Java pour la programmation concurrente

Chap #3

Introduction

- La programmation concurrente est un style de programmation permettant l'exécution **simultanée** de plusieurs **tâches interagissant entre elles**.

- Ces tâches sont communément implémentées sous forme de **processus** ou de **threads** au sein d'une même application.

- La concurrence est indispensable lorsque l'on souhaite écrire des applications interagissant avec le monde réel (qui est concurrent) ou tirant parti de multiples unités centrales (dans les système multiprocesseurs ou répartis).

- Même dans une architecture non répartie, c'est un bon moyen d'utiliser au mieux les capacités (processeur) d'une machine par la minimisation des temps morts.

Applications réparties

- La concurrence est souvent liée à la “répartition” d’une application.

- **Une application dont l'exécution est répartie sur plusieurs hôtes est intrinsèquement concurrente:** plusieurs tâches (processus, threads) de l’application sont alors exécutées simultanément sur plusieurs hôtes et doivent **coopérer**.

- Pourtant, ce n'est pas toujours explicitement le cas:

  L'utilisation de protocoles de communication entre parties distribuées d'une application, comme les RPC (appels de procédure à distance) ou RMI (invocation de méthode à distance), permet au développeur de s’abstraire de certaines problématiques de concurrence (toutes ?).

Paradigmes

- La communication entre les différentes tâches concurrentes d'une application (répartie ou non) peut s'effectuer en suivant l'un des 2 paradigmes suivants:

  - **Communication par partage de mémoire: (Java, C#)**
    Les composants concurrents communiquent par altération du contenu d'une zone mémoire partagée. Ce type de programmation concurrente nécessite souvent l'utilisation (explicite ou non) d'une forme particulière de verrou ("lock") pour coordonner les tâches.

  - **Communication par passage de message: (Erlang)**
    Les composants concurrents communiquent en échangeant des messages de manière asynchrone ("send and pray") ou synchrone (l'envoyeur bloque tant qu'il n'a pas reçut de réponse). Ce type de communication est considéré plus robuste que le partage de mémoire, mais aussi beaucoup plus lent.

Programmation Java

- Possibilité d'exécuter des **traitements concurrents** au sein d'une JVM.
- **La concurrence en Java est basée quasi-exclusivement sur les threads** (cf. chapitre suivant).
- La gestion de la concurrence permet par exemple :
  - la programmation événementielle (ex. IHM)
  - **des entrées/sorties non bloquantes**
  - la mise en place de "timers", déclenchements périodiques de certains traitements (méthodes)
  - **à un serveur de répondre simultanément à plusieurs clients**

Programmation concurrente

Threads

Chap #3.1

Concurrence
Unités d'exécution

- **Processus**: traitements concurrents situés dans des espaces mémoire séparés et gérés par le système d'exploitation.
  - Une application peut être constituée d'un ou plusieurs processus.
  - Une JVM = un processus.
- **Thread**: traitements concurrents n'existants qu'au sein de processus, on les appelle des "processus légers".
  - **Les threads partagent un même espace d'adressage: elles peuvent partager les mêmes variables ⇒ Danger !**
  - Chaque thread dispose par contre de sa propre pile d'exécution.
  - Dans le cas de Java, les threads sont gérées "au dessus" de l'OS par la JVM.
  - L'exécution multithreadée est une fonctionnalité essentielle de la plate-forme Java. **Chaque application possède une thread implicite** (la *main thread*) qui à la possibilité d'engendrer d'autres threads.

Threads vs. Processus

- **Threads:**
  - (+) Il est souvent plus efficace de coder une application nécessitant des traitements parallèles avec des threads (plus simple à coder, plus léger à instancier).
  - (-) Le dysfonctionnement d'une thread peut perturber les autres threads.
- **Processus:**
  - (+) Le dysfonctionnement d'un processus n'a pas (normalement) d'incidence sur les autres processus.
  - (-) Beaucoup moins souple d'utilisation (espaces d'adressages séparés → communication difficile entre les processus).
- → **Approches mixtes**: plusieurs processus ayant chacun plusieurs threads.

Threads Java

... et programmation objet

> **Rappel:** En Programmation orientée objet, on appelle instance d'une classe, un objet avec un comportement correspondant à cette classe et un état initial.

- Les traitements effectués par une thread ne sont pas liées à des instances particulières.

- Les threads exécutent des traitements sur (éventuellement) plusieurs instances.

- **Les threads sont vues “extérieurement” comme des instances de la classe** `java.lang.Thread` **par le programmeur.**

Threads Java

Définir une thread

- Il suffit d'écrire une classe héritant de `java.lang.Thread`

```
public class myThread extends java.lang.Thread {
  ...
}
```

- ou implémentant l'interface `java.lang.Runnable`

```
public class myThread implements java.lang.Runnable {
  ...
}
```

- Dans les 2 cas, les instructions propres à la thread doivent être définies dans la méthode `run()`. La méthode `run()` d'une thread est similaire à la méthode `main()` d'un programme Java classique.

```
public class myThread (...) {
 public void run() {
 // code à insérer ici
 }
}
```

Threads Java

Lancer une thread 1/3

- Dans le cas où la classe à été définie comme héritant de `java.lang.Thread`, il faut faire appel à la méthode d'instance `start()`.

```
public class MyThread extends Thread {
public void run() {
   do{} while(true); //boucle infinie
}

public static void main(String [] args) {
   MyThread thread = new MyThread();
   thread.start();
  System.out.println("Thread Launched"); //est-ce que ça s'affiche ?
 }
}
```

Threads Java

Lancer une thread 2/3

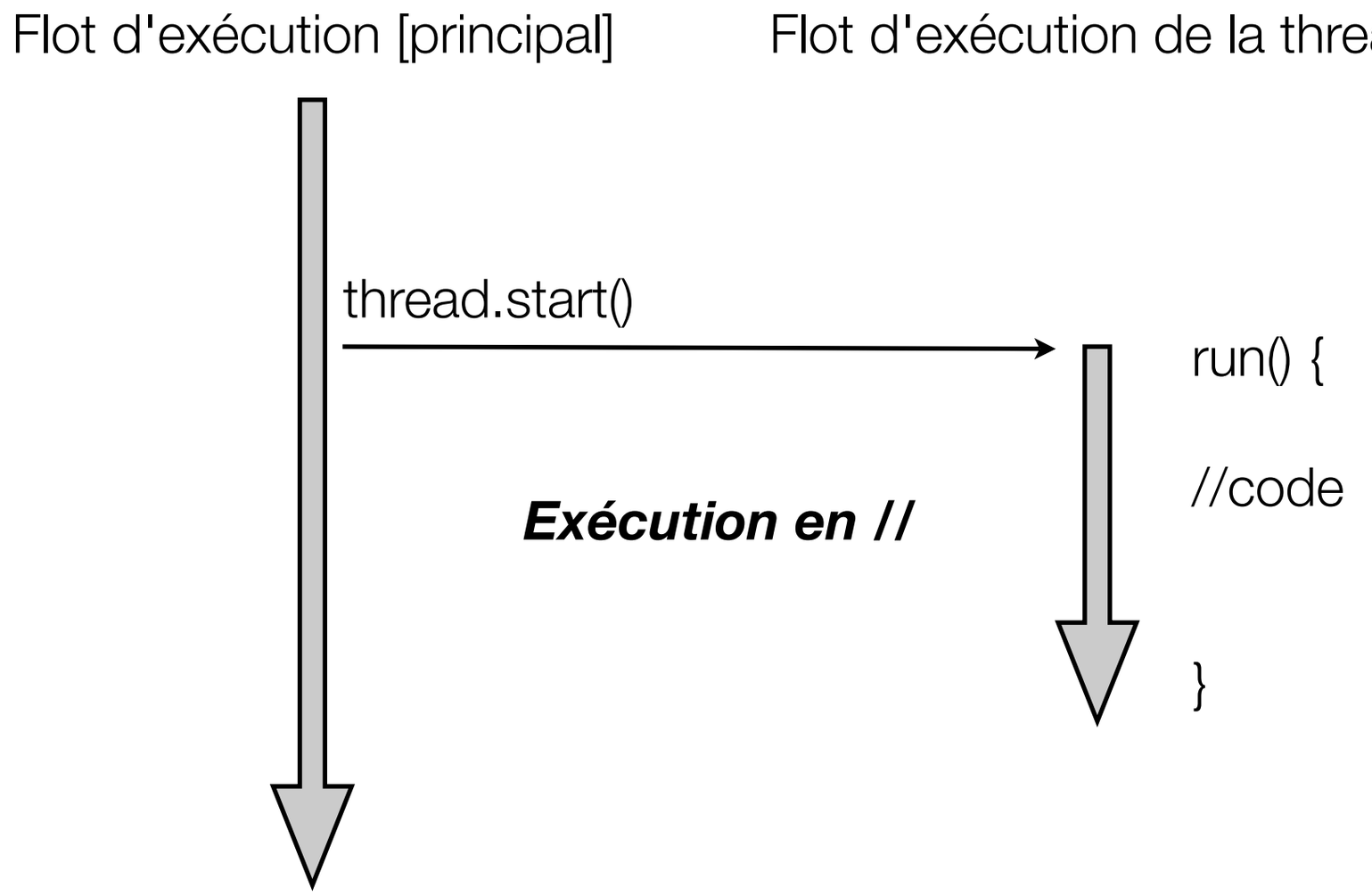
- Dans le cas ou la classe à été définie comme implémentant `java.lang.Runnable`, il faut utiliser le constructeur `public Thread(Runnable)` de la classe `Thread` avant de pouvoir appeler `start()`.

```
public class MyThread implements Runnable {
public void run() { (...) }

public static void main(String [] args) {
   MyThread runnable = new MyThread();
   Thread thread = new Thread(runnable);
   thread.start();
 }
}
```

Threads Java

Lancer une thread 3/3

- Lorsque `start()` est appelée, un nouveau flot d'exécution se “détache” et le contenu de la méthode `run()` est exécuté en // du flot d'exécution principal.



Threads Java

Remarques

- Il est possible de créer plusieurs threads d'une même classe.

- **Mais appel de la méthode `start()` une seule fois pour chaque instance de thread !** Pour créer une nouvelle thread il faut re-instancier la classe de la thread.

```
MyThread thread1 = new MyThread();
thread1.start();
~~thread1.start();~~
MyThread thread2 = new MyThread();
thread2.start();
```

- Une thread meurt lorsque sa méthode `run()` se termine.

- On n'appelle jamais directement la méthode `run()`, il faut passer par la méthode `start()` après l'instanciation.

Threads Java

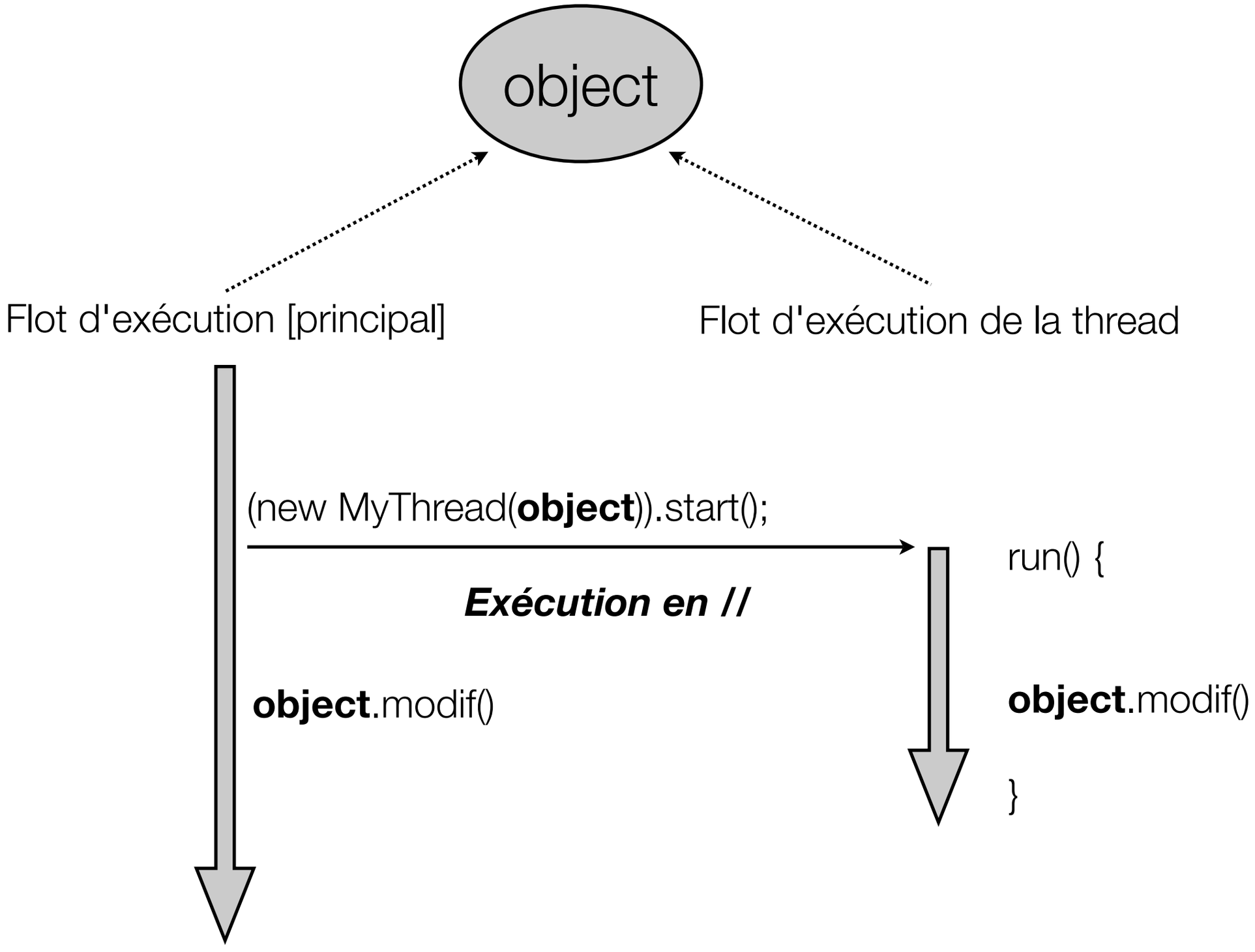
Passage de paramètres

- Il n'est pas possible de passer des paramètres à une thread via la méthode `start()`
- Solution:
  - Définir des variables d'instance
  - Les initialiser via les paramètres du constructeur lors de l'instanciation de la thread

```
public class MyThread extends Thread {
	String message;

	public MyThread(String message) {
		this.message = message;
	}

	public void run() {
		do {System.out.println(message);}while(true);
	}

	public static void main(String [] args) {
		(new MyThread("Hello World!")).start();
		System.out.println("Thread Launched");
	}
}
```

Threads Java

Passage de paramètres



Threads Java

Suspendre l'exécution

- La méthode `static Thread.sleep(m)` permet de suspendre l'exécution d'une thread pour un temps `m` déterminé (en milli-secondes).

- Le respect exact du temps de suspension **n'est pas garanti** dans la mesure où son implémentation dépend du système d'exploitation sous-jacent.

- C'est un moyen simple pour libérer du temps processeur pour d'autres threads d'une même application.

- L'exemple suivant affiche un message toutes les 4 secondes :

```
public class SleepMessages {
    public static void main(String args[]) throws InterruptedException {
        String importantInfo[] = { "Mares eat oats","Does eat oats",
            "Little lambs eat ivy","A kid will eat ivy too" };

        for (int i = 0; i < importantInfo.length; i++) {
            //Pause for 4 seconds
            Thread.sleep(4000);
            //Print a message
            System.out.println(importantInfo[i]);
        }
    }
}
```

Threads Java
Interruptions

- Une interruption est une indication pour une thread quelle devrait arrêter son exécution et faire quelque-chose d'autre.

- C'est à la charge du programmeur de décider comment répondre à une interruption !

  Le plus courant est de terminer purement et simplement l'exécution de la thread qui reçoit cette indication.

- Une thread envoie une interruption à une autre thread via la méthode d'instance `interrupt()` de la classe `Thread`.

- **Pour que le mécanisme d'interruption fonctionne correctement, la thread réceptrice du message doit gérer sa propre interruption.**

  - **Comment ?**

Threads Java
Interruptions

- **Cas #1 :**
  Si la thread invoque fréquemment des méthodes qui lèvent des exceptions `InterruptedException` alors elle peut traiter l'interruption dans le *bloc catch* afférent.

- Beaucoup de méthodes comme `sleep()` qui lèvent des `InterruptedException`, sont conçues pour s'arrêter et retourner immédiatement lorsque la thread exécutrice reçoit une interruption.

```
public class MyThread implements Runnable {
    public void run() {
    ...
        for (int i = 0; i < importantInfo.length; i++) {
        //Pause for 4 seconds
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            //We've been interrupted: no more messages.
            return;
        }
        //Print a message
        System.out.println(importantInfo[i]);
    }
}
```

Threads Java
Interruptions

- Mais que faire si une thread reste longtemps sans invoquer de méthode qui lève des `InterruptedException` ?

- **Cas #2 :**
  La thread invoque périodiquement la méthode statique `Thread.Interrupted()`, qui retourne `True` si la thread à reçu un message d'interruption.

On "retourne" dans la méthode run, ce qui à pour conséquence de terminer l'exécution de la thread :

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        //We've been interrupted:
        //no more crunching.
        return;
    }
}
```

On fait suivre l'interruption, de manière à centraliser son traitement dans un bloc catch :

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        //We've been interrupted:
        //no more crunching.
        throw new InterruptedException();
    }
}
```

Threads Java

Interruptions

- Le mécanisme d'interruption est basé sur l'utilisation d'un "flag" interne connu sous le nom de "interrupt status".
- Invoquer `Thread.interrupt()` positionne ce flag.
- Lorsqu'une thread vérifie si elle à été interrompue en invoquant la méthode statique `Thread.interrupted()`, le flag est réinitialisé.
- La méthode d'instance `Thread.isInterrupted` est utilisée par une thread pour vérifier le statut d'interruption d'une autre thread. Dans ce cas le flag n'est pas réinitialisé.
- Par convention, une méthode qui s'interrompt en levant une `InterruptedException` (comme `sleep()`) procède simultanément à la réinitialisation du flag.

Threads Java

Joins

- La méthode `join()` permet à une thread en cours d'exécution d'attendre la complétion d'une tâche effectuée par une autre thread.
- Si t est une instance de la classe `Thread`, dont la thread associée est en cours d'exécution, l'instruction `t.join()` :
  - cause la suspension de la thread courante (celle qui appele `t.join()`)
  - jusqu'à la terminaison de la thread `t`.
- Il existe des variantes de la méthode `join()` qui permettent de spécifier un temps d'attente.
  - Mais comme `sleep()`, `join()` ne garantie pas le respect précis du temps demandé.
- `join()` répond à l'interruption de la thread courante en levant une `InterruptedException`.

Inconsistances mémoire

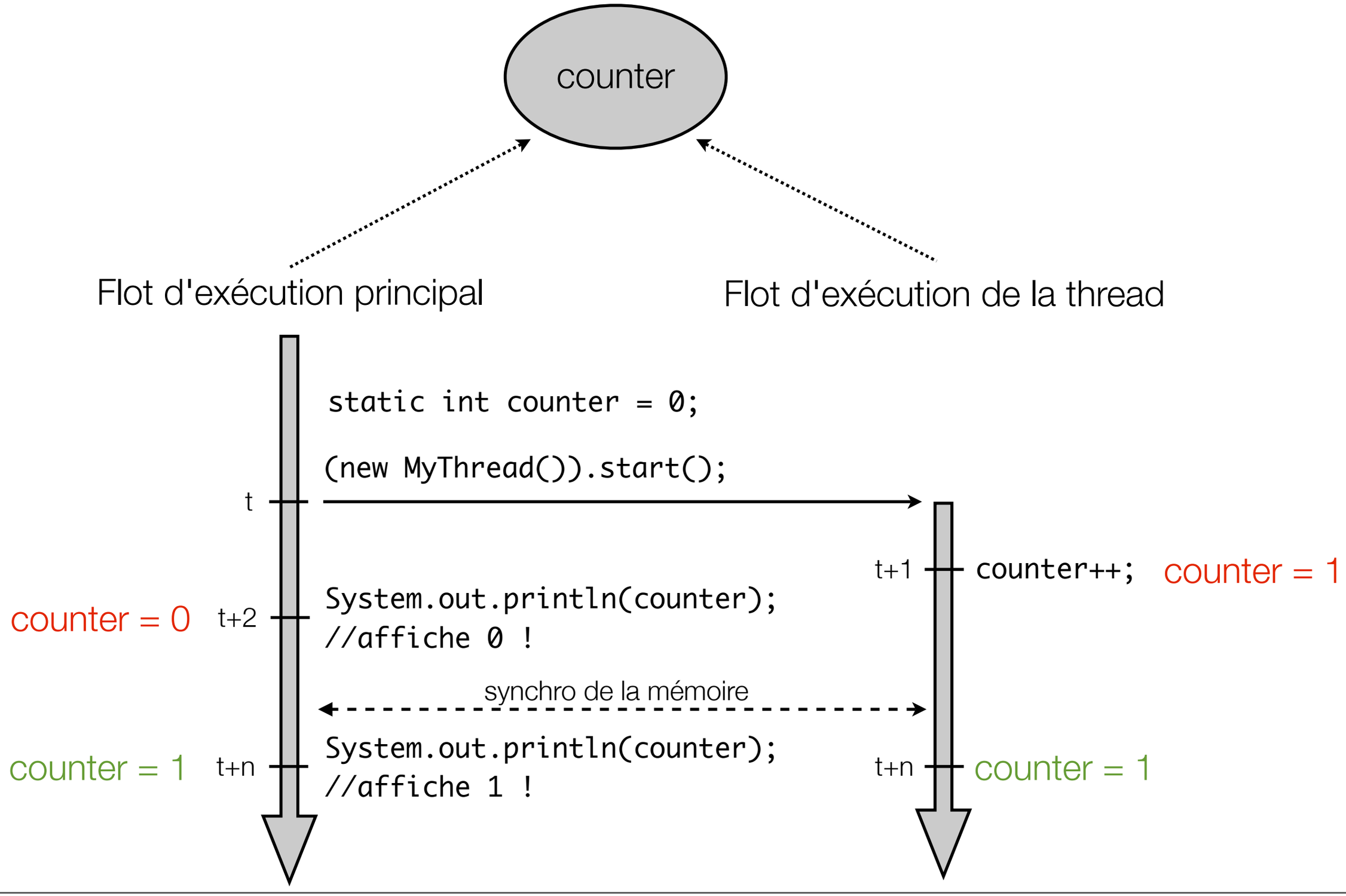
- Les inconsistances mémoire apparaissent lorsque différentes threads ont des vues différentes de ce qui devrait être une seule et même donnée.
- Exemple d'inconsistance mémoire potentielle sur la variable `counter` :

```
public class Exemple {
    static int counter = 0;

    private static class MyThread extends Thread {
        public void run() {
          counter++;
        }
    }

    public static void main(String args[]) throws InterruptedException {
        (new MyThread()).start();
        Thread.sleep(2000);
        System.out.println(counter);
    }
}
```

Inconsistances mémoire

- Si la “main thread” effectue un affichage de `counter`, **absolument rien ne garantie que la valeur affichée sera bien “1”**.
- En effet, en Java chaque thread peut posséder temporairement une copie “locale” des variables partagées. Ainsi, les données vue par une thread donnée peuvent être différentes de celles vues par d’autres threads.
- Dans l’exemple précédent, les modification apportés par la seconde thread au contenu de la variable `counter` pourraient très bien ne pas encore avoir été répercutées sur la mémoire principale et donc invisiblent à la “main thread”.
- **Dans la pratique, ce genre de situation arrive rarement. Il ne faut malgré tout faire aucune supposition à ce sujet dans votre code.**

Inconsistances mémoire

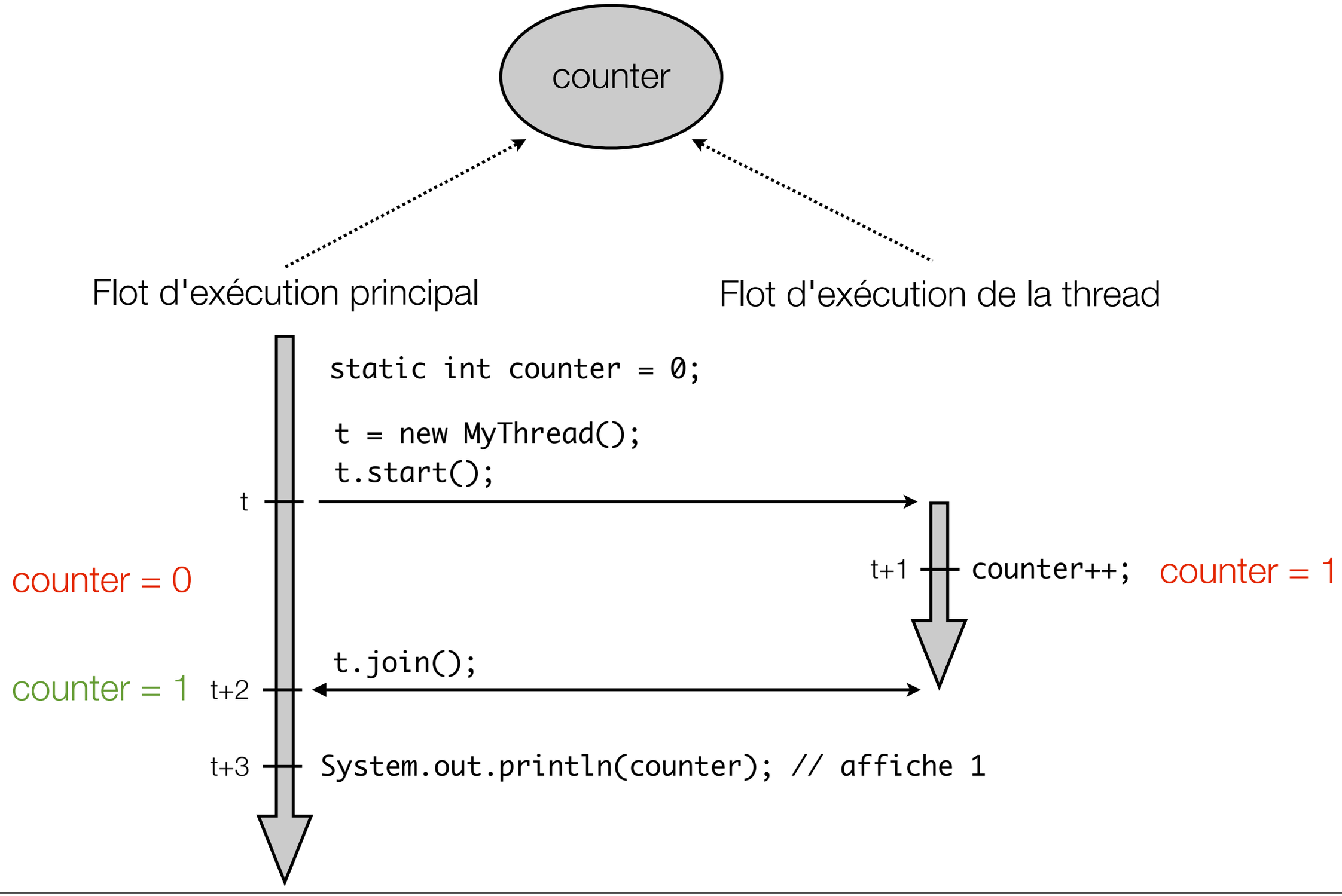


Inconsistances mémoire

- Pour s'assurer de la visibilité des modifications dans toutes les threads concurrentes il faut créer des liens de dépendance ou **liens "happens-before"** entre les opérations de lecture et les opération d'écriture de données situées dans les différentes threads.
- Différentes techniques existent pour créer ce type de liens :
  - Toute instruction exécutée avant un `Thread.start` est en relation "happens-before" avec la thread fraîchement lancée : ce qui équivaut à dire que les modifications de la mémoire apportées par ces instructions sont visibles pour la thread.
  - Les instruction de la thread `t` sont en relation "happens-before" avec les instructions suivant un `t.join()` : les effets de `t` sont alors visibles.
  - Une autre technique sera vue dans le cours suivant... (cf. Chap #3.2).
- Il est aussi possible d'utiliser des variables déclarées `volatile` pour éviter que des threads possède des vues mémoire différentes pour une variable donnée (mais coûteux à l'exécution).

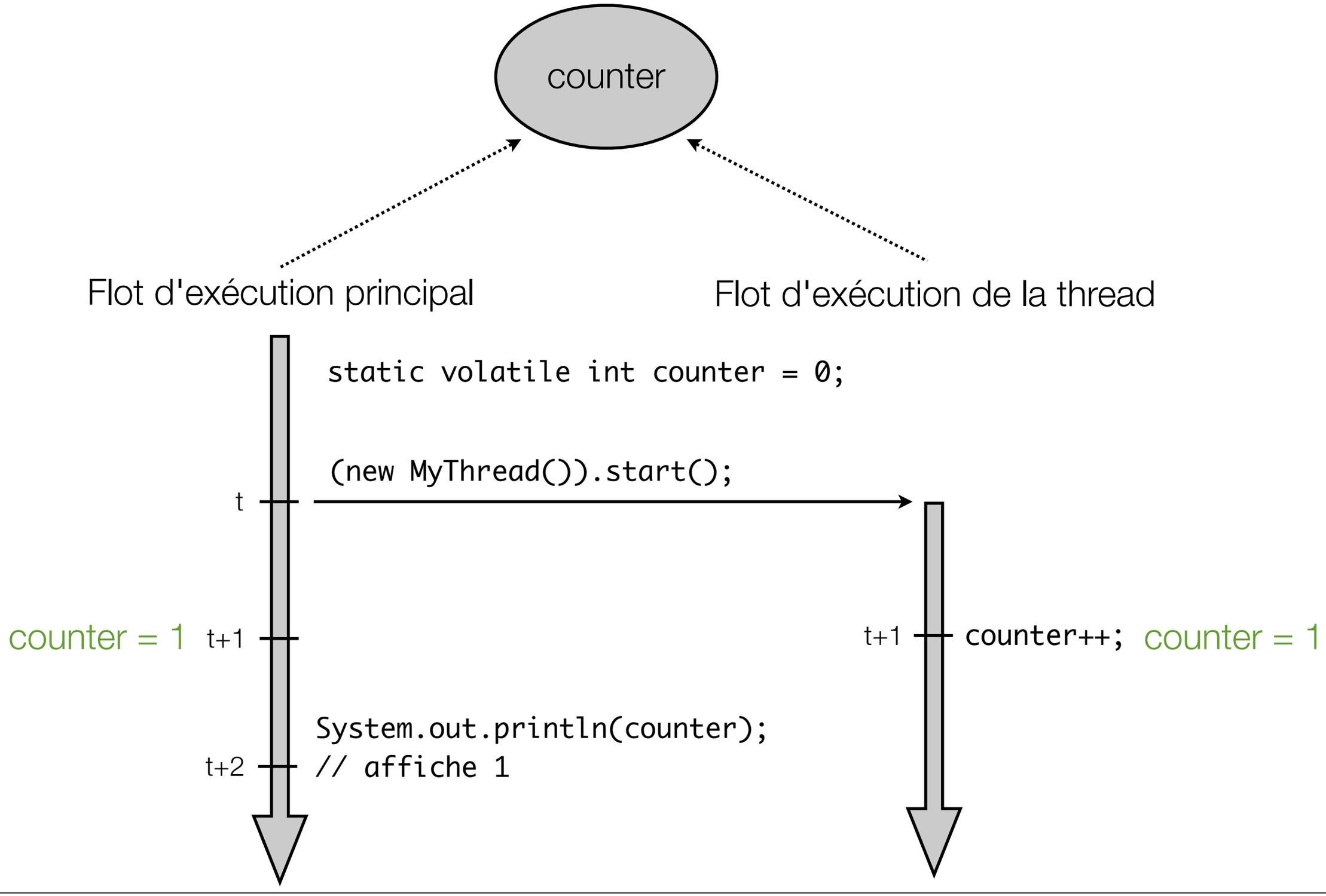
Inconsistances mémoire

Utilisation de Thread.join



Inconsistances mémoire

Utilisation de variables volatiles



Fin du cours #3