Programmation concurrente

Problématiques associées

Chap #3.2

Introduction

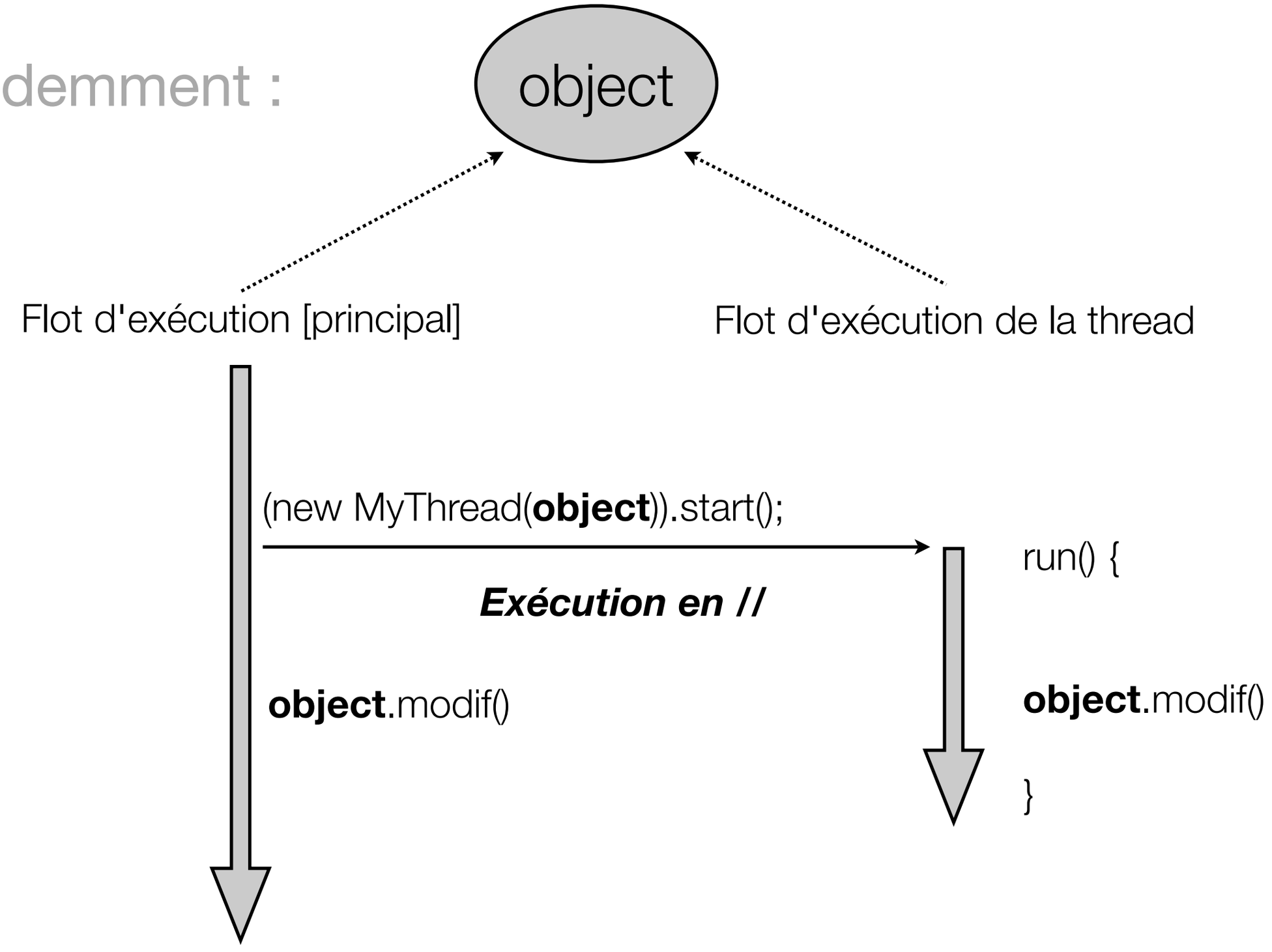
- **L’exécution simultanée de plusieurs tâches concurrentes ne s’effectue pas sans difficultés.**
- Plusieurs threads peuvent accéder de manière concurrente à une même portion de code (qui manipule des données).
- Les threads communiquent essentiellement par le **partage d’accès sur une même zone mémoire**. 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.**

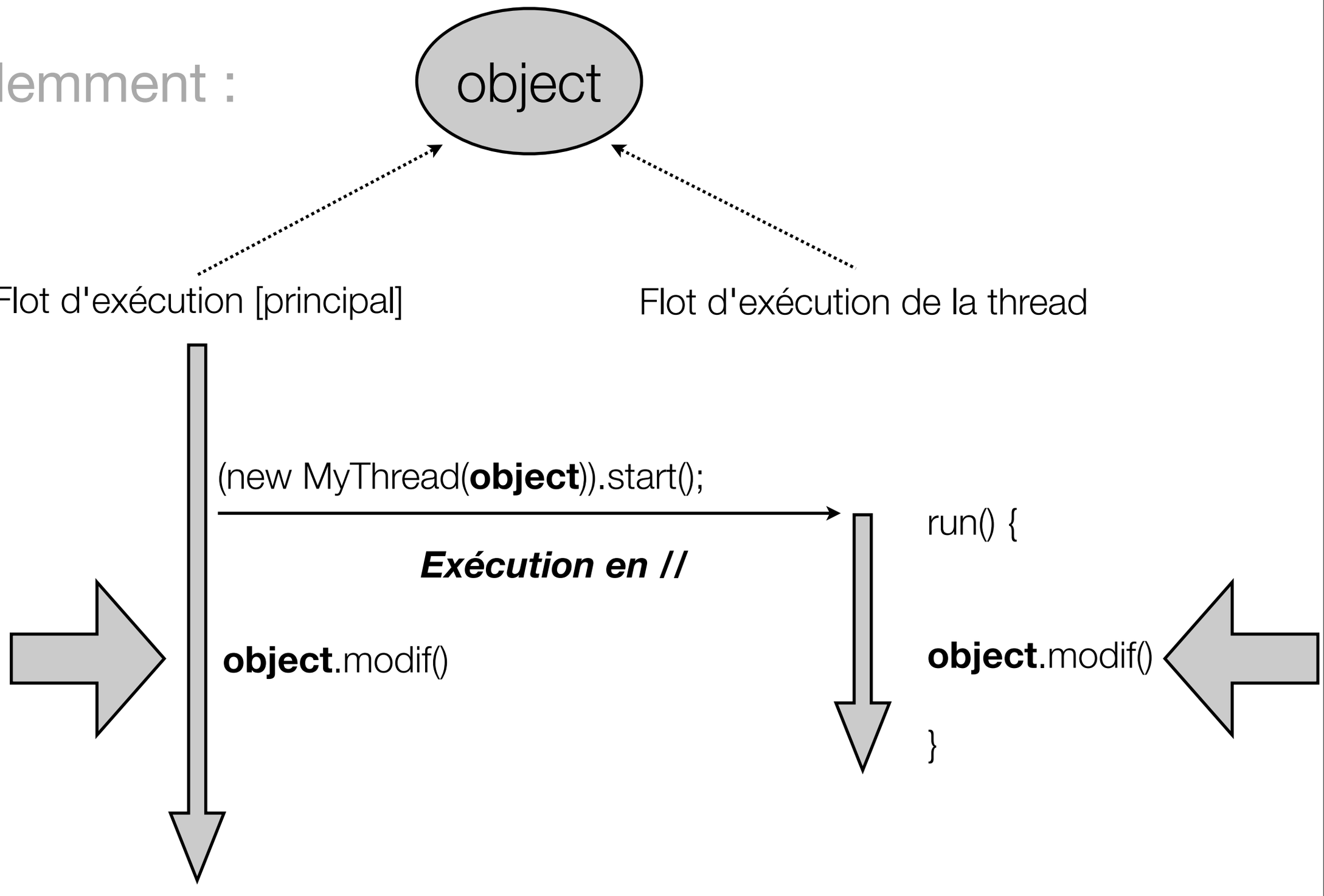
Introduction

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

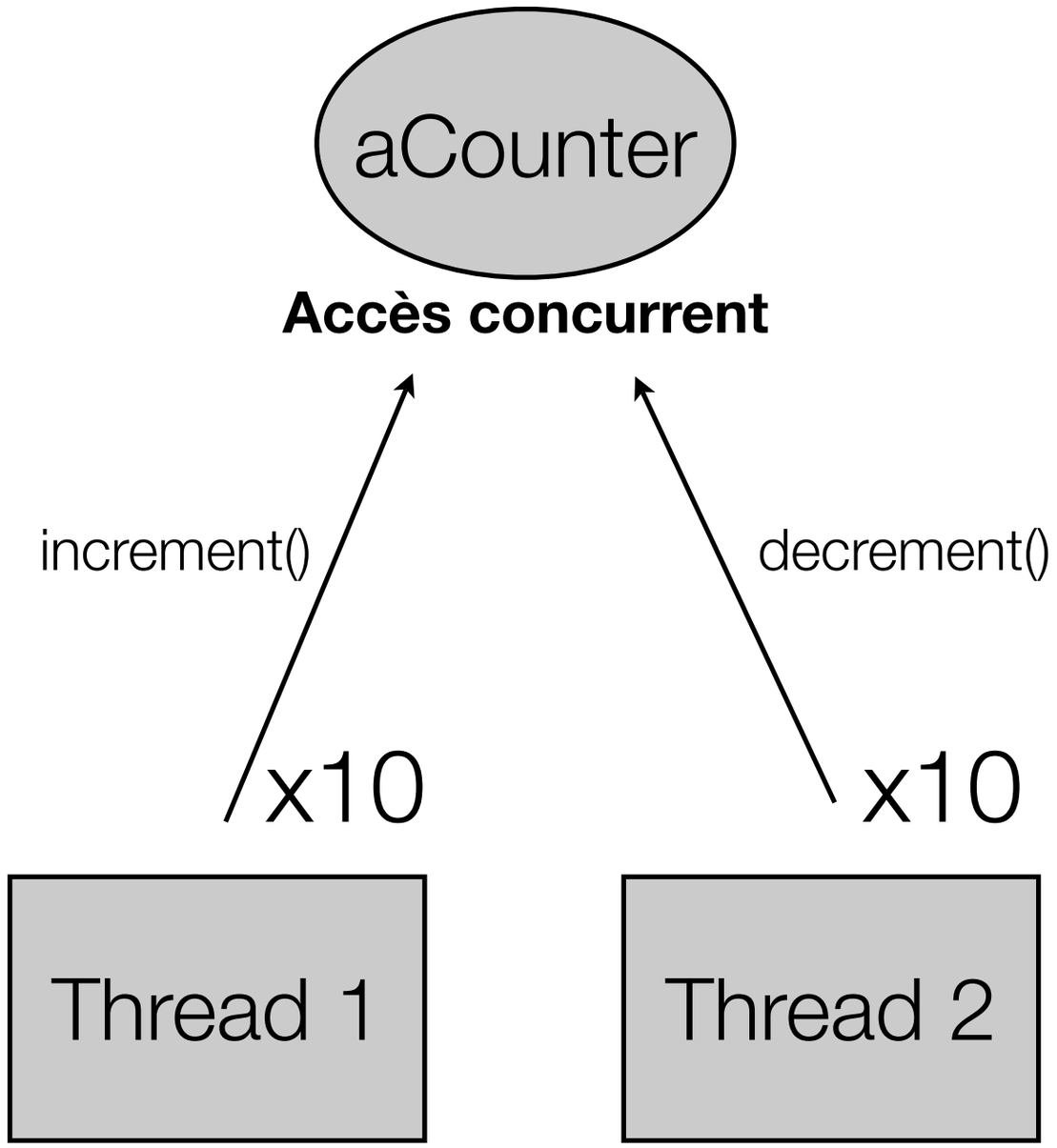
- Supposons maintenant qu'une thread A invoque la méthode *increment()* en même temps qu'une autre thread B invoque *decrement()* sur une même instance de *Counter.*

- 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

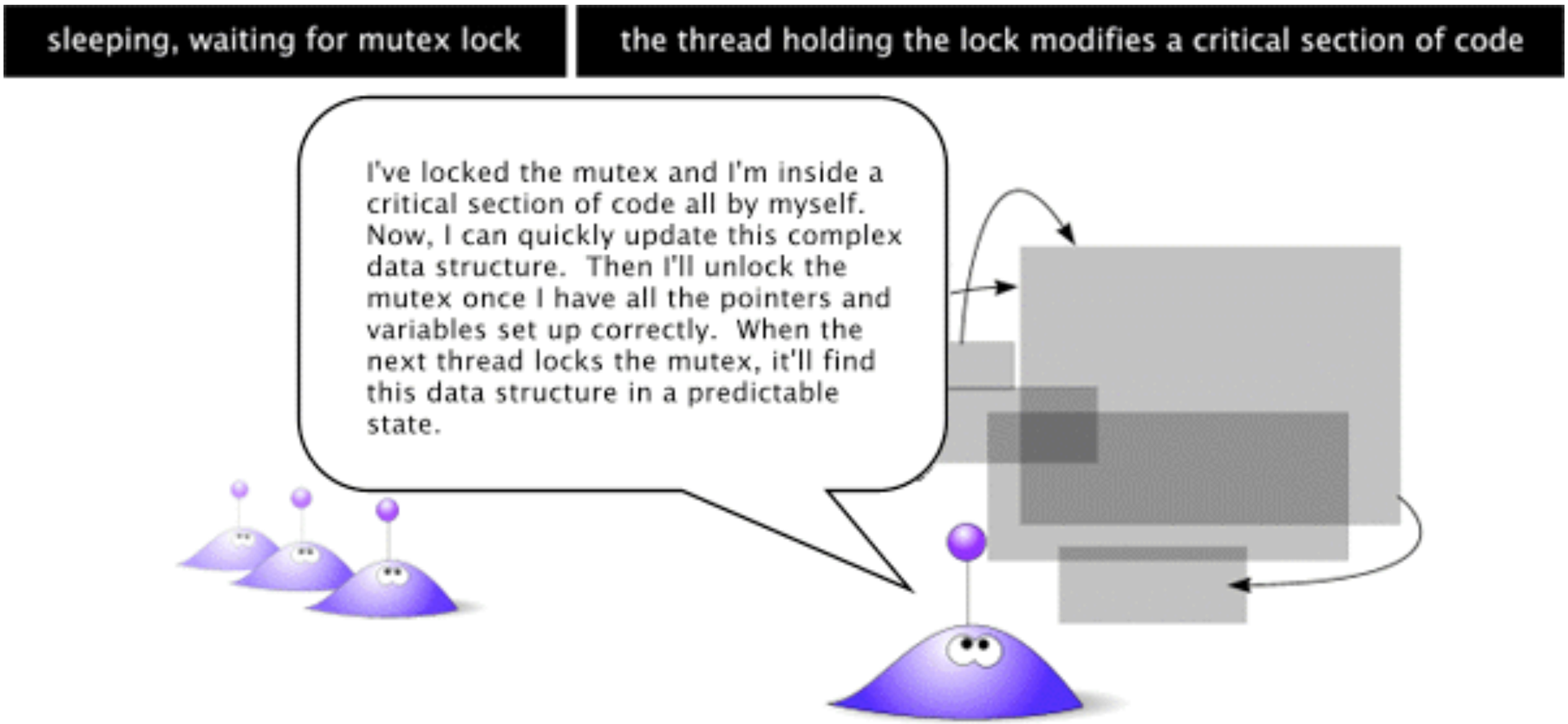


image © IBM, 2000

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
Problématiques associées

- On présente ici les 2 grands types de problématiques **liées à l'utilisation de techniques de synchronisation**. Il s'agit d'effet indésirables qu'il n'est pas toujours facile d'éviter ou de prévoir, surtout en utilisant les techniques de synchronisation dites de bas niveau:

  - **Interblocages (deadlocks)**

  - **Famines (livelocks)**

Synchronisation
Interblocage

- L'interblocage (de l'anglais deadlock) est un risque important puisqu'il fait entrer le programme dans une arrêt irrémédiable. Ce phénomène est aussi appelé boucle létale.

- Il se produit lorsque une thread T1 demande l'accès à une ressource R2, déjà bloquée par une thread T2. Or cette dernière attend une ressource R1 qui est bloquée par la thread T1. **Les deux threads s'attendent mutuellement** et ne peuvent sortir de cette situation.

- Plusieurs méthodes existent pour les éviter :

    - Elimination lors de la conception par une analyse détaillée des algorithmes.

    - Système préventif qui détecte un risque de deadlock avant que celui-ci ne se produise durant l'exécution.

    - Système de récupération si un deadlock se produit, le système doit pouvoir repartir dans un état valide.

Synchronisation
Interblocage - Exemple

- Alphonse et Gaston sont deux amis très courtois.
- Une grande règle de courtoisie est que lorsque l'on fait une révérence à quelqu'un, on doit rester penché tant que l'autre personne n'a pas répondu à la révérence.
- Malheureusement cette règle **ne permet pas de tenir compte de la possibilité que nos deux amis se saluent exactement en même temps !**

Synchronisation

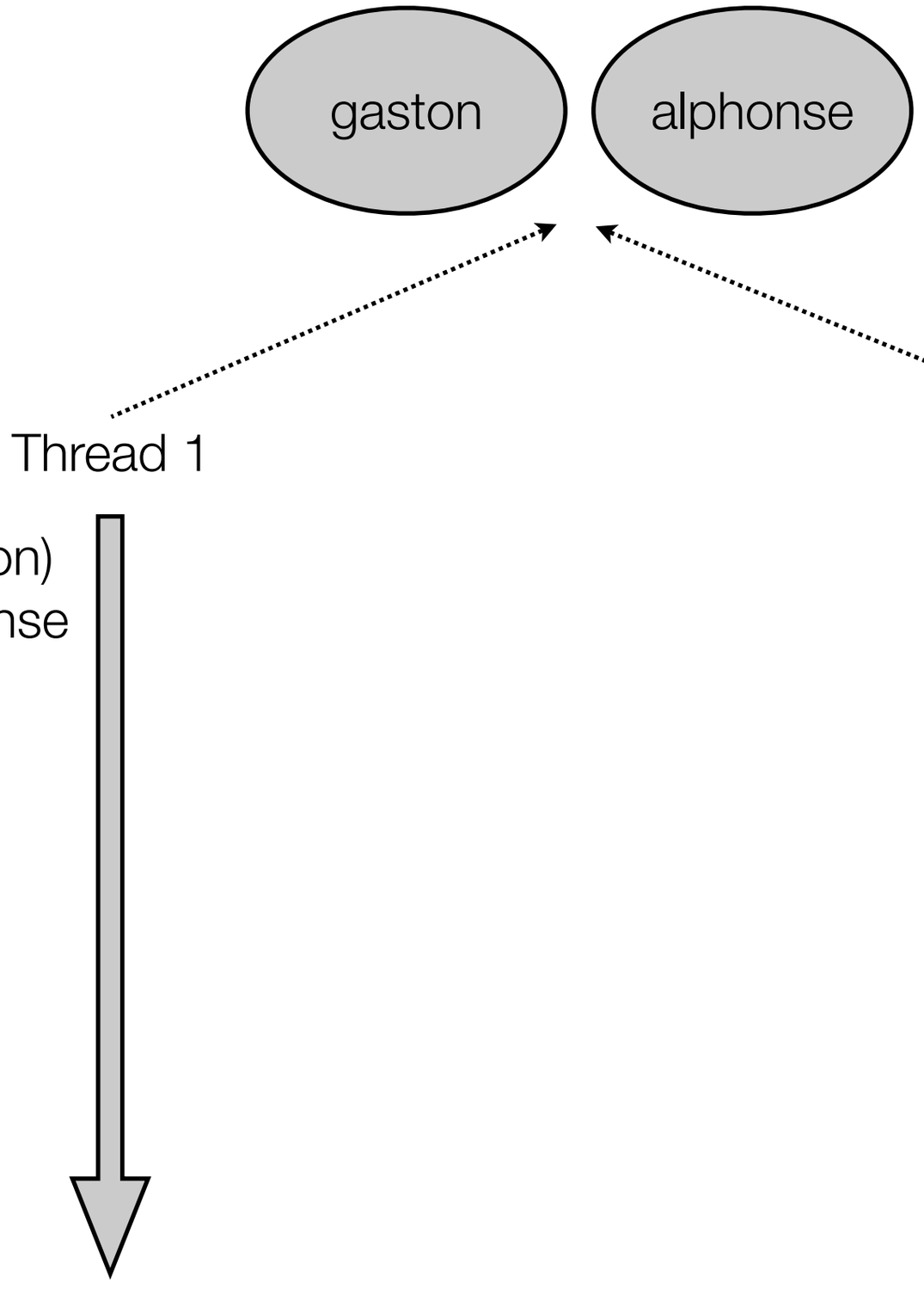
Interblocage - Exemple


```
public class Friend {
   private final String name;

   public Friend(String name) {
      this.name = name;
   }
   public String getName() {
       return this.name;
   }
   public synchronized void bow(Friend bower) {
      System.out.format(
            "%s: %s has bowed to me!%n",
            this.name, bower.getName());
       bower.bowBack(this);
   }
   public synchronized void bowBack(Friend bower) {
      System.out.format(
             "%s: %s has bowed back to me!%n",
             this.name,bower.getName());
   }
}
```

```
public static void main(String[] args) {

   final Friend alphonse =
      new Friend("Alphonse");

   final Friend gaston =
      new Friend("Gaston");


   //gaston salue alphonse
   new Thread(new Runnable() {
      public void run() {
         alphonse.bow(gaston);
      }}).start();

   //alphonse salue gaston
   new Thread(new Runnable() {
      public void run() {
         gaston.bow(alphonse);
      }}).start();
}
```

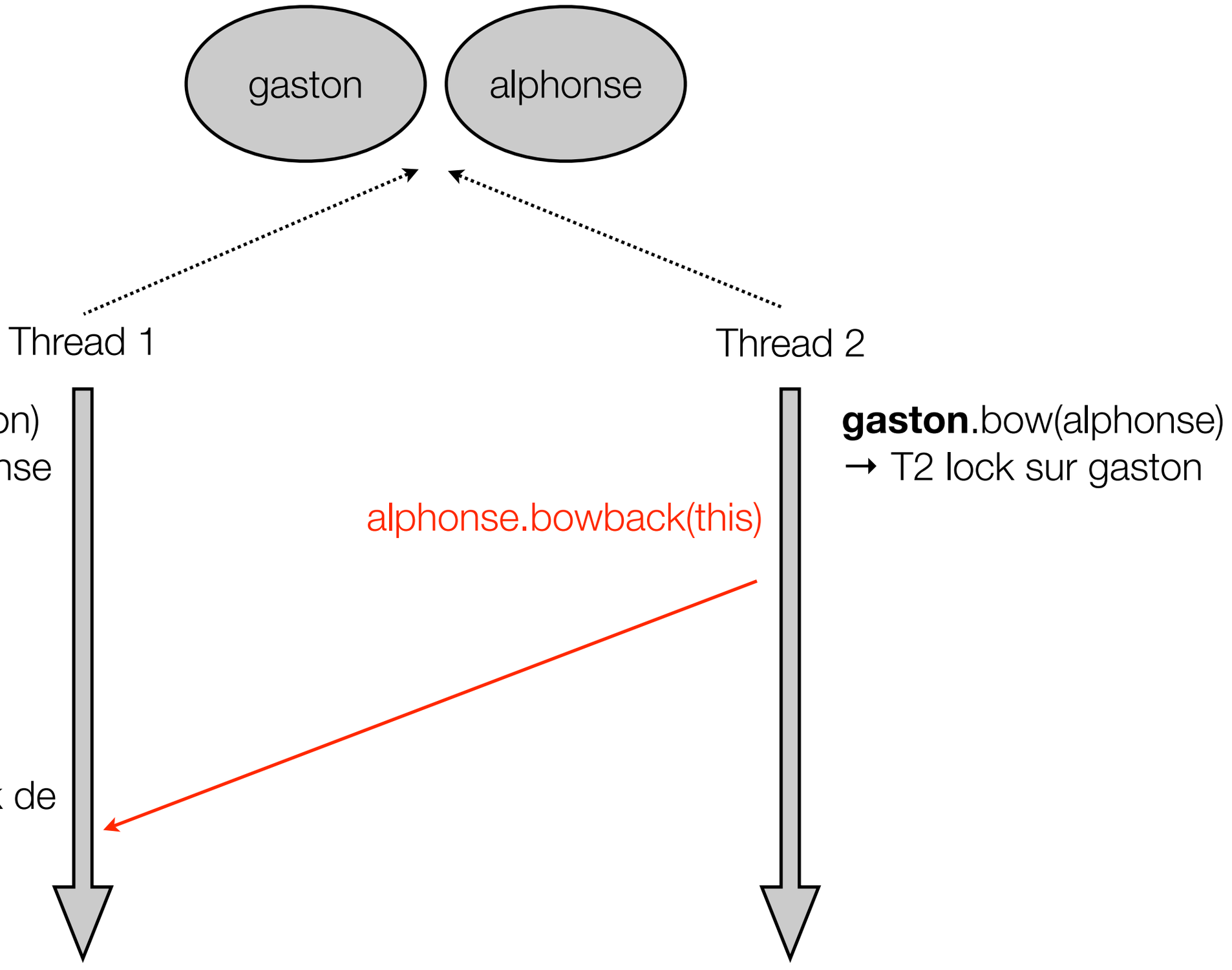
Synchronisation

Interblocage - Exemple



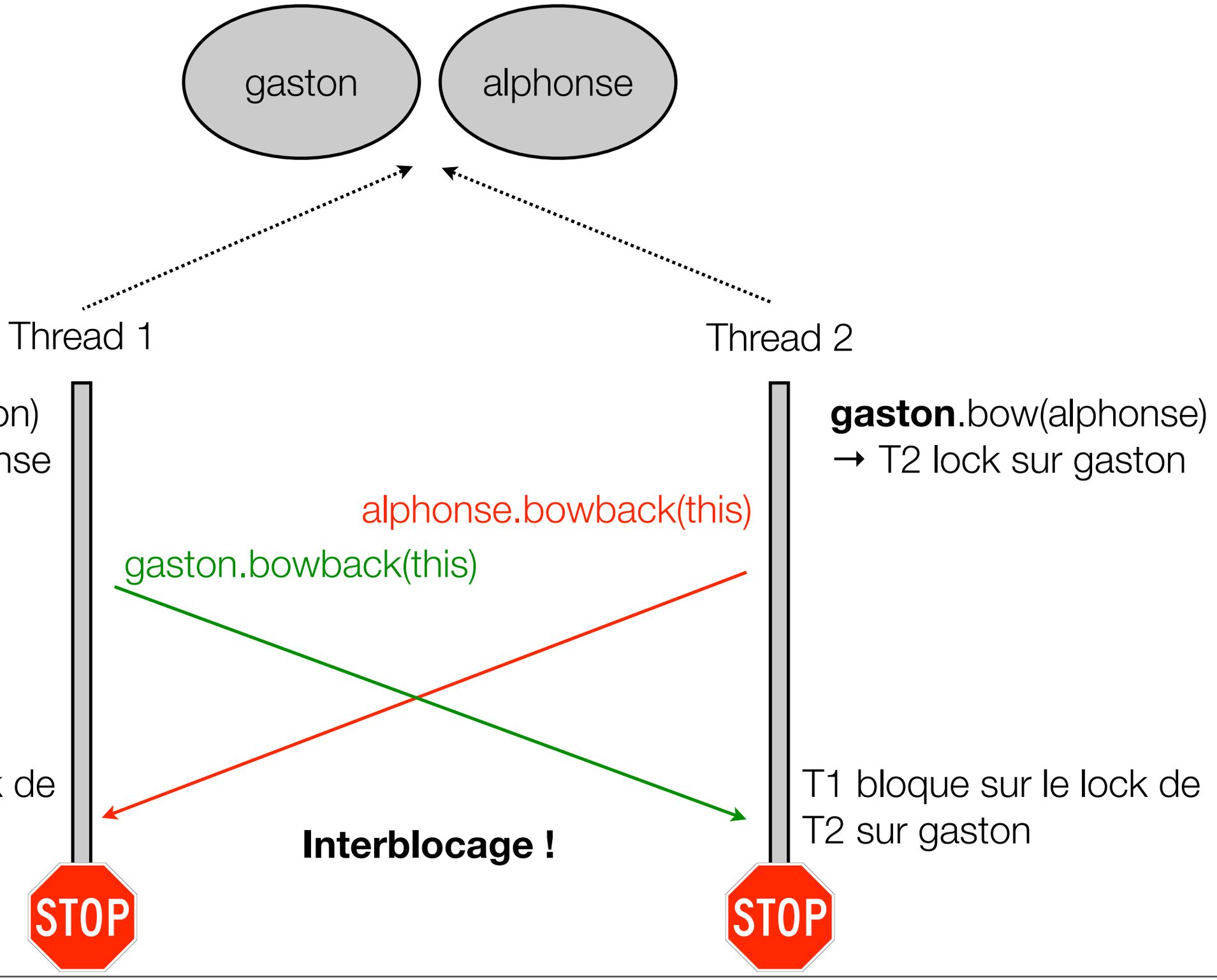
Synchronisation

Interblocage - Exemple



Synchronisation

Interblocage - Exemple



Synchronisation
Famine

- La famine est un problème que peut avoir un algorithme d'exclusion mutuelle.
- Elle se produit lorsqu'un algorithme n'est pas équitable, c'est-à-dire qu'il ne garantit pas à tous les threads souhaitant accéder à une section critique une probabilité non nulle d'y parvenir en un temps fini.
- Sans l'exécution de toutes ses threads, l’application n’est pas en mesure de mener à bien toutes ses tâches.
- Un exemple d'algorithme d'exclusion mutuelle pouvant conduire à une famine est l'algorithme de Dekker.
- Une des causes possibles de famine provient d'algorithmes qui ne garantissent pas que les threads qui souhaitent entrer dans une section critique y entrent dans leur ordre de demande (c'est-à-dire un comportement FIFO). **Un exemple typique de primitive de synchronisation qui ne garantit pas cet ordre est le synchronized du langage java utilisé avec la méthode notify().**

Synchronisation

Modèles de synchronisation

- **Modèle lecteurs-rédacteur**
- **Modèle producteur(s)-consommateur(s)**

- Ils utilisent la notion de moniteur.

Synchronisation

Modèle lecteurs-rédacteur

- Ce modèle traite de l'accès concurrent en lecture et en écriture à une ressource simple. Plusieurs thread peuvent lire en même temps la ressource, mais il ne peut y avoir qu'une et une seul thread en écriture.
- Deux types d’acteurs: lecteur, rédacteur.
- Soit 1 seul rédacteur et plusieurs lecteurs:
  - demande de lecture: bloquer si écriture en cours
  - demande d’écriture: bloquer si écriture ou lecture en cours
- Tous les acteurs bloqués sont réveillés en fin d’écriture ou en fin de lecture.
- Il nous faut un booléen rédacteur et un compteur lecteurs.

```
boolean redacteur = false;
int lecteurs = 0;
```

Synchronisation

Modèle lecteurs-rédacteur

- On implémente “demande de lecture: bloquer si écriture en cours” et “tous les acteurs sont réveillés en fin de lecture”.

```
void demandeLecture() throws InterruptedException {
    synchronized(this) {
        while(redacteur) { wait(); }
        lecteurs++;
    }
}

void finLecture() throws InterruptedException {
    synchronized(this) {
        lecteurs--;
        notifyAll();
    }
}
```

Synchronisation

Modèle lecteurs-rédacteur

- On implémente “demande d’écriture: bloquer si écriture ou lecture en cours” et “tous les acteurs sont réveillés en fin d’écriture”.

```
void demandeEcriture() throws InterruptedException {
   synchronized(this) {
      while(redacteur || lecteurs>0) { wait(); }
      redacteur = true;
   }
}

void finEcriture() throws InterruptedException {
   synchronized(this) {
      redacteur = false;
      notifyAll();
   }
}
```

Synchronisation

Modèle lecteurs-rédacteur

- Méthodes pour la lecture et l'écriture implémentent le modèle lecteurs-rédacteur:

```
void ecrire() throws InterruptedException {
      demandeEcriture();
      //ecrire
      finEcriture();
}

void lire() throws InterruptedException {
      demandeLecture();
      //lire
      finLecture();
}
```

Synchronisation

Modèle producteur(s)-consommateur(s)

- Lorsque des threads souhaitent communiquer entre elles, elles peuvent le faire par l'intermédiaire d'un tampon (une file). Mais, il faut définir le comportement à avoir lorsqu'une thread souhaite lire depuis le tampon mais qu'il est vide et lorsqu'une thread souhaite écrire dans le tampon mais qu'il est plein.

- Deux types d’acteurs: producteur et consommateur.

- Soit 1 ou plusieurs producteur et 1 ou plusieurs lecteurs:

  - demande de production: bloquer si tampon plein

  - demande de consommation: bloquer si tampon vide

- Tous les acteurs bloqués sont réveillés en fin de production et en fin de consommation.

- Initialisation du tampon:

```
public class TamponProducteurConsommateur {
    int max;               //taille max du tampon
    Object[] tampon;       //le tampon (FIFO)
    int taille = 0;        //nombre d'elts en cours dans le tampon

    public TamponProducteurConsommateur(int max) {
        this.max = max;
        tampon = new Object[max];
    }

    (...)
```

Synchronisation

Modèle producteur(s)-consommateur(s)

- On implémente “demande de production: bloquer si tampon plein” et “tous les acteurs bloqués sont réveillés en fin de production”.

```
synchronized void demanderProd() throws InterruptedException {
    while (taille == max) { wait(); }
}

synchronized void finProd() {
    notifyAll();
}
```

Synchronisation

Modèle producteur(s)-consommateur(s)

- On implémente “demande de consommation: bloquer si tampon vide” et “tous les acteurs bloqués sont réveillés en fin de consommation”.

```
synchronized void demanderCons() throws InterruptedException {
    while (taille == 0) { wait(); }
}

synchronized void finCons() {
    notifyAll();
}
```

Synchronisation

Modèle producteur(s)-consommateur(s)

- Méthodes pour la production et la consommation implémentent le modèle producteur(s)-consommateur(s):

```
void produire(Object data) throws InterruptedException {
    demanderProd();
    //produire → mettre les données dans la tampon
    //incrementer le nb d'elts en cours
    finProd();
}

Object consommer() throws InterruptedException {
    demanderCons();
    //consommer → prendre les données dans le tampon
    //decrementer le nb d'elts en cours
    finCons();
}
```

Synchronisation
Compléments

- **Entrées sorties non-bloquantes (asynchrones):**
  - **Problème**: pouvoir lire des données sur un flux sans rester bloquer en cas d'absence de données.
  - **Solution**:
    - Une thread qui lit en permanence et stocke les données dans un buffer.
    - Une méthode read d'accès indirect aux données qui lit dans le buffer.

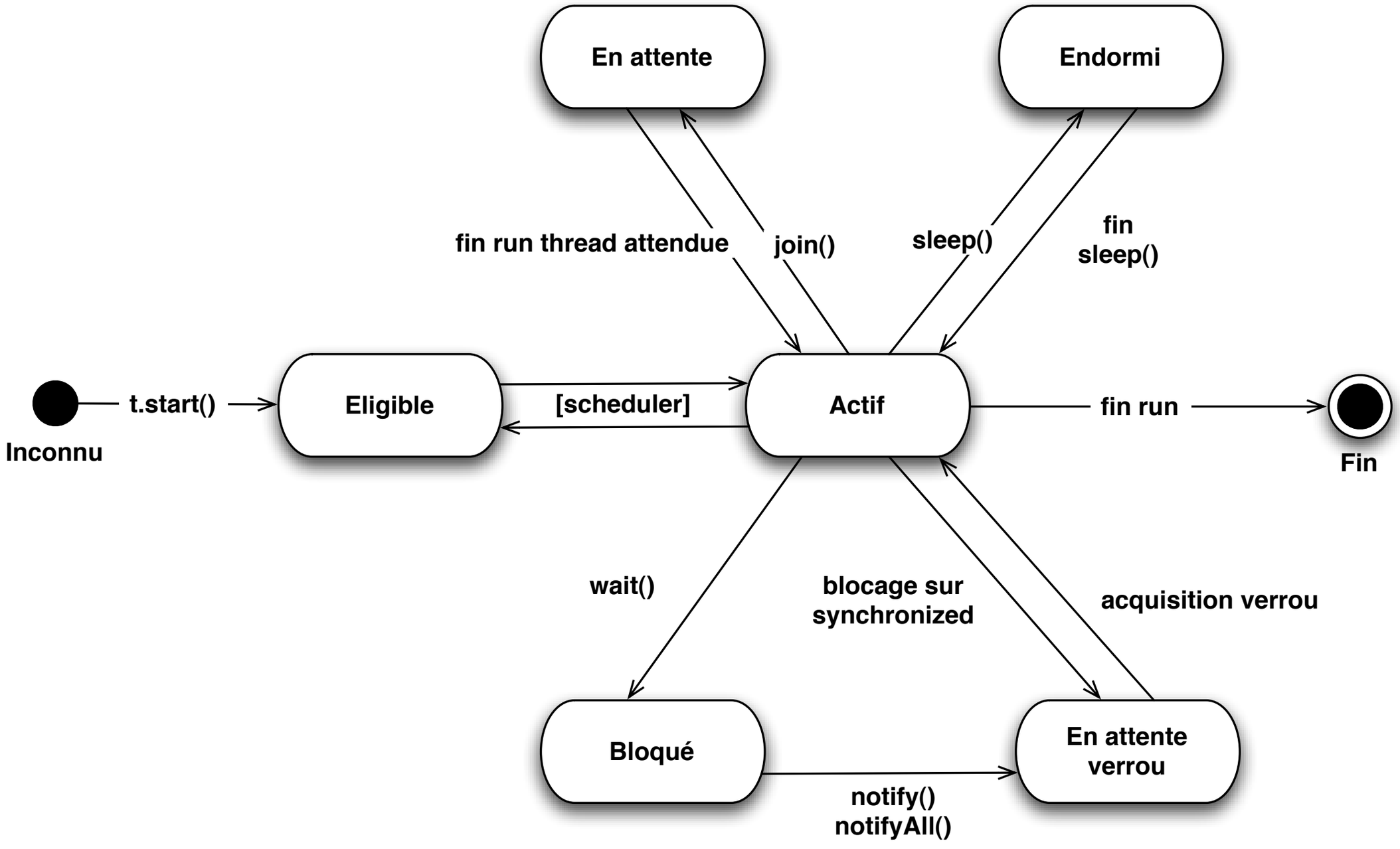
Synchronisation
Compléments

- **Entrées sorties non-bloquantes (asynchrones):**

```
public class AsyncInputStream extends FilterInputStream implements Runnable {
	int[] buffer;

	protected AsyncInputStream(InputStream in) {
		super(in);
		//initialiser le buffer
		new Thread(this);
	}

	public void run() {
		try {
			int b = in.read();
			while(b != -1) {
				//stocker b dans le buffer
				b = in.read(); //bloque si pas de donnŽes, -1 si fin des donnŽes
			}
		} catch (IOException e) {}
	}

	public int read() {
		return ...
		// retourne la 1ere donnŽe disponible dans le buffer
	}
}
```

Synchronisation
Compléments

- Cycle de vie d'une thread:



Synchronisation

Se documenter sur...

- Les pools de thread
- Les priorités de thread

Synchronisation

Techniques de haut niveau

- Nous avons vu précédemment que l'utilisation des primitives de bas niveau pour la synchronisation de threads n'étaient pas sans inconvénients pour le programmeur.

- Des abstractions de plus haut niveau ont donc été développées dans certains langages afin de minimiser certains inconvénients liés à l'usage des primitives de synchronisation

- A partir de la version 1.5 du langage Java, nous disposons notamment des :
  - Objets verrous
  - Collections concurrentes
  - Variables atomiques

Synchronisation
Objet verrous

- Package `java.util.concurrency.locks`
- Interface générale `Lock` : les objets de type `Lock` ont un fonctionnement très proche du verrou implicite des instructions `synchronized` classiques :
  - Seule une thread peut posséder un objet de type Lock à la fois.
  - Les objets verrous ou “lock objects” supportent le mécanisme wait()/ notify() vu précédemment (via leurs objets Condition associés).
- Mais le plus gros avantage des objets verrous comparativement au verrou implicite réside dans la possibilité de ne pas bloquer sur un essai de prise de verrou.
  - Utilisation de la classe `ReentrantLock` qui implémente l’interface `Lock` et dispose d’une implémentation de la méthode `tryLock`.
- Trouver une solution pour Gaston et Alphonse en TP !

Synchronisation
Objet verrous

- Contrairement aux verrous implicites Java, une précaution supplémentaire doit être prise lorsque l'on manipule des objets verrous :
  - En effet, **ils ne sont pas associés à l'instruction** `synchronized` et doivent donc être déverrouillés à la main après utilisation.
  - Afin de s'assurer à 100% que l'instruction de déverrouillage sera effectuée même si le code en section critique est interrompu on utilise la construction try { ... } finally { ... } :

```
class X {
   private final ReentrantLock lock = new ReentrantLock();

   public void m() {
     lock.lock();
     try {
        // ... method body
     } finally {
       lock.unlock()
     }
   }
 }
```

Synchronisation
Collections concurrentes

- Package `java.util.concurrent`
  - `BlockingQueue` : définie une structure de données FIFO (file) qui bloque la thread qui essaye d'y ajouter un élément si elle est pleine ou qui essaye d'y retirer un élément si elle est vide.
  - `ConcurrentMap` : est une sous-interface de `java.util.Map` qui définie des opérations atomiques utilitaires sur une structure de données de type "clé-valeur". Rendre ces opérations atomiques (indivisibles et donc ininterruptibles) permet d'éviter les problématiques de synchronisation. L'implémentation la plus courante de cette interface est la classe `ConcurrentHashMap`.
  - `ConcurrentNavigableMap` : est une sous-interface de `ConcurrentMap` qui supporte l'appariement approximatif. L'implémentation la plus courante de cette interface est la classe `ConcurrentSkipListMap`.

Synchronisation
Variables atomiques

- Package `java.util.concurrent.atomic`
- Les classes disponibles dans ce package définissent des types équivalents aux types primitifs Java (`int`, `long`, `boolean`, ...) qui permettent des opérations atomiques sur des variables déclarées :
  - `AtomicInteger`
  - `AtomicLong`
  - `AtomicBoolean`, ...
- La manipulation atomique des valeurs passe par l'utilisation de méthodes `get` et `set` définies par ces classes.
- Leur utilisation est sensiblement différente de celle des variables déclarées '`volatil`'. Cf. http://www.javaperformancetuning.com/news/qotm030.shtml
- On peut utiliser ce type de classes pour résoudre simplement le problème de compteur que nous avons évoqué précédemment: cela représente une solution plus légère et localisée (seul l'accès à la variable est atomique) que l'emploi de méthodes `synchronized`.

Synchronisation
Variables atomiques


```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet(); //atomique, contrairement à c++
    }

    public void decrement() {
        c.decrementAndGet(); //atomique, contrairement à c--
    }

    public int value() {
        return c.get();
    }

}
```

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