

Concurrence		Concurrence		Concurrence		Parallélisme	
Quoi?	Où et quand?	Quoi?	Où et quand?	Quoi?	Où et quand?	Quoi?	Où et quand?
<p>Compléments en P00 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Concurrence Introduction Concurrence et parallélisme Les abstractions Théorie en Java Brûler le JMM API de haut niveau Interfaces graphiques Gestion des erreurs et exceptions</p>	<p>Naturelle et nécessaire dans des situations variées en programmation<sup>1</sup>:</p> <ul style="list-style-type: none"> <li>• <b>Serveurs web</b>: un même serveur doit pouvoir servir de nombreux clients indépendamment les uns des autres sans les faire attendre.</li> <li>• <b>Interfaces homme-machine</b>: le programme doit pouvoir, tout en prenant en compte, sans délitai, les actions de l'utilisateur, continuer à exécuter d'éventuelles tâches de fond, jouer des animations, etc.</li> <li>• De manière générale, c'est utile dans tout programme qui doit réagir immédiatement à des événements de causes et origines variées et indépendantes.</li> <li>• Possible<sup>2</sup> pour de nombreux algorithmes décomposables en étapes indépendantes.</li> </ul> <p>Utilise de programmer ces algorithmes de façon concurrente car on peut profiter du <u>parallelisme pour les accélérer</u> (cf. page suivante).</p>	<p>Compléments en P00 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Concurrence Introduction Concurrence et parallélisme Les abstractions Théorie en Java Brûler le JMM API de haut niveau Interfaces graphiques Gestion des erreurs et exceptions</p>	<p>1. Et pas seulement en programmation, mais c'est le sujet qui nous intéresse!</p> <p>2. Et discutablement naturelle aussi.</p>	<p>Ainsi, l'enjeu de la programmation concurrente est double :</p> <ul style="list-style-type: none"> <li>• Nécessité : pouvoir programmer des fonctionnalités intrinsèquement concurrentes (serveur web, IG, etc.).</li> <li>• Opportunité : tirer partie de toute la puissance de calcul du matériel contemporain.</li> </ul> <p>En effet : des travaux indépendants (concurrents) peuvent naturellement être confiés à des unités d'exécution distinctes (parallèles).</p> <p>Malheureusement, la programmation concurrente est un <u>art difficile</u>...</p>	<p>Compléments en P00 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Concurrence Introduction Concurrence et parallélisme Les abstractions Théorie en Java Brûler le JMM API de haut niveau Interfaces graphiques Gestion des erreurs et exceptions</p>	<p>Deux travaux<sup>1</sup> s'exécutent en parallèle, s'ils exécutent en même temps.</p> <ul style="list-style-type: none"> <li>• Simultanéité au niveau le plus bas : si 2 travaux s'exécutent en parallèle, à un instant t, s'exécutent en même temps une instruction de l'un et de l'autre.</li> <li>• → exécution sur 2 lieux physiques différents (e.g. 2 coeurs, 2 circuits, ...).</li> <li>• <b>Degré de parallélisme</b><sup>2</sup> = nombre de travaux simultanément exécutables.</li> </ul> <p>Pour des raisons économiques et technologiques, les microprocesseurs modernes (multi-cœur<sup>3</sup>) ont typiquement un degré de parallélisme <math>\geq 2</math>.</p> <p>C'est une opportunité qu'il faut savoir saisir!</p>	<p>1. Pour ne pas dire « processus », qui a un sens un peu trop précis en informatique. 2. D'une plateforme d'exécution. 3. Sur les CPU de moyenne et haut de gamme, le degré de parallélisme est généralement de 2 par cœur, grâce au SMT (<i>simultaneous multithreading</i>), appelé <i>hyperthreading</i> chez Intel</p>

Concurrence		Questions d'ordonnancement				
Compléments en 200	Aldric Degorre	Compléments en 200	Aldric Degorre			
<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrence</p> <p>Introduction</p> <p>Concurrency et partage des abstractions</p> <p>Les abstractions</p> <p>Théâtre en Java</p> <p>Dormir le JMM</p> <p>API de haut niveau</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>	<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrence</p> <p>Introduction</p> <p>Concurrency et partage des abstractions</p> <p>Les abstractions</p> <p>Théâtre en Java</p> <p>Dormir le JMM</p> <p>API de haut niveau</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>	<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrence</p> <p>Introduction</p> <p>Concurrency et partage des abstractions</p> <p>Les abstractions</p> <p>Théâtre en Java</p> <p>Dormir le JMM</p> <p>API de haut niveau</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>	<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrence</p> <p>Introduction</p> <p>Concurrency et partage des abstractions</p> <p>Les abstractions</p> <p>Théâtre en Java</p> <p>Dormir le JMM</p> <p>API de haut niveau</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>			
<p>Exécuter deux travaux réellement concurrents en parallèle est facile<sup>1</sup>, mais la réalité est souvent plus compliquée :</p> <ul style="list-style-type: none"> <li>• Si (degré de) concurrence &gt; (degré de) parallélisme, alors <b>partage du temps</b></li> <li>• Concurrence de 2 sous-programmes jamais parfaite<sup>2</sup> car nécessité de se <u>transmettre/partager</u> des résultats et de se <b>synchroniser</b>.<sup>3</sup></li> </ul> <p>→ Différentes <u>abstractions</u> pour aider à programmer de façon correcte et, si possible, intuitive, tout en prenant en compte ces réalités de diverses façons.</p> <ol style="list-style-type: none"> <li>1. On en affecte un à chaque cœur, pourvu qu'il y ait 2 cœurs disponibles, et on n'en parle plus!</li> <li>2. Sinon ce ne seraient des sous-programmes mais des programmes indépendants à part entière!</li> <li>3. En fait, ces deux aspects sont indissociables.</li> </ol>	<p>Exécuter deux travaux réellement concurrents en parallèle est facile<sup>1</sup>, mais la réalité est souvent plus compliquée :</p> <ul style="list-style-type: none"> <li>• Si (degré de) concurrence &gt; (degré de) parallélisme, alors <b>partage du temps</b></li> <li>• Concurrence de 2 sous-programmes jamais parfaite<sup>2</sup> car nécessité de se <u>transmettre/partager</u> des résultats et de se <b>synchroniser</b>.<sup>3</sup></li> </ul> <p>→ Différentes <u>abstractions</u> pour aider à programmer de façon correcte et, si possible, intuitive, tout en prenant en compte ces réalisités de diverses façons.</p> <ol style="list-style-type: none"> <li>1. On en affecte un à chaque cœur, pourvu qu'il y ait 2 cœurs disponibles, et on n'en parle plus!</li> <li>2. Sinon ce ne seraient des sous-programmes mais des programmes indépendants à part entière!</li> <li>3. En fait, ces deux aspects sont indissociables.</li> </ol>	<p>Plusieurs techniques de transmission de résultats :</p> <ul style="list-style-type: none"> <li>• <b>variables partagées</b> : variables accessibles par plusieurs tâches concurrentes.</li> <li>• Données partagées de façon transparente, sans synchronisation a priori, mais le langage permet d'insérer des primitives de <u>synchronisation explicite</u><sup>1</sup>.</li> <li>• <b>passage de message</b> : données « envoyées »<sup>2</sup> d'une tâche à l'autre.</li> </ul> <p>Synchronisation implicite de l'émission et de la réception du message : par exemple, une tâche en attente de réception est <u>bloquée</u> tant qu'elle n'a rien reçu.<sup>3</sup></p> <p>La réalité physique est plus proche du modèle des variables partagées<sup>4</sup>, mais le <b>passage de message</b> est un paradigme plus sûr<sup>5</sup>.</p> <ol style="list-style-type: none"> <li>1. En Java : <code>start()</code>, <code>join()</code>, <b>volatile</b>, <b>synchronized</b> et <code>wait()</code>/<code>notify()</code>.</li> <li>2. Sous-entendu : l'envoyeur ne peut plus accéder à ce qui a été envoyé.</li> <li>3. C'est une possibilité. On peut aussi bloquer la tâche émettrice (canal borné, « rendez-vous »).</li> <li>4. Mémoire centrale lisible par plusieurs CPU.</li> <li>5. Pour lequel la sûreté d'un programme est plus facile à prouver.</li> </ol>	<p>Un <b>ordonnanceur</b> est un programme chargé de répartir les tâches concurrentes sur les unités d'exécutions disponibles. Il s'agit souvent d'un sous-système du noyau de l'OS<sup>1</sup>. L'ordonnanceur peut mettre en œuvre :</p> <ul style="list-style-type: none"> <li>• un fonctionnement <b>multi-tâches préemptif</b> : l'ordonnanceur choisit quand mettre en pause une tâche pour reprendre l'exécution d'une autre. Cela peut arriver (presque) à tout moment.</li> </ul> <p>C'est le cas pour la gestion des processus dans les OS modernes pour ordinateur personnel.</p> <ul style="list-style-type: none"> <li>• ou bien un fonctionnement <b>multi-tâches coopératif</b> : chaque tâche signale à l'ordonnanceur quand elle peut être mise en attente (par exemple en faisant un appel bloquant).</li> </ul> <p>1. Operating System/système d'exploitation</p>	<p>Un <b>ordonnanceur</b> est un programme chargé de répartir les tâches concurrentes sur les unités d'exécutions disponibles. Il s'agit souvent d'un sous-système du noyau de l'OS<sup>1</sup>. L'ordonnanceur peut mettre en œuvre :</p> <ul style="list-style-type: none"> <li>• un fonctionnement <b>multi-tâches préemptif</b> : l'ordonnanceur choisit quand mettre en pause une tâche pour reprendre l'exécution d'une autre. Cela peut arriver (presque) à tout moment.</li> </ul> <p>C'est le cas pour la gestion des processus dans les OS modernes pour ordinateur personnel.</p> <ul style="list-style-type: none"> <li>• ou bien un fonctionnement <b>multi-tâches coopératif</b> : chaque tâche signale à l'ordonnanceur quand elle peut être mise en attente (par exemple en faisant un appel bloquant).</li> </ul> <p>1. Operating System/système d'exploitation</p>	<p>Mais on peut simuler le passage de message :</p> <pre>public final class MailBox&lt;T&gt; { // classe réutilisable, simulant un passage de message avec "rendez-vous"     private T content; // mémoire partagée, encapsulée     public synchronized void sendMessage(T message) throws InterruptedException {         while (content != null) wait(); // attend la condition content != null         content = message;         notifyAll(); // débloque les autres threads en attente sur cette MailBox     }     public synchronized T receiveMessage() throws InterruptedException {         while (content == null) wait(); // attend la condition content == null         T ret = content;         content = null;         notifyAll(); // débloque les autres threads en attente sur cette MailBox         return ret;     } }</pre> <p>public final class PingPong {     public static void main(String[] args) {         var box = new MailBox&lt;String&gt;();         new Thread(() -&gt; {             try {                 while (true) box.sendMessage("ping !");             } catch (Exception e) { throw new RuntimeException(e); }         }).start(); // producteur/écrivain         new Thread(() -&gt; {             try {                 while (true) System.out.println(((box.receiveMessage() == "ping !") ? "pong !" : "error !"));             } catch (Exception e) { start(); // consommateur/lecteur             }         }).start();     } }</p>	<p>Mais on peut simuler le passage de message :</p> <pre>public final class MailBox&lt;T&gt; { // classe réutilisable, simulant un passage de message avec "rendez-vous"     private T content; // mémoire partagée, encapsulée     public synchronized void sendMessage(T message) throws InterruptedException {         while (content != null) wait(); // attend la condition content != null         content = message;         notifyAll(); // débloque les autres threads en attente sur cette MailBox     }     public synchronized T receiveMessage() throws InterruptedException {         while (content == null) wait(); // attend la condition content == null         T ret = content;         content = null;         notifyAll(); // débloque les autres threads en attente sur cette MailBox         return ret;     } }</pre> <p>public final class PingPong {     public static void main(String[] args) {         var box = new MailBox&lt;String&gt;();         new Thread(() -&gt; {             try {                 while (true) box.sendMessage("ping !");             } catch (Exception e) { throw new RuntimeException(e); }         }).start(); // producteur/écrivain         new Thread(() -&gt; {             try {                 while (true) System.out.println(((box.receiveMessage() == "ping !") ? "pong !" : "error !"));             } catch (Exception e) { start(); // consommateur/lecteur             }         }).start();     } }</p>

## Questions de synchronisation

Recevoir des messages

- fonctions bloquantes** : la tâche réceptrice appelle une fonction fournie par la bibliothèque, qui la  bloque  jusqu'à ce que la valeur attendue soit disponible.

```

ForkJoinTask<Result> task = ForkJoinTask.adapt(() -> {
    ... // tâche 1
    () . fork () ;
    ...
    return result ;
}) . fork () ;
... // plus loin
Result x = task . join () ; // appel à fonction bloquante join ()
... // tâche 2 : fait qqc avec le résultat x de tâche 1

```

- fonctions de rappel (callbacks)** : on passe à la bibliothèque une fonction que celle-ci appellera sur le résultat attendu dès qu'il sera disponible.

```

CompletableFuture . supplyAsync () -> {
    ... // tâche 1
    return result ;
}) . thenApply ((x) -> { // corps de la fonction de rappel
    ... // tâche 2 : fait qqc avec le résultat x de tâche 1
}) ;

```

Envoyer des messages

Envoyer le résultat x d'un calcul, ça peut être simplement :

- retourner x à la fin d'une fonction (tâche productrice), c'est le cas dans les exemples précédents (« **return result** »).
- passer x en paramètre d'un appel de méthode. Par exemple, on peut soumettre la valeur à une file d'attente synchronisée :

```

... // calcule x
queue . offer (x) ;
... // fait autre chose (avec interdiction de toucher à x !)

```

Dans ce dernier cas, la tâche consommatrice reçoit le message en appelant queue . take() (fonction bloquante).

- Remarque : cela est similaire à l'exemple de la classe **MailBox** donné plus tôt<sup>1</sup>.
1. Différence : **MailBox** ne stocke qu'un seul message (= « Rendez-vous »), alors qu'une file d'attente peut en stocker plusieurs, permettant au consommateur et au producteur de ne pas suivre le même rythme.

Exemple simple (1)

Compléments

Exemple simple (1)

- Exemple de deux threads, l'un qui compte jusqu'à 10 alors que l'autre récite l'alphabet :
- ```

class ReciteNombres extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++)
            System.out.print(i + " ");
    }
}

class ReciteAlphabet extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 26; i++)
            System.out.print((char) ('a'+i) + " ");
    }
}

```
- Abstraction concurrente consistant en une séquence d'instructions dont l'exécution **simule une exécution séquentielle** (en interne)<sup>1</sup> et **parallelise** à celle des autres threads.

### Definition (**Thread** ou fil d'exécution)

- Un nombre quelconque de threads s'exécute sur une plateforme de degré *n* parallélisme quelconque<sup>2</sup>. Un ordonnanceur partage les ressources de la plateforme pour que cela soit possible.

- Ainsi, *n* threads en exécution simultanée simulent un parallélisme de degré *n* parallélisme quelconque<sup>2</sup>. Un ordonnanceur partage les ressources de la plateforme pour que cela soit possible.
- Un nombre quelconque de threads s'exécute sur une plateforme de degré *n* parallélisme quelconque<sup>2</sup>. Un ordonnanceur partage les ressources de la plateforme pour que cela soit possible.
- Un processus<sup>3</sup> (= 1 application en exécution) peut utiliser plusieurs threads qui ont accès aux mêmes données (mémoire partagée).

1. Ce qui permet de le programmer avec les principes habituels de programmation impérative : séquences d'instructions, boucles, branchements, pile d'appels de fonctions, ...  
 2. Même inférieur au nombre de threads  
 3. Cette fois-ci au sens où on l'entend en informatique.

Envoyer des messages

## Notion de *Thread*

Exemple simple (2)

## Notion de *thread*

Un concept décliné à plusieurs niveaux (1)

### Alors

```
public class Exemple {
    public static void main(String[] args) {
        new ReciteNombres().start();
    }
}
```

peut afficher

```
0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v w x y z
```

mais également

```
0 1 2 3 a b c d 4 5 e 6 f 7 g 8 h 9 i j k l m n o p q r s t u v w x y z
```

ou encore

```
0 1 2 3 4 a 5 b 6 c 7 d 8 e 9 f g h i j k l m n o p q r s t u v w x y z
```

Compléments en 200  
Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrency

Concurrency et parallélisme

Les abstractions

Théories en Java

Démarrer le JVM

APIs de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

Exemple

Compléments en 200  
Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrency

Concurrency et parallélisme

Les abstractions

Théories en Java

Démarrer le JVM

APIs de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

Compléments en 200  
Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrency

Concurrency et parallélisme

Les abstractions

Théories en Java

Démarrer le JVM

APIs de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

## À propos des *threads* système

Avantages et inconvénients

- Ce sont des *threads*.

**Avantage** : se programment séquentiellement (respectent les habitudes).

**Inconvénient** : la synchronisation doit être explicitée par le programmeur.<sup>1</sup>

- Multi-tâche préemptif : l'ordonnanceur peut suspendre un *thread* (au profit d'un autre), à tout moment

**Avantage** : pas besoin de signaler quand le programme doit « laisser la main ».  
**Inconvénient** : changements de contexte fréquents et coûteux.

- Implémentation dans le noyau :

**Avantage** : compatible avec tous les exécutables de l'OS (pas seulement JVM)

**Inconvénient** : fonctionnalités rudimentaires. P. ex., chaque *thread* a une pile de taille fixe (1024 ko pour les *threads* de la JVM 64bits) → peu économique!

## Notion de *thread*

Un concept décliné à plusieurs niveaux (2)

## Notion de *thread*

Avantages et inconvénients

- Dans le **runtime des langages de programmation** : des langages de programmation (Erlang, Go, Haskell, Lua, ...), mais pas actuellement Java), contiennent une notion de *thread* « léger » (différents noms : green thread, fibre, coroutine, goroutine, ...), s'exécutant par dessus un ou des *threads* système.

**Langage/runtime**    **Abstractions (fibres, coroutines, acteurs, futurs, événements, ...)**

| thread       | thread | thread | thread | thread | thread | ... |
|--------------|--------|--------|--------|--------|--------|-----|
| Ordonnanceur |        |        |        |        |        |     |

**OS (noyau)**

| Proc. logique    | Proc. logique | Proc. logique | Proc. logique |
|------------------|---------------|---------------|---------------|
| SMT              |               | SMT           |               |
| Cœur             |               | Cœur          |               |
| CPU <sup>2</sup> |               |               |               |

1. Sous-entendu : « *thread* système » (« *thread* » sans précision = « *thread* système »).
2. Possible aussi : plusieurs CPUs (plusieurs coeurs par CPU, plusieurs processeurs logiques par cœur...).

1. On va voir dans la suite comment. Pour l'instant, retenez qu'il n'y a aucune synchronisation, donc aucun partage de données sûr entre *threads* si on n'ajoute pas « quelque chose ».

| Aldric Degorre | Introduction<br>Généralités<br>Style<br>Objets et classes<br>Types et polymorphisme<br>Héritage<br>Généricité<br>Concurrency<br>Introduction<br>Concurrent et parallèle<br>Les abstractions<br>Threads en Java<br>Bomber le JMM<br>APIs de haut niveau                                                                                                                                                                                                                                                                                                                                                  | Interfaces graphiques<br>Gestion des erreurs et exceptions |
|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|
|                | <p>→ <b>les langages de programmation proposent des mécanismes, utilisant les threads système, pour pallier leurs inconvénients tout en essayant [1] de garder leur avantages.</b></p> <p><b>Au moins deux objectifs :</b></p> <ul style="list-style-type: none"> <li>• <b>limiter le nombre de threads système utilisés, afin de diminuer l'empreinte mémoire et la fréquence des changements de contexte</b></li> <li>• <b>forcer des procédés sûrs pour le partage de données; ou à défaut, faciliter les bonnes pratiques de synchronisation.</b></li> </ul> <p>1. avec plus ou moins de succès</p> |                                                            |

| Aldric Degorre | Java                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | Interfaces graphiques<br>Gestion des erreurs et exceptions |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|
|                | <ul style="list-style-type: none"> <li>• utilise directement les threads système, via la classe <code>Thread</code>.</li> <li>• a historiquement (Java 1.1) utilisé des <i>green threads</i><sup>1</sup>, abandonnés pour des raisons de performance<sup>2</sup>.</li> <li>• pourrait néanmoins, dans le futur, supporter les fibres<sup>3</sup> via le projet Loom.</li> <li>• dispose actuellement d'un grand nombre d'APIs facilitant où rendant plus sûre l'utilisation des threads : les boucles d'événements Swing et JavaFX, <code>ThreadPoolExecutor</code>, <code>ForkJoinPool</code>/<code>ForkJoinTask</code>, <code>CompletableFuture</code>, <code>Stream</code>...</li> </ul> <p>1. Une sorte de threads légers.</p> <p>2. Ils étaient ordonnancés sur un seul thread système, empêchant d'utiliser plusieurs processeurs.</p> <p>3. Autre type de threads légers. Cette fois-ci, le travail peut être distribué sur plusieurs threads système.</p> <p>Des implémentations de fibres pour Java existent déjà : bibliothèques Quasar et Kiliim. Mais pour fonctionner, celles-ci doivent modifier le code-octet généré par <code>javac</code>.</p> |                                                            |

| Aldric Degorre | Threads en Java | Concrètement |
|----------------|-----------------|--------------|
|                |                 |              |

| Aldric Degorre | Threads en Java                                                                                                                                                                                                                                                                                                                                               | Concrètement                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                | <p>Introduction<br/>Généralités<br/>Style<br/>Objets et classes<br/>Types et polymorphisme<br/>Héritage<br/>Généricité<br/>Concurrency<br/>Introduction<br/>Threads en Java<br/>Bomber le JMM<br/>APIs de haut niveau</p> <p>Threads en Java<br/>Bomber le JMM<br/>APIs de haut niveau</p> <p>Interfaces graphiques<br/>Gestion des erreurs et exceptions</p> | <ul style="list-style-type: none"> <li>Tous les threads ont accès au même tas (mêmes objets) et à la même zone statique (mêmes classes)... mais pas à la même pile!</li> <li>Les threads communiquent grâce aux <b>variables partagées</b>, stockées dans le tas.</li> <li>Une même méthode peut être appelée depuis n'importe quel thread (pas de séparation syntaxique du code associé aux différents threads).</li> <li>Pour démarrer un thread : <code>unObjetThread.start()</code> ; (où <code>unObjetThread</code> instance de la classe <code>Thread</code>).<br/>→ aussitôt, appel de <code>unObjetThread.run()</code> dans le thread associé à cet objet.</li> <li>À chaque thread correspond une pile d'appels de méthode. En bas de la pile : <ul style="list-style-type: none"> <li>pour le thread <code>main</code>, le frame de la méthode <code>main</code>,</li> <li>pour les autres, celui de l'appel initial à <code>run</code> sur l'objet représentant le <i>thread</i>.</li> </ul> </li> </ul> <p>1. Pour Swing : <i>Event Dispatching Thread</i> (EDT). Pour JavaFX : <i>JavaFX Application Thread</i>.</p> <p>2. Et l'intérêt de n'avoir qu'un seul thread pour cela : la sûreté du fonctionnement de l'IG. Pas d'entrelacements entre 2 événements, pas d'accès compétition.</p> |

- Définir et instancier une classe héritant de la classe *Thread* :

```
public class HelloThread extends Thread {
    @Override public void run() { System.out.println("Hello from a thread!"); }
    // plus loin
    new HelloThread().start();
```

- Implémenter *Runnable* et appeler le constructeur *Thread(Runnable target)* :

```
public class HelloRunnable implements Runnable {
    @Override public void run() { System.out.println("Hello from a thread!"); }
    // plus loin
    new Thread(new HelloRunnable()).start();
```

- Mais pour un *thread* simple, on préférera écrire une lambda-expression :

```
new Thread(() -> { System.out.println("Hello from a thread!"); }).start();
```

- 1. => *Thread* est ainsi un décorateur de *Runnable*.

- L'interface *Runnable* a pour seule méthode (abstraite) **void run()**.
  - Cette interface n'a, *a priori*, aucun rapport avec les *threads*, mais :
    - ses instances sont souvent passées au constructeur de *Thread* pour programmer leur exécution sur un nouveau *thread*;
    - Thread* implémente *Runnable* (et possède d'autres méthodes, voir la suite);
    - la méthode *run* de *Thread* appelle la méthode *run* du *Runnable* passé en paramètre (le cas échéant).<sup>1</sup>
- L'approche consistant à définir des tâches en implémentant directement *Runnable* plutôt qu'en étendant *Thread* laisse la possibilité d'hériter d'une autre classe :

```
public class MaClasse extends JFrame implements Runnable {
    public void run() {
        ...
    }
}
```

- 1. => *Thread* est ainsi un décorateur de *Runnable*.

- Introduction
- Généralités
- Style
- Objets et classes
- Types et polymorphisme
- Héritage
- Généricité
- Concurrence
- Introduction
- Threads en Java
- Introduction
- La classe *Thread*
- Synchronisation
- Boucle de JMM
- APIs de haut niveau
- Interfaces graphiques
- Gestion des erreurs et exceptions
- Détails

- Une instance de *thread* est toujours dans un des états suivants :
- **NEW** : juste créé, pas encore démarré.
- **RUNNABLE** : en cours d'exécution.
- **BLOCKED** : en attente de moniteur (voir la suite).
- **WAITING** : en attente d'une condition d'un autre *thread* (voir *notify()*/*wait()*).
- **TIME\_WAITING** : idem pour attente avec temps limite.
- **TERMINATED** : exécution terminée.

Mais attendons la suite pour en dire plus sur ces états...

- Introduction
- Généralités
- Style
- Objets et classes
- Types et polymorphisme
- Héritage
- Généricité
- Concurrence
- Introduction
- Threads en Java
- Introduction
- La classe *Thread*
- Synchronisation
- Boucle de JMM
- APIs de haut niveau
- Interfaces graphiques
- Gestion des erreurs et exceptions
- Détails

- String *getName()* : récupérer le nom d'un *thread*.
- void join()** : attendre la fin de ce *thread* (voir synchronisation).
- void run()** : la méthode qui lance tout le travail de ce *Thread*. C'est la méthode qu'il faudra redéfinir à chaque fois que *Thread* sera étendue !.
- static void sleep (long millis)** : met le *thread* courant (i.e. en cours d'exécution) en pause pendant tant de ms. (NB : c'est une méthode **static**. Le *thread* mis en pause est celui qui appelle la méthode. Il n'y a pas de **this**!).
- void start()** : démarre le *thread* (conséquence : *run()* est exécutée dans le nouveau *thread* : celui décrit par l'objet, pas celui de l'appelant!)..
- void interrupt()** : interrompt le *thread* (déclenche *InterruptedException* si le *thread* était en attente sur *wait()*, *join()*, *sleep()*...).
- static boolean interrupted()** : teste si un autre *thread* a demandé l'interruption du *thread* courant.
- Thread.getState()** : retourne l'état du *thread*.

- Si `t` est un *thread*, l'appel `t.interrupt()` demande l'interruption de celui-ci.
  - Si `t` est en train d'exécuter une méthode `interruptible`<sup>1</sup>, celle-ci quitte tout de suite.
  - L'interruption est propagée le long des méthodes de la pile d'appel qui quittent une à une... jusqu'à la méthode principale de la tâche<sup>2</sup> qui quitte aussi.
  - Le résultat (non garanti<sup>3</sup>) est la terminaison de la tâche exécutée sur `t`<sup>4</sup>.
  - La propagation de l'interruption est implémentée par la propagation de l'exception `InterruptedException` et par le contrôle du booléen `Thread.interrupted()` (détails juste après).
1. C'est le cas de toutes les méthodes bloquantes de l'API `Thread.wait()`, `sleep()`, `join()`...
2. Habituellement : `run`.
3. Si les méthodes exécutées sur `t` n'ont pas prévu d'être interrompues, rien ne se passe.
4. Si exécution directe dans le *thread*, terminaison du *thread*, sinon, si exécution dans un *thread pool*, le *thread* est juste rendu de nouveau disponible.

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- Pour écrire une méthode interruptible `f` :
  - Quand une interruption est détectée la bonne pratique est de quitter (`return` ou `throw`) au plus tôt, tout en libérant les ressources utilisées.
  - L'interruption peut être détectée de deux façons :
    - soit une méthode auxiliaire `g` appelée depuis `f` quitte sur `InterruptedException`
    - soit on a obtenu `true` en appelant `Thread.interrupted()`.
  - Le premier cas (exception) doit être traité en mettant tout appelle à `g` dans un block `try/finally` (libération explicite des ressources de `f` dans le `finally`) ou bien `try-with-resource` (libération implicite).
  - Remarque : il faut absolument vérifier `Thread.interrupted()` dans toute boucle de `f` ne faisant pas d'appel à une méthode interruptible comme `g`.
  - Dans tous les cas, il faut veiller à propager le statut « interrompu » au contexte d'exécution, pour qu'il puisse, lui aussi, prendre en compte le fait qu'une interruption a eu lieu. 2 cas de figure (voir la suite).

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- Pour écrire une méthode interruptible `f` :
  - Quand une interruption est détectée la bonne pratique est de quitter (`return` ou `throw`) au plus tôt, tout en libérant les ressources utilisées.
  - L'interruption peut être détectée de deux façons :
    - soit une méthode auxiliaire `g` appelée depuis `f` quitte sur `InterruptedException`
    - soit on a obtenu `true` en appelant `Thread.interrupted()`.
  - Le premier cas (exception) doit être traité en mettant tout appelle à `g` dans un block `try/finally` (libération explicite des ressources de `f` dans le `finally`) ou bien `try-with-resource` (libération implicite).
  - Remarque : il faut absolument vérifier `Thread.interrupted()` dans toute boucle de `f` ne faisant pas d'appel à une méthode interruptible comme `g`.
  - Dans tous les cas, il faut veiller à propager le statut « interrompu » au contexte d'exécution, pour qu'il puisse, lui aussi, prendre en compte le fait qu'une interruption a eu lieu. 2 cas de figure (voir la suite).

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- Pour écrire une méthode interruptible `f` :
  - Quand une interruption est détectée la bonne pratique est de quitter (`return` ou `throw`) au plus tôt, tout en libérant les ressources utilisées.
  - L'interruption peut être détectée de deux façons :
    - soit une méthode auxiliaire `g` appelée depuis `f` quitte sur `InterruptedException`
    - soit on a obtenu `true` en appelant `Thread.interrupted()`.
  - Le premier cas (exception) doit être traité en mettant tout appelle à `g` dans un block `try/finally` (libération explicite des ressources de `f` dans le `finally`) ou bien `try-with-resource` (libération implicite).
  - Remarque : il faut absolument vérifier `Thread.interrupted()` dans toute boucle de `f` ne faisant pas d'appel à une méthode interruptible comme `g`.
  - Dans tous les cas, il faut veiller à propager le statut « interrompu » au contexte d'exécution, pour qu'il puisse, lui aussi, prendre en compte le fait qu'une interruption a eu lieu. 2 cas de figure (voir la suite).

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- 2 cas de figure, selon que la signature de `f` est imposée ou non :
- Si ce n'est pas le cas, on propage le statut « interrompu » en quittant sur `InterruptedException`. 2 cas de figure :
    - si une méthode appellée depuis `f` a elle-même lancé `InterruptedException` : dans ce cas on ne met pas de `catch` et la propagation est automatique.
    - sinon, on peut ajouter `throw new InterruptedException();`
  - `InterruptedException` étant une exception sous contrôle, il faut aussi ajouter `throws InterruptedException` à la signature de `f`.

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- 2 cas de figure, selon que la signature de `f` est imposée ou non :
- Si ce n'est pas le cas, on propage le statut « interrompu » en quittant sur `InterruptedException`. 2 cas de figure :
    - si une méthode appellée depuis `f` a elle-même lancé `InterruptedException` : dans ce cas on ne met pas de `catch` et la propagation est automatique.
    - sinon, on peut ajouter `throw new InterruptedException();`
  - `InterruptedException` étant une exception sous contrôle, il faut aussi ajouter `throws InterruptedException` à la signature de `f`.

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- Simon, si la signature de `f` est imposée par l'interface implémentée (ex : `Runnable`) et ne contient pas `throws InterruptedException`, on ne peut alors pas quitter sur `InterruptedException`.
- Solution : avant `return` on appelle `System.currentThread().interrupt()` (ce qui fait que le prochain appel à `interrupt()` retournera bien `true`).

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- Exemples de méthodes interruptibles :
- ```
// avec while et acquisition/libération de resource (bloc "try-with-resource")
Data f(Data x) throws InterruptedException {
    try (Scanner s = new Scanner(System.in)) {
        while(test(x)) {
            x = transform(x, s.next());
            if (Thread.interrupted()) throw new InterruptedException(); // <-- ici !
        }
        return x;
    } // s.close() appelée implicitement à la sortie du bloc (par throw ou par return)
}
```
- // exemple sans boucle, mais avec appel bloquant  
**void sleep5s()** throws InterruptedException {  
 System.out.println("Acquisition potentielle de ressource");  
 try {  
 Thread.sleep(5000); // on attend 5s  
 } finally { System.out.println("Libération de la même ressource"); }  
 // Pas de "catch". Si sleep() envoie InterruptedException, elle est propagée.  
}

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- Exemples de méthodes interruptibles :
- ```
// avec while et acquisition/libération de resource (bloc "try-with-resource")
Data f(Data x) throws InterruptedException {
    try (Scanner s = new Scanner(System.in)) {
        while(test(x)) {
            x = transform(x, s.next());
            if (Thread.interrupted()) throw new InterruptedException(); // <-- ici !
        }
        return x;
    } // s.close() appelée implicitement à la sortie du bloc (par throw ou par return)
}
```
- // exemple sans boucle, mais avec appel bloquant  
**void sleep5s()** throws InterruptedException {  
 System.out.println("Acquisition potentielle de ressource");  
 try {  
 Thread.sleep(5000); // on attend 5s  
 } finally { System.out.println("Libération de la même ressource"); }  
 // Pas de "catch". Si sleep() envoie InterruptedException, elle est propagée.  
}

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- Exemples de méthodes interruptibles :
- ```
// avec while et acquisition/libération de resource (bloc "try-with-resource")
Data f(Data x) throws InterruptedException {
    try (Scanner s = new Scanner(System.in)) {
        while(test(x)) {
            x = transform(x, s.next());
            if (Thread.interrupted()) throw new InterruptedException(); // <-- ici !
        }
        return x;
    } // s.close() appelée implicitement à la sortie du bloc (par throw ou par return)
}
```
- // exemple sans boucle, mais avec appel bloquant  
**void sleep5s()** throws InterruptedException {  
 System.out.println("Acquisition potentielle de ressource");  
 try {  
 Thread.sleep(5000); // on attend 5s  
 } finally { System.out.println("Libération de la même ressource"); }  
 // Pas de "catch". Si sleep() envoie InterruptedException, elle est propagée.  
}

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- Exemples de méthodes interruptibles :
- ```
// avec while et acquisition/libération de resource (bloc "try-with-resource")
Data f(Data x) throws InterruptedException {
    try (Scanner s = new Scanner(System.in)) {
        while(test(x)) {
            x = transform(x, s.next());
            if (Thread.interrupted()) throw new InterruptedException(); // <-- ici !
        }
        return x;
    } // s.close() appelée implicitement à la sortie du bloc (par throw ou par return)
}
```
- // exemple sans boucle, mais avec appel bloquant  
**void sleep5s()** throws InterruptedException {  
 System.out.println("Acquisition potentielle de ressource");  
 try {  
 Thread.sleep(5000); // on attend 5s  
 } finally { System.out.println("Libération de la même ressource"); }  
 // Pas de "catch". Si sleep() envoie InterruptedException, elle est propagée.  
}

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

- Exemples de méthodes interruptibles :
- ```
// avec while et acquisition/libération de resource (bloc "try-with-resource")
Data f(Data x) throws InterruptedException {
    try (Scanner s = new Scanner(System.in)) {
        while(test(x)) {
            x = transform(x, s.next());
            if (Thread.interrupted()) throw new InterruptedException(); // <-- ici !
        }
        return x;
    } // s.close() appelée implicitement à la sortie du bloc (par throw ou par return)
}
```
- // exemple sans boucle, mais avec appel bloquant  
**void sleep5s()** throws InterruptedException {  
 System.out.println("Acquisition potentielle de ressource");  
 try {  
 Thread.sleep(5000); // on attend 5s  
 } finally { System.out.println("Libération de la même ressource"); }  
 // Pas de "catch". Si sleep() envoie InterruptedException, elle est propagée.  
}

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Respecte le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Détails

## Deux principaux problèmes :

- 1 Les entrelacements non maîtrisés** : les instructions de 2 threads **s'entrelacent**<sup>1</sup> et accèdent (lecture et écriture) aux mêmes données dans un ordre imprévisible. Ce phénomène est « naturel » (l'ordonnanceur est libre de faire avancer un *thread*, puis l'autre au moment où il veut); il est parfois gênant, parfois non.
- 2 Les incohérences dues aux optimisations matérielles**<sup>2</sup> : la JVM<sup>3</sup> laisse une marge d'interprétation assez large au matériel pour qu'il puisse exécuter le programme efficacement. Principales conséquences :

- ordre des instructions donné dans le code source pas forcément respecté
- modifications de variables partagées pas forcément vues par les autres *threads*.

Pour l'instant, concentrons nous sur le problème 1.

1. *interleave*
2. en particulier dans le microprocesseur
3. La JVM s'appuie sur le **JMM** : Java Memory Model, un modèle d'exécution relativement laxe.

Qu'est-ce qui est affiché quand on exécute le programme suivant ?

```
public class ThreadInterferences extends Thread {
    static int x = 0;
    public ThreadInterferences(String name){ super(name); }

    @Override
    public void run() {
        while(x++ < 10) System.out.println("x incrémenté par " + getName() + ", sa
                                         nouvelle valeur est " + x + ".");
    }
}

public static void main(String[] args){
    new ThreadInterferences("t1").start();
    new ThreadInterferences("t2").start();
}
```

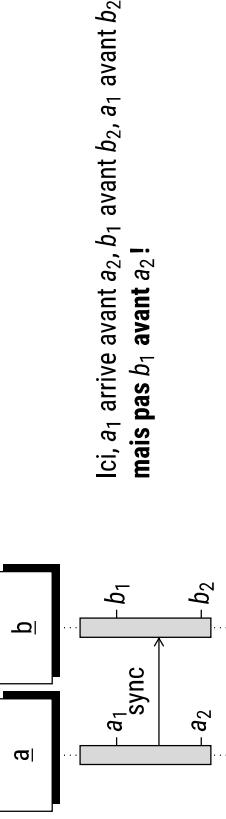
On s'attend à voir tous les entier de 1 à 10 s'afficher dans l'ordre.

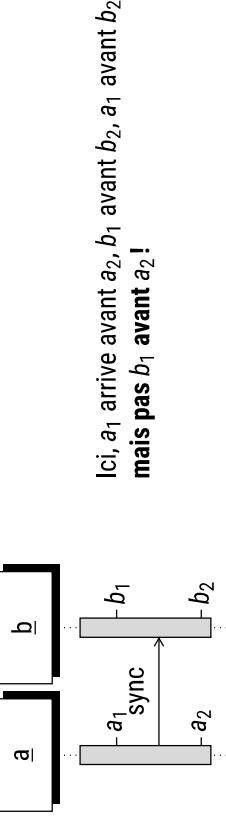
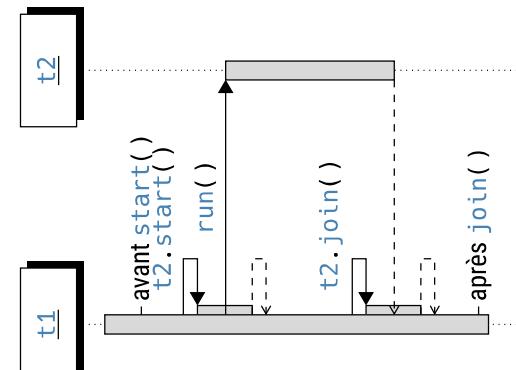
```
x incrémenté par t2, sa nouvelle valeur est 2.
x incrémenté par t1, sa nouvelle valeur est 2.
x incrémenté par t2, sa nouvelle valeur est 3.
x incrémenté par t1, sa nouvelle valeur est 4.
x incrémenté par t2, sa nouvelle valeur est 5.
x incrémenté par t1, sa nouvelle valeur est 6.
x incrémenté par t2, sa nouvelle valeur est 7.
x incrémenté par t2, sa nouvelle valeur est 8.
x incrémenté par t2, sa nouvelle valeur est 9.
x incrémenté par t2, sa nouvelle valeur est 10.
x incrémenté par t1, sa nouvelle valeur est 8.
```

Contrairement à ce qu'on pourrait attendre : les nombres ne sont pas dans l'ordre, certains se répètent, d'autres n'apparaissent pas.

- 1. Notion abordée plus loin.
- 2. Idem. Dans ce cas, remplacer « accédant aux mêmes données » par « utilisant le même verrou ».
- Exemple d'opération non atomique : `++` (peut se décomposer ainsi : copie `x` en pile, empile 1, additionne, copie le sommet de pile dans `x`).

1. Notion abordée plus loin.  
2. Idem. Dans ce cas, remplacer « accédant aux mêmes données » par « utilisant le même verrou ».

La synchronisation	
<p><b>Synchronisation :</b></p> <ul style="list-style-type: none"> <li>consiste, pour un <i>thread</i>, à attendre le « feu vert » d'un autre <i>thread</i> avant de continuer son exécution;</li> <li>interdit certains entrelacements;</li> <li>contribute à établir la relation "arrivé-avant", limitant les optimisations autorisées<sup>1</sup>.</li> </ul> 	<p>Synchronisation simple : attendre la terminaison d'un <i>thread</i> avec <code>join()</code><sup>1</sup> :</p> <pre>public class ThreadJoin extends Thread {     static int x = 0;     @Override     public void run(){ System.out.println(x); }     public static void main(String[] args){         Thread t = new ThreadJoin();         t.start();         t.join();         x++;     } }</pre>
<p>1. À suivre...</p>	<p>Affiche 0 alors que le même code sans l'appel à <code>join()</code> affichera probablement 1.</p> <p>Tout ce qui est exécuté dans le <i>thread</i> <code>t</code> arrive-avant ce qui suit le <code>join()</code> dans le <i>thread</i> <code>main</code> (ici, l'incrémentation de <code>x</code>).</p> <p>1. Existe aussi en version temporisée : on bloque jusqu'au délai donné en paramètre maximum.</p>

La synchronisation	
<p><b>Synchronisation :</b></p> <ul style="list-style-type: none"> <li>consiste, pour un <i>thread</i>, à attendre le « feu vert » d'un autre <i>thread</i> avant de continuer son exécution;</li> <li>interdit certains entrelacements;</li> <li>contribute à établir la relation "arrivé-avant", limitant les optimisations autorisées<sup>1</sup>.</li> </ul> 	<p>Synchronisation avec <code>join()</code></p> 
<p>1. À suivre...</p>	<p>Le verrou intrinsèque (ou moniteur).</p> <p>Qu'est-ce, à quoi cela sert-il ?</p>

Le verrou intrinsèque	
<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrence</p> <p>Introduction</p> <p>Threads en Java</p> <p>Introduction</p> <p>La classe Thread</p> <p>Synchronisation</p> <p>Transférer le JMM</p> <p>Arts de haut niveau</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>	<p>Qu'est-ce, à quoi cela sert-il ?</p> <ul style="list-style-type: none"> <li>En Java tout objet contient un <b>verrou intrinsèque</b> (ou <b>moniteur</b>).</li> <li>À tout moment, le moniteur est soit <u>libre</u>, soit <u>détenu</u> par un (seul) <i>thread</i> donné.</li> <li>Ainsi un moniteur met en œuvre le principe d'exclusion mutuelle.</li> <li>Lors de son exécution, un <i>thread</i> <code>t</code> peut demander à prendre un moniteur.             <ul style="list-style-type: none"> <li>S'il le moniteur est déjà pris, <code>t</code> est alors mis en attente jusqu'à ce que le moniteur se libère pour lui (il peut y avoir une liste d'attente).</li> <li>Un <i>thread</i> peut à tout moment libérer un moniteur qu'il possède.</li> </ul> </li> </ul> <p><b>Conséquence</b> : tout ce qui se produit dans un <i>thread</i> avant qu'il libère un moniteur arrive-avant ce qui se produit dans le prochain <i>thread</i> qui obtiendra le moniteur, après l'obtention de celui-ci.</p>
<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrence</p> <p>Introduction</p> <p>Threads en Java</p> <p>Introduction</p> <p>La classe Thread</p> <p>Synchronisation</p> <p>Transférer le JMM</p> <p>Arts de haut niveau</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>	

## Bloc synchronisé :

```
class AutreCompteur{
    private int valeur;
    private Object verrou = new Object(); // peu importe le type déclaré
    public void incr(){{
        synchronized(verrou){ // méthode contenant bloc synchronisé
            valeur++;
        }
    }
}
```

**Sémantique** : le thread qui exécute ce bloc demande le moniteur de verrou en y entrant et le libère en en sortant.

**Conséquence** : pour une instance donnée de AutreCompteur, le bloc n'est exécuté que par un seul thread en même temps (exclusion mutuelle). Les autres threads qui essayent d'y entrer sont suspendus (BLOCKED).

## Bloc synchronisé :

```
class Compteur {
    private int valeur;
    // méthode contenant bloc synchronisé
    public synchronized void incr(){{
        valeur++;
    }
}
```

équivalent à...

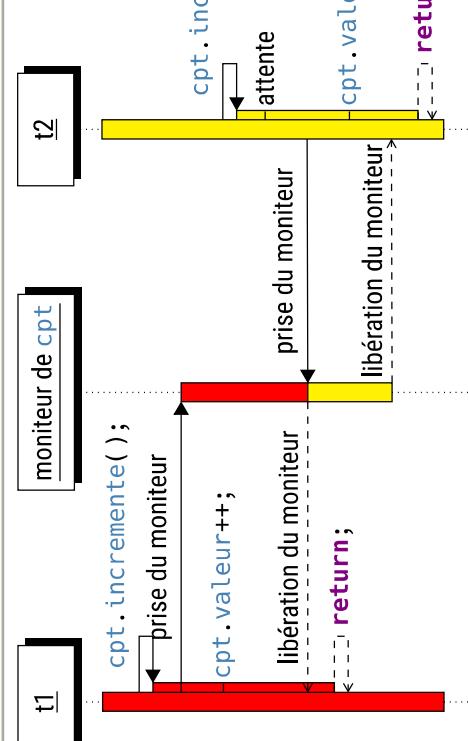
**Méthode synchronisée** : cas particulier avec synchronisation de tout le corps de la méthode sur moniteur de this. → syntaxe plus légère, plus souvent utilisée en pratique.

**Note** : l'exclusion mutuelle porte sur le moniteur (1 par objet) et non sur le bloc synchronisé (souvent plusieurs par moniteur).

**Conséquence** : 1 bloc synchronisé n'a qu'une seule exécution simultanée. De plus, aucun autre bloc synchronisé sur le même moniteur ne sera exécuté en même temps.

## Le verrou intrinsèque

```
Compteur cpt = new Compteur();
new Thread(cpt::incremente).start(); new Thread(cpt::incremente).start();
```



- 3 méthodes concernées (classe Object) : notify(), notifyAll() et wait().
- Ces méthodes sont appelables seulement dans un bloc synchronisé sur l'objet récepteur de l'appel : synchronized(x){ x.wait(); }.
- wait() : met le thread en sommeil et libère le moniteur (getState() passe de RUNNABLE à WAITING).

Le thread restera dans cet état tant qu'il n'est pas réveillé (par notifyAll() ou notify()). Il sera alors en attente pour récupérer le moniteur (WAITING → BLOCKED).

- notifyAll() : réveille tous les threads en attente sur l'objet. Ceux-ci deviennent candidats à reprendre le moniteur quand il sera libéré.
- notify() : réveille un thread en attente sur l'objet.

- On utilise `wait()` pour attendre une condition `cond`.

- Mais plusieurs threads peuvent être en attente. Un autre pourrait être libéré et récupérer le moniteur avant, rendant la condition à nouveau fausse.

- aucune garantie que `cond` soit vraie au retour de `wait()`.

Ainsi, il faut tester à nouveau jusqu'à satisfaire la condition :

```
synchronized(obj) { // conseil : mettre wait dans un while
    while(!condition(obj)) obj.wait();
    ... // insérer ici instructions qui avaient besoin de condition()
}
```

### Il faut absolument retenir la formule ci-dessus!!!

(utilisée dans 99,9% des cas d'usage corrects de `wait...`)

### Variantes acceptables :

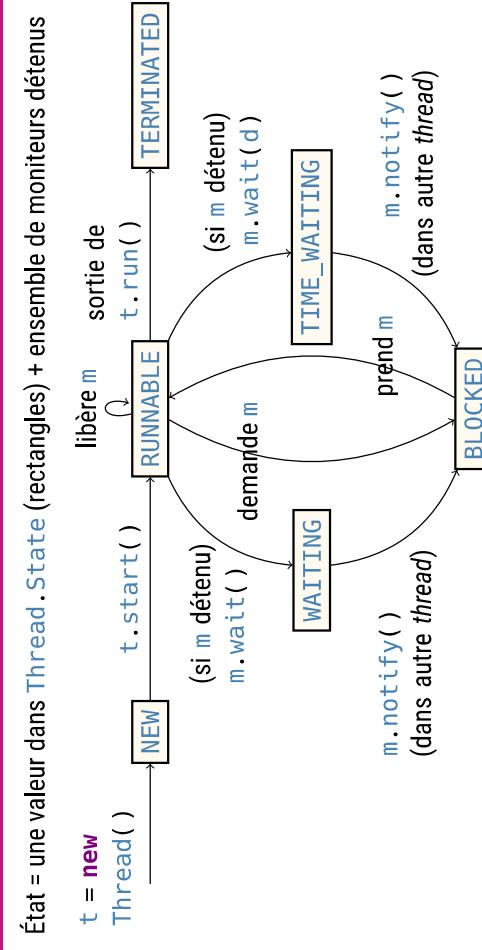
- `while( ! condition() ) Thread.sleep(temps);`  
→ utile quand on sait qu'aucun thread ne notifiera quand la condition sera vraie.
  - `while( ! condition() ) Thread.onSpinWait(); (Java ≥ 9) : attente active`  
(c'est-à-dire : ni blocage ni attente, le thread reste **RUNNABLE**).  
→ on évite le coût de la mise en attente et du réveil, cette approche est donc conseillée quand on s'attend à ce que la condition soit vraie très vite.
- Déconseillé<sup>1</sup> :** `while(!condition(obj)) *rien*; ; attente active « bête »`
- c'est l'ancienne façon de faire, remplacée avantagusement par la variante avec `onSpinWait`. En effet, `onSpinWait` signale à l'ordonnanceur que le *thread* peut être mis en pause (laisser sa place sur le processeur) prioritaiement en cas de besoin.

1. Sauf pour faire cuire une omlette sur son microprocesseur...

- moniteurs = principe d'exclusion mutuelle + mécanisme d'attente/notification;
- mais il existe d'autres façons de synchroniser des threads par rapport à l'usage d'une ressource (exemple : lecteurs/rédacteur);
- fonctionnalités possibles : savoir à qui appartient le verrou, qui est en attente, etc. ;
- bibliothèque de verrous divers dans `java.util.locks`, implémentant l'interface `java.util.concurrent.locks.Lock`.

L'interface `java.util.concurrent.locks.Lock` :

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    Condition newCondition();
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
    void unlock();
}
```



En réalité, racourci directement vers RUNNABLE plutôt que BLOCKED quand le moniteur est déjà disponible.

## Dangers de la synchronisation

quand elle est utilisée à mauvais escient ou à l'excès

### Verrous explicites (2)

Compléments  
en 200

Aldric Degorre

Comme le verrouillage et le déverrouillage se font par appels explicites aux méthodes lock et un lock, ces verrous sont appelés **verrous explicites**.

**Inconvénient :** l'occupation du verrou n'est pas délimitée par un bloc lexical tel que **synchronized { ... }**.

La logique du programme doit assurer que toute exécution de **lock** soit suivie d'une exécution de **unlock**.

**Avantages :**

- Nombreuses options de configuration.
- Flexibilité dans l'ordre d'accquisition et de libération. 2

1. Mais on peut programmer un tel bloc à la main à l'aide d'une fonction d'ordre supérieur, et encapsuler un tel verrou dans une classe dont l'interface ne permettrait d'accéder que via cette FOS.
2. Concurrent Programming in Java (2.5.1.4) montre un exemple de liste chaînée concurrente où, lors d'un parcours, il est nécessaire d'exécuter une chaîne d'acquisitions/libérations croisées de la forme :  
`m1.lock(); ... ; m2.lock(); m1.unlock(); ... ; m3.lock(); m2.unlock(); ... ; m4.lock(); m3.unlock(); ... ; m5.lock(); m4.unlock(); ... ;`

**Un dernier avertissement :** la synchronisation doit rester raisonnable!

En général, plus il y a de synchronisation, moins il y a de parallélisme... et plus le programme est ralenti. Pire, il peut bloquer.

**Pathologies typiques :**

- **dead-lock** : 2 threads attendent chacun une ressources que seul l'autre serait à même de libérer (en fait 2 ou plus : dès lors que la dépendance est cyclique).
- **famine (starvation)** : une ressource est réservée trop souvent/trop longtemps toujours par la même tâche, empêchant les autres de progresser.
- **live-lock** : boucle infinie causée par plusieurs threads se faisant réagir mutuellement, sans pour autant faire avancer le programme. 1

1. S'imaginer deux individus essayant de se croiser dans un couloir, entamant simultanément une manœuvre d'évitement du même côté, mettant les deux personnes à nouveau l'une face à l'autre, provoquant une nouvelle manœuvre d'évitement, et ainsi de suite...

## Dangers de la synchronisation

quand elle est utilisée à mauvais escient ou à l'excès

Compléments  
en 200

Aldric Degorre

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrence  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Empêcher le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Résumé

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrence  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Empêcher le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions

### Éviter les dead-locks

À l'aide d'une utilisation raisonnée des verrous

Compléments  
en 200

Aldric Degorre

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrence  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Empêcher le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrence  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Empêcher le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions

**Exemple de dead lock**

```
class SynchronizedObject {
    public synchronized void use() {
        ...
    }
}

public class DeadLock extends Thread {
    private final SynchronizedObject obj1, obj2;
    private DeadLock(SynchronizedObject obj1, SynchronizedObject obj2) {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }
}

@Override public void run() {
    obj1.useWith(obj2);
    System.out.println("Thread.currentThread() + "claims_monitor_on_ + this");
}

public static void main(String args[]) {
    SynchronizedObject o1 = new SynchronizedObject();
    o2 = new SynchronizedObject();
    / dead lock, sauf si le 1er thread arrive à terminer avant que le 2e ne commence
    new DeadLock(o1, o2).start();
    new DeadLock(o2, o1).start();
}
```

## Dangers de la synchronisation

quand elle est utilisée à mauvais escient ou à l'excès

Compléments  
en 200

Aldric Degorre

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrence  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Empêcher le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Résumé

Introduction  
Généralités  
Style  
Objets et classes  
Types et polymorphisme  
Héritage  
Généricité  
Concurrence  
Introduction  
Threads en Java  
Introduction  
La classe Thread  
Synchronisation  
Empêcher le JMM  
API de haut niveau  
Interfaces graphiques  
Gestion des erreurs et exceptions

**Dangers de la synchronisation**

Exemple de dead lock

Principe pour éviter les **dead-locks** : **toujours acquérir les verrous dans le même ordre** 1 et les libérer dans l'ordre inverse 2 (ordre LIFO, donc).

En effet : dans l'exemple précédent, une exécution de **run** veut acquérir **o1** puis **o2**, alors que l'autre exécution veut faire dans l'autre sens.

→ quand on écrit un programme concurrent, il faut documenter un ordre unique pour prendre les verrous.

L'autre voie est de se reposer sur des abstractions de plus haut niveau, sur lesquelles il est plus aisés de raisonner (cf. la suite).

1. Pas évident en pratique : verrous créés dynamiquement, difficile de savoir quels verrous existeront à l'exécution. On peut aussi ne pas savoir quels verrous donner une méthode donnée d'une classe tierce utilise.

2. Pour les verrous intrinsèques, ordre inverse imposé par l'imbrication des blocs **synchronized**. Mais rien de tel pour les verrous explicites. La preuve de l'absence de **dead-lock** doit alors se faire au cas par cas.

Paradigme des threads	
Introduction	Consistance de la mémoire
<p>Généralités</p> <ul style="list-style-type: none"> <li>Rappel : <i>thread</i> = abstraction</li> <li>simulant la séquentialité dans le <i>thread</i>;</li> <li>permettant une communication instantanée inter-thread via une <u>mémoire partagée</u>;</li> <li>(et simulant le parallélisme parfait<sup>1</sup> entre <i>threads</i>).</li> </ul>	
<p>Style</p> <ul style="list-style-type: none"> <li>Est-ce vraiment la réalité ?</li> <li>→ Divulgârage : NON !</li> </ul>	
<p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrence</p> <p>Interactions</p> <p>Threads en Java</p> <p>Domicile le JMM</p> <p>API de l'outil Java</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>	<p>En réalité, <u>paradigme idéal trop contraignant</u>, empêchant les <u>optimisations matérielles</u>.</p> <p>Modèle d'exécution réellement implémenté par la JVM : le <b>JMM</b><sup>2</sup>.</p> <p>Seule garantie : <u>sous condition</u>, ce qu'on observe est indistinguable du paradigme idéal.</p> <p>1. Hors synchronisation, évidemment.</p> <p>2. Java Memory Model</p>
Analyse	1. Mémoire locale propre au cœur, plus proche physiquement et plus rapide que la mémoire centrale.

Modèle de mémoire Java et optimisations	
Introduction	Consistance de la mémoire
<p>Généralités</p> <ul style="list-style-type: none"> <li>Réalité physique : chaque cœur de CPU dispose de son propre cache<sup>1</sup> de mémoire.</li> <li>Interprétation : Chaque <i>thread</i> utilise potentiellement un cache de mémoire différent. Ainsi, les données partagées existent en de multiples copies pas forcément à jour. (on parle de problèmes de <u>visibilité</u> des changements et de <u>cohérence</u> de la mémoire)</li> </ul>	
<p>Style</p> <ul style="list-style-type: none"> <li>Solution naïve : répercuter immédiatement les changements dans tous les caches immédiatement.</li> </ul>	
<p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrence</p> <p>Interactions</p> <p>Threads en Java</p> <p>Domicile le JMM</p> <p>API de l'outil Java</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>	<p>Problème : cette opération est coûteuse et supprime le bénéfice du cache.</p> <p>→ <b>le JMM</b> : ne garantit donc pas une cohérence parfaite (mais un minimum quand-même...)</p>
Analyse	1. Mémoire locale propre au cœur, plus proche physiquement et plus rapide que la mémoire centrale.

Modèle de mémoire Java et optimisations	
Introduction	Consistance de la mémoire
<p>Généralités</p> <ul style="list-style-type: none"> <li>Réalité physique : Les CPU sont dotés de mécanismes permettant de <u>réordonner des instructions</u><sup>1</sup> qu'il sait devoir exécuter (afin de mieux occuper tous ses composants).</li> <li>Interprétation : l'ordre du programme n'est pas toujours respecté (même sur 1 thread). En plus, optimisations différentes d'une architecture matérielle à une autre (p. ex : comportement différent entre x86 et ARM) → ordre peu prévisible.</li> </ul>	
<p>Style</p> <ul style="list-style-type: none"> <li>Solution naïve : ajouter des barrières<sup>2</sup> partout dans le code compilé.</li> </ul>	
<p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrence</p> <p>Interactions</p> <p>Threads en Java</p> <p>Domicile le JMM</p> <p>API de l'outil Java</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>	<p>Problème : ne garantit pas le respect exact de l'ordre du programme... mais promet que certaines choses importantes restent bien ordonnées.</p> <p>→ <b>le JMM</b> : peut afficher « <b>Bug !</b> ».</p>
Analyse	L'appel <b>test(100)</b> peut afficher « <b>Bug !</b> ».

Modèle de mémoire Java et optimisations	
Introduction	Consistance de la mémoire
<p>Généralités</p> <ul style="list-style-type: none"> <li>Réalité physique : Les CPU sont dotés de mécanismes permettant de <u>réordonner des instructions</u><sup>1</sup> qu'il sait devoir exécuter (afin de mieux occuper tous ses composants).</li> <li>Interprétation : l'ordre du programme n'est pas toujours respecté (même sur 1 thread). En plus, optimisations différentes d'une architecture matérielle à une autre (p. ex : comportement différent entre x86 et ARM) → ordre peu prévisible.</li> </ul>	
<p>Style</p> <ul style="list-style-type: none"> <li>Solution naïve : ajouter des barrières<sup>2</sup> partout dans le code compilé.</li> </ul>	
<p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Concurrence</p> <p>Interactions</p> <p>Threads en Java</p> <p>Domicile le JMM</p> <p>API de l'outil Java</p> <p>Interfaces graphiques</p> <p>Gestion des erreurs et exceptions</p>	<p>Problème : si un des <i>threads</i> finit d'exécuter « <b>x = true; y = true;</b> » et un autre reçoit, dans son cache, la modification sur <b>y</b> mais pas sur <b>x</b>.</p> <p>1. Le JMM autorise cette possibilité théorique, mais probablement vous ne verrez jamais ce message !</p>
Analyse	1. <i>out-of-order execution</i>

### Exemple :

- Supposons qu'initialement `x == 0 && y == 0`. On veut exécuter :
    - sur le thread 1 : (1) `a = x`; (2) `y = 1`;
    - et, sur le thread 2 : (3) `b = y`; (4) `x = 2`;
  - (1) arrive-avant (2) et (3) avant (4)
  - à la fin, il semble impossible d'avoir à la fois `a == 2` et `b == 1`.
  - Or c'est pourtant possible !
- En effet, sur chaque *thread* isolé, inverser les 2 instructions ne change pas le résultat. Comme il n'y a pas de synchronisation, rien n'interdit donc ces inversions.
- Il est donc possible d'exécuter les 4 instructions dans l'ordre suivant :
- $$y = 1; x = 2; a = x; b = y;$$

Exemple

Donc

- mono-thread → aucune différence visible due à ces optimisations;
  - mais multi-thread → différences possibles si synchronisation insuffisante.
- au programmeur de faire en sorte d'avoir une synchronisation suffisante afin que ces optimisations ne soient pas un problème.
- Remarque :** il reste à définir précisément ce qu'on entend par interférence et synchronisation suffisante.

### Ordre arrivé-avant :

- **Ordre partiel** sur les événements d'une exécution, indiquant leur relation de causalité (toute modification causée par ce qui arrive-avant est « vue » par ce qui arrive-après).
- Il est induit par :
  - l'ordre d'exécution des instructions sur un même *thread* tel que demandé par la logique du programme (**ordre du programme**) ;
    - les synchronisations (le réveil d'un *thread* arrive-après l'événement qui l'a réveillé) ;
    - et la causalité entre la lecture d'une variable **volatile** ou **final** et la dernière écriture de celle-ci avant cette lecture.

- **Ordre d'exécution** : ordre chronologique réel d'exécution des instructions.
- Dans une exécution correcte, on voudrait que cet ordre respecte « notre » logique.

1. Par opposition aux instructions d'un programme donné.

### Pour une exécution donnée :

- **Ordre d'exécution** = réalité objective, non interprétée, de celle-ci.
- Celui-ci est difficile à prévoir, dépendant des optimisations opérées par le CPU.
- Il ne respecte pas forcément l'ordre du programme, et donc a fortiori, pas non plus l'ordre arrivé-avant d'une exécution donnée.
- De très nombreux ordres d'exécution sont possibles pour un même programme.
- **Ordre arrivé-avant** = interprétation idéale de la réalité (qui considère la logique du programme et des synchronisations).
- Il est défini sans ambiguïté et facile à déduire à partir d'un code source et d'une trace d'exécution (p. ex. : depuis un ordre d'exécution).

On pouvera qu'un programme est correct en raisonnant sur les ordres arrivé-avant de certaines exécutions particulières : les **exécutions séquentiellement cohérentes** [1].

1. À suivre!

## Accès en compétition

### Variable partagée : variable accessible par plusieurs threads.

Tout attribut est (à moins de prouver le contraire) une variable partagée. Les autres variables (locales ou paramètres de méthodes) ne sont jamais partagées.<sup>1</sup>

**Accès conflictuels** : dans une exécution, 2 accès à une même variable sont conflictuels si au moins l'un des deux est en écriture.

**Accès en compétition (data race)** : 2 accès conflictuels à une variable partagée, tels que l'un n'arrive-pas avant l'autre<sup>2</sup>.

1. Mais les attributs de l'objets référencé peuvent être partagés!
2. C'est à dire : 2 accès qui ne sont pas reliés par une chaîne de synchronisations et d'ordres imposés par l'ordre des instructions du programme.

## Accès en compétition

Programme avec accès en compétition :	Programme sans accès en compétition :
<pre>class Boite {     int x; }  public class Compétition {     public static void main(String args[]) {         Boite b = new Boite();         new Thread() -&gt; { b.x = 1; }.start();         new Thread() -&gt; {             System.out.println(b.x);         }.start();     } }</pre>	<pre>class BoiteSynchro {     private int x;     public synchronized int getX() {         return x;     }     public synchronized void setX(int x) {         this.x = x;     } }  public class PasCompétition {     public static void main(String args[]) {         BoiteSynchro b = new BoiteSynchro();         new Thread() -&gt; {             b.setX(1);         }).start();         new Thread() -&gt; {             System.out.println(b.getX());         }).start();     } }</pre>

Ici, rien n'impose que la lecture de `b.x` arrive avant son affectation ou bien le contraire.

### Exemple

## Modèle d'exécution

La vérité, enfin !

L'ordre d'exécution exact d'un programme est imprévisible, mais ce qui suit est garanti :
<u>Si le programme est correctement synchronisé<sup>1</sup>,</u>
<u>alors son exécution est indiscernable d'une exécution séquentiellement cohérente.</u>
<b>Concrètement</b> , pour que les optimisations ne provoquent pas d'incohérences, il « suffit » donc qu'il n'y ait <b>pas de compétition</b> .

Remarque : le plus dur reste de trouver quels accès sont en compétition...

Heureusement : d'après la propriété ci-dessus, **il suffit de vérifier exécutions « normales »** seulement pour prouver qu'aucune exécution ne se comporte de façon visiblement anormale.

1. Note : si la synchronisation est incorrecte, cela ne veut pas dire qu'on ne sait rien. En fait, la spécification donne tout un ensemble de règles basées sur un critère de causalité... règles trop compliquées et donnant des garanties trop faibles pour être raisonnablement utilisables en pratique.

## Programme correctement synchronisé

La vérité, enfin !

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Introduction	Threads en Java	Demande le JMM	APIs de haut niveau	Interfaces graphiques	Gestion des erreurs et exceptions	Exemple		
Compléments en 200	Aldric Degorre	Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Introduction	Threads en Java	Demande le JMM	APIs de haut niveau	Interfaces graphiques	Gestion des erreurs et exceptions	Résumé

### Exécution séquentiellement cohérente : exécution

- qui suit un ordre total,
- respectant l'ordre du programme,
- et telle que pour toute lecture d'un emplacement mémoire, la dernière écriture, dans le passé, sur cet emplacement est prise en compte.

⇒ Une exécution séquentiellement cohérente est donc une exécution idéale, intuitive, du programme, non affectée par les réordonnements et incohérences de cache.

Exemple : si `x = 0, x = 1 et printLn(x)` s'exécutent dans cet ordre et qu'entre `x = 1 et printLn(x)` il n'y a pas d'affectation à `x`, alors c'est bien **1** qui s'affiche.

**Programme correctement synchronisé** : se dit d'un programme dont toute exécution séquentiellement cohérente est sans accès en compétition.

## Éviter les problèmes, version courte

Compléments en '00 Aldric Degorre	Supprimer les compétitions
Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Concurrence Introduction Threads en Java Dépêche le JMM API de haut niveau Interfaces graphiques Gestion des erreurs et exceptions Résumé	Comment éviter les compétitions ? <ul style="list-style-type: none"><li>• éviter de partager les variables quand ce n'est pas nécessaire → préférer les variables locales (jamais partagées) aux attributs;</li><li>• quand ça suffit, privilégier les données partagées en lecture seule → privilégier les structures immuables (voir ci-après);</li><li>• sinon, renforcer la relation arrivé-avant :<ul style="list-style-type: none"><li>• utiliser les mécanismes de synchronisation déjà présentés,</li><li>• marquer des attributs comme <b>final</b> ou <b>volatile</b> (voir ci-après);</li><li>• utiliser des classes déjà écrites et garanties « <i>thread-safe</i> ».</li></ul></li><li>• Souvent, rien de tout ça ne convient : on peut avoir besoin d'attributs modifiables sans synchronisation ! Mais il faudra s'assurer qu'un seul <i>thread</i> peut y accéder.</li></ul>
Compléments en '00 Aldric Degorre	Les mot-clés <b>volatile</b> et <b>final</b> (2)
Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Concurrence Introduction Threads en Java Dépêche le JMM API de haut niveau Interfaces graphiques Gestion des erreurs et exceptions Discussion	<p><b>Téchnique infaillible :</b> tous les attributs <b>volatile</b> (ou <b>final</b>) ⇒ accès en compétition impossible. Cependant pas idéale car :</p> <ul style="list-style-type: none"><li>• <b>non réaliste</b> : un programme utilise des classes faites par d'autres personnes;</li><li>• <b>non efficace</b> : <b>volatile</b> empêche les optimisations ⇒ exécution plus lente.<sup>2</sup></li></ul> <p>En plus, <b>volatile</b> ne permet pas de rendre les méthodes atomiques ⇒ entrelacements toujours non maîtrisés ⇒ <b>volatile</b> ne suffit pas toujours pour tout.</p> <p><b>Exemple :</b> avec <b>volatile int x = 0</b>; si on exécute 2 fois en parallèle <b>x++</b>, on peut toujours obtenir 1 au lieu de 2.</p> <hr/> <p>1. Mais on peut encapsuler leurs instances dans des classes à méthodes synchronisées... au prix d'encore un peu moins de performance. 2. Remarque : pour <b>final</b>, la question ne se pose qu'au début de la vie de l'objet. À ce stade, accéder à une ancienne version de l'attribut n'aurait aucun sens. L'optimisation serait nécessairement fausse.</p>

## Les objets immuables

# Les objets immuables

Compléments en 200  
Aldric Degorre

Compléments en 200  
Aldric Degorre

- **Rappel : immutable** = non modifiable. Le terme s'applique aux objets et, par extension, aux classes dont les instances sont des objets immuables.
- Ces objets ont généralement des champs tous **final**. Conséquence : relation « arrivé-avant » entre l'initialisation de l'objet et tout accès ultérieur.
- Pendant la vie de l'objet : pas d'accès en écriture ⇒ pas d'accès en compétition.  
⇒ non seulement l'utilisation qui en est faite dans un *thread* n'influe pas sur l'utilisation dans un autre *thread*<sup>1</sup>, mais en plus il ne peut pas y avoir d'incohérence de cache par rapport au contenu d'un objet immuable.<sup>2</sup>

Remarque : tout cela reste vrai quand on parle des champs **final** d'un objet quelconque.

1. Donc tout objet immuable est *thread-safe*.
2. Si l'objet immuable est correctement publié, tous les *threads* sont d'accord sur l'ensemble des valeurs publiées.

## Variables atomiques

# Autres objets *thread-safe*

Compléments en 200  
Aldric Degorre

Compléments en 200  
Aldric Degorre

`java.util.concurrent.atomic` propose un certain nombre de classes de **variables atomiques** (classes mutables *thread-safe*).

- Exemples : `AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, ...
- Leurs instances représentent des booléens, des entiers, des tableaux d'entiers, ...
- Accès simples : comportement similaire aux variables volatiles.
- Disposent, en plus, d'opérations plus complexes et malgré tout atomiques (typiquement : incrémentation).<sup>1</sup>

1. L'accès atomique est garanti sans synchronisation, grâce à des appels à des instructions dédiées des processeurs, telles que CAS (compare-and-set). Ainsi ces classes ne sont en réalité pas implémentées en Java, car elles sont compilées en tant que code spécifique à l'architecture physique (celle sur laquelle tourne la JVM).

- Typiquement, une étape de calcul consiste à créer un nouvel objet immuable à partir d'objets immuables existants (puisque l'on ne peut pas les modifier).
- Un tel calcul peut être réalisé à l'aide d'une **fonction pure**<sup>1</sup>
- **Inconvénient** : implique d'allouer un nouvel objet pour chaque étape de calcul.
- Le résultat doit être correctement publié pour être utilisable par un autre *thread* :
  - grâce aux mécanismes (méthodes) de passage de message prévus par l'API utilisée,
  - ou bien « à la main », en l'enregistrant dans une variable partagée (soit **volatile**, soit **private** avec accesseurs **synchronized**), modifiable.
- **Exemple** : `SharedResources.setX(f(SharedResources.getX()));` ; (où `getX` et `setX` sont **synchronized** et `f` est une fonction pure).

1. Fonction sans effet de bord, notamment sans modification d'état persistente.

2. Notamment si l'escape analysis détermine que l'objet n'est que d'usage local → la JVM l'alloue en pile.

Cela dit, ceci ne concerne que les calculs intermédiaires car la variable partagée est stockée dans le tas.

Compléments en 200  
Aldric Degorre

Compléments en 200  
Aldric Degorre

## Les nombreux inconvénients de l'API threads

APIs pour la concurrence	
Compléments en POO Aldric Degorre	Schéma de base, pour limiter le nombre de threads

Utiliser directement les *threads* et les moniteurs → nombreux inconvénients :

- Chaque *thread* utilise beaucoup de mémoire. Et les instancier prend du temps.
- Trop de *threads* → changements de contexte fréquents (opération coûteuse).
- Nécessité de communiquer par variables partagées → risque d'accès en compétition (et donc d'incohérences)
- En cas de synchronisation mal faite, risque de blocage.

**Heureusement :** de nombreuses API de haut niveau<sup>1</sup> aident à contourner ces écueils.  
→ on travaillera plutôt avec celles-ci que directement avec les *threads* et les moniteurs.

1. programmées par-dessus les *threads* et les moniteurs

APIs pour la concurrence	
Compléments en POO Aldric Degorre	Schéma de base, pour limiter le nombre de threads

**Idée :** réutiliser un même *thread* pour plusieurs exécuter plusieurs tâches tour à tour.

**Exécuteur :** objet qui gère un certain ensemble de *threads*

- en distribuant des **tâches** sur ceux-ci, selon politique définie;
- en évitant de créer plus de *threads* que nécessaire<sup>1</sup>;
- et en évitant de détruire un *thread* aussitôt qu'il est libre (pour éviter d'en re créer).

**Tâche :**

- séquence d'instructions (en pratique : une fonction) à exécuter sur un *thread*
  - ne peut pas être mise en pause pour libérer le *thread* au profit d'une autre.<sup>2,3</sup>
1. Selon politique de l'exécuteur. Plusieurs possibles. Par exemple : nb. max. *threads* ≤ nb. cœur(s).
  2. Le *thread*, lui, peut être mis en pause par le noyau pour libérer un processeur au profit d'un autre *thread*.
  3. En principe, car avec `ForkJoinPool`, notamment, certains appels de méthodes permettent la mise en pause → multi-tâche coopératif.

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Introduction	Threads en Java	Dompson le JMM	APIs de haut niveau	Interfaces graphiques	Gestion des erreurs et exceptions
Compléments en POO Aldric Degorre								Introduction	Threads en Java	Dompson le JMM	APIs de haut niveau		

## APIs pour la concurrence

APIs pour la concurrence	
Compléments en POO Aldric Degorre	Differents styles d'APIs pour les tâches

Pour synchroniser et faire communiquer des tâches interdépendantes, 2 styles d'API : (dans les 2 cas, **passage de messages** plutôt que variables partagées)

- **API bloquante** : appel de méthode bloquante pour attendre la fin d'une autre tâche (comme `join` pour les *threads*) et obtenir son résultat (si applicable).
- **Exemple :**

```
ForkJoinTask<Integer> f = ForkJoinTask.adapt(() -> scanner.nextInt());
ForkJoinTask.adapt(() -> {
    f.fork(); // lancement d'une sous-tâche
    System.out.println("join()"); // appel bloquant avec récupération du résultat
}).fork(); // lancement de la tâche principale
```

**Dans le JDK :** `Thread`, `Future` et `ForkJoinTask` suivent ce principe.<sup>1</sup>

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Introduction	Threads en Java	Dompson le JMM	APIs de haut niveau	Interfaces graphiques	Gestion des erreurs et exceptions
Compléments en POO Aldric Degorre								Introduction	Threads en Java	Dompson le JMM	APIs de haut niveau		

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Introduction	Threads en Java	Dompson le JMM	APIs de haut niveau	Interfaces graphiques	Gestion des erreurs et exceptions
Compléments en POO Aldric Degorre								Introduction	Threads en Java	Dompson le JMM	APIs de haut niveau		

- **API non bloquante** : une tâche qui dépend d'un résultat fourni par une autre est passée en tant que **fondction de rappel** (`callback`)<sup>1</sup>. Cette dernière sera déclenchée par l'arrivée du résultat attendu (plus généralement : un évènement). **Exemple :**

```
CompletableFuture
// tâche 1 : d'abord programme la lecture d'un Scanner
.supplyAsync(scanner::nextInt)
// tâche 2 : dès qu'un entier est fourni, affiche-le
.thenAccept(System.out::println)
```

**Dans le JDK :** `Swing`, `CompletableFuture`, `Stream`, `Flow`.<sup>2</sup>

1. Sur le principe, une fonction de première classe → en Java traditionnel un objet avec une méthode dédiée, en Java moderne, une lambda-expression.
  2. Hors JDK : Akka (implémentation des « acteurs »), diverses implémentations de la spécification Reactive Streams (autres que `Flow`), JavaFX (en effet, JavaFX n'est plus dans le JDK depuis Java 11), ...
1. Hors JDK, citons le principe des « fibres » dans la bibliothèque Quasar (qui implémente aussi les « acteurs » en API bloquante).

## Interfaces Executor et Runnable

### Interfaces ExecutorService, Callable<V> et Future<V>

Compléments en 200	Aldric Degorre	<h4>Interfaces Executor et Runnable</h4>
Introduction		<ul style="list-style-type: none"> <li>En Java, les exécuteurs sont les instances de l'interface <code>Executor</code> :</li> </ul>
Généralités		<pre>public interface Executor { void execute(Runnable command); }</pre>
Style		<p>L'appel <code>unExecutor.execute(unRunnable)</code> exécute la méthode <code>run()</code> de <code>unRunnable</code>. Ainsi, dans ce cas, les tâches sont des instances de <code>Runnable</code>.</p>

Compléments en 200	Aldric Degorre	<h4>Interfaces ExecutorService, Callable&lt;V&gt; et Future&lt;V&gt;</h4>
Introduction		<ul style="list-style-type: none"> <li><code>ExecutorService</code> étend <code>Executor</code> en y ajoutant :           <ul style="list-style-type: none"> <li><code>&lt;T&gt; Future&lt;T&gt; submit(Callable&lt;T&gt; task)</code> : programme une tâche.</li> <li>Des méthodes pour demander et/ou attendre la terminaison des tâches en cours.</li> </ul> </li> </ul>
Généralités		<ul style="list-style-type: none"> <li><code>Callable&lt;V&gt;</code> est comme <code>Runnable</code>, mais sa méthode retourne un résultat :</li> </ul>
Style		<pre>public interface Callable&lt;V&gt; { V call(); }</pre>

Compléments en 200	Aldric Degorre	<h4>Interfaces ExecutorService, Callable&lt;V&gt; et Future&lt;V&gt;</h4>
Introduction		<ul style="list-style-type: none"> <li><code>Future&lt;V&gt;</code> = objets pour accéder à un résultat de type <code>V</code> promis dans le futur :</li> </ul>
Généralités		<pre>public interface Future&lt;V&gt; {     public boolean cancel(boolean mayInterruptIfRunning);     V get() throws InterruptedException, ExecutionException;     V get(long timeout, TimeUnit unit) throws TimeoutException;     ExecutionException getException();     boolean isCancelled();     boolean isDone(); }</pre>
Style		<p>Les méthodes <code>get</code> sont bloquantes jusqu'à disponibilité du résultat.</p>

Compléments en 200	Aldric Degorre	<h4>Interfaces ExecutorService, Callable&lt;V&gt; et Future&lt;V&gt;</h4>
Introduction		<h4>Interfaces ExecutorService, Callable&lt;V&gt; et Future&lt;V&gt;</h4>
Généralités		<p>Exemple plus complet</p>
Style		<pre>public class TestCall implements Callable&lt;Integer&gt; {     private final int x;     public TestCall(int x) { this.x = x; }     @Override public Integer call() { return x; } }  public class Exemple {     public static void main(String[] args){         ExecutorService executor = Executors.newSingleThreadExecutor();         Future&lt;String&gt; futur1 = executor.submit(new TestCall(1));         Future&lt;String&gt; futur2 = executor.submit(new TestCall(2));         try { System.out.println(futur1.get() + futur2.get()); } // affiche "3"         catch(ExecutionException e) { ... }         finally { executor.shutdown(); }     } }</pre>

Compléments en 200	Aldric Degorre	<h4>Interfaces ExecutorService, Callable&lt;V&gt; et Future&lt;V&gt;</h4>
Introduction		<h4>Usage (schéma général)</h4>
Généralités		
Style		

Compléments en 200	Aldric Degorre	<h4>Interfaces ExecutorService, Callable&lt;V&gt; et Future&lt;V&gt;</h4>
Introduction		<h4>Usage (schéma général)</h4>
Généralités		
Style		

Types et polymorphisme		<p>... Callable&lt;String&gt; task = ...; // task sera exécuté dans thread choisi par es : Future&lt;String&gt; result = es.submit(task); ... // le programme attend puis affiche le résultat : System.out.println("Résultat : " + result.get());</p>
Héritage		
Généralité		

Ici, l'exécuteur exécute les 2 tâches l'une après l'autre (un seul *thread* utilisé), mais en même temps que la méthode `main()` continue à s'exécuter.

Cette dernière finit par attendre les résultats des 2 tâches pour les additionner.

## Instancier des exéuteurs

# Les limites de ThreadPoolExecutor (et des fabricues newCachedThreadPool() et newFixedThreadPool())

Compléments en POO  
Aldric Degorre

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Introduction	Threads en Java	Demande le JMM	API de haut niveau	Interfaces graphiques	Gestion des erreurs et exceptions	Discussion
<p><b>À l'aide de la classe Executor :</b></p> <ul style="list-style-type: none"><li>= bibliothèque de <u>fabriques statiques</u> d'ExecutorService.</li><li><b>static ExecutorService newSingleThreadExecutor()</b> : crée un ExecutorService utilisant un worker thread unique.</li><li><b>static ExecutorService newCachedThreadPool()</b> : crée un exéuteur dont les threads du pool se créent, à la demande, sans limite, mais sont réutilisés s'ils redeviennent disponibles.</li><li><b>static ExecutorService newFixedThreadPool(<int :<="" b="" nthreads)=""> même chose, mais avec limite fixée à <b>n threads</b><sup>1</sup>.</int></b></li></ul> <p>On peut aussi utiliser directement les constructeurs de <b>ThreadPoolExecutor</b> ou de <b>ScheduledThreadPoolExecutor</b><sup>2</sup> (nombreuses options).</p> <ol style="list-style-type: none"><li>Choisir <b>n</b> en rapport avec le nombre d'unités de calcul/cœurs.</li><li>Implémenter <b>ScheduledExecutorService</b>, permettant de programmer des tâches périodiques et/ou différées. Les futurs retournés implémentent <b>Schedule</b> ledFuture&lt;V&gt;. Regardez la documentation.</li></ol>	Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Threads en Java	Demande le JMM	API de haut niveau	Interfaces graphiques	Gestion des erreurs et exceptions	Discussion

<ul style="list-style-type: none"><li>But d'un <b>thread pool</b> = réduire le nombre de <b>threads</b> → petit nombre de <b>threads</b>.</li><li>Or les <b>threads</b> bloqués (par appel bloquant, comme <b>f.get()</b>, au sein de la tâche) ne sont pas réattribuables à une autre tâche<sup>1</sup>. → moins de <b>threads</b> disponibles dans le <b>pool</b> → <u>ralentissement</u>.</li><li><b>Cas extrême</b> : si grand nombre de tâches concurrentes avec interdépendances, il arrive que tout le <b>pool</b> soit bloqué par des tâches en attente de tâches bloquées ou non démarrees → rien ne viendra débloquer la situation. Cette situation s'appelle un <b>thread starvation deadlock</b><sup>2</sup>.</li></ul>	<p><b>Comment concilier pool/de taille bornée et garantie d'absence de blocage ?</b></p> <hr/> <ol style="list-style-type: none"><li>Il est impossible de « sortir » une tâche déjà en exécution sur un <b>thread</b> pour le libérer.</li><li>Du point de vue des <b>threads</b>, c'est bien un <b>deadlock</b> : dépendance cyclique entre <b>threads</b>. Du point de vue des tâches, dépendance pas forcément cyclique, mais blocage car multi-tâche non-préemptif s'exécutant sur un nombre limité d'unités d'exécution.</li></ol>
---	---

## Work-stealing strategy

# Work-stealing strategy

Pourquoi ça marche

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Concurrence	Introduction	Threads en Java	Demande le JMM	API de haut niveau	Interfaces graphiques	Gestion des erreurs et exceptions	Description
<p><b>Solution</b> : stratégie de vol de travail (<b>work stealing</b>). Principe :</p> <ul style="list-style-type: none"><li>une file d'attente de tâches par <b>thread</b> au lieu d'une commune à tout le <b>pool</b> ;</li><li>tâches générées par une autre tâche → ajoutées sur file du même <b>thread</b> ;</li><li>quand un <b>thread</b> veut du travail, il prend une tâche en priorité dans sa file, sinon il en « vole » une dans celle d'un autre <b>thread</b> ;</li><li><b>le plus important</b> : si le résultat attendu n'est pas disponible, <b>get</b> (et <b>join</b>) exécute d'abord les tâches en file au lieu de bloquer le <b>thread</b> tout de suite. ⇒ C'est là que se met en place la coopération entre tâches.</li></ul>	Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Threads en Java	Demande le JMM	API de haut niveau	Interfaces graphiques	Gestion des erreurs et exceptions	Description

## ForkJoinPool et ForkJoinTask

L'implémentation de la *work-stealing* strategy en Java

## Work-stealing strategy

Indications et contre-indications

Compléments en POO	Aldric Degorre
Introduction	
Généralités	
Style	<p>• Petites tâches à dépendances acycliques (notamment, algorithmes récursifs)</p> <p>→ stratégie très intéressante (pas de <i>thread starvation deadlock</i>)</p> <p>• Tâches sans dépendances</p> <p>→ stratégie inutile et plus lourde que la stratégie « naïve »<sup>1</sup>.</p> <p>• Tâches à dépendances cycliques</p> <p>→ <i>deadlocks</i> assurés (mais aucune stratégie ne peut fonctionner dans ce cas!).</p>
Discussion	<p>1. Sans compter qu'en Java, l'implémentation de celle-ci (<code>ThreadPoolExecutor</code>) est plus configurable que l'implémentation de la <i>work-stealing</i> strategy (<code>ForkJoinPool</code>).</p>

## ForkJoinPool et ForkJoinTask

L'implémentation de la *work-stealing* strategy en Java

Compléments en POO	Aldric Degorre
Introduction	
Généralités	
Style	<p>Ainsi Java propose :</p> <ul style="list-style-type: none"><li>la classe <code>ForkJoinPool</code> : implémentation d'<code>Executor</code> utilisant cette stratégie</li><li>la classe <code>ForkJoinTask</code> : tâches capables de générer des sous-tâches (<code>&lt; fork()</code>) et d'attendre leurs résultats pour les utiliser (<code>&lt; join()</code>).</li></ul>
Concurrence	<p><code>ForkJoinPool</code> est considéré suffisamment efficace pour que le <i>thread pool</i> par défaut (utilisé implicitement par plusieurs API concurrentes) soit une instance de cette classe.</p>
Discussion	<p>Obtenir le <i>thread pool</i> par défaut : <code>ForkJoinPool.commonPool()</code>.</p>

## ForkJoinPool et ForkJoinTask

Compléments en POO	Aldric Degorre
Introduction	
Généralités	
Style	<p>• Classe abstraite. Ses objets sont les tâches exécutables par <code>ForkJoinPool</code>.</p> <p>• On préfère étendre une de ses sous classes (abstraites aussi)<sup>1</sup> :</p> <ul style="list-style-type: none"><li>• <code>RecursiveAction</code> : tâche sans résultat (exemple : modifier les feuilles d'un arbre)</li><li>• <code>RecursiveTask&lt;V&gt;</code> : tâche calculant un résultat de type <code>V</code> (exemple : compter les feuilles d'un arbre)</li></ul>
Concurrence	<p>Dans les 2 cas, on étend la classe en définissant la méthode <code>compute()</code> qui décrit les actions effectuées par la tâche :</p> <ul style="list-style-type: none"><li>• pour <code>RecursiveAction</code> : <code>protected void compute();</code></li><li>• pour <code>RecursiveTask&lt;V&gt;</code> : <code>protected V compute();</code></li></ul>
Discussion	<p>• Et <code>ForkJoinTask</code> ?</p> <ul style="list-style-type: none"><li>• 3 fabricres statiques <code>ForkJoinTask&lt;V&gt;</code>, <code>adapt(...)</code><sup>2</sup> permettent de créer des tâches à partir de <code>Runnable</code> et de <code>Callable&lt;V&gt;</code>.</li></ul>
Détails	<p>1. Ces deux classes servent juste à faciliter l'implémentation. 2. Le nom <code>adapt</code> provient du patron <code>adapter</code>.</p>

## ForkJoinPool et ForkJoinTask

Et car rien ne vaut un exemple...

### La tâche récursive :

```
class Fibonacci extends RecursiveTask<Integer> {
    Fibonacci(int n) { this.n = n; }

    @Override protected Integer compute() {
        if (n <= 1) return n;
        Fibonacci f1 = new Fibonacci(n - 1);
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        return f2.compute() + f1.join();
    }
}
```

### Et l'appel initial (dans main()) :

```
System.out.println((new ForkJoinPool()).invoke(new Fibonacci(12)));
```

## Une alternative : ComplettableFuture

Compléments en 200

Aldric Degorre

### Introduction

### Généralités

### Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Introduction

Théâtre en Java

Déroulé le JMM

APIs de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

### Exemple

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Introduction

Théâtre en Java

Déroulé le JMM

APIs de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

Compléments en 200

Aldric Degorre

### Introduction

### Généralités

### Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Introduction

Théâtre en Java

Déroulé le JMM

APIs de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

### Exemple

## Les reactive streams<sup>2</sup> avec Flow

java.util.concurrent.Flow :

- Implémentation standardisée dans le JDK (Java 9) de la spécification reactive streams (2015), issue d'une initiative des sociétés Netflix, Pivotal et Lightbend.
- C'est une API non bloquante, inspirée du patron Observateur/Observé.
- Idée :** 2 sortes d'objets, Publisher et Subscriber. Le premier peut produire une séquence de messages, alors que le second réagit aux données qu'on lui envoie.
- Pour cela, le Subscriber doit d'abord s'abonner à un Publisher (et un seul).
- En cas d'utilisation « normale »<sup>1</sup>, le Subscriber traite les messages reçus séquentiellement (pas de synchronisation à gérer dans ses méthodes).

- Plusieurs Subscriber peuvent s'abonner au même Publisher (« fan out »).
- Un seul abonnement, et Publisher correctement implémenté.  
Cela dit, il est possible de coordonner plusieurs abonnements (« fan in »), mais il faut le faire « à la main » à l'aide de plusieurs instances de Subscriber et un peu de synchronisation.
  - C'est le nom de la spécification, cela n'a pas de rapport avec l'API stream de Java.

## Une alternative : CompletableFuture

Exemple : de ForkJoinTask à CompletableFuture

Compléments en 200

Aldric Degorre

### Introduction

### Généralités

### Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Concurrence

Introduction

Théâtre en Java

Déroulé le JMM

APIs de haut niveau

Interfaces graphiques

Gestion des erreurs et exceptions

### Devient :

```
CompletableFuture<Rf>cff = CompletableFuture.supplyAsync(Boulot::f);
CompletableFuture<Rg>cfg = cfg.thenApplyAsync(Boulot::g);
CompletableFuture<Rh>cfh = cff.thenApply(Boulot::h);
Ri ri = Boulot.(rf, frg.join());
System.out.println("Résultat : " + ri);
System.out.println("Résultat : " + ri);
```

- CompletableFuture<Rf>cff = CompletableFuture.supplyAsync(Boulot::f);
- CompletableFuture<Rg>cfg = cfg.thenApplyAsync(Boulot::g);
- CompletableFuture<Rh>cfh = cff.thenApply(Boulot::h);
- CompletableFuture<Ri>cfl = cfh.thenCombine(cfr, Boulot::i);
- cfl.thenAccept(ri -> { System.out.println("Résultat : " + ri); })

```
public class PrintSubscriber implements Flow.Subscriber<String> {
    private Flow.Subscription subscription;
    @Override public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1); // Je suis prêt à recevoir 1 premier message !
        System.out.println(this + " : je suis inscrit !");
    }
    @Override public void onNext(String item) { // Je suis prêt à recevoir 1 autre message !
        System.out.println(this + " : " + item + " (" + Thread.currentThread().getName() + ")");
    }
    @Override public void onError(Throwable throwable) { System.out.println(this + " : Oups ?"); }
    @Override public void onComplete() { System.out.println(this + " : C'est fini !"); }
}
```

### On connecte les morceaux et on lance :

```
public static void main(String[] args) throws InterruptedException {
    var publisher = new SubmissionPublisher<String>(); // Submission publisher fourni par JDK
    publisher.subscribe(new PrintSubscriber()); // on abonne une instance
    publisher.subscribe(new PrintSubscriber()); // puis une autre
    List.of("Lorem", "ipsum", "dolor", "sit", "amet").forEach(publisher::submit);
    ForkJoinPool.commonPool().awaitTermination(1000, TimeUnit.MILLISECONDS);
}
```

Quelques points restent à expliquer.

### Les abonnements :

- Un abonnement est symbolisé par un objet intermédiaire, instance de `Flow.Subscription`.
- C'est le `Flow.Publisher` qui est chargé d'instancier l'abonnement et de le passer à l'abonné (via méthode `onSubscribe`).
- `Flow.Subscription` est généralement implémentée dans ou avec l'implémentation de `Flow.Publisher`.

### La gestion de la « contre-pressure » (ou `backpressure`) :

- Un abonné ne reçoit pas plus de messages que le nombre qu'il a demandé à recevoir (via appel à `request(n)`).
- Ainsi le `Publisher` doit gérer une file d'attente (messages prêts à être publiés, mais qui n'ont pas encore été envoyés à tous les abonnés).

```
PrintSubscriber@284682b2: Je suis inscrit !
PrintSubscriber@1446b42: Je suis inscrit !
PrintSubscriber@1446b42: Lorem (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: Lorem (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@284682b2: ipsum (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: ipsum (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@1446b42: dolor (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: dolor (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@1446b42: sit (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: sit (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@284682b2: amet (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: amet (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@1446b42: amet (thread ForkJoinPool.commonPool-worker-5)
```

Quelques points restent à expliquer.

### Les abonnements :

- Un abonnement est symbolisé par un objet intermédiaire, instance de `Flow.Subscription`.
- C'est le `Flow.Publisher` qui est chargé d'instancier l'abonnement et de le passer à l'abonné (via méthode `onSubscribe`).
- `Flow.Subscription` est généralement implémentée dans ou avec l'implémentation de `Flow.Publisher`.

### La gestion de la « contre-pressure » (ou `backpressure`) :

- Un abonné ne reçoit pas plus de messages que le nombre qu'il a demandé à recevoir (via appel à `request(n)`).
- Ainsi le `Publisher` doit gérer une file d'attente (messages prêts à être publiés, mais qui n'ont pas encore été envoyés à tous les abonnés).

```
public class PrintSubscriber implements Flow.Subscriber<T> {
    private Flow.Subscription subscription;
    @Override public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1); // non instanciable
    }
    public static interface Publisher<T> {
        public void subscribe(Subscriber<T> subscriber);
    }
    public static interface Subscription {
        public void onSubscribe(Subscriber<T> subscriber);
        public void onDispose();
        public void onError(Throwable throwable);
        public void onComplete();
    }
    public static interface Processor<T,R> extends Subscription {
        public void onProcess(T item);
    }
}
```

### On connecte les morceaux et on lance :

```
public static void main() {
    var publisher = new Processor<String, String>() {
        @Override public void onProcess(String item) {
            System.out.println(item);
        }
    };
    publisher.subscribe(new PrintSubscriber());
    publisher.onDispose(() -> System.out.println("Publisher disposed"));
    publisher.onError((throwable) -> System.out.println("Publisher error: " + throwable));
    publisher.onComplete(() -> System.out.println("Publisher completed"));
    publisher.request(1);
}
```

Quelques points restent à expliquer.

### Les abonnements :

- Un abonnement est symbolisé par un objet intermédiaire, instance de `Flow.Subscription`.
- C'est le `Flow.Publisher` qui est chargé d'instancier l'abonnement et de le passer à l'abonné (via méthode `onSubscribe`).
- `Flow.Subscription` est généralement implémentée dans ou avec l'implémentation de `Flow.Publisher`.

### La gestion de la « contre-pressure » (ou `backpressure`) :

- Un abonné ne reçoit pas plus de messages que le nombre qu'il a demandé à recevoir (via appel à `request(n)`).
- Ainsi le `Publisher` doit gérer une file d'attente (messages prêts à être publiés, mais qui n'ont pas encore été envoyés à tous les abonnés).

## Les reactive streams avec Flow

En pratique

En pratique, le JDK fournit déjà une implémentation de `Publisher<T>` : `SubmissionPublisher<T>` (contenant une implémentation de `Subscription`).

Cette classe :

- utilise un *thread pool* (soit celui passé en paramètre, soit `ForkJoinPool.commonPool()`)
- une méthode **public int submit(T item)** qui permet de fournir à ce *publisher* les prochains messages qu'il va diffuser à ses abonnés.
- quand un message doit être envoyé à un abonné, sa méthode **onNext** est appelée dans une tâche soumise au *thread pool* (donc possibilité de répartition sur plusieurs threads).

Ainsi, pour créer des graphes de `Flow` sans effort, on peut implémenter des `Publisher` et `Processor` basés sur cette classe (qui implémente déjà tout ce qui est un peu compliqué).

En pratique, le JDK fournit déjà une implémentation de `Publisher<T>` : `SubmissionPublisher<T>` (contenant une implémentation de `Subscription`).

Cette classe :

- utilise un *thread pool* (soit celui passé en paramètre, soit `ForkJoinPool.commonPool()`)
- une méthode **public int submit(T item)** qui permet de fournir à ce *publisher* les prochains messages qu'il va diffuser à ses abonnés.
- quand un message doit être envoyé à un abonné, sa méthode **onNext** est appelée dans une tâche soumise au *thread pool* (donc possibilité de répartition sur plusieurs threads).

Ainsi, pour créer des graphes de `Flow` sans effort, on peut implémenter des `Publisher` et `Processor` basés sur cette classe (qui implémente déjà tout ce qui est un peu compliqué).

Exemple de Processor

Exemple de Processor

Exemple de Processor

Ensuite on peut étendre cette classe abstraite pour écrire un `Processor` complet :

```
public class DoublerProcessor extends AbstractProcessor<String, String> {
    Flow.Subscription subscription;
    @Override public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
    }
    @Override public void onNext(String item) {
        subscription.request(1); // requête de nouveau mot à chaque fois, mais on pourrait faire autrement :
        // par exemple, attendre qu'au moins un des abonnés en ait besoin (voire tous les abonnés)
        submit(item + item); // pour chaque mot "mot" reçu, transmet "motmot" à ses abonnés.
    }
    @Override public void onError(Throwable throwable) {
    }
    @Override public void onComplete() {
    }
}
```

Écrire directement `DoublerProcessor` sans écrire par `AbstractProcessor` était possible, mais cette dernière peut être réutilisée pour d'autres concréétisation. En version très courte on aurait pu avoir :

```
public class DoublerProcessor extends SubmissionPublisher<String> implements Flow.Processor<String, String> {
    String str;
    // héritage de SubmissionPublisher, pour monter autre chose que la composition
    // redéfinitions de onSubscribe, onNext, onError, onComplete comme ci-dessus ...
}
```

Si certains programmes peuvent être réalisés différemment avec plusieurs APIs, certaines sont clairement plus adaptées à certains cas d'usage... Par ailleurs, certaines API correspondent mieux à votre façon de penser. Pour vous faire une idée, **expérimentez !**