

M.3.3.2 Programmation Objet Avancée

Multithreading Java





Plan



جامعة محمد الخامس بالرباط
Université Mohammed V de Rabat



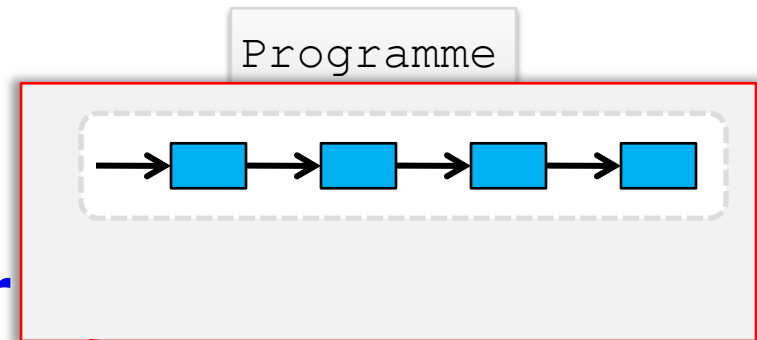
- **Multithreading ?**
- **Intérêt des threads**
- **Définition de threads**
- **Cycle de vie des Threads**
- **Concurrence d'accès**

Multithreading ?

→ Etend l'idée du multitâche

- Un programme effectue plusieurs tâches «en même temps»

→ Processus = flot de contrôle (séquence d'instructions)



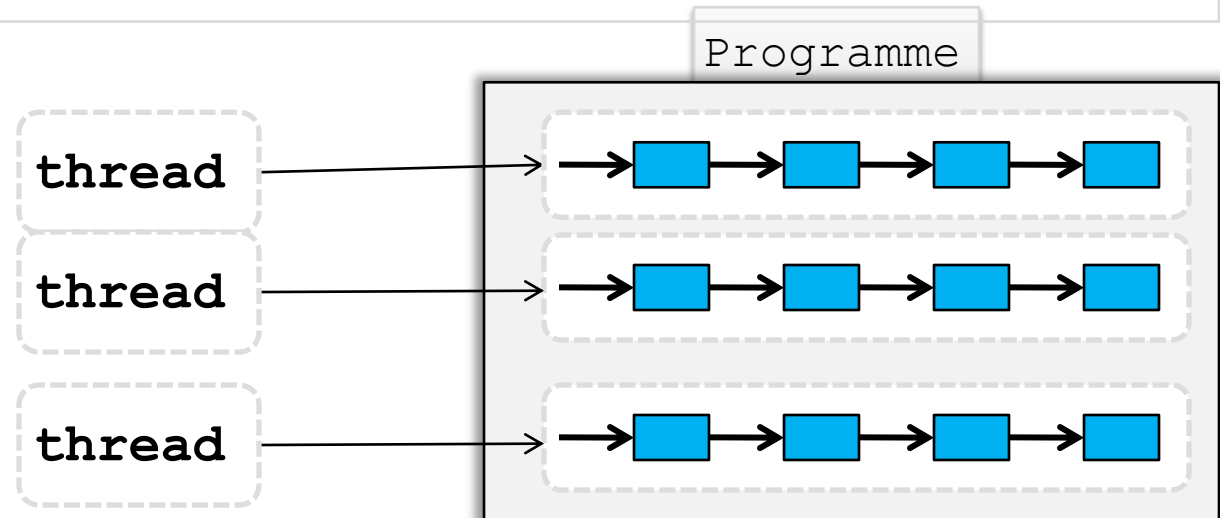
→ Thread = Processus léger (*lightweight process*)

- il s'exécute dans le contexte d'un programme
- utilise les ressources allouées pour ce programme et son environnement

Multithreading ?

→ Programmation concurrente (multithreading)

- Possibilité d'exécuter plusieurs threads simultanément dans le même programme

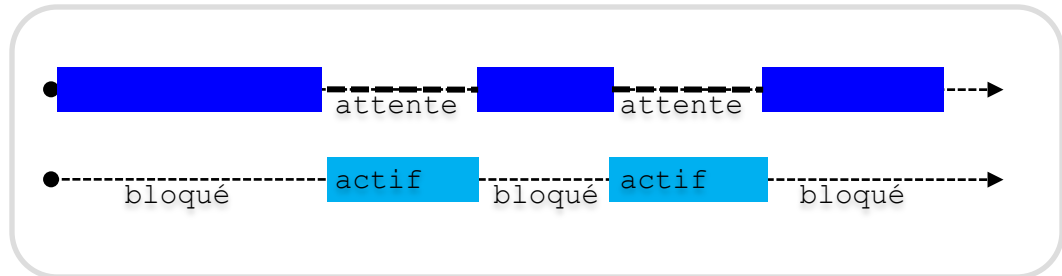


Intérêt des Threads

→ Faire un traitement en tâche de fond

Thread **principal**

Thread **secondaire**



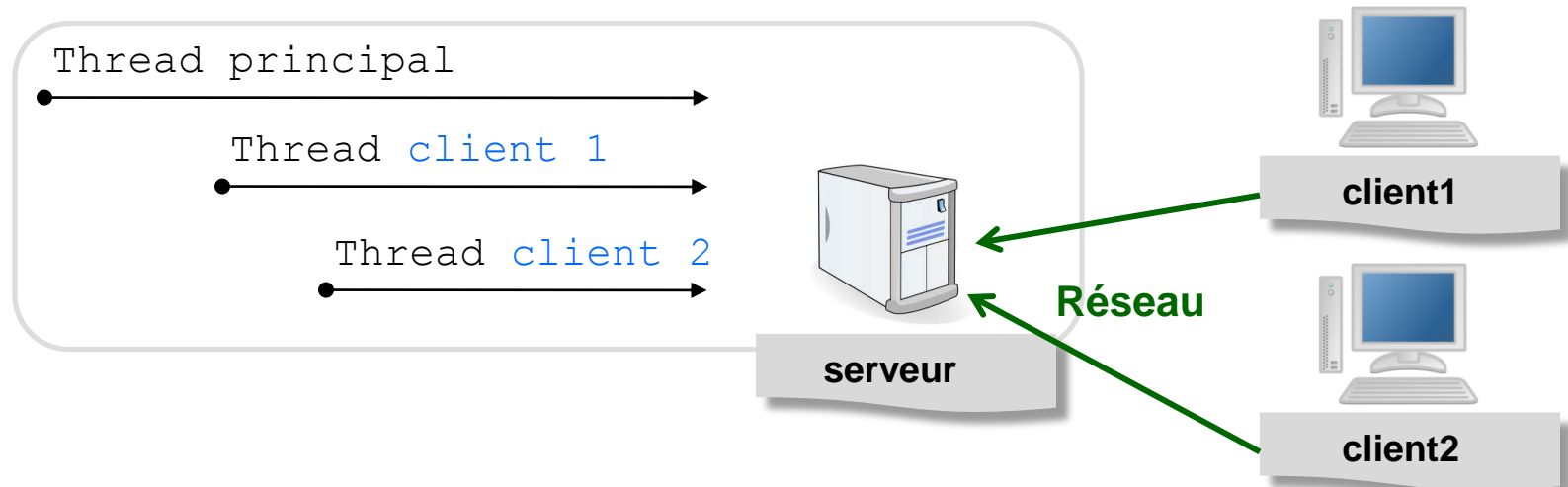
↕ Lancé avec une priorité inférieure, il s'exécute en permanence sans perturber le traitement principal

Exemples

- ▶ Un **GC** lancé en arrière plan dans un thread séparé
- ▶ Coloration syntaxique des **éditeurs**
- ▶ 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

➔ Optimise *le temps d'attente* des clients

- ▶ **Serveur web** qui doit servir plusieurs requêtes concurrentes
- ▶ **Navigateur** qui doit télécharger plusieurs images
simultanément



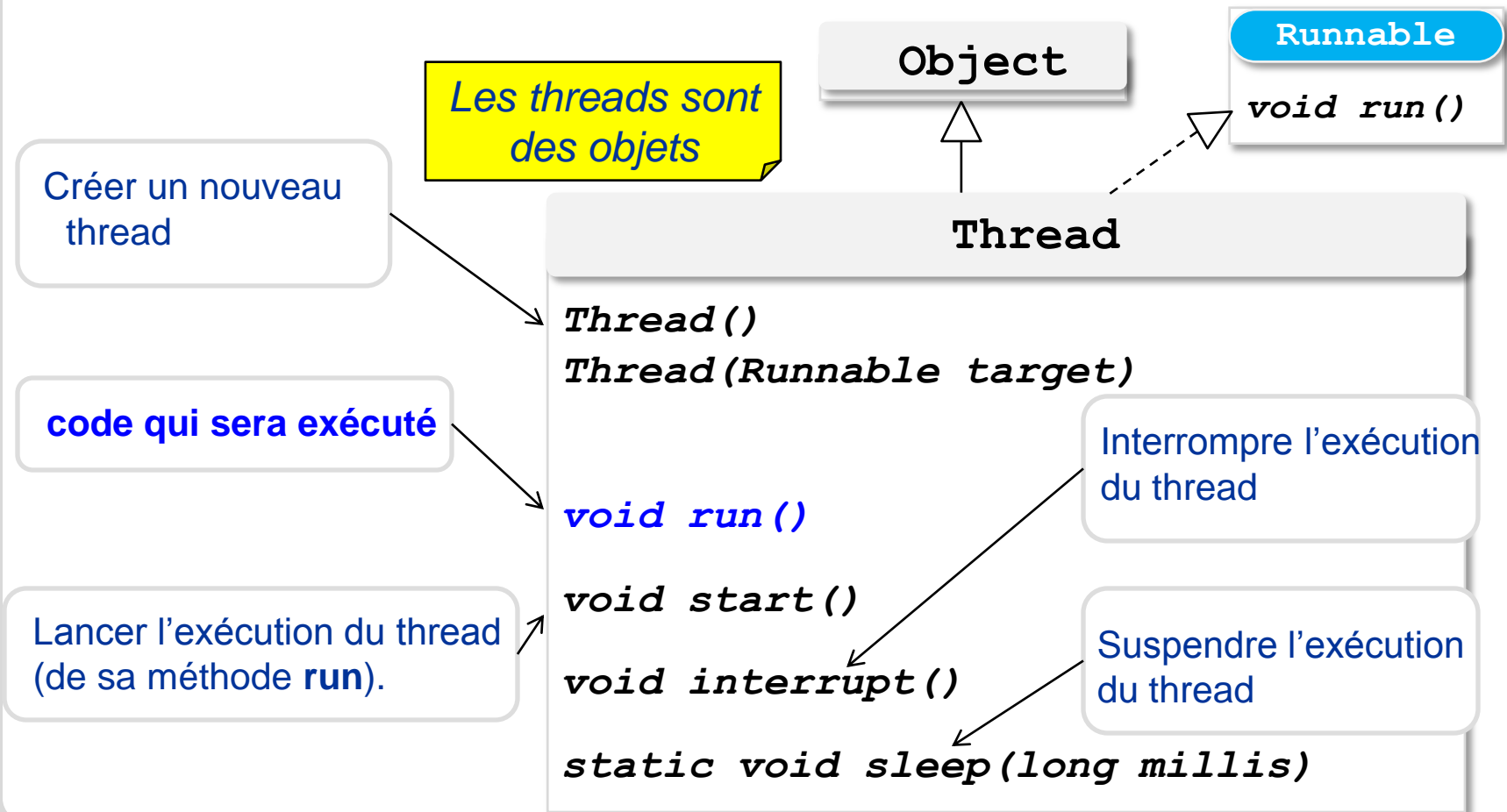
Intérêt des Threads



- ➔ **Puissance d'exécution** (*parallélisme*)
- ➔ **Meilleures performances**
 - ◆ Plus légers donc *plus faciles à créer et à détruire*
- ➔ **Partage des ressources système**
 - ◆ les threads partagent les données
 - *les processus ont leurs propres variables*
 - ◆ La communication inter-threads devient plus simple
 - *Pratique pour les E/S*
- ➔ **Intégrés au langage** (package **java.lang**)

Définition de threads

➔ **java.lang.Thread** au centre de la gestion des threads



Définition de threads

→ Les threads doivent implémenter l'interface **Runnable**

Solution 1

1. Sous classer Thread

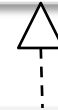
```
Class ThreadSorter extends Thread {
    List list;
    public ThreadSorter(list) {
        this.list = list;
    }
}
```

2. Redéfinir run()

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

Runnable

void run()



Thread

```
Thread()
Thread(Runnable)
...
void run()
```



ThreadSorter

void run()

Définition de threads

Solution 1

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);
sorter2.start();
...
```

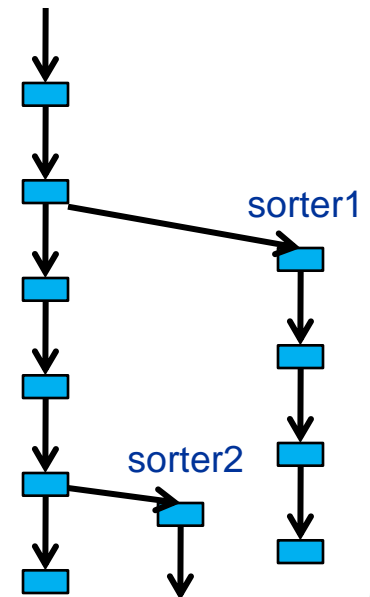
Appeler **run()** directement,
exécute la tâche dans le
même thread (aucun
nouveau thread n'est lancé)

Thread



ThreadSorter

void run()



Définition de threads

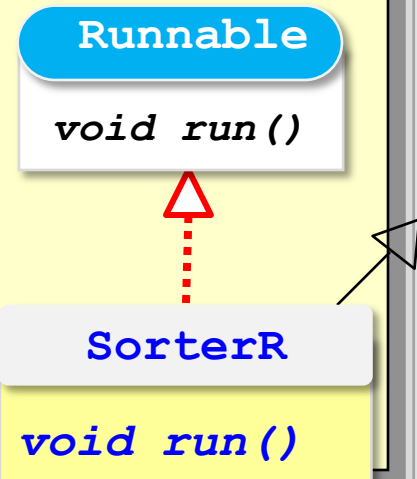
Solution 2

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

Solution 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

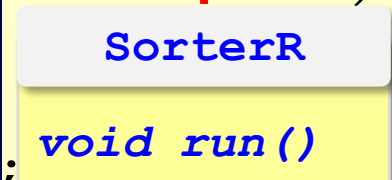
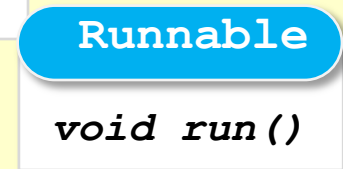
```
th1.start();
```

```
...
```

```
Thread th2 = new Thread(...);
```

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

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





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();
    }

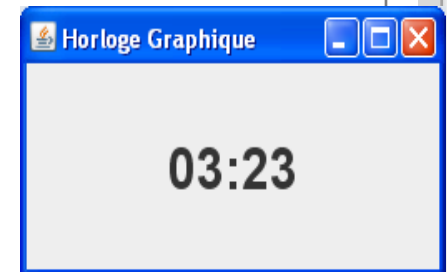
    public static void main(String[] args) {
        JFrame f= new JFrame("Horloge Graphique");
        f.getContentPane().add(new Horloge());
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
}
```

Runnable

void run()

Afficheur

void run()



Définition de threads

Exemple

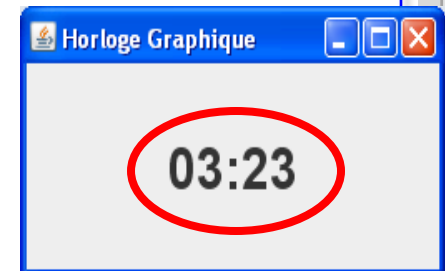
```
private class Afficheur implements Runnable {
    int s=0, m=0; String min="", sec="";
    public void run() {
        while(true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                return;
            }
            s++; if(s==60)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

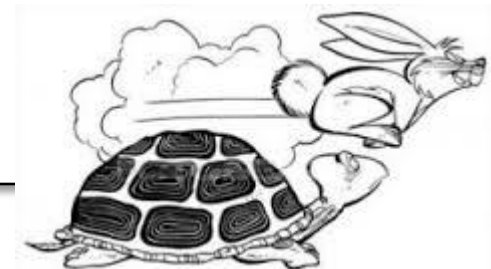


InterruptedException ?

Exemple

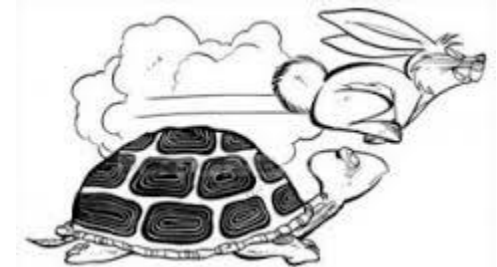
Implémenter la fable du **lièvre et de la tortue**

- 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



Exemple

```
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;  
        }  
    }  
}
```

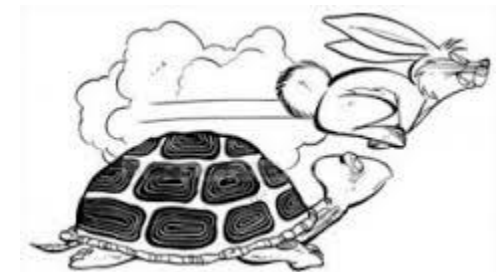


Exemple

```
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





Interruption de threads

Le thread termine lorsque sa méthode `run()` retourne

- Nous voulons rarement qu'une activité **cesse immédiatement** : ~~`Thread.stop()`~~ *deprecated*
... on verra pourquoi
- Nous utilisons plutôt `Thread.interrupt()` pour demander la terminaison d'un thread

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

→ La méthode `run()` a souvent la forme :

```
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 *demande 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

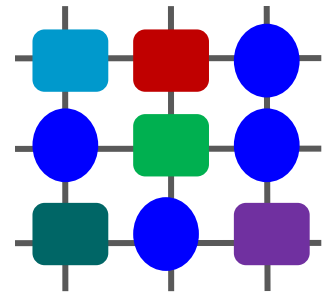
```

} 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;
}
}
}

```

Cycle de vie des Threads

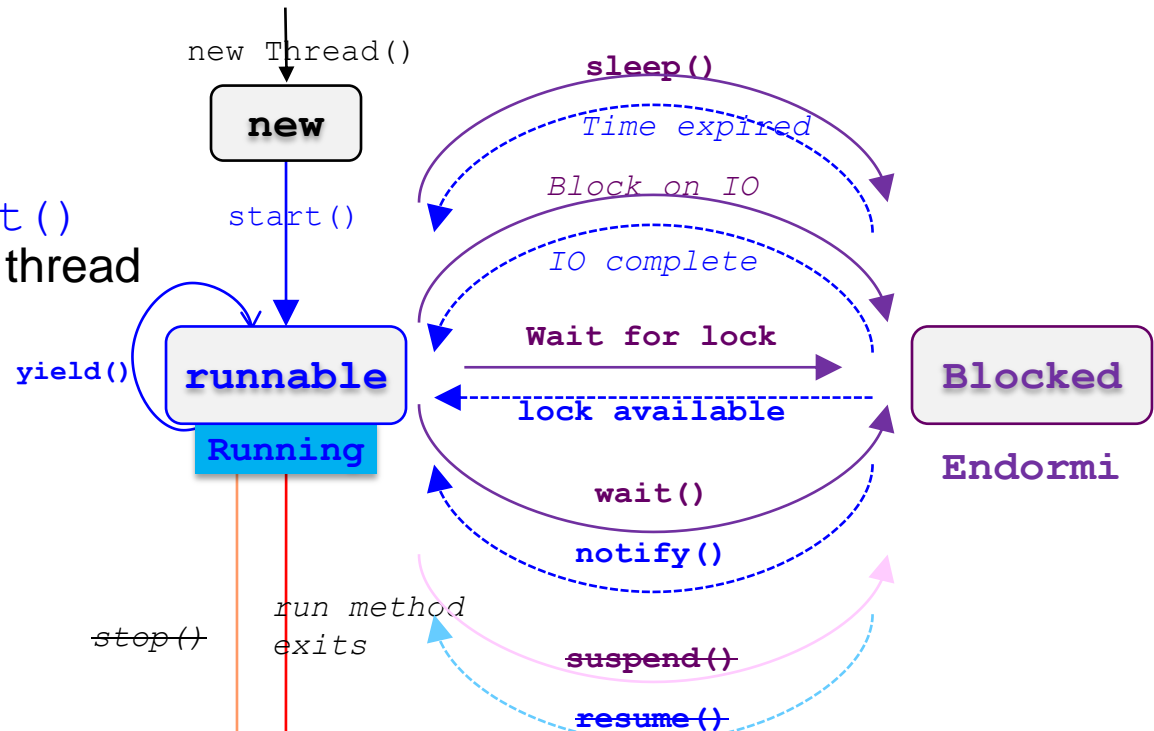


Cycle de vie des threads

Créé

→ Non encore actif : `start()`
lance `run()` et active le thread

Actif



`public final void notify()`

Réveille un seul thread en attente sur le moniteur de cet objet.

`public final void wait()`

Le thread courant attend « **dort** » jusqu'à ce qu'un autre thread appelle `notify()` ou `notifyAll()` pour cet objet.

`static void sleep(long millis)`

Endort le thread pendant un intervalle de temps.

Cycle de vie des threads

CPU

Priorité

- ➔ 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 :
 - **Entre** `[MIN_PRIORITY, MAX_PRIORITY]`
- ➔ Modifier les priorités (niveaux absolus) des threads :
 - `setPriority(int)` throws `IllegalArgumentException`



Cycle de vie des threads

Priorité

Ordonnancement préemptif

- La JVM choisit d'exécuter le thread actif qui a la plus haute priorité : ***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()`, implicitement en passant à l'état endormi (`wait()`, `sleep()` ou `suspend()`)



Cycle de vie des threads



Exemple : priorités

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

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

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

Cycle de vie des threads


Exemple : priorités

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

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

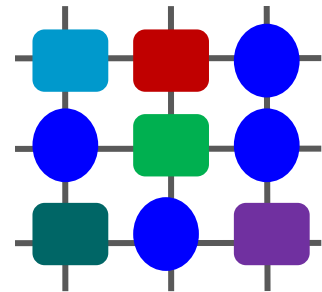
Avec yield()

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



M.3.3.2 Programmation Objet Avancée

Concurrence d'accès

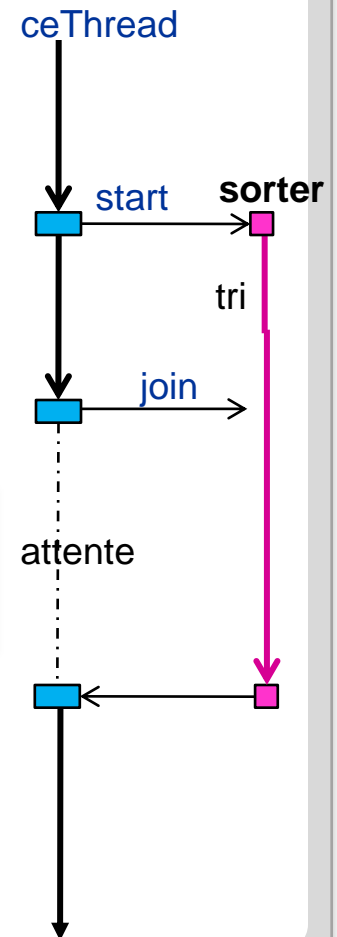


Concurrence d'accès

➔ Un thread **peut attendre** qu'un autre se termine (`join`)

```
Liste maListe;
...
Thread sorter = new MonThreadSorter(maListe);
sorter.start();

byte[] data = readData(); //la liste doit être triée
// il faut donc attendre la fin du thread sorter
try {
    sorter.join();
}
catch (InterruptedException e) {
    // ignorée : lancée si un autre thread l'a interrompu
}
```

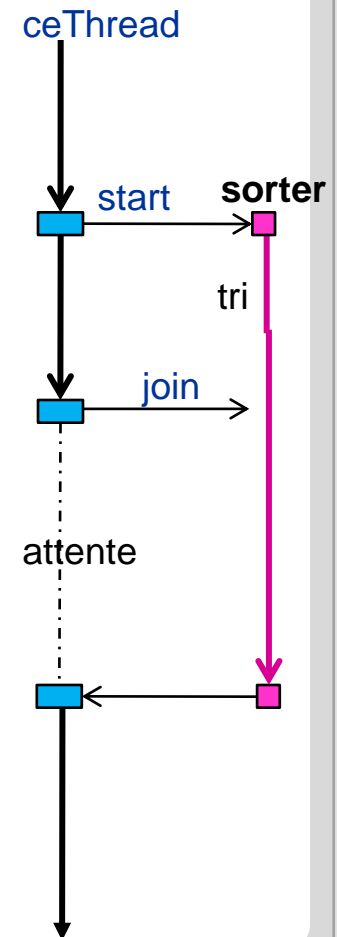


Concurrence d'accès

→ Ce type de synchronisation reste insuffisant

- 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

→ Il faut garantir **l'accès exclusif** à un objet



Concurrence d'accès



Banque

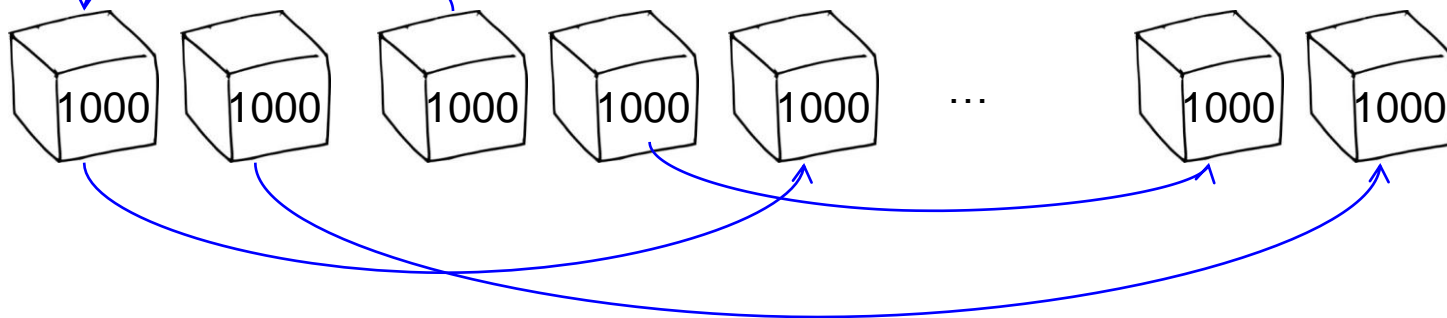
```
final double[] comptes;
```

```
Banque(nbCompt, soldeInit)
```

```
transferer(int,int,double)
```

```
double totalSoldes()
```

```
Banque banque = new Banque(100, 1000);
```



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

Concurrence d'accès

→ La classe 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) {}
    }
}
```

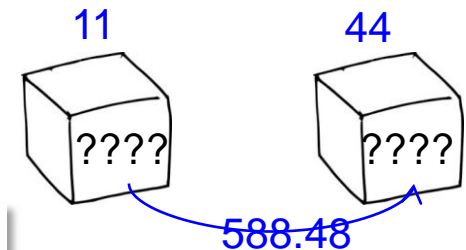


Concurrence d'accès



→ 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 !

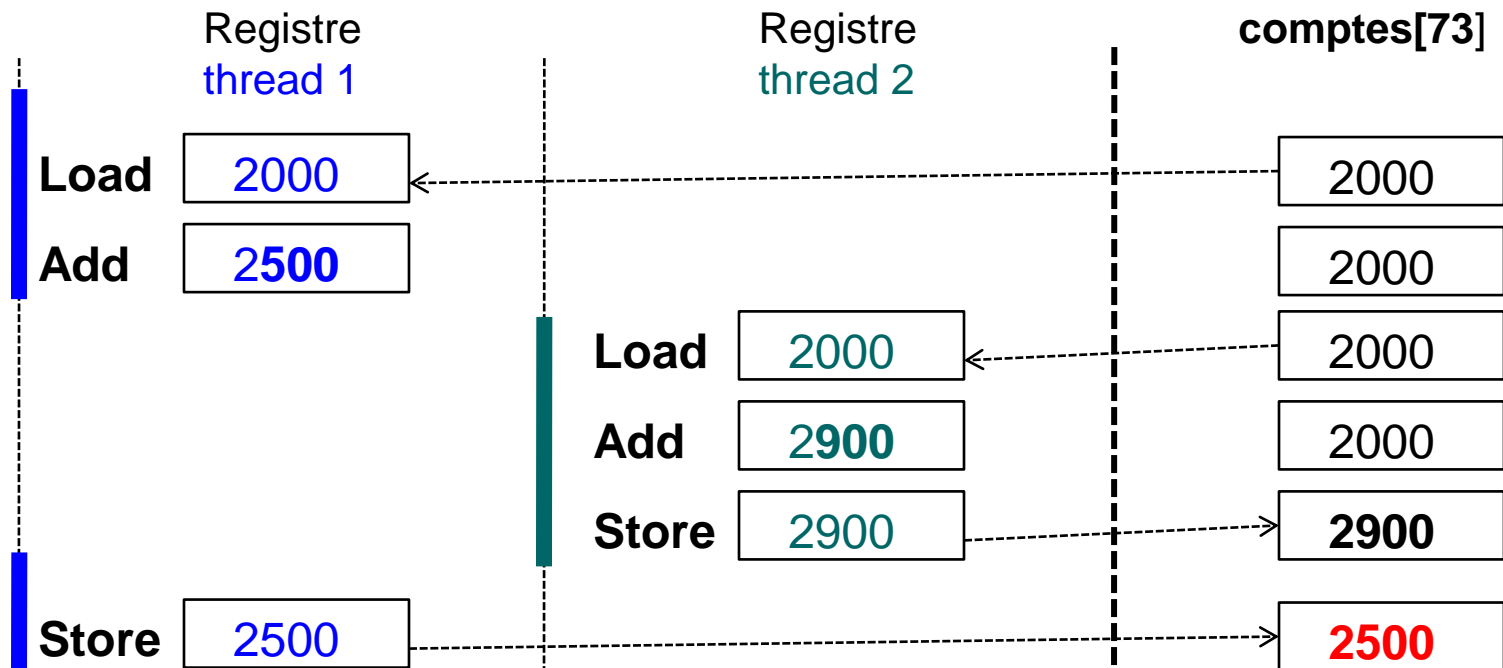
Ce n'est pas très sûr !!!!

Concurrence d'accès

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



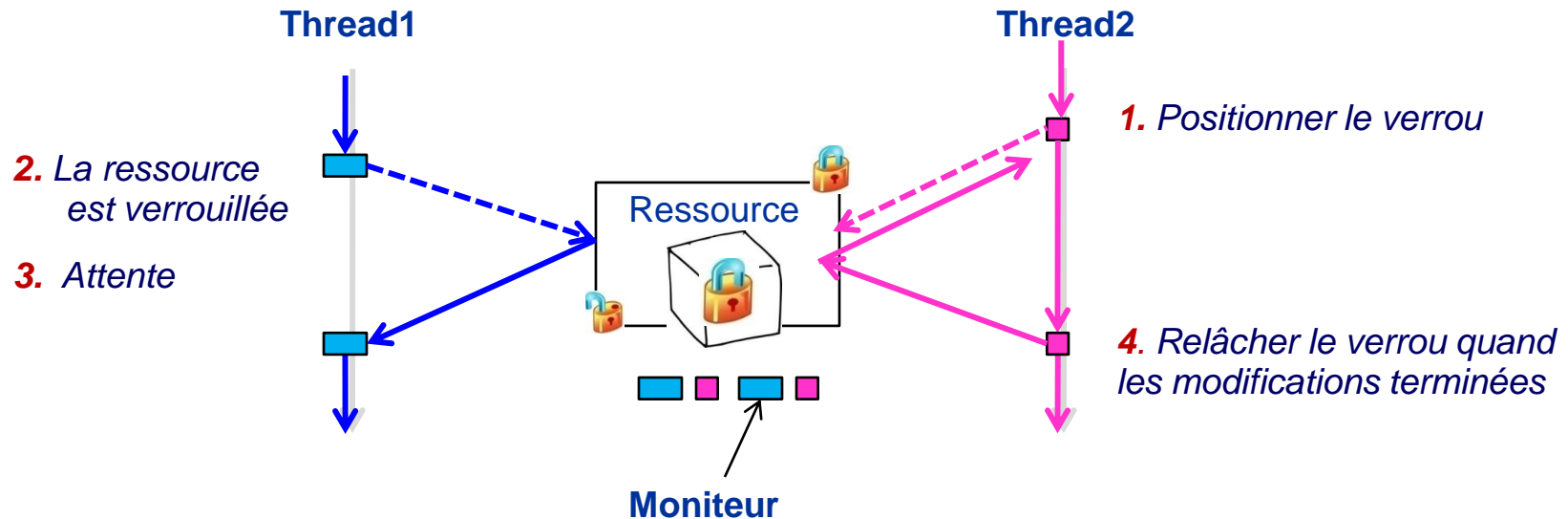
1. **Load** `comptes[vers]` dans un registre
2. **Add** `montant`
3. **Store** remettre le résultat dans `comptes[vers]`



Concurrence d'accès

Synchronisation JDK5.0

- 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

Synchronisation

JDK5.0

A P I

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;

A P I

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

Synchronisation

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

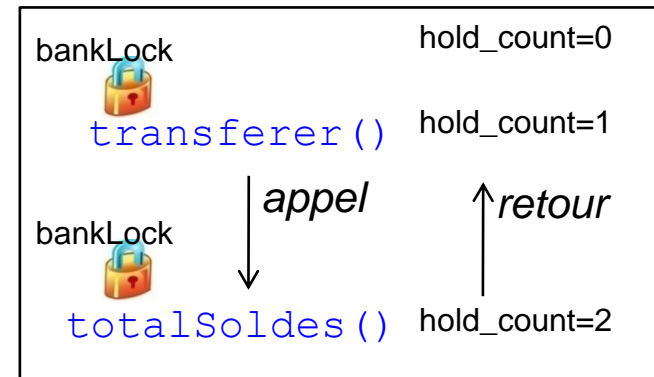
Concurrence d'accès

Synchronisation

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

Synchronisation

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

Synchronisation

Solde insuffisant ?

```
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

Synchronisation

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

Synchronisation

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

Synchronisation

Conditions

A P I

java.util.concurrent.locks.**Lock** 5.0

interface

- void **await()**
puts this thread on the wait set for this condition.
- void **signalAll()**
***unblocks all** threads in the wait set for this condition*
- void **signal()**
***unblocks one randomly** selected thread in the wait set for this condition*

- 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

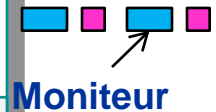
Synchronisation JDK1.0

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

Équivalent à

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

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





Concurrence d'accès



Synchronisation JDK1.0

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



➔ Le verrou est libéré soit en quittant la partie synchronisée soit en appelant la méthode **wait()**.

A P I

java.lang.**Object** 1.0

- **void wait()**
*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()**
unblocks the threads that called wait on this object
- **void wait(long millis)**
...

Concurrence d'accès

Synchronisation JDK1.0

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



Moniteur

`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

Concurrence d'accès

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



→ Ne pas synchroniser :

- les données peuvent être *corrompues*

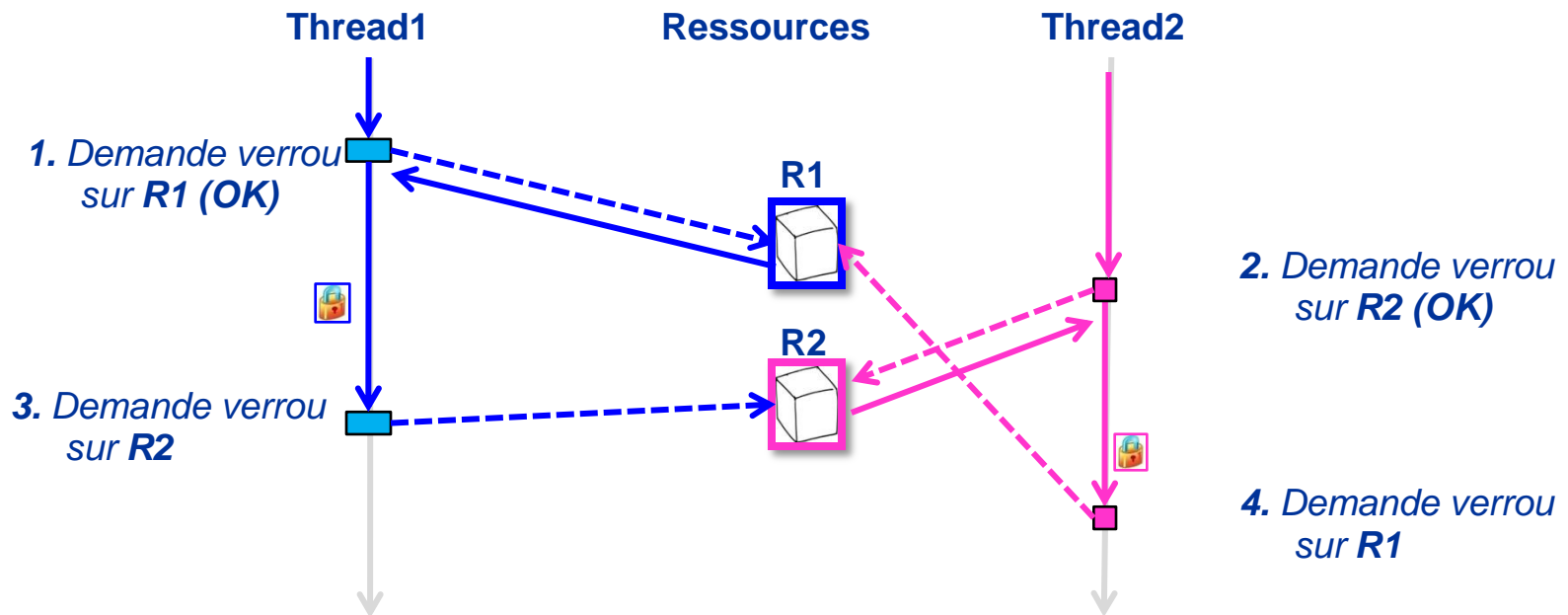
→ Mal synchroniser :

- *Les performance sont réduites* : l'entrée dans une méthode `synchronized` est coûteuse (prise du verrou)
- *Risque d'interblocage (deadlock) ...*

Concurrence d'accès

Interblocage

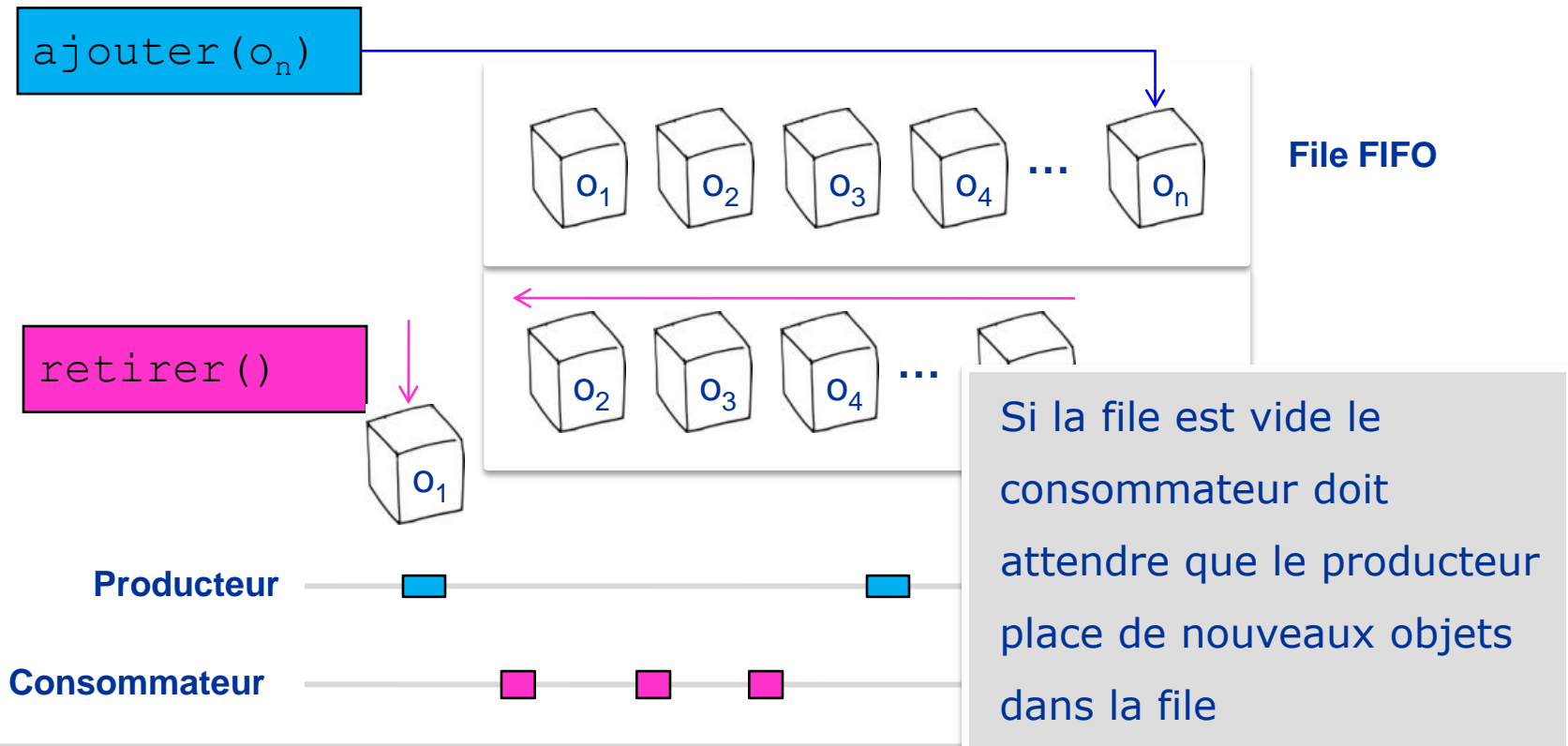
→ *chaque thread attend que l'autre relâche un verrou*



Concurrence d'accès

Interblocage

→ File FIFO partagée par deux threads



Concurrence d'accès

Interblocage

```
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

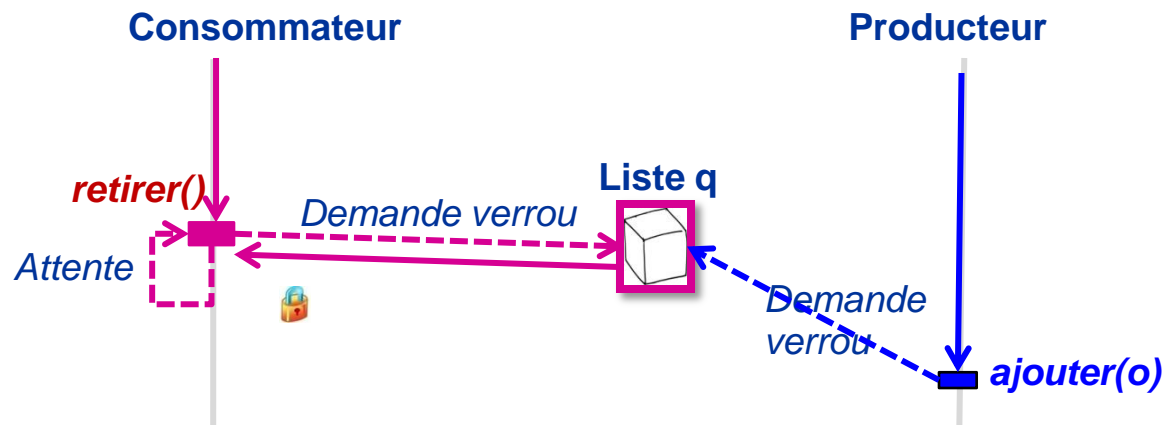
Si la liste q est vide **attendre**
qu'un objet soit mis dans la file

Interblocage

Concurrence d'accès

Interblocage

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



- Le producteur est interrompu.
- Il doit attendre que le consommateur relâche son verrou pour pouvoir faire un ajout

Concurrence d'accès

Interblocage

Solution

```
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(0);
    }

    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

Interblocage

Attention

```
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

Pourquoi ...

~~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*)



M.3.3.2

Programmation Objet Avancée