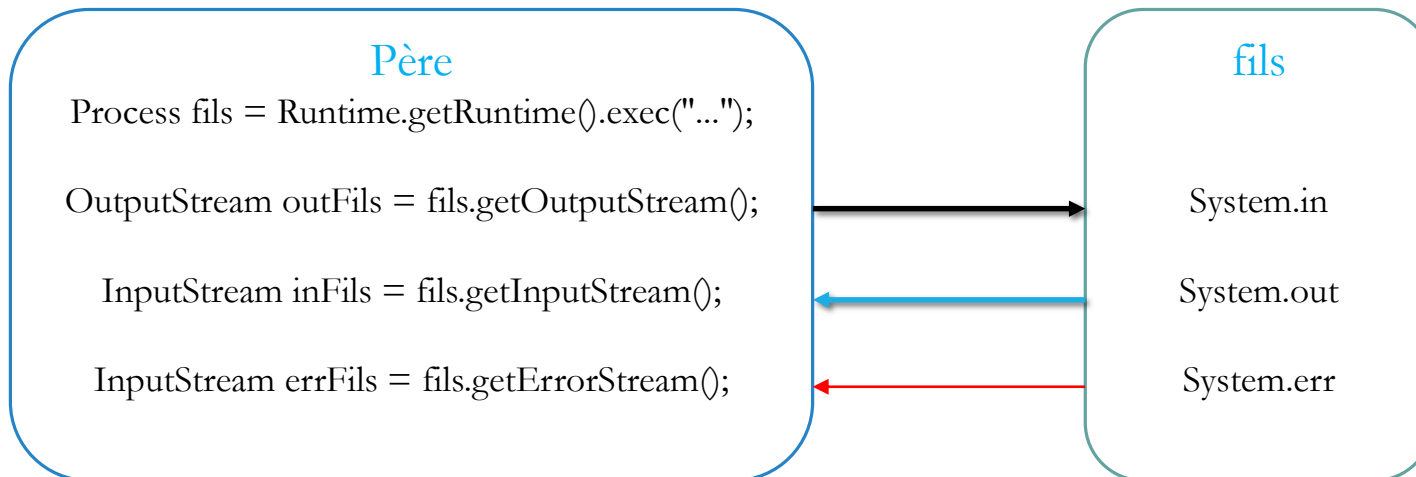
- Elu : il utilise le processeur
- Prêt : le processus est prêt à s'exécuter, mais n'a pas le processeur (occupé par un autre processus en exécution)
- Bloqué

Environnement d'exécution en Java

- L'environnement d'exécution d'une JVM est disponible dans la JVM elle-même sous la forme d'un objet de type `java.lang.Runtime`.
 - Un seul exemplaire d'un tel objet (Pattern Singleton) : impossible de créer un objet de cette classe.
- L'instance unique peut être récupérée par appel à la méthode statique `Runtime.getRuntime()`;
- De nombreuses méthodes sont disponibles dans la classe `Runtime`:
 - Méthodes permettant de connaître l'environnement d'exécution : `total/free/maxMemory()`, `availableProcessors()`
 - Contrôle du processus : `gc()`, `exit()`, `halt()`
 - Création d'autres processus : `Process exec(...)`;

Processus en Java

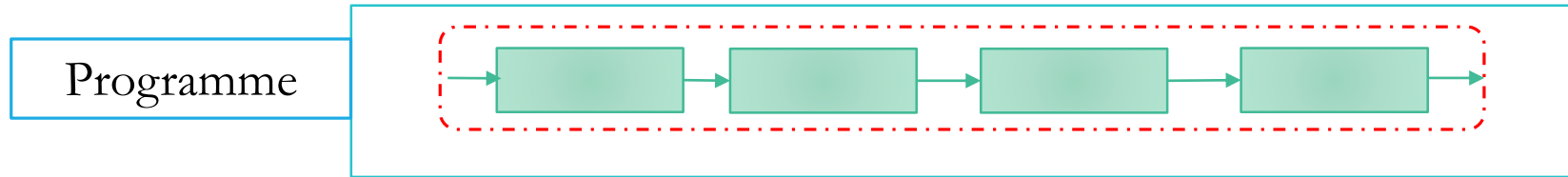
- les processus dont l'exécution (externe) a été commandée par une JVM sont représentés dans celle-ci sous la forme d'objet `java.lang.Process`. Ces objets permettent de
 - communiquer avec les processus externes correspondants
 - se synchroniser avec leur exécution
- On peut obtenir les flux d'entrée/sorties du processus externe :



- D'autres Méthodes :
 - `waitFor()` : attends la fin de l'exécution du fils
 - `exitValue()` : valeur renvoyée lors de la fin du fils (par `exit(n)`)
 - `destroy()` : tue le fils

Notion de Thread

- Idée : Créer des Processus légers **Thread** (lightweight process)
 - s'exécutant dans le contexte d'un programme
 - Utilisant les ressources allouées pour ce programme et son environnement

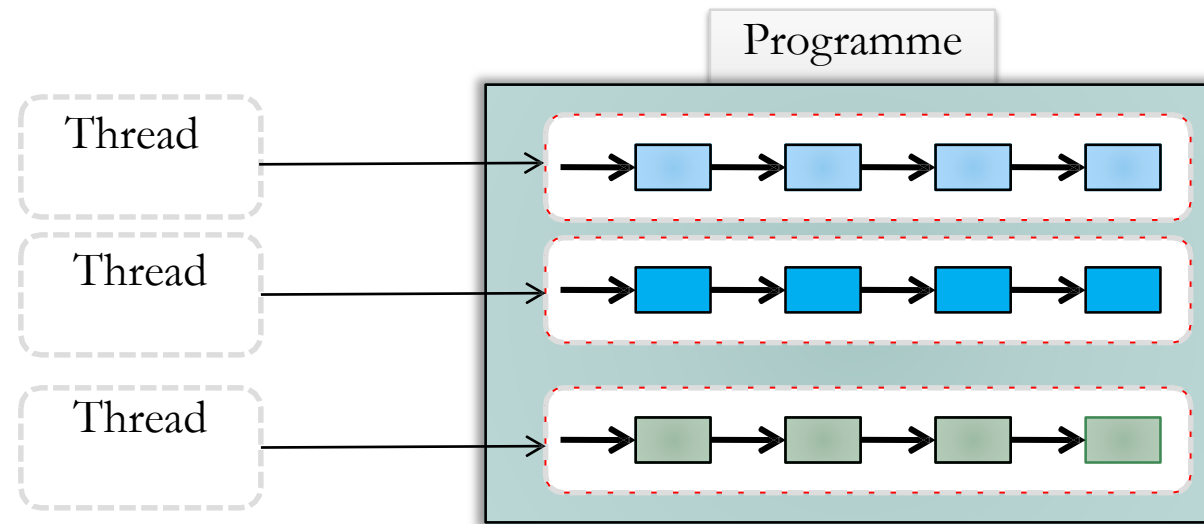


- Avantages :
 - Légèreté grâce au partage des données
 - Meilleures performances au lancement et en exécution
 - Partage les ressources système (pratique pour les I/O)

Multithreading

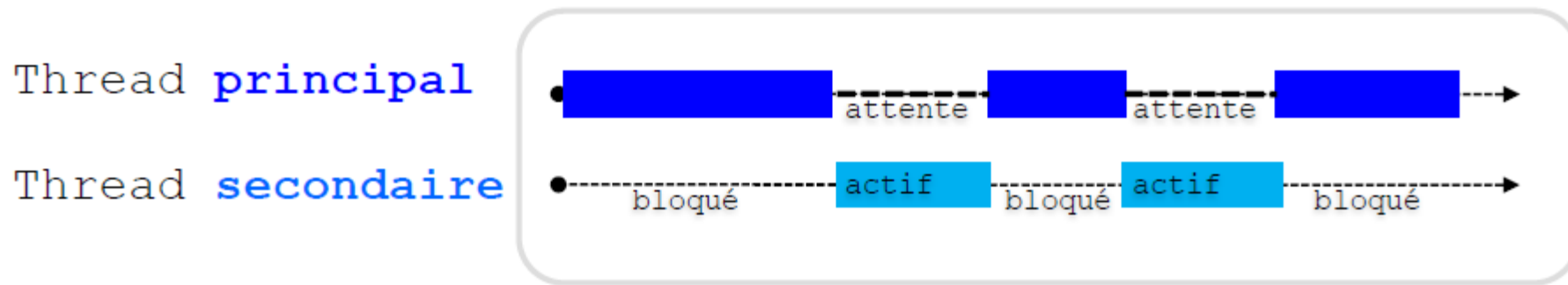
- Multithreading : Programmation concurrente

Possibilité d'exécuter plusieurs threads
simultanément dans le même programme



Intérêt des Threads

- Faire un traitement en tâche de fond : Lancer des threads avec une priorité inférieure, il s'exécute en permanence sans perturber le traitement principal.

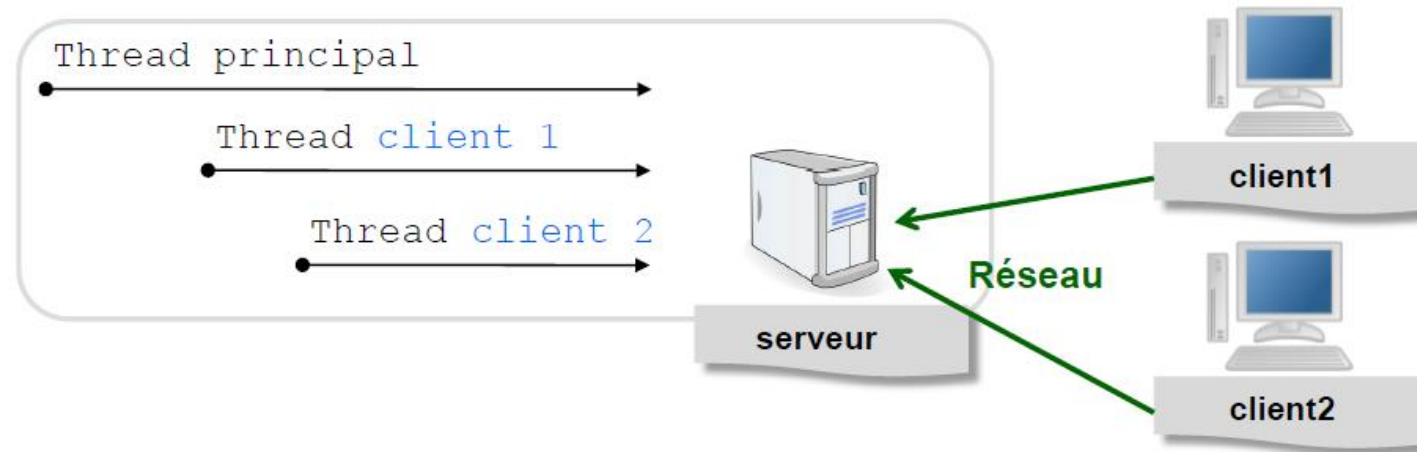


- Exemples :
 - Un GC lancé en arrière plan dans un thread séparé
 - Les GUIs utilisent un thread à part pour séparer les événements du GUI de l'environnement système.

Intérêt des Threads

- Plusieurs clients qui partagent un serveur : Les services proposés aux clients sont traités en parallèle

→ Optimisation du temps d'attente des clients



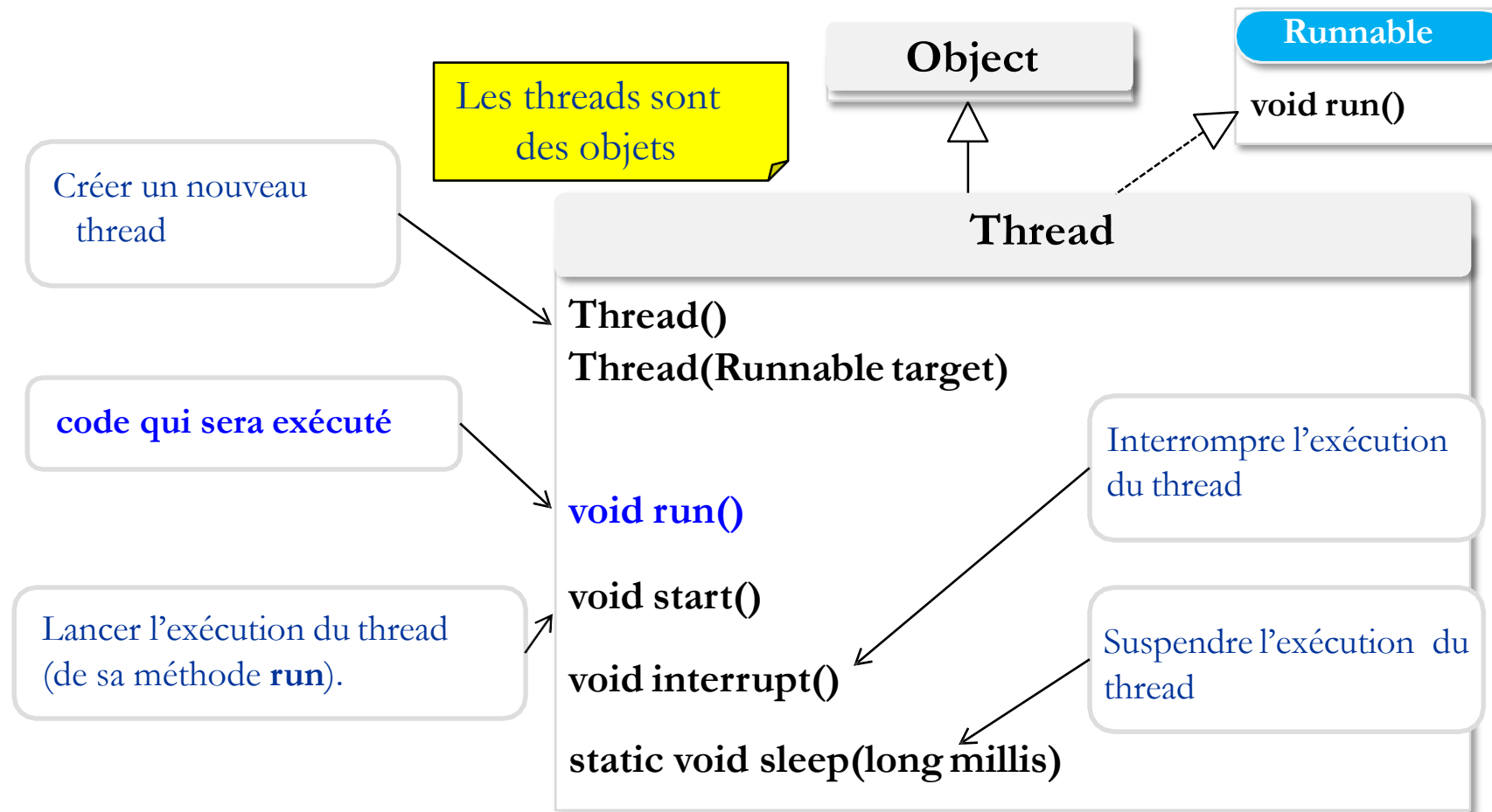
- Exemples :
 - Serveur web qui doit servir plusieurs requêtes concurrentes
 - Navigateur qui doit télécharger plusieurs images simultanément

Intérêt des Threads

- Étant donnée une exécution de Java (une JVM)
 - un seul processus (au sens système d'exploitation)
 - disposer de multiples fils d'exécution internes
 - possibilités de contrôle plus fin (priorité, interruption...)
 - c'est la JVM qui assure l'ordonnancement (concurrency)
 - espace mémoire commun entre les différents threads

Définition de threads

- java.lang.Thread au centre de la gestion des threads



Définition de Threads

- Première Méthode : un thread est définie comme instance d'une classe dérivée de la classe Thread.

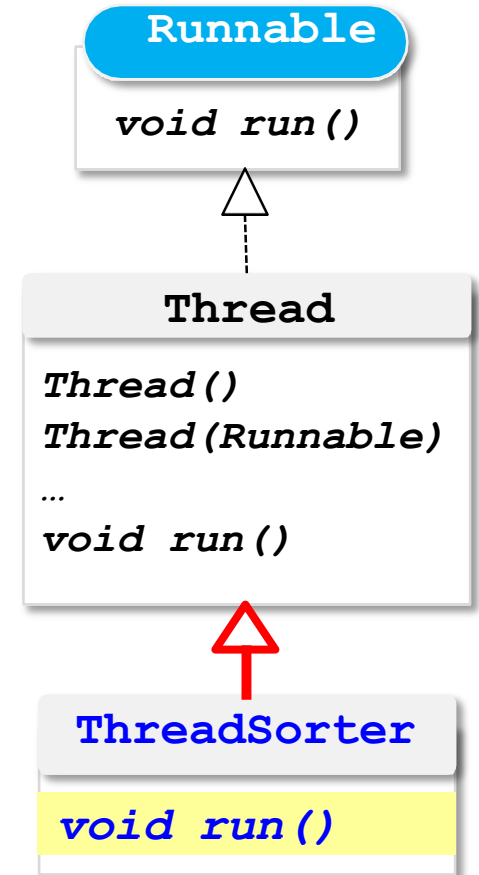
1. Sous classer Thread

```
Class ThreadSorter extends Thread {  
    List list;  
    public ThreadSorter (List list)  
        { this.list = list;  
        }  
}
```

2. Redéfinir run()

```
public void run() {  
    Collections.sort(list);  
}}
```

Méthode 1



Définition de Threads

- Première Méthode : un thread est définie comme instance d'une classe dérivée de la classe Thread.

3. Instancier

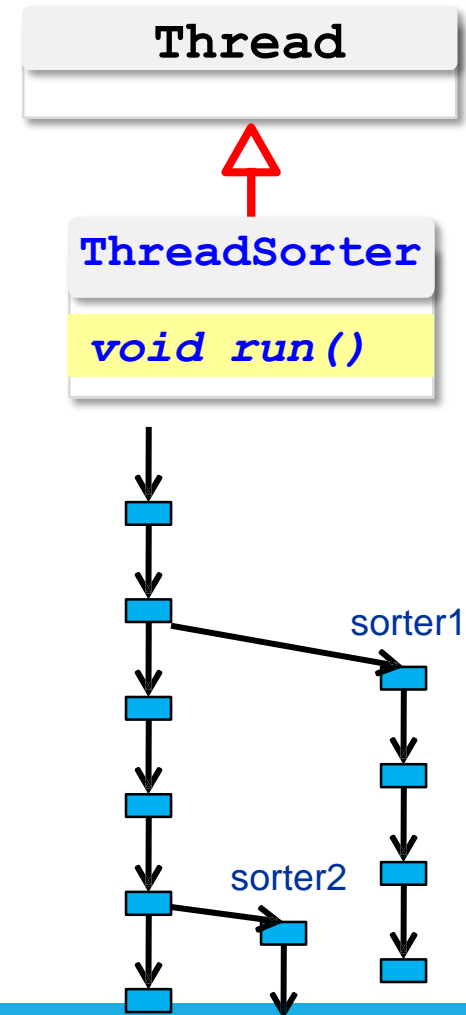
```
List liste1 = new ArrayList(); // ...  
List liste2 = new ArrayList(); // ...  
  
Thread sorter1 = new ThreadSorter (liste1);
```

4. lancer l'exécution de la méthode run ()

```
sorter1.start();  
...  
Thread sorter2 = new ThreadSorter (liste2);
```

Appeler *run()* directement exécute la tâche dans le même Thread
so
.. → Aucun nouveau Thread n'est lancé

Méthode 1



Définition de Threads

- Deuxième Méthode : un thread est définie comme implémentation de l'interface Runnable.

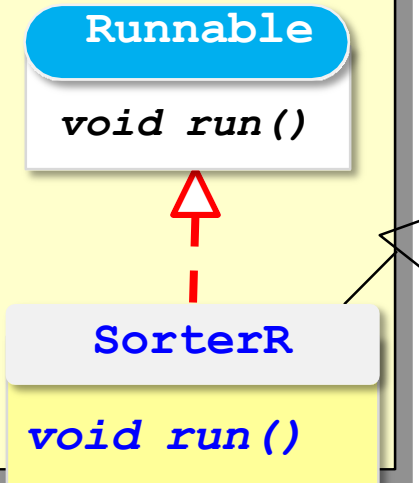
Méthode 2

1. Implémenter l'interface Runnable

```
class SorterR extends JApplet implements Runnable {  
    public void init() { ... }
```

2. Définir `run()`

```
    public void run() {  
        // tri et mise à jour de l'affichage  
    }
```



Définition de Threads

- Deuxième Méthode : un thread est définie comme implémentation de l'interface Runnable.

Méthode 2

3. Instancier l'implémentation de Runnable

```
Thread th1 = new Thread(new SorterR());
```

Objet implémentant Runnable

4. Exécuter la méthode run() de l'objet Runnable associé

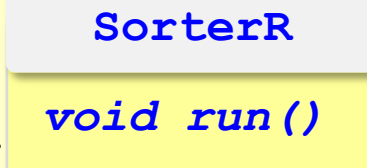
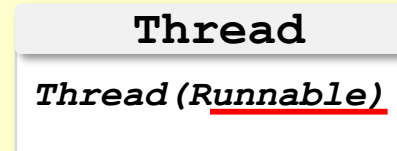
```
th1.start();
```

```
...
```

```
Thr
```

```
th2.start();...
```

Approche recommandée car vaut mieux séparer la tâche à exécuter du mécanisme de son exécution



Exercice :

Ecrivez un programme qui utilise deux threads en parallèle :

- Le premier affichera les 26 lettres de l'alphabet : de A-Z
- Le deuxième affichera les nombres impairs de 1 à 60.

→ que peut on remarquer ?!!!

Définition de Threads

Exemple

```
public class Horloge extends JLabel{  
    public Horloge() {  
        this.setHorizontalAlignment(JLabel.CENTER);  
        Font font = this.getFont();  
        this.setFont(new Font(font.getName(), font.getStyle(), 30));  
        Runnable afficheur = new Afficheur();  
        Thread thread = new Thread(afficheur);  
        thread.start();  
    }  
    public static void main(String[] args) {  
        JFrame f= new JFrame("Horloge Graphique");  
        f.getContentPane().add(new Horloge());  
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        f.setVisible(true);  
    }  
}
```

Runnable

void run()

Afficheur

void run()



Définition de Threads

Exemple

```
private class Afficheur implements Runnable {
    int s=0, m=0; String min="", sec="";
    public void run() {
        while(true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                return;
            }
            s++; if(s==60) { s=0; m++; }
            if(m==60) { m=0; }
            sec = (s<10 ? "0": "") + String.valueOf(s);
            min = (m<10 ? "0": "") + String.valueOf(m);
            this.setText(min + ":" + sec);
        }
    }
}
```

Runnable

void run()

Afficheur

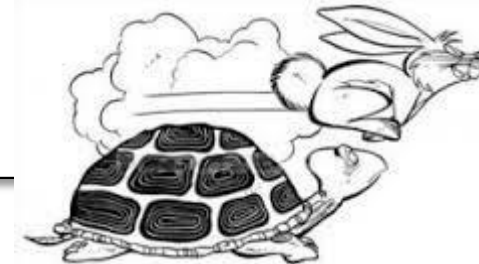
void run()



InterruptedException ?

Exercice :

- Implémenter la fable du lièvre et de la tortue
 - Le lièvre et la tortue débutent la course
 - Le lièvre dort au milieu de la course pensant qu'il est trop rapide que la tortue
 - La tortue continue à se déplacer lentement, sans s'arrêter, et gagne la course



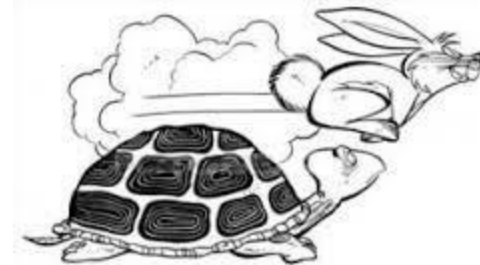
Exercice :

- Implémenter la fable du lièvre et de la tortue

```
public class Coureur implements Runnable {
    private static String vainqueur;
    private static final int distanceTotale=100;
    private int pas; //vitesse du coureur !!

    public void run() {
        courir();
    }

    private void courir() {
        String threadName = Thread.currentThread().getName();
        for(int dist = 1; dist <= distanceTotale; dist++){
            // lièvre et tortue (threads) font chacun sa course
            System.out.println(threadName+" : "+dist+" m");
            // Si quelqu'un a déjà gagné : fin de course
            if(courseDejaGagnee(dist)) break;
        }
    }
}
```

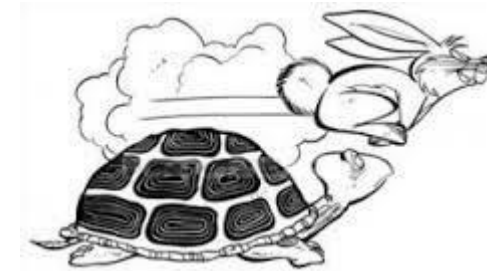


Exercice :

- Implémenter la fable du lièvre et de la tortue

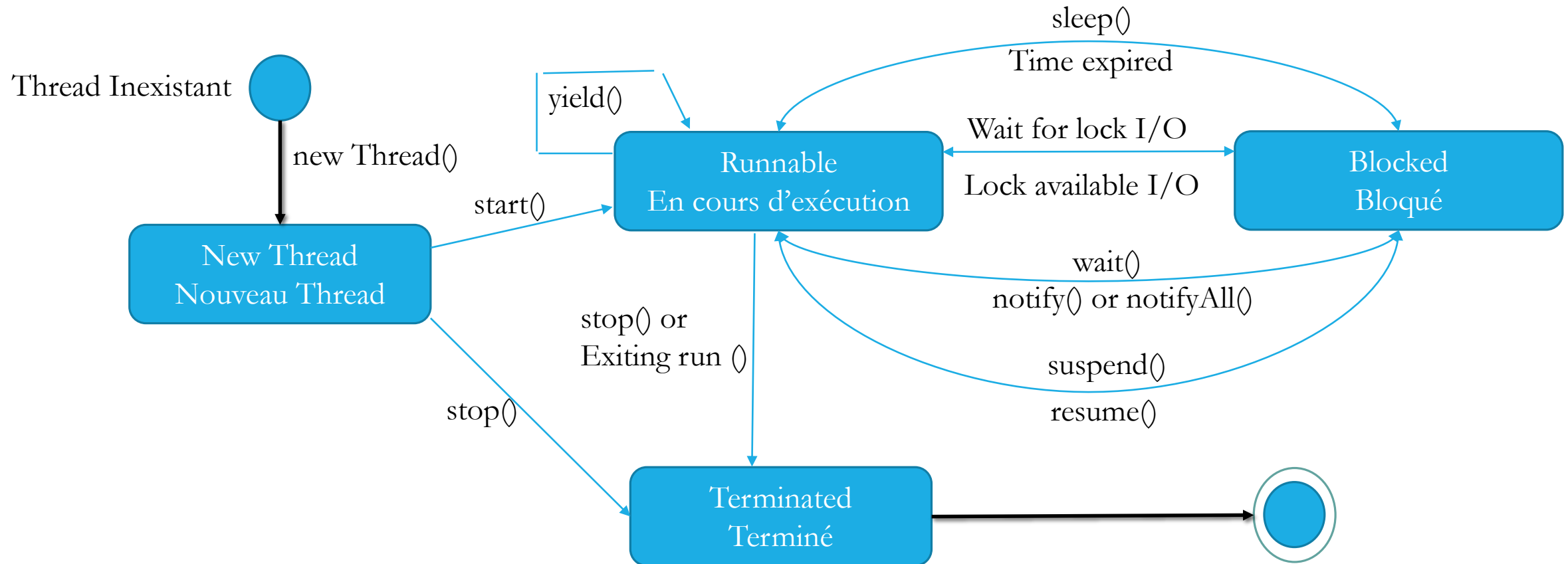
```
public class CourseDemo {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        Thread lievre = new Thread (new Coureur (80) , "Lievre");  
        Thread tortue = new Thread (new Coureur (20) , "Tortue");  
        //A vos marques, prêt partez !!  
        tortue.start();  
        lievre.start();  
    }  
}
```

vitesse



Cycle de vie d'un Thread

Cycle de vie d'un Thread



Etats d'un thread

- Créé :
 - comme n'importe quel objet Java
 - pas encore actif
- Actif :
 - après la création, il est activé par `start()` qui lance `run()`.
 - il est alors ajouté dans la liste des threads actifs pour être exécuté par l'OS en temps partagé
 - peut revenir dans cet état après un `resume()` ou un `notify()`

Exemple

```
class ThreadCompteur extends Thread {
    int no_fin;
    ThreadCompteur (int fin) {no_fin = fin;} // Constructeur
    // On redéfinit la méthode run()
    public void run () {
        for (int i=1; i<=no_fin ; i++) {
            System.out.println(this.getName()+":"+i);} }

    public static void main (String args[]) {
        // On instancie les threads
        ThreadCompteur cp1 = new ThreadCompteur (100);
        ThreadCompteur cp2 = new ThreadCompteur (50);
        cp1.start();
        cp2.start();
    } }
```

Etats d'un thread

- Endormi ou bloqué :
 - après `sleep()` : endormi pendant un intervalle de temps (ms)
 - `suspend()` endort le Thread mais `resume()` le réactive
 - une entrée/sortie bloquante (ouverture de fichier, entrée clavier) endort et réveille un Thread
- Mort :
 - si `stop()` est appelé explicitement
 - quand `run()` a terminé son exécution

Exemple

```
class ThreadCompteur extends Thread {
    int no_fin; int attente;
    ThreadCompteur (int fin,int att) {
        no_fin = fin; attente=att;}
    // On redéfinit la méthode run()
    public void run () {
        for (int i=1; i<=no_fin ; i++) {
            System.out.println(this.getName()+":"+i);
            try {sleep(attente);}
            catch (InterruptedException e) {};}
    }
    public static void main (String args[]) {
        // On instancie les threads
        ThreadCompteur cp1 = new ThreadCompteur (100,100);
        ThreadCompteur cp2 = new ThreadCompteur (50,200);
        cp1.start();
        cp2.start();
    } }
```

Les Priorités

Principes

- Seuls les Threads actifs peuvent être exécutés et donc accéder au CPU
- Par défaut, chaque nouveau Thread a la même priorité que le Thread qui l'a créé.
- Java permet de modifier les priorités (niveaux absolus) des Threads par la méthode `setPriority()`

Méthodes :

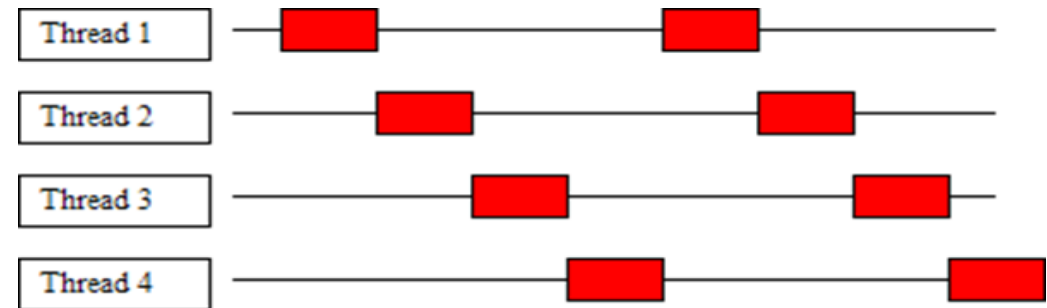
- `setPriority(int)` : fixe la priorité du receveur.
 - le paramètre doit appartenir à : `[MIN_PRIORITY, MAX_PRIORITY]`
 - sinon `IllegalArgumentException` est levée
- `int getPriority()` : pour connaître la priorité d'un Thread
 - `NORM_PRIORITY` : donne le niveau de priorité "normal"

Gestion du CPU entre Threads

Time-slicing (ou round-robin scheduling) :

- La JVM répartit de manière équitable le CPU entre tous les threads de même priorité.

→ Ils s'exécutent en "parallèle"



Préemption (ou priority-based scheduling) :

- Le premier thread du groupe des threads à priorité égale monopolise le CPU. Il peut le céder :
 - **Involontairement** : sur entrée/sortie
 - **Volontairement** : appel à la méthode statique `yield()` qui permet, pour le thread COURANT, de passer la main à un autre thread de priorité égale ou supérieure.
- Attention : ne permet pas à un thread de priorité inférieure de s'exécuter.
- **Implicitement** : en passant à l'état endormi (`wait()`, `sleep()` ou `suspend()`)

Les Priorités : exemple

Ecrivez un programme qui utilise deux threads en parallèle :

- Le premier représentera un producteur qui propose quatre Items
- Le deuxième représentera un consommateur qui va utiliser les Items

→ Proposer deux solutions :

1. La production se termine complètement avant de commencer la consommation.
2. La production et la consommation se font en parallèle.

Les Priorités

Exemple

```
public class PriorityExample
{
    public static void main(String[] args)
    {
        Thread producer = new Producer();
        Thread consumer = new Consumer();

        producer.setPriority(Thread.MAX_PRIORITY);
        consumer.setPriority(Thread.MIN_PRIORITY);

        producer.start();
        consumer.start();
    }
}
```

Les Priorités et gestion du CPU

Exemple

```
class Producer extends Thread{
    public void run()
    {
        for (int i = 0; i < 4; i++){
            System.out.println("Producer : Item " + i);
        }
    }
}

class Consumer extends Thread{
    public void run() {
        for (int i = 0; i < 4; i++){
            System.out.println("Consumer : Item " + i);
        }
    }
}
```

Avec yield()

Producer : Item 0
Consumer : Item 0
Producer : Item 1
Consumer : Item 1
Producer : Item 2
Consumer : Item 2
Producer : Item 3
Consumer : Item 3

Producer : Item 0
Producer : Item 1
Producer : Item 2
Producer : Item 3
Consumer : Item 0
Consumer : Item 1
Consumer : Item 2
Consumer : Item 3

Fermeture de threads

- Le thread termine lorsque sa méthode **run()** retourne
 - Nous voulons rarement qu'une activité cesse immédiatement :

Thread.stop():

Doc

Thread.stop () : *deprecated*

- *This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack).*
- *If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior.*

Modification dans
un compte bancaire

Fermeture de threads : boucle conditionnelle

- Solution 1 : Mettre une condition dans la boucle while(running) avec running un booléen. Pour changer la valeur de running, deux possibilités :
 - Mettre une condition dans la boucle infinie.
 - écrire une méthode.

```
class MonThread extends Thread {  
    public Monthread () {  
        // Mon constructeur  
    }  
  
    @Override  
    public void run() {  
        // Initialisation  
        while(true) { // Boucle infinie pour  
                        effectuer des traitements.  
            // Traitement à faire  
        } }  
}
```



```
class MonThread extends Thread {  
    protected volatile boolean running = true;  
    public Monthread () {  
        // Mon constructeur    }  
    public arret() { // Méthode 2  
        running = false;    }  
    @Override  
    public void run() {  
        // Initialisation  
        while(running) { // Boucle infinie pour effectuer des traitements.  
            // Traitement à faire  
            if (/*Condition d'arret*/) // Méthode 1  
                running = false;  
        } } }  
}
```

→ Si le thread fasse de longues attentes (comme un Thread.sleep de 5 minutes, un wait , un Thread.join ou autre), il faudrait alors attendre que l'attente de ce dernier se termine avant qu'il ne se ferme.

Interruption de thread

- Pour qu'un thread interrompe vraiment son exécution, il doit participer activement à sa propre interruption
- 2 cas :
 - le thread est bloqué, par les méthodes `sleep`, `wait`, `join`, ou en attente d'une entrée/sortie interruptible
 - ❖ L'appel de `interrupt` provoque la levée d'une `InterruptedException` (ou `ClosedByInterruptException` pour les E/S) dans le thread en attente
 - ❖ Le thread «interrompu» doit gérer cette interruption
 - toutes les autres situations : Le thread doit vérifier (périodiquement) s'il a été interrompu en appelant la méthode statique `interrupted()`

Interruption de threads

```
while (moreworktodo)
{
    //do more work
}
```

Thread.interrupt()

Si ce thread exécute une méthode **bloquante** de bas niveau

interruptible : `Thread.sleep()`, `Object.wait()`, ...

- il débloque et
- lance **InterruptedException** (**rôle ?**)

Sinon

- Fixe simplement : **interrupted_status** = true

Interruption de threads

Doc

```
public class InterruptedException extends Exception
```

- *Thrown when a thread is **waiting**, **sleeping**, or otherwise **occupied**, and the thread is **interrupted**, either before or during the activity.*
- *Occasionally a method may wish to test whether the current thread has been interrupted, and if so, to immediately throw this exception.*

Interruption de threads

```
public void run() {  
    try{  
        while( !Thread.currentThread().isInterrupted()  
                && moreworktodo)  
        {  
            //do more work  
        }  
    } catch (InterruptedException e) {  
        // Thread was interrupted  
        // during sleep or wait ...  
    } finally{  
        //cleanup, if required  
    }  
    // exiting the run method terminates the thread  
}
```

Chaque thread doit vérifier occasionnellement s'il a été interrompu

Si appel à `sleep()` dans la boucle, pas besoin d'appeler `isInterrupted()`: *Un thread bloqué ne peut pas faire cette vérification*

Interruption de threads

- L'interruption est un mécanisme coopératif
 - C'est une façon de demander poliment à un autre thread « s'il veut bien arrêter ce qu'il fait et à sa convenance ».
 - Le thread interrompu n'arrête pas nécessairement immédiatement ce qu'il fait.
 - Il peut vouloir nettoyer les travaux en cours ou aviser les autres activités de l'annulation avant de terminer.

Interruption de threads

Exemple

```
public class PlayerMatcher {
    private PlayerSource players;
    public PlayerMatcher(PlayerSource players) {
        this.players = players;
    }

    public void matchPlayers() throws InterruptedException {
        Player playerOne, playerTwo;
        try {
            while (true) {
                playerOne = playerTwo = null;
                // Wait for two players to arrive and start a new game
                playerOne = players.waitForPlayer();
                // could throw IE
                playerTwo = players.waitForPlayer();
                // could throw IE
                startNewGame(playerOne, playerTwo);
            }
        }
    }
}
```

Interruption de threads

Exemple

```
} catch (InterruptedException e) {  
    /*  
    *   If we got one player and were interrupted, put that  
    *   player back  
    */  
    if (playerOne != null)  
        players.addFirst(playerOne);  
  
    // Then propagate the exception  
    throw e;  
}  
}
```

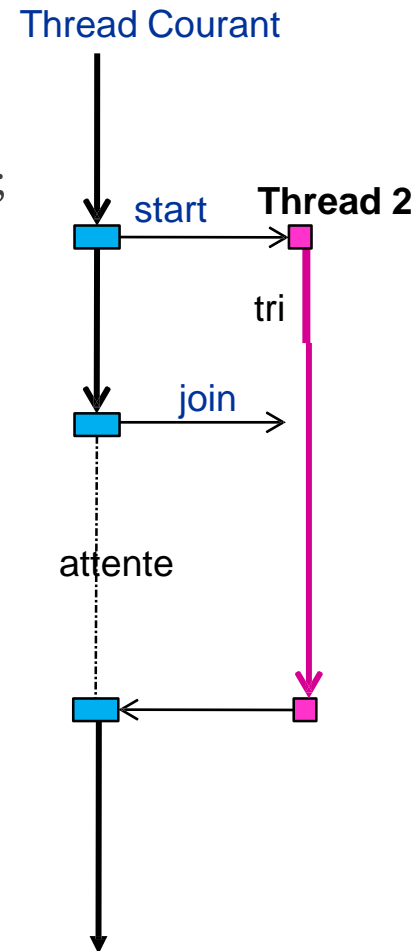
Concurrence d'accès

Concurrence d'accès : fonction join

- Méthode Join : bloque le thread Courant jusqu'à la fin de l'exécution du thread contrôlé par Thread2.
- Exemple : on voudrait exécuter que l'instruction : `System.out.println(« Terminer »);` qu'après que trois threads lancés 'Afficheur' termine leur travaux

```
class exemplejoin {  
    public static void main(String[] argv) {  
        Afficheur r1,r2,r3;  
        r1 = new Afficheur (« Etudiants»);  
        r2 = new Afficheur (« Professeurs»);  
        r3 = new Afficheur (« Administration»);  
        r1.start();  
        r2.start();  
        r3.start();  
        try{  
            r1.join();  
            r2.join();  
            r3.join(); }  
        catch(InterruptedException exc) {}  
        System.out.println(« Terminer »); } }
```

```
class Afficheur extends Thread  
{  
    String chaine;  
    Afficheur(String chaine)  
    {  
        this.chaine=chaine;  
    }  
    public void run()  
    {  
        System.out.println(chaine);  
        setPriority(Thread.MIN_PRIORITY);  
        System.out.println(chaine);  
    }  
}
```



Concurrence d'accès

- Les threads d'un même processus partagent le même espace mémoire
 - pas de "mémoire privée"
 - Les threads peuvent accéder simultanément à une même ressource
 - Les données risquent d'être corrompues
- L'utilisation de threads peut entraîner des besoins de synchronisation pour éviter les problèmes liés aux accès simultanés aux variables
- En programmation parallèle, on appelle section critique, une partie du code qui ne peut être exécutée en même temps par plusieurs threads sans risquer de provoquer des anomalies de fonctionnement

Exemple de problème

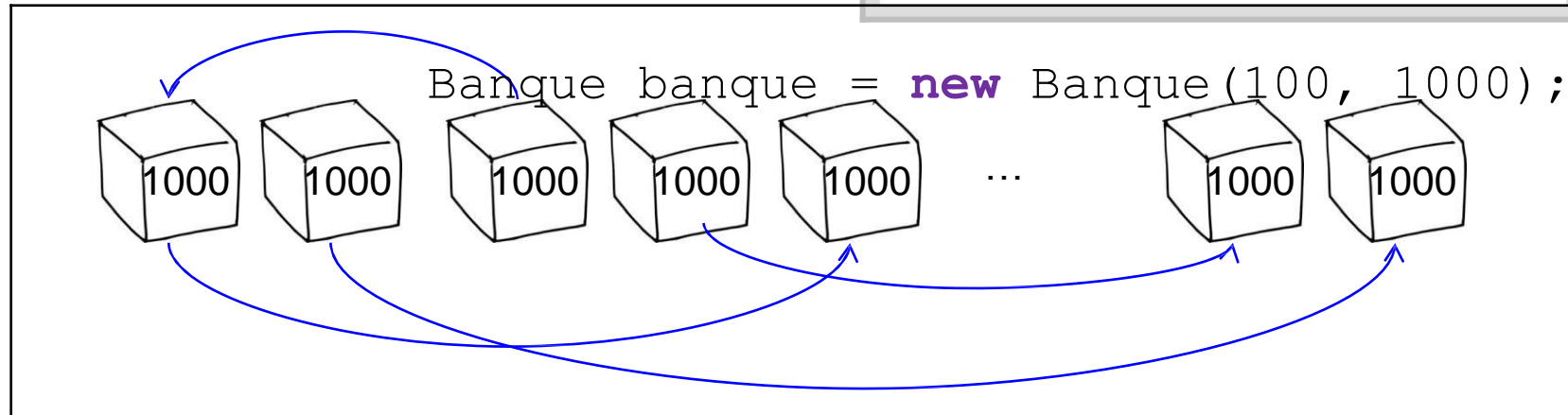
- Si $a = 5$, le code $a = a + 1$; exécuté par 2 threads, peut donner en fin d'exécution 6 ou 7 suivant l'ordre d'exécution.
- Accès simultané à un compte.

Concurrence d'accès : Exemple Banque



Banque

```
final double[] comptes;  
Banque(nbCompt, soldeInit)  
transferer(int,int,double)  
double totalSoldes()
```



Chaque compte (**thread**) effectue des transferts vers d'autres comptes.

Concurrence d'accès

→ La classe Transfert implements Runnable

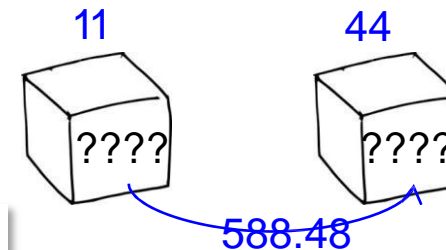
```
class Transfert implements Runnable{
    private Banque banque;
    private int de;
    public Transfert(Banque b, int de){
        banque =b; this.de=de;
    }
    public void run(){
        try{
            while (true) {
                int vers = (int) (banque.size()*Math.random());
                double montant = 1000*Math.random()); // <1000
                banque.transferer(de, vers, montant);
                Thread.sleep(10);
            }
        }catch (InterruptedException e) {}
    }
}
```

Concurrence d'accès

→ La méthode `transferer()`

```
public void transferer(int de, int vers, double montant) {  
    if (comptes[de] < montant) return;  
    System.out.print(Thread.currentThread());  
    comptes[de] -= montant;  
    System.out.printf("%10.2f (%d -->%d)", montant, de, vers);  
    comptes[vers] += montant;  
    System.out.printf(" Total : %10.2f\n", totalSoldes());  
}
```

Thread[Thread-11,5,main] 588.48 (11 --> 44) Total : 100000.00



Concurrence d'accès

→ La simulation

```
public static void main(String[] args) {  
    Banque b = new Banque(100, 1000);  
    for (int de = 0; de < 100; de++) {  
        Thread th = new Thread(new Transfert(b, de));  
        th.start();  
    }  
}
```

→ Le total des soldes doit rester le même !


```
...  
Thread[Thread-11,5,main] 588.48 (11 --> 44) Total : 100000.00  
Thread[Thread-12,5,main] 976.11 (12 --> 22) Total : 100000.00  
...  
Thread[Thread-36,5,main] 401.71 (36 --> 73) Total : 99291.06  
Thread[Thread-36,5,main] 691.46 (36 --> 77) Total : 99291.06  
...
```

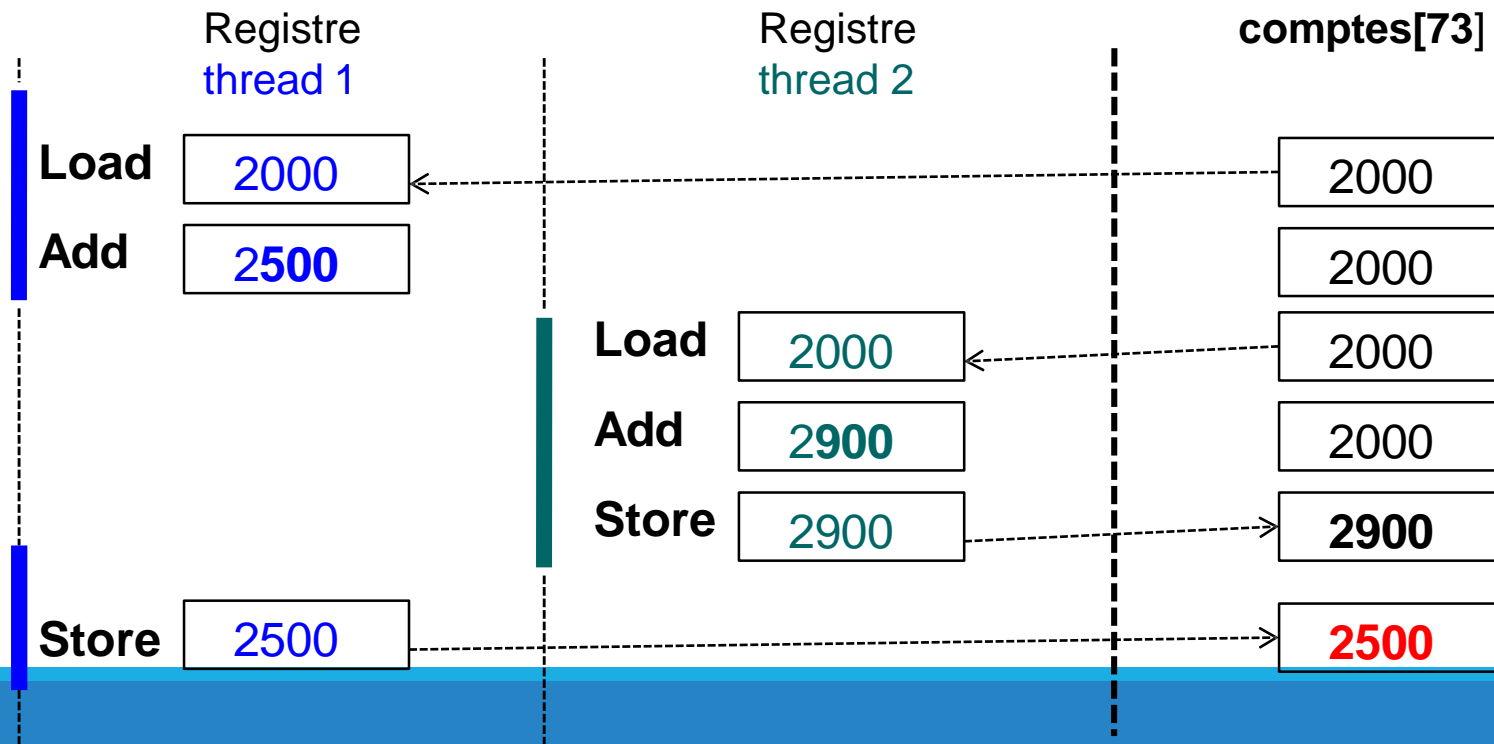
problème !

Ce n'est pas très sûr !!!!

Concurrence d'accès

L'instruction `comptes[vers] += montant;` n'est pas atomique

- 
1. **Load** `comptes[vers]` dans un registre
 2. **Add** `montant`
 3. **Store** remettre le résultat dans `comptes[vers]`

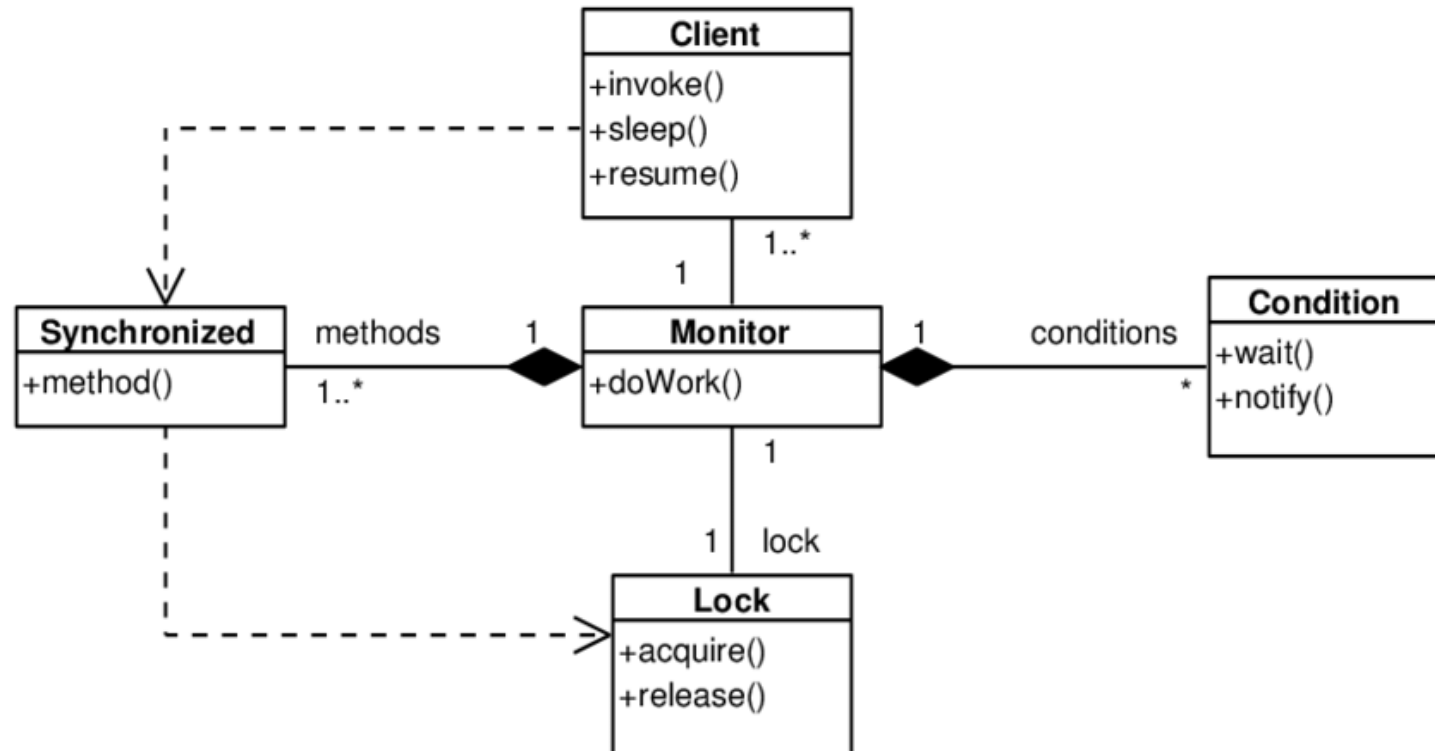


Concurrence d'accès

- L'exécution simultanée par plusieurs threads d'une même portion de code peut engendrer des problèmes d'accès concurrents sur certains objets.
 - Comment accéder à une ressource partagée dans un environnement Multithreading
 - Assurer la consistance du comportement de la ressource
 - Difficile à gérer par les instances clients (solution globale)
- Utiliser des mécanismes de verrouillage qui vont restreindre l'exécution d'une portion de code critique à un seul thread en permettant d'inhiber les accès concurrents qui sont réalisés dans cette portion de code.
- Java propose deux types de verrous :
 - les moniteurs : un mécanisme implicite dont la mise en œuvre est intégrée au langage et à la JVM
 - les Locks : c'est un mécanisme explicite dont la mise en œuvre utilise des classes de l'API du JDK

Concurrence d'accès : Pattern Monitor

- Moniteur : Contrôler l'accès concurrent à une ressource partagée pour limiter les résultats non prévisibles



The Monitor Object Pattern class Diagram

Concurrence d'accès : bloc synchronisé

- Un bloc synchronisé : tout bloc portant une déclaration **synchronized** et fonctionne avec un **Moniteur**. Il peut être utilisé de trois façons :
 - Un modificateur d'une méthode statique : le Moniteur de synchronisation est la classe qui possède ce bloc.
 - Un modificateur d'une méthode non statique : le Moniteur de synchronisation est l'instance de cette classe dans laquelle on se trouve. On peut accéder à cette instance par le mot-clé `this`.
 - Un bloc commençant par le mot-clé **synchronized** : le moniteur de synchronisation doit être passé explicitement à l'ouverture de ce bloc (Ce peut être tout objet Java.)

```
public class Chauffage {
    private Object key = new Object();

    /* méthode statique synchronisée, le Moniteur de
    synchronisation est l'objet Chauffage.class */
    public static synchronized boolean getNombreChauffage () {
        // corps de la méthode
    }

    / * méthode non statique synchronisée, le Moniteur de
    synchronisation est l'objet this */
    public synchronized boolean plusChaud() {
        // corps de la méthode
    }

    public boolean plusFroid() {
        /* synchronization sur l'objet key on peut aussi
        synchroniser sur l'objet this */
        synchronized(key) {
            // bloc synchronisé
        }
    }
}
```

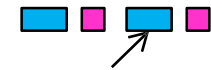
Concurrence d'accès : Pattern Monitor

Synchronisation JDK1.0

```
public synchronized void transferer(...) {  
    <corps de la méthode> //section critique  
}
```

Équivalent à

```
public void transferer(...) {  
    //positionner verrou  
    try{  
        <corps de la méthode>  
    }finally{  
        //libérer verrou  
    }  
}
```



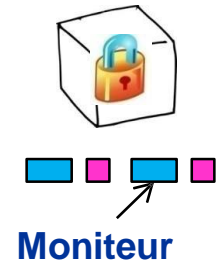
Moniteur

Chaque objet possède un **verrou implicite** et une **condition implicite** associée

Concurrence d'accès : Synchronized

Synchronisation JDK1.0

```
public synchronized void transferer(...) {  
    <corps de la méthode> //section critique  
}
```



- Le verrou est libéré :
 - en quittant la partie synchronisée.
 - en appelant la méthode `wait()`.

API

java.lang.Object 1.0

- **`void wait()` , `void wait(long millis)`**

Causes a thread to wait until it is notified. This method throws an **`IllegalMonitorStateException`** if the current thread is not the owner of the object's lock. It can only be called from within a synchronized method.

- **`void notifyAll()` , `void notify()`**

unblocks the threads (one single random Thread) that called wait on this object

Concurrence d'accès : Synchronized

Synchronisation

➔ On peut effectuer une synchronisation plus fine

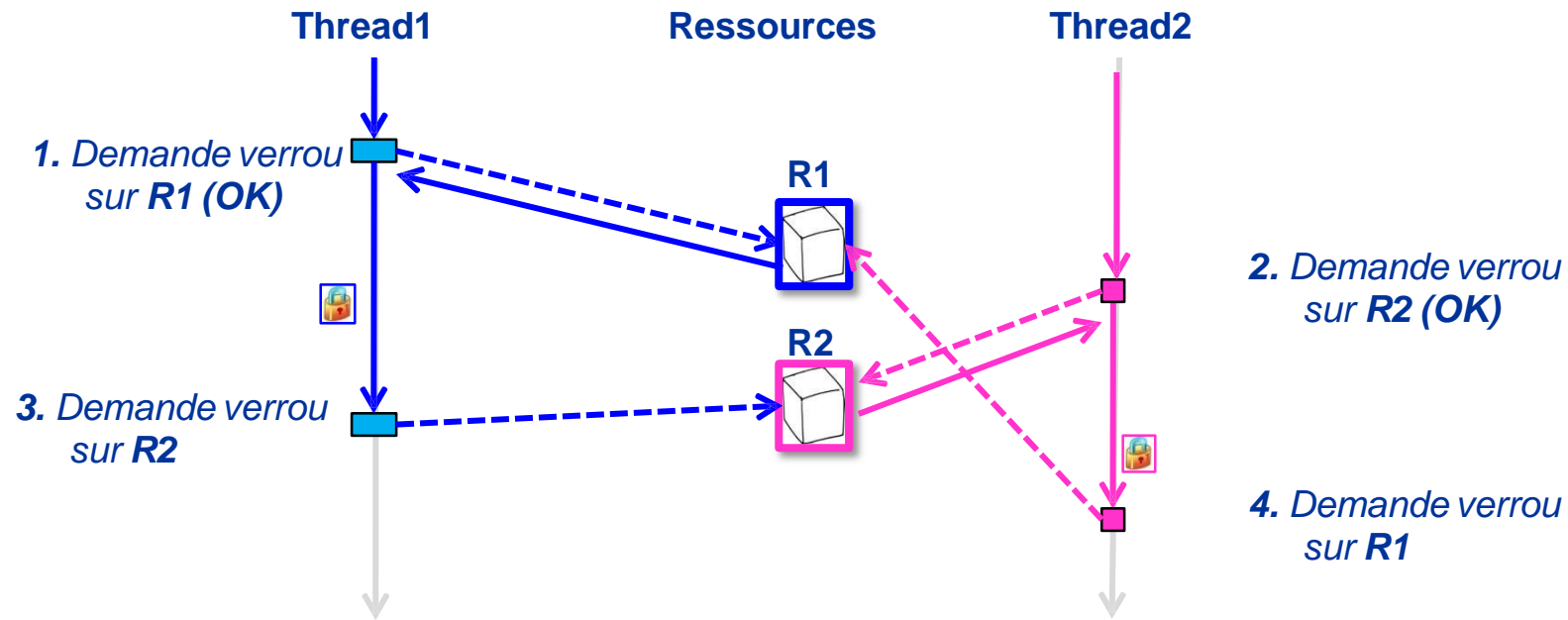
```
...  
public static void swap(Object[] tab, int i1, i2) {  
    synchronized(tab) {  
        Object tmp = tab[i1];  
        tab[i1] = tab[i2];  
        tab[i2] = tmp;  
    }  
}
```

Bloc synchronisé

Concurrence d'accès : Synchronized et Deadlock

Interblocage

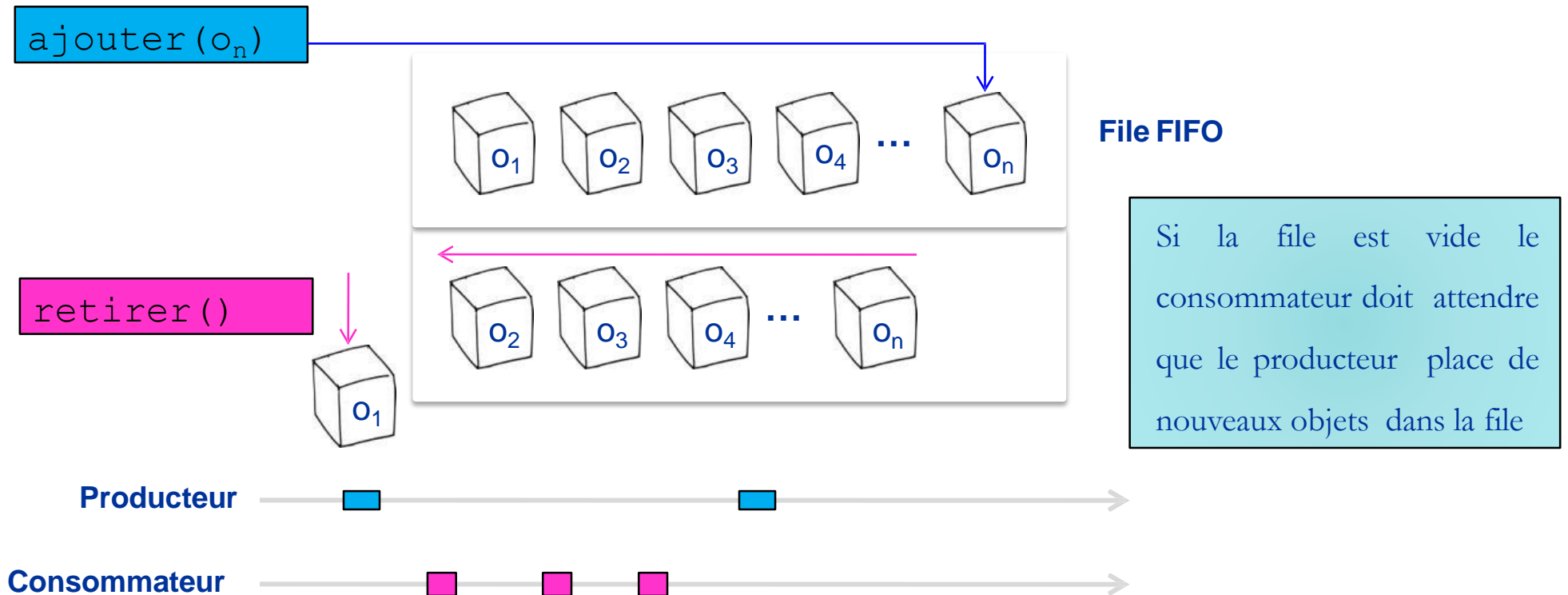
→ *chaque thread attend que l'autre relâche un verrou*



Concurrence d'accès : Synchronized et Deadlock

Interblocage : Exemple

→ *File FIFO partagée par deux threads*



Concurrence d'accès : Synchronized et Deadlock

Interblocage : Exemple

```
import java.util.LinkedList;
public class FileFIFO {
    LinkedList q = new LinkedList();
    public synchronized Object retirer() {
        while (q.size()==0) { ; }
        // NE RIEN FAIRE

        return q.remove(0);
    }
    public synchronized void ajouter(Object o) {
        q.add(o);
    }
}
```

Nécessaire si plusieurs
consommateurs

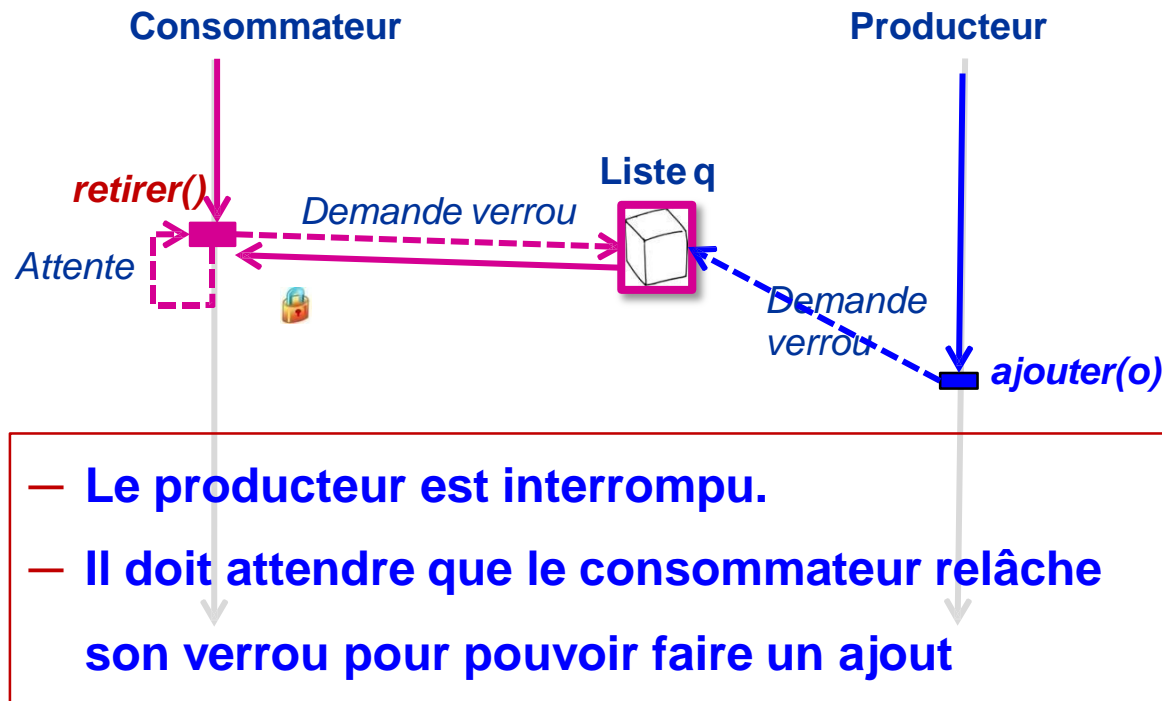
Si la liste q est vide **attendre**
qu'un objet soit mis dans la file

Interblocage

Concurrence d'accès : Synchronized et Deadlock

Interblocage : Exemple

➔ **Interblocage** quand un consommateur est en attente



Concurrence d'accès : Synchronized et Deadlock

Interblocage : Exemple

```
import java.util.LinkedList;

public class FileFIFO {
    LinkedList q = new LinkedList();

    public synchronized Object retirer() {

        if (q.size() == 0) {
            try { this.wait(); }
            catch (InterruptedException e) {}
        }

        return q.remove(o);
    }

    public synchronized void ajouter(Object o) {
        q.add(o);
        this.notify();
    }
}
```

1. Interrompre le **consommateur**

2. Permettre au **producteur** d'accéder à la file

3. Relancer le **consommateur**

Concurrence d'accès : Synchronized et Deadlock

Interblocage : Exemple

```
import java.util.LinkedList;

public class FileFIFO {
    LinkedList q = new LinkedList();

    public synchronized Object retirer() {
        if (q.size() == 0) {
            try { this.wait(); }
            catch (InterruptedException e) {}
        }
        return q.remove(o);
    }

    public synchronized void ajouter(Object o) {
        q.add(o);
        this.notify();
    }
}
```

1. Interrompre le **consommateur**

2. Permettre au **producteur** d'accéder à la file

3. Relancer le **consommateur**

Concurrence d'accès : Synchronized et Deadlock

Interblocage : Exemple

```
import java.util.LinkedList;
public class FileFIFO {
    LinkedList q = new LinkedList();
    public synchronized Object retirer() {
        while (q.size() == 0) {
            try { this.wait(); }
            catch (InterruptedException e) {}
        }
        return q.remove(0);
    }
    public synchronized void ajouter(Object o) {
        q.add(o);
        this.notify();
    }
}
```

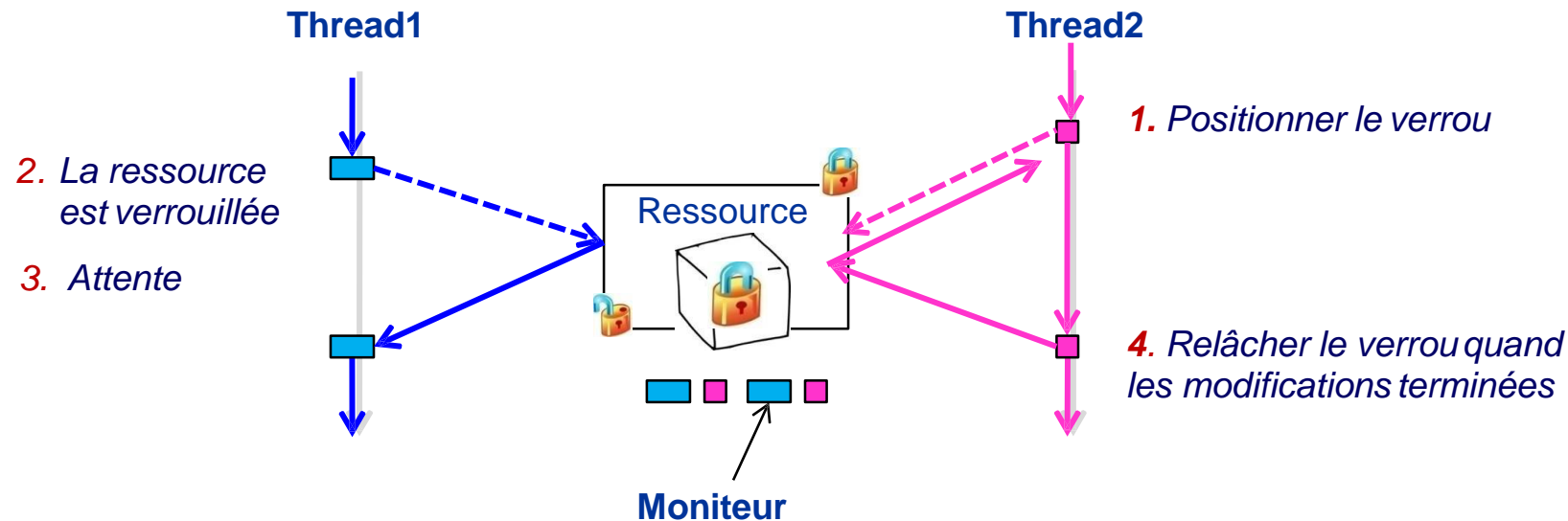
quand le thread en attente est réveillé il est en concurrence avec d'autres thread

IL FAUT vérifier à nouveau la condition qui l'a mis en attente avant de poursuivre son exécution

Concurrence d'accès

A partir de JDK 5.0

- Système de **verrous** pour protéger du code « sensible »
- Chaque objet maintient une **liste de threads** en attente
- Les RDV entre threads sont gérés au sein d'un **moniteur**



Concurrence d'accès

A partir de JDK 5.0

API `java.util.concurrent.locks.Lock` 5.0 *interface*

- `void lock()`
acquires this lock; blocks if the lock is currently owned by another thread
- `void unlock()`
Releases this lock;

API `java.util.concurrent.locks.ReentrantLock` 5.0

- `ReentrantLock()`
Constructs a reentrant lock that can be used to protect a critical section

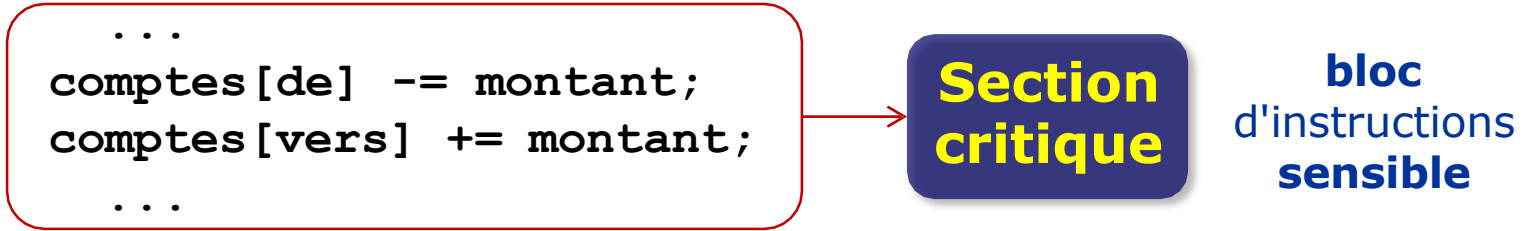


Moniteur

Concurrence d'accès

Classe ReentrantLock

```
public class Banque{  
    private Lock bankLock = new ReentrantLock();  
    public void transferer(int de, int vers, int montant){  
        bankLock.lock();  
        try{  
            ...  
            comptes[de] -= montant;  
            comptes[vers] += montant;  
            ...  
        }  
        finally{  
            bankLock.unlock();  
        }  
    }  
}
```



Section critique

bloc d'instructions sensible

Chaque objet `Banque` a son propre verrou pour un accès exclusif à la section critique

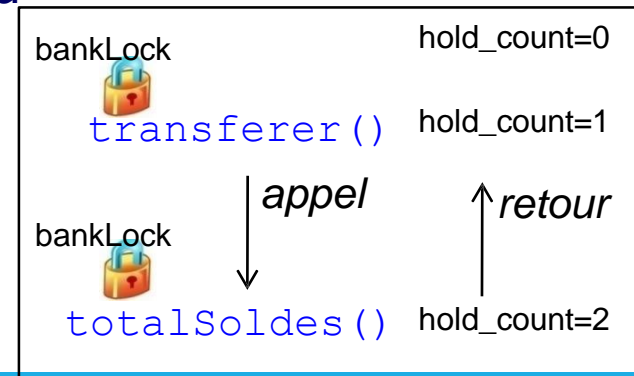
Concurrence d'accès

Classe ReentrantLock

```
public class Banque{  
    private Lock bankLock = new ReentrantLock();  
}
```

- Reentrant parcequ'un thread peut acquérir en boucle un verrou qu'il possède déjà :
 - Une méthode protégée par un verrou peut appeler une autre méthode protégée par le même verrou

- Le verrou dispose d'un compteur de nombre d'appels à la méthode `lock()`



Concurrence d'accès

Solde insuffisant ?

```
if (comptes[de] >= montant)
    //Thread might be deactivated at this point
    banque.transferer(de, vers, montant);
```

- **Problème** *Dans le temps et après plusieurs exécutions, le compte pourra avoir un solde insuffisant et transférer comme même !*
- Il faut être sûr que le thread ne sera pas interrompu entre le test et l'insertion : donc *protéger les deux par un verrou.*

Concurrence d'accès

Utilisation des Verrous

```
public class Banque{  
    private Lock bankLock = new ReentrantLock();  
    public void transferer(int de, int vers, int montant){  
        bankLock.lock();  
        try{  
            if (comptes[de] < montant) {  
                // wait  
            }  
            //Transférer l'argent  
            comptes[de] -= montant;  
            comptes[vers] += montant;  
        }  
        finally{  
            bankLock.unlock();  
        }  
    }  
}
```

*Je dois attendre qu'un
autre thread me fasse
un transfert !*

Impossible

*car je détiens le verrou
et donc l'accès exclusif !*

Il y a attente d'un verrou et attente sur une condition

Concurrence d'accès

Utilisation des Conditions

```
class Banque{  
    private Condition fondsSuffisants ;  
    public Banque () {  
        . . .  
        soldeSuffisant = bankLock.newCondition() ;  
    }  
}
```

API `java.util.concurrent.locks.Lock` 5.0 *interface*

- Condition `newCondition()`
returns a condition object that is associated with this lock

- **Condition** : sert à gérer les threads qui ont acquis un verrou mais ne peuvent pas faire un travail utile.
- Un verrou peut avoir plusieurs "**conditions**"

Concurrence d'accès

Classe ReentrantLock : Conditions

```
public void transfer(int de, int vers, int montant) {  
    bankLock.lock();  
    try{  
        while (comptes[de] < montant) {  
            soldeSuffisant.await();  
            // Le thread courant est bloqué et rend le verrou.  
            // Il reste bloqué jusqu'à ce qu'un autre thread  
            // appelle signalAll() sur la même condition  
        }  
        // transfert d'argent ...  
        //Lorsqu'un autre thread transfère de l'argent...  
        soldeSuffisant.signalAll();  
        // débloque tous les threads attendant cette condition  
    }  
    finally{  
        bankLock.unlock();  
    }  
}
```

Concurrence d'accès

Classe ReentrantLock : Conditions

A P I	java.util.concurrent.locks.Lock 5.0	interface
▪ void <code>await()</code>		
<i>puts this thread on the wait set for this condition.</i>		
▪ void <code>signalAll()</code>		
<i>unblocks all threads in the wait set for this condition</i>		
▪ void <code>signal()</code>		
<i>unblocks one randomly selected thread in the wait set for this condition</i>		

- Un thread qui appelle `await()` devient éligible.
- Une fois élu et le verrou libre, il continue la où il était.
- Un thread ne peut pas appeler `await()` ou `signalAll()` sur une condition que s'il possède le verrou associé

Concurrence d'accès

Comparaison Synchronized vs Lock

```
public Synchronized void transferer(...) {  
    <corps de la méthode> //section critique  
}
```

`wait()` de Object



`await()` de Condition

`notifyAll()` de Object



`signalAll()` de Condition



Simple



Une seule condition par verrou : pas toujours efficace.



On ne peut pas interrompre un thread qui essaye de récupérer un verrou

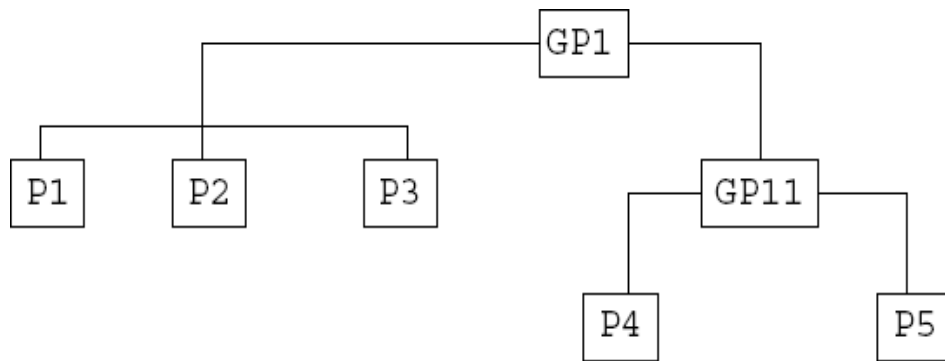
Threads Daemons

- Un thread peut être déclaré comme daemon :
 - comme le Thread "garbage collector", le Thread "afficheur d'images", ...
 - en général de faible priorité, il tourne dans une boucle infinie.
 - arrêt implicite dès que le programme se termine.
- Les méthodes :
 - `setDaemon()` : déclare un thread daemon (avant le démarrage du Thread, sinon une exception est levée de type `IllegalThreadStateException`.)
 - `isDaemon()` : renvoie un booléen qui précise si le thread est un démon.

Les « ThreadGroup »

- Plusieurs processus (Thread) peuvent s'exécuter en même temps, il serait utile de pouvoir les manipuler comme une seule entité
 - pour les suspendre
 - pour les arrêter, ...
- Java offre cette possibilité via l'utilisation des groupes de threads : `java.lang.ThreadGroup`
 - on groupe un ensemble nommé de threads
 - ils sont contrôlés comme une seule unité
- Fonctionnement :
 - la JVM crée au minimum un groupe de threads nommé main
 - par défaut, un thread appartient au même groupe que celui qui l'a créé (son père)
 - `getThreadGroup()` : pour connaître son groupe

Les « ThreadGroup »



```
ThreadGroup group1 = new ThreadGroup("GP1");  
Thread p1 = new Thread(group1, "P1");  
Thread p2 = new Thread(group1, "P2");  
Thread p3 = new Thread(group1, "P3");  
ThreadGroup groupe11 = new ThreadGroup(group1, "GP11");  
Thread p4 = new Thread(groupe11, "P4");  
Thread p5 = new Thread(groupe11, "P5");
```

- la classe ThreadGroup permet de constituer une arborescence de Threads et de ThreadGroups
- elle donne des méthodes classiques de manipulation récursives d'un ensemble de threads :
suspend(), stop(), resume(), ...
- et des méthodes spécifiques : setMaxPriority(), ...

Méthodes deprecated

~~stop, suspend, resume, destroy~~ : deprecated

- Leur usage entraîne souvent des situations d'interblocage
 - Le thread **A** arrête le thread **B**
 - **B** arrête toutes les méthodes en suspens y compris `run()`
 - **B** rend les verrous de tous les objets qu'il a verrouillé.
- A** n'a pas de vision sur le point d'arrêt de **B** et
B ne coopère pas.
- Cela peut mettre des objets dans un état inconsistant
(*exemple virement de compte à compte*)

Multithreading Java