Programmation concurrente

Problématiques associées

Chap #3.2

Introduction

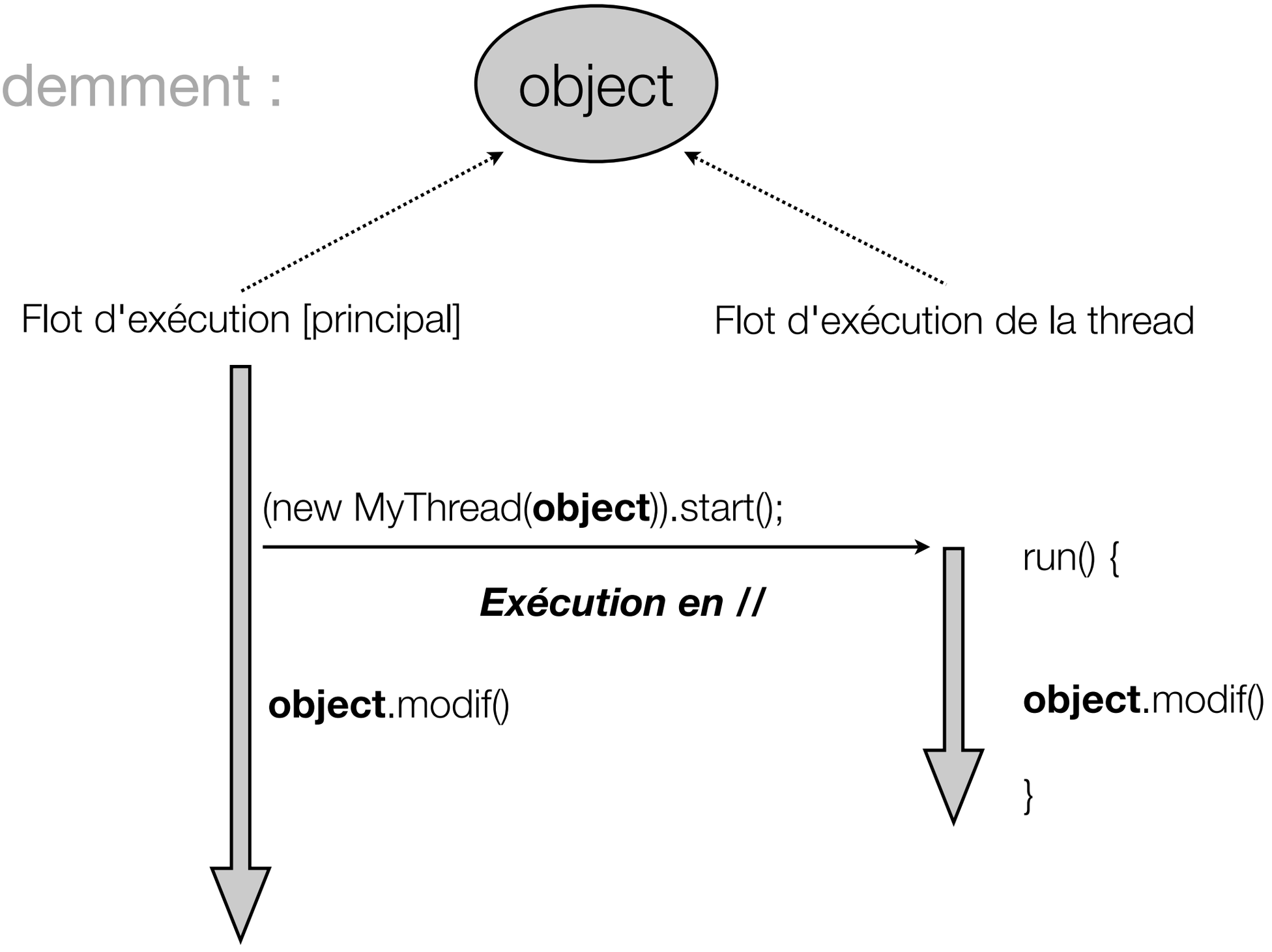
- **L’exécution simultanée de plusieurs tâches concurrentes ne s’effectue pas sans difficultés.**
- Plusieurs threads peuvent accéder de manière concurrente à une même portion de code (qui manipule des données).
- Les threads communiquent essentiellement par le **partage d’accès sur une même zone mémoire**. Il peut s’agir d’une variable globale (en procédural), statique (en orienté-objet), ou tout simplement parce-que deux variables pointent la même zone mémoire dans le tas.
- Cette forme de communication est très efficace mais fait apparaître deux types de problèmes:

  1. **L'interférence entre les threads**
  2. **Les inconsistances mémoire** (vues précédemment, cf. Chap #3.1).

- **Les techniques de synchronisation permettent de prévenir l’apparition de ces 2 problèmes.**

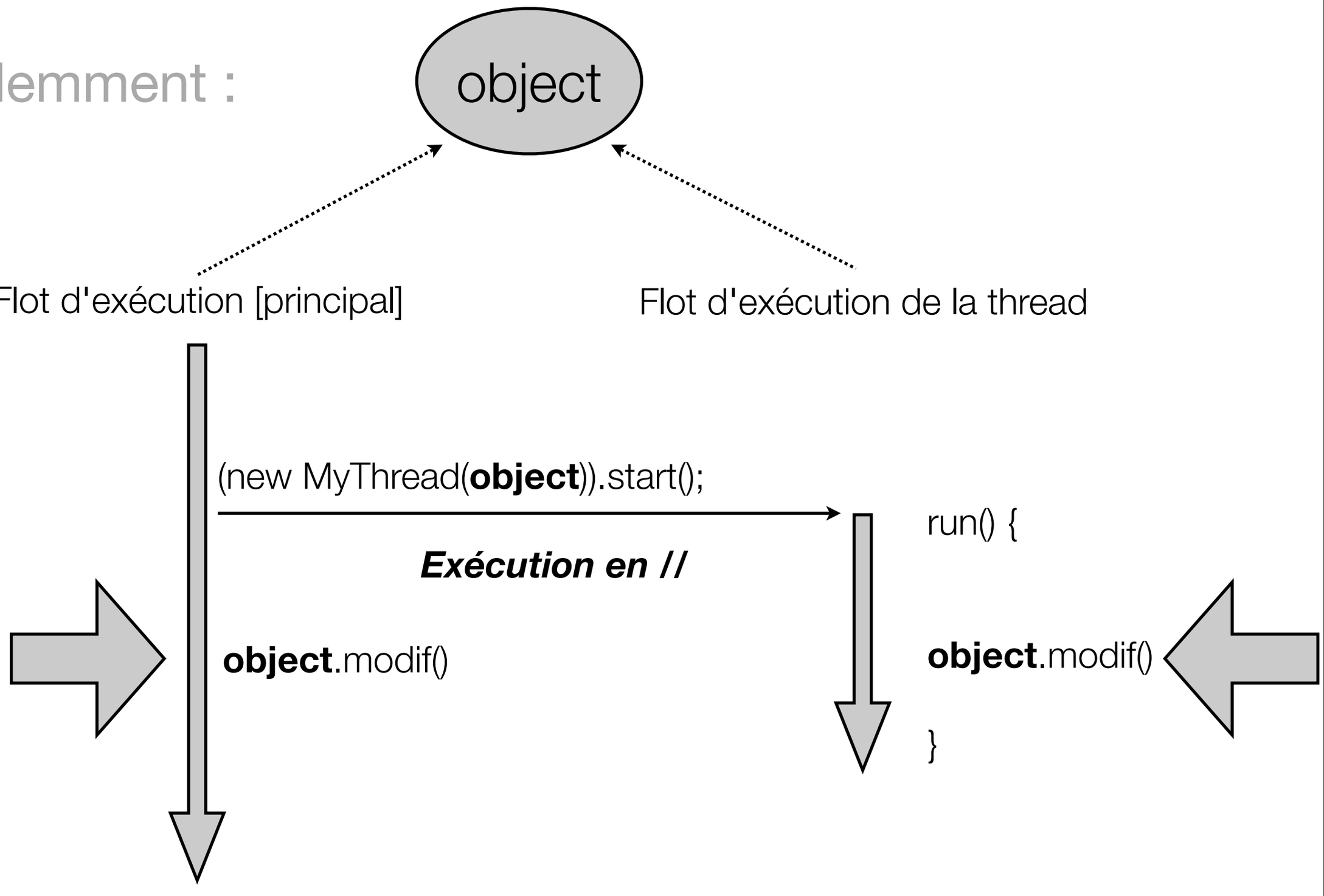
Introduction

Précédemment :



Clash potentiel

Précédemment :



Interférence entre threads

- Des erreurs peuvent apparaître si plusieurs threads accèdent à **des données partagées :**
  - Une même instance partagé par plusieurs threads qui offre des modificateurs sur l'état de l'instance.
  - Un même modificateur peut être appelé de manière concurrente par plusieurs threads possédant une même référence sur cette instance.

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
```

c++ = 3 étapes distinctes

1. obtenir la valeur de c
2. incrémenter la valeur de 1
3. stocker la valeur incrémentée dans c

Interférence entre threads

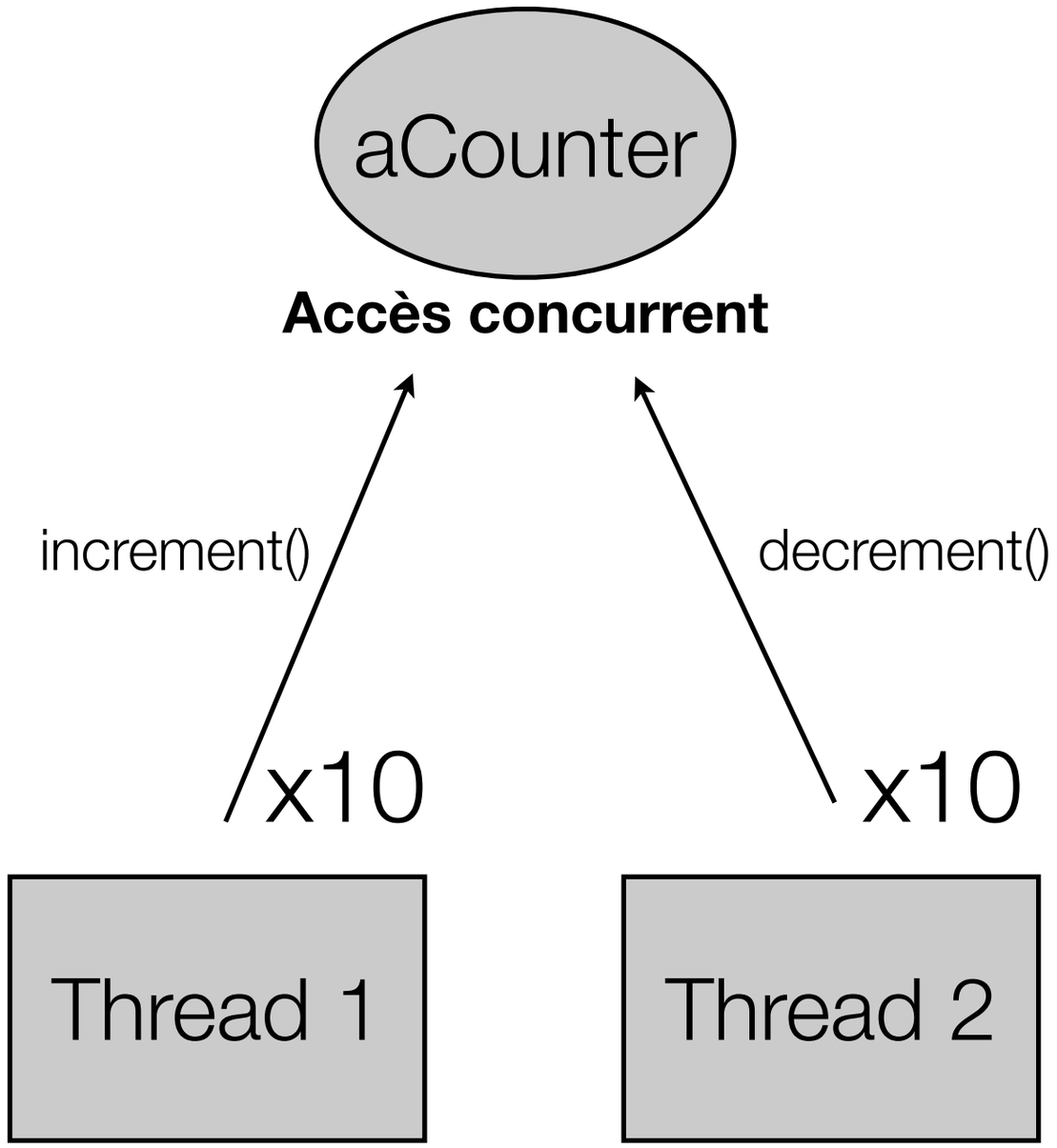
- Supposons maintenant qu'une thread A invoque la méthode *increment()* en même temps qu'une autre thread B invoque *decrement()* sur une même instance de *Counter.*

- La valeur initiale de *c* est 0, la séquence d'action pourrait se retrouver emmêlée de cette façon:

  - Thread A: obtenir la valeur de *c*
  - Thread B: obtenir la valeur de *c*
  - Thread A: incrémenter la valeur de 1 → 1
  - Thread B: décrémenter la valeur de 1 → -1
  - Thread A: stocker la valeur incrémentée dans *c* → *c* == 1
  - Thread B: stocker la valeur incrémentée dans *c* → *c* == -1
  - **Au final *c* vaut -1**

Interférence entre threads


```
public static void main(String [] args) {
      Counter aCounter = new Counter();
      (new ThreadType1(aCounter)).start();
      (new ThreadType2(aCounter)).start();
}

public class ThreadType1 extends Thread {
      Counter counter;

      public ThreadType1(Counter counter) {
            this.counter = counter;
      }

      public void run() {
            for(int i=0;i<10;i++) { counter.increment() };
      }
}

public class ThreadType2 extends Thread {
      Counter counter;

      public ThreadType2(Counter counter) {
            this.counter = counter;
      }

      public void run() {
            for(int i=0;i<10;i++) { counter.decrement() };
      }
}
```



Synchronisation
Section critique

- **En programmation concurrente, une section critique est une portion de code dans laquelle il doit être garanti qu'il n'y aura jamais plus d'une thread simultanément.**

- **Il est nécessaire de définir des sections critiques lorsqu'il y a accès à des ressources partagées par plusieurs threads**, de manière à ce que les opérations de lecture et d'écriture de données soient **atomiques** (≠ décomposables).

- Dans l'exemple précédent (Counter), si l'on définit les méthodes d'incrémentation et de décrémentation comme sections critiques, alors on évite le problème d'interférence.

- Une section critique peut être protégée par un mutex, un sémaphore ou d'autres techniques de programmation concurrente plus avancées.

Synchronisation

Outils de synchronisation

- Ces différentes primitives implémentent toutes une forme plus ou moins évoluée de verrouillage qui sert à mettre en place la synchronisation des entités concurrentes (sur une ressource, ou plus généralement sur une section critique):

  - **Mutex**
  - **Sémaphores**
  - **Moniteurs**

Synchronisation
Mutex, principes

- Un Mutex (anglais : Mutual exclusion, Exclusion mutuelle) est la primitive la plus simple de synchronisation à utiliser pour éviter que des ressources partagées d'un système ne soient utilisées en même temps.

- **Les deux seules opérations possibles sur un mutex sont le verrouillage et le déverrouillage**. **Elles sont elle même des sections critiques** dont l'implémentation est effectuée au plus bas niveau par masquage des interruptions, lecture/écriture en un cycle, ...)

- **2 threads ne peuvent pas avoir le même mutex verrouillé en même temps.**

- **Si une thread B essaye de verrouiller un mutex alors qu'il a déjà été verrouillé par une autre thread A, alors la thread B reste bloquée sur le mutex tant qu'il n'a pas été relâché par la thread A.**

Synchronisation

Mutex et section critique

- Implémentation très simple d'une section critique à l'aide d'un mutex. Lorsqu'une thread arrive sur la section critique elle doit:

1. **Verrouiller le mutex associé à la section critique.**
2. **Exécuter la section critique (qui lit ou met à jour des données,...).**
3. **Déverrouiller le mutex.**

- Lorsque l'on ne verrouille pas une ressource pour les opérations de lecture, il existe un risque de corruption de données lors d'une opération d'écriture.

- Le besoin d'atomicité n'existe cependant pas lorsqu'une variable statique est initialisée au démarrage du processus, puis jamais modifiée.

Synchronisation

Mutex et section critique

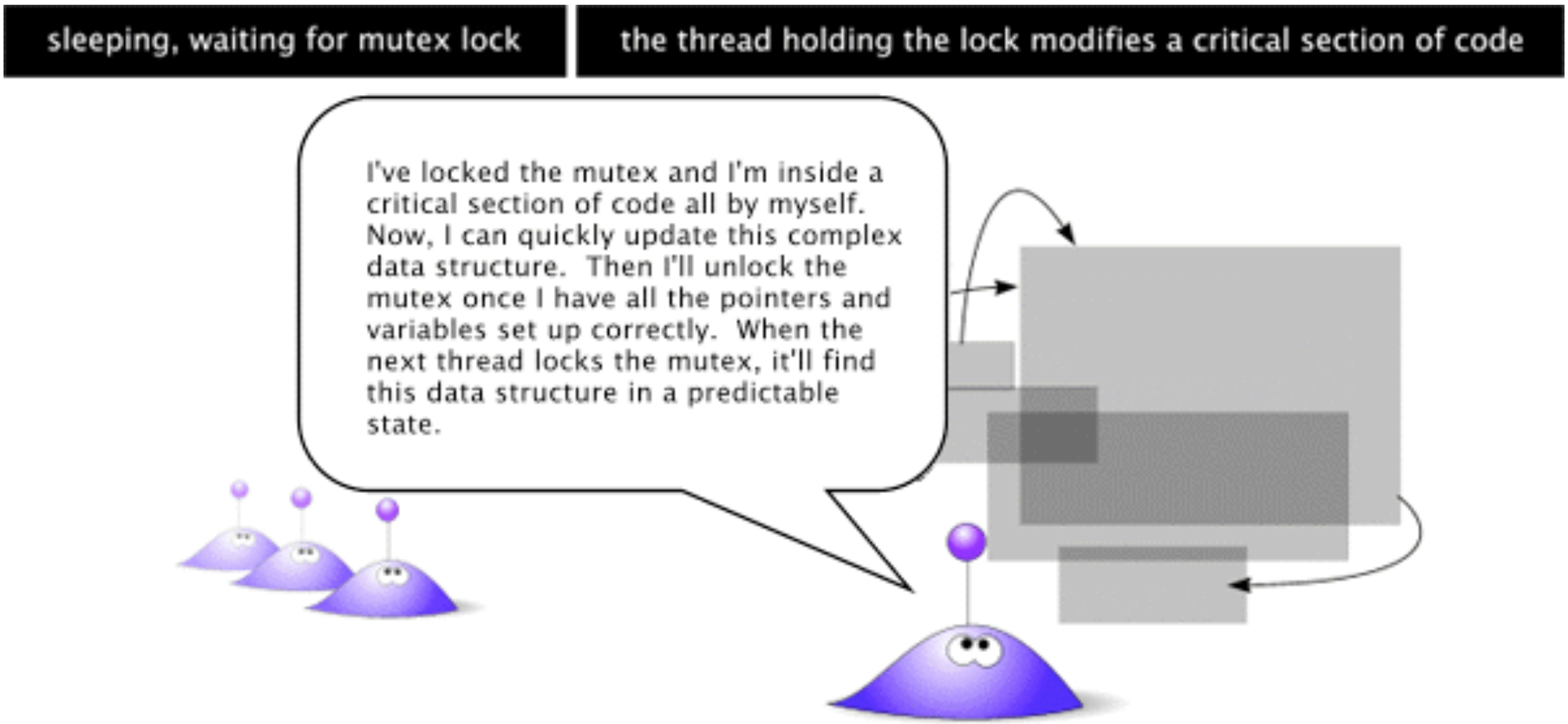


image © IBM, 2000

Synchronisation

Mutex et Java

- Java ne possède d'implémentation **explicite** de la notion de mutex.
- Mais, tout objet Java peut être utilisé comme mutex via l'instruction *synchronized* sur un bloc de code (cf. transparents suivants).
- Coder la classe mutex en TP !

Synchronisation

Instruction `synchronized`

- Elle s’utilise sur une méthode ou un bloc de code : la méthode ou le bloc de code devient ainsi une zone d’exclusion mutuelle.

- **Principe #1:** au plus une thread simultanément en train d'exécuter du code dans une zone synchronized (elle possède le verrou).

- **Principe #2:** Si une thread en train d'exécuter du code en zone synchronized alors les autres thread demandeuses restes bloquées à l’entrée.

- **Principe #3:** Dès que la thread en cours d'exécution sort de la zone synchronized (libère le verrou) alors **la 1er thread** restée bloquée est libérée (et prend le verrou), les autres restent en attente.

Synchronisation
Bloc synchronized


```
public void ecrire(...) {
...
 synchronized(objet) { ... } // section critique
...
}
```

- *objet* : objet verrou de référence pour assurer l'exclusion mutuelle. **Il s'agit d'un mutex** "de facto", mais on ne le manipule pas directement.

- L'**entrée** par une thread dans le bloc *synchronized* est associée implicitement à une demande de **verrouillage** sur le mutex *objet.*

- La **sortie** du bloc *synchronized* est associée implicitement à une demande de **déverrouillage** sur le mutex *objet*.

- **Attention:** on n'écrit pas explicitement d'instruction de verrouillage ou déverrouillage du mutex.

Synchronisation
Méthode `synchronized`

- `public [static] synchronized void ecrire(...) { ... }`

- Contrairement au bloc de code synchronisé, il n'est pas nécessaire de préciser l'objet de référence pour le verrouillage.

- En effet, chaque instance Java est automatiquement associé à un verrou de référence, c'est ce verrou (un mutex en pratique) qui est utilisé par la méthode *synchronized*.

- **Conséquence logique:** une seule thread en cours d'exécution en même temps dans toutes les méthodes *synchronized* d'une instance (puisqu'elles se synchronisent toutes sur le même verrou).

- **Attention:** le contrôle de concurrence s'effectue au niveau de l'objet → plusieurs exécutions d'une même méthode d'instance *synchronized* possibles dans des instances différentes.

Synchronisation
Remarques

- **Quand une méthode synchronized se termine elle crée automatique un lien “happens-before” avec toute invocation subséquente des autres méthodes synchronisées sur le même verrou de référence** (en pratique les méthodes du même objet).

- Attention au coût d’une zone *synchronized*: l’appel à une méthode est au moins 4x plus long qu’un appel de méthode classique.

- La JVM garantit l’atomicité d’accès aux *byte*, *char*, *short*, *int*, *float* et aux références d’objet.

- **Mais pas d’atomicité sur long ni double !**

Synchronisation
Sémaphores

- Un sémaphore est un mutex “généralisé”, il peut ne pas être binaire.

- La particularité du sémaphore par rapport aux mutex est lié au fait qu’un sémaphore ajoute au verrouillage la notion de **compteur** : tant que le compteur d’un sémaphore n’a pas atteint 0, il est possible pour une thread (et une autre,...) d’“acquérir” à nouveau le sémaphore.

- Trois opérations possible: **initialisation** (avec la valeur du compteur), **acquérir** (historiquement nommée ‘P’ pour ‘Proberen’ en néerlandais) et **relâcher** (historiquement nommée ‘V’ pour ‘Verhogen’).

- Un sémaphore binaire (initialisé à 1) est équivalent à un mutex.

- Un sémaphore non-binaire ne permet pas d’implémenter une section critique en exclusion mutuelle puisque plus d’une thread peuvent acquérir simultanément le sémaphore.

Synchronisation
Sémaphores, aujourd'hui

- Les sémaphores sont toujours utilisés dans les langages de programmation qui n'implémentent pas intrinsèquement d'autres formes de synchronisation.

- Ils sont le mécanisme primitif de synchronisation de beaucoup de systèmes d'exploitation.

- La tendance dans le développement des langages de programmation est de **s'orienter vers des formes plus structurées de synchronisation comme les moniteurs.**

- Outre les problèmes d'interblocage qu'ils peuvent provoquer comme les mutex, les sémaphores ne protègent pas les programmeurs de l'erreur courante qui est de prendre un sémaphore par un processus qui a déjà pris ce même sémaphore, ou d'oublier de libérer un sémaphore qui a été bloqué. Hoare, Hansen, Andrews, Wirth, et même Dijkstra ont jugé le **sémaphore obsolète**.

Synchronisation
Sémaphores et Java

- Java ne possède d'implémentation de la notion de sémaphore.

- Coder la classe sémaphore en TP !

Synchronisation
Moniteurs

- Le concept de moniteur a été mis au point pour apporter **plus de structure et de “discipline”** dans la synchronisation qu’avec l’utilisation directe de mutex ou de sémaphores.
- Un moniteur est une approche pour synchroniser deux ou plusieurs tâches qui utilisent des ressources partagées.
- **Un moniteur est constitué :**
  - d’un ensemble de procédures permettant l'interaction avec la ressource partagée,
  - d’un verrou d'exclusion mutuelle,
  - de variables associées à la ressource,
  - d’un invariant (souvent implicite) que s’engagent à respecter les procédures.

Synchronisation
Moniteurs et Java

- En programmation objet, un moniteur est un objet avec un mécanisme intégré de synchronisation et de gestion de l'exclusion mutuelle (sections critiques).

- **En Java, un moniteur est un objet dont certaines méthodes (d'instance) sont déclarées *synchronized*:**

  - "Il dispose de procédures permettant l'interaction avec la ressource partagée" : ce sont les méthodes d'instance *synchronized* de l'objet.

  - "Il dispose d'un verrou d'exclusion mutuelle" : le mutex implicitement associé à chaque objet.

  - "Il dispose de variables associées à la ressource" : les variables d'instance de l'objet.

- Dans tout moniteur, on dispose de 2 méthodes spéciales pour gérer la synchronisation : **wait() et notify()**.

Synchronisation

Moniteur Java, méthode wait()

- Cette méthode nécessite un accès exclusif à l'objet exécutant, **elle n'est donc disponibles qu'au sein d'un bloc ou d'une méthode synchronized.**
- Lors d'un appel à wait(): `synchronized(this) { ... wait(); ... }`
  - **mise en attente** de la thread exécutante
  - **relâchement de l'accès exclusif** sur la zone synchronized
  - **attente d'un appel à notify()** par une autre thread
  - lorsque notifiée la thread libérée reste en **attente de réacquisition de l'accès exclusif** (Attention: différent de l'état "wait"):
    - OK → Exécution du code "en dessous" du wait()
- Lors de la sortie de la zone synchronized l'accès exclusif à l'objet est relâché.

Synchronisation

Moniteur Java, méthode notify()

- Cette méthode nécessite un accès exclusif à l'objet exécutant, **elle n'est donc disponibles qu'au sein d'un bloc ou d'une méthode synchronized.**

```
synchronized(this) { ... notify(); ... }
```

- La méthode notify() permet de réactiver **une** thread mis en attente sur les instructions wait() **liées au même verrou** (implicite ou non) que le notify().

- Si plusieurs threads sont en attente alors n'y a aucune garantie sur laquelle des threads sera réactivée. Mais la plupart des implémentations de JVM réactivent la première thread bloquée (ne pas reposer sur ce principe !).

Synchronisation

Moniteur Java, méthode notifyAll()

- La méthode notify() n'est pas toujours suffisante pour certains modèles de synchronisation, notamment lors de la compétition pour l'accès à une ressource. Dans ce cas, on donne leur chance à toutes les threads en attente grâce à la méthode **notifyAll()**.

- La méthode notifyAll() réactive **toutes** les threads en attente sur les instructions wait() liées au même verrou (implicite ou non) que le notifyAll().

- **Attention:** attendre le verrou != attendre sur un wait()

- **Corollaire:** cela ne veut pas dire que les threads réactivées auront un accès concurrent au code après le wait(), **seule une thread va réussir à prendre le verrou**. Les autres vont rester en attente du verrou mais **il ne sera pas nécessaire d'appeler à nouveau notify() pour les réactiver**.

- En pratique, si plusieurs thread peuvent potentiellement être en attente, il est recommandé de ne faire aucune supposition sur l'ordre de réactivation des threads et d'utiliser la méthode notifyAll().

Synchronisation
Conclusion

- On ne connaît pas, aujourd'hui, d'algorithmes ou de techniques parfaites pour gérer la synchronisation des threads.
- Les méthodes présentées ici sont toutes faillibles dans des conditions précises.
- Le bon usage des primitives de synchronisation repose sur les épaules du programmeur qui doit essayer de prévoir autant que possible les situations d’interblocage et de famine.

Bibliographie

- Roscoe, A. W. (1997). The Theory and Practice of Concurrency. Prentice Hall. ISBN 0-13-674409-5.

- Taubenfeld, Gadi (2006). Synchronization Algorithms and Concurrent Programming. Pearson / Prentice Hall, 433. ISBN 0131972596.

- **Consulter l'API Java :**
  http://java.sun.com/j2se/1.5.0/docs/api/

Fin du cours #4