Definition (Concurrence)

Deux actions, instructions, travaux, tâches, processus, etc. sont **concurrents** si leurs exécutions sont **indépendantes** l'une de l'autre (l'un n'attend pas de résultat de l'autre).

- **Conséquence :** deux actions concurrentes peuvent s'exécuter simultanément, si la plateforme d'exécution le permet.
- Un **programme concurrent** est un programme dont certaines portions de code sont indépendantes les unes des autres et tel que la plateforme d'exécution sait [1] exploiter ce fait pour optimiser l'exécution. [2].

1. Le plus souvent, cette connaissance nécessite que les portions concurrentes soient signalées dans le code source.
2. Notamment en exécutant simultanément, **en parallèle**, ces portions de code si c'est possible.

Concurrence

Où et quand ?

- Naturelle et nécessaire dans des situations variées en programmation [1] :
  - **Serveurs web :** un même serveur doit pouvoir servir de nombreux clients indépendamment les uns des autres sans les faire attendre.
  - **Interfaces homme-machine :** le programme doit pouvoir, tout en prenant en compte, sans délai, les actions de l'utilisateur, continuer à exécuter d'éventuelles tâches de fond, jouer des animations, etc.
  - De manière générale, c'est utile dans tout programme qui doit réagir immédiatement à des évènements de causes et origines variées et indépendantes.
- Possible [2] pour de nombreux algorithmes décomposables en étapes indépendantes.

  Utile de programmer ces algorithmes de façon concurrente car on peut profiter du parallélisme pour les accélérer (cf. page suivante).

1. Et pas seulement en programmation, mais c'est le sujet qui nous intéresse !
2. Et discutablement naturelle aussi.

Parallélisme

Deux travaux [1] s'exécutent **en parallèle**, s'ils exécutent en même temps.

- Simultanéité au niveau le plus bas : si 2 travaux s'exécutent en parallèle, à un instant t, s'exécutent en meme temps une instruction de l'un et de l'autre.
- $\rightarrow$ exécution sur 2 lieux physiques différents (*e.g.* 2 cœurs, 2 circuits, ...).
- **Degré de parallélisme** [2] = nombre de travaux simultanément exécutables.

Pour des raisons économiques et technologiques, les microprocesseurs modernes (multi-cœur [3]) ont typiquement un degré de parallélisme ≥ 2.

C'est une opportunité qu'il faut savoir saisir !

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 (*simultaneous multithreading*), appelé *hyperthreading* chez Intel

Ainsi, l'enjeu de la programmation concurrente est double :

- **Nécessité :** pouvoir programmer des fonctionnalités intrinsèquement concurrentes (serveur web, IG, etc.).
- **Opportunisme :** tirer partie de toute la puissance de calcul du matériel contemporain.
  En effet : des travaux indépendants (concurrents) peuvent naturellement être confiés à des unités d'exécution distinctes (parallèles).

Malheureusement, la programmation concurrente est un art difficile...

Exécuter deux travaux réellement concurrents en parallèle est facile [1], mais la réalité est souvent plus compliquée :

- Si (degré de) concurrence $>$ (degré de) parallélisme, alors **partage du temps** des unités d'exécution ($\rightarrow$ **ordonnanceur** nécessaire).
- Concurrence de 2 sous-programmes jamais parfaite [2] car nécessité de se transmettre/partager des résultats et de se **synchroniser**. [3]

$\rightarrow$ Différentes abstractions pour aider à programmer de façon correcte et, si possible, intuitive, tout en prenant en compte ces réalités de diverses façons.

1. On en affecte un à chaque cœur, pourvu qu'il y ait 2 cœurs disponibles, et on n'en parle plus !
2. Sinon ce ne seraient des sous-programmes mais des programmes indépendants à part entière !
3. En fait, ces deux aspects sont indissociables.

Questions d'ordonnancement

Multi-tâche préemptif vs. coopératif

Un **ordonnanceur** 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 [1].

L'ordonnanceur peut mettre en œuvre :

- un fonctionnement **multi-tâches préemptif** : l'ordonnanceur choisit quand mettre en pause une tâche pour reprendre l'exécution d'une autre. Cela peut arriver (presque) à tout moment.
  C'est le cas pour la gestion des processus dans les OS modernes pour ordinateur personnel.
- ou bien un fonctionnement **multi-tâches coopératif** : chaque tâche signale à l'ordonnanceur quand elle peut être mise en attente (par exemple en faisant un appel bloquant).

1. *Operating System*/système d'exploitation

Questions de synchronisation

Transmettre des résultats d'une tâche à l'autre

Plusieurs techniques de transmission de résultats :

- **variables partagées** : variables accessibles par plusieurs tâches concurrentes. Données partagées de façon transparente, sans synchronisation a priori, mais le langage permet d'insérer des primitives de synchronisation explicite [1].
- **passage de message** : données « envoyées » [2] d'une tâche à l'autre. Synchronisation implicite de l'émision et de la réception du message : par exemple, une tâche en attente de réception est bloquée tant qu'elle n'a rien reçu. [3]

La réalité physique est plus proche du modèle des variables partagées [4], mais le passage de message est un paradigme plus sûr [5].

1. En Java : `start()`, `join()`, **`volatile`**, **`synchronized`** et `wait()`/`notify()`.
2. Sous-entendu : l'envoyeur ne peut plus accéder à ce qui a été envoyé.
3. C'est une possibilité. On peut aussi bloquer la tâche émettrice (canal borné, « rendez-vous »).
4. Mémoire centrale lisible par plusieurs CPU.
5. Pour lequel la sûreté d'un programme est plus facile à prouver.

Questions de synchronisation

Se transmettre des résultats

Mais on peut simuler le passage de message :

```java
public final class MailBox<T> { // 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;
    }
}

public final class PingPong {
    public static void main(String[] args) {
        var box = new MailBox<String>();
        new Thread(() -> {
            try { while(true) box.sendMessage("ping!"); }
            catch (Exception e ) { throw new RuntimeException(e); }
        }).start(); // producteur/écrivain
        new Thread(() -> {
            try { while(true) System.out.println((box.receiveMessage() == "ping!")?"pong!":"error!"); }
            catch (Exception e ) { throw new RuntimeException(e); }
        }).start(); // consommateur/lecteur
    }
}
```

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
    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
});
```

Questions de synchronisation

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 2 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);
... // fais 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 [1].

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.

Notion de *thread*

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

Abstraction concurrente consistant en une séquence d'instructions dont l'exécution **simule une exécution séquentielle** (en interne) [1] et **parallèle** à celle des autres *threads*.

- Un nombre quelconque de *threads* s'exécute sur une plateforme de degré de parallélisme quelconque [2]. 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*
- Un **processus** [3] (= 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 habitules 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.

Notion de *thread*

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) + " ");
    }
}
```

Notion de *Thread*

Exemple simple (2)

Alors

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

Notion de *thread*

Un concept décliné à plusieurs niveaux (1)

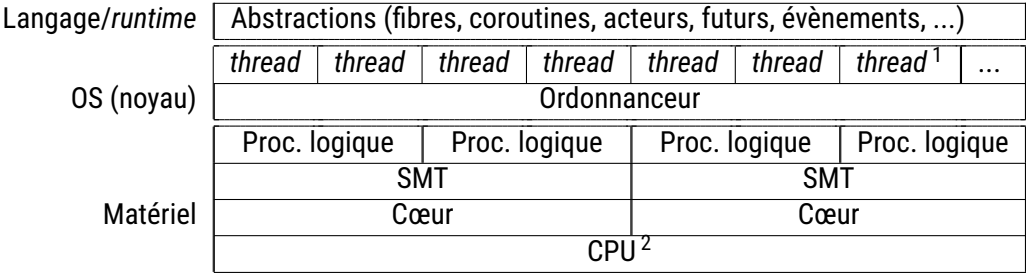
- **Dans le matériel :** p. ex., dans les processeurs *Intel Core i7*, un même cœur exécute 2 *threads* simultanés pour pouvoir utiliser optimalement tous les composants du *pipeline* (SMT/*hyperthreading*).
  Ces 2 *threads* sont présentés à l'OS comme des processeurs séquentiels à part entière (ainsi un *i7* à 4 cœurs, apparaît, pour le système, comme 8 processeurs).
- **Dans les OS multi-tâches :** afin que plusieurs logiciels puissent s'exécuter en même temps, un OS est capable d'instancier un « grand » [1] nombre de *threads* (on parle de « ***threads* système** »).
  L'OS contient un ordonnanceur affectant tour à tour les *threads* aux différents processeurs [2] en gérant les changements de contexte. [3]

1. On parle typiquement de milliers, pas de millions. La limite pratique est la mémoire disponible.
2. réels ou simulés, cf. *hyperthreading*
3. Contexte = pointeur de pile, pointeur ordinal, différents registres...

Notion de *thread*

Un concept décliné à plusieurs niveaux (2)

- **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.



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

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. [1]
- 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 !

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 ».

Des APIs de haut niveau pour ne pas manipuler les *threads*

$\rightarrow$ les langages de programmation proposent des mécanismes, utilisant les *threads* système, pour pallier leurs inconvénients tout en essayant [1] de garder leur avantages.

Au moins deux objectifs :

- limiter le nombre de *threads* système utilisés, afin de diminuer l'empreinte mémoire et la fréquence des changements de contexte
- forcer des procédés sûrs pour le partage de données ; ou à défaut, faciliter les bonnes pratiques de synchronisation.

1. avec plus ou moins de succès

La situation de Java

Java

- utilise directement les *threads* système, via la classe `Thread`.
- a historiquement (Java 1.1) utilisé des *green threads* [1], abandonnés pour des raisons de performance [2].
- pourrait néanmoins, dans le futur, supporter les fibres [3] via le projet Loom.
- dispose actuellement d'un grand nombre d'APIs facilitant où rendant plus sûre l'utilisation des *threads* : les boucles d'évènements Swing et JavaFX, `ThreadPoolExecutor`, `ForkJoinPool`/`ForkJoinTask`, `CompletableFuture`, `Stream`...

---

1. Une sorte de *threads* légers.
2. Ils étaient ordonnancés sur un seul *thread* système, empêchant d'utiliser plusieurs processeurs.
3. Autre type de *threads* légers. Cette fois-ci, le travail peut être distribué sur plusieurs *threads* système. Des implémentations de fibres pour Java existent déjà : bibliothèques Quasar et Kilim. Mais pour fonctionner, celles-ci doivent modifier le code-octet généré par `javac`.

Threads en Java

- 1 *thread* est associé à 1 pile d'appel de méthodes
  Thread principal en Java = pile des méthodes appelées depuis l'appel initial à `main()`
  → **vous utilisez déjà des *threads* !**
- Interfaces graphiques (Swing, JavaFX, ...) : un *thread* [1] ($\neq$ `main`) est dédié aux évènements de l'IG :
  - Programmation événementielle → méthodes gestionnaires d'événement
  - Événements → mis en file d'attente quand ils surviennent.
  - Quand le *thread* des évènements est libre, le gestionnaire correspondant au premier événement de la file est appelé et exécuté sur ce *thread*.

  Intérêt : [2] pas besoin de prévoir des interruptions régulières dans le *thread* `main` pour vérifier et traiter les événements en attente (l'IG resterait figée entre deux)

1. Pour Swing : *Event Dispatching Thread* (EDT). Pour JavaFX : *JavaFX Application Thread*.
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 évènements, pas d'accès compétition.

- Tous les *threads* ont accès au même tas (mêmes objets) et à la même zone statique (mêmes classes)... mais pas à la même pile !
- Les *threads* communiquent grâce aux **variables partagées**, stockées dans le tas.
- Une même méthode peut être appelée depuis n'importe quel *thread* (pas de séparation syntaxique du code associé aux différents *threads*).
- Pour démarrer un *thread* : `unObjetThread.start();` (où `unObjetThread` instance de la classe `Thread`).
  → aussitôt, appel de `unObjetThread.run()` dans le *thread* associé à cet objet.
- À chaque *thread* correspond une pile d'appels de méthode. En bas de la pile :
  - pour le *thread* `main`, le *frame* de la méthode `main` ;
  - pour les autres, celui de l'appel initial à `run` sur l'objet représentant le *thread*.

Créer un *thread* en Java

Quelques techniques classiques pour créer un *thread*

- 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 préfèrera écrire une lambda-expression :

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

Créer un *thread* en Java

Remarques

- 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). [1]
- 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`.

La classe Thread

Méthodes importantes

- `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.State getState()` : retourne l'état du *thread*.

La classe Thread

États d'un *thread*

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...

La classe Thread

Interruptions (1)

- 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 [1], 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 [2] qui quitte aussi.
- Le résultat (non garanti [3]) est la terminaison de la tâche exécutée sur `t` [4].
- 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.

La classe Thread

Interruptions (2)

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 appel 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).

La classe Thread

Interruptions (3)

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 appelée 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`.
- Sinon, 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 `interrupted()` retournera bien **`true`**).

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.
}
```

Problèmes liés au *multithreading*

Deux principaux problèmes :

1. **Les entrelacements non maîtrisés** : les instructions de 2 threads **s'entrelacent** [1] 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** [2] : la JVM [3] 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.

Entrelacements de threads

Exemple (1)

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.

Entrelacements de threads

Exemple (2)

Exécution possible :

```
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 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.

Accès atomique

Avec quelle granularité les entrelacements se font-ils ? Peut-on s'arrêter au milieu d'une affectation, faire autre chose sur la même variable, puis finir ? → notion clé : **atomicité**

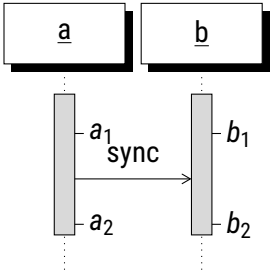
- **Atomique** = non séparable (étym.), non entrelaçable (ici).
  Aucune autre instruction, accédant aux mêmes données, ne peut être exécutée pendant celle des instructions d'une opération atomique.
- Quelques exemples d'opérations atomiques :
  - lecture ou affectation de valeur 32 bits (`boolean`, `char`, `byte`, `short`, `int`, `float`) ;
  - lecture ou affectation de référence (juste la référence, pas le contenu de l'objet) ;
  - lecture ou affectation d'attribut `volatile` [1] ;
  - exécution d'un bloc `synchronized` [2]
- Exemple d'opération non atomique : `x++` (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

Synchronisation :

- consiste, pour un *thread*, à attendre le « feu vert » d'un autre *thread* avant de continuer son exécution ;
- interdit certains entrelacements ;
- contribue à établir la relation "arrivé-avant", limitant les optimisations autorisées [1].



Ici, a_1 arrive avant a_2, b_1 avant b_2, a_1 avant b_2, **mais pas b_1 avant a_2 !**

1. À suivre...

Avec la classe Thread : méthode `join()`

Synchronisation simple : attendre la terminaison d'un *thread* avec `join()` [1] :

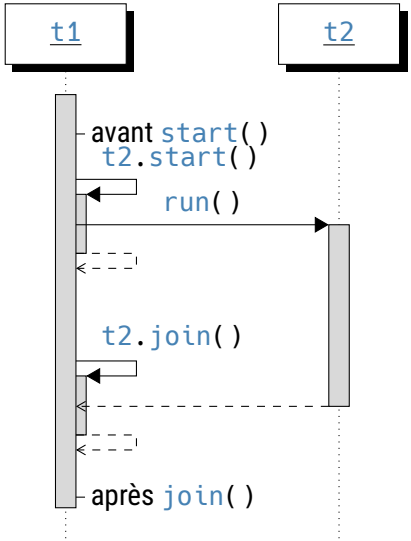
```
public class ThreadJoin extends Thread {
    static int x = 0;

    @Override
    public void run(){ System.out.println(x); }

    public static void main(String[] args){
        Thread t = new ThreadJoin();
        t.start();
        t.join();
        x++;
    }
}
```

affiche `0` alors que le même code sans l'appel à `join()` affichera probablement `1`.

Tout ce qui est exécuté dans le *thread* `t` arrive-avant ce qui suit le `join()` dans le *thread* `main` (ici, l'incrémentation de `x`).

1. Existe aussi en version temporisée : on bloque jusqu'au délai donné en paramètre maximum.

Synchronisation avec `join()`



Le verrou intrinsèque

Qu'est-ce, à quoi cela sert-il ?

- En Java tout objet contient un **verrou intrinsèque** (ou **moniteur**).
- À tout moment, le moniteur est soit libre, soit détenu par un (seul) *thread* donné. Ainsi un moniteur met en œuvre le principe d'exclusion mutuelle.
- Lors de son exécution, un *thread* `t` peut demander à prendre un moniteur.
  - Si le moniteur est libre, alors il est pris par `t`, qui continue son exécution.
  - Si le moniteur est déjà pris, `t` est alors mis en attente jusqu'à ce que le moniteur se libère pour lui (il peut y avoir une liste d'attente).
- Un *thread* peut à tout moment libérer un moniteur qu'il possède.

Conséquence : tout ce qui se produit dans un *thread* avant qu'il libère un moniteur arrive-avant ce qui se produit dans le prochain *thread* qui obtiendra le moniteur, après l'obtention de celui-ci.

Le verrou intrinsèque

Utilisation en pratique : le mot-clé **synchronized**

Bloc synchronisé :

```
class AutreCompteur{
    private int valeur;
    private Object verrou = new Object(); // peu importe le type déclaré
    public void incr(){
        synchronized(verrou) { //    <--- ici !
            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`).

Le verrou intrinsèque

Utilisation en pratique : le mot-clé **synchronized**

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.

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

équivalent à...

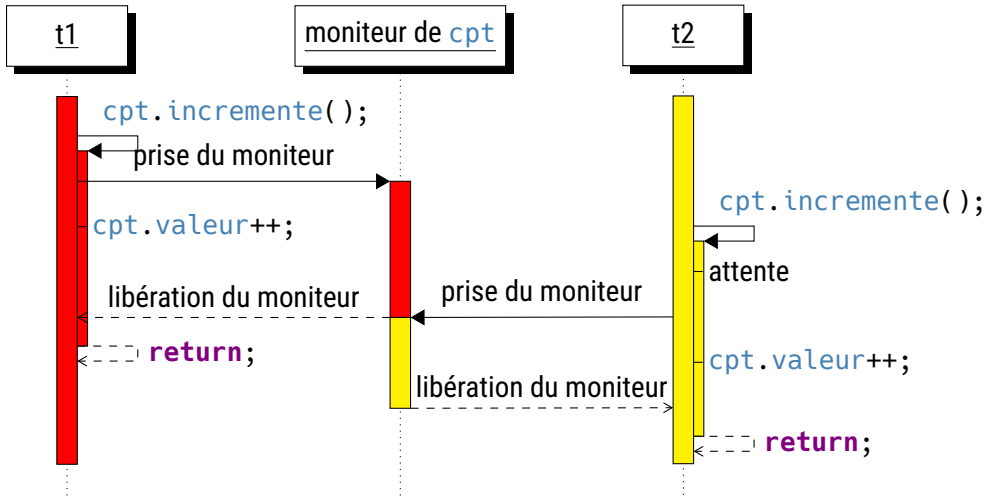
```
class Compteur {
    private int valeur;
    // méthode synchronisée
    public synchronized void incr() {
        valeur++;
    }
}
```

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();
```



Le verrou intrinsèque

Le mécanisme `notifyAll()`/`notify()`/`wait()`

- 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.

Le verrou intrinsèque

Le mécanisme `notifyAll()`/`notify()`/`wait()` – conseil d'utilisation

- 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 $\geq$ 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.

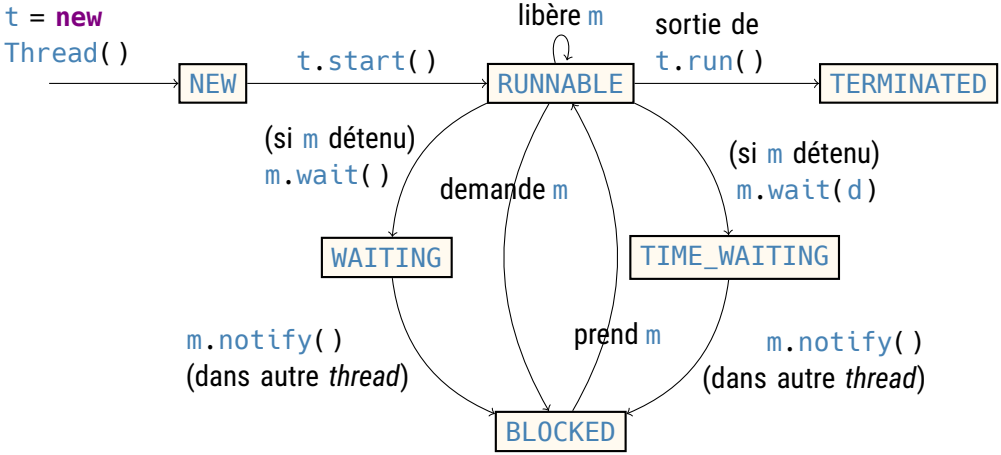
Déconseillé [1] **:** ~~`while(!condition(obj)/*rien*/;`~~ : attente active « bête »

→ c'est l'ancienne façon de faire, remplacée avantageusement 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) prioritairement en cas de besoin.

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

Retour sur les états d'un *thread*

État = une valeur dans `Thread.State` (rectangles) + ensemble de moniteurs détenus



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

Verrous explicites (1)

- 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 resource (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();
}
```

Verrous explicites (2)

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

Inconvénient : l'occupation du verrou n'est pas délimitée par un bloc lexical tel que `synchronized { ... }`[1].
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'acquisition 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'acquérir le verrou 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() ; ... .`

Dangers de la synchronisation

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

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 a nouveau l'une face à l'autre, provoquant une nouvelle manœuvre d'évitement, et ainsi de suite...

Dangers de la synchronisation

Exemple de *dead lock*


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

    public synchronized void useWith(SynchronizedObject other) {
        for (int i = 0; i < 1000; i++); // on simule un long travail
        System.out.println(Thread.currentThread() + " claims monitor on " + this);
        other.use(); // ça, ça sent mauvais...
    }
}

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() + " is done.");
    }

    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();
    }
}
```

Éviter les dead-locks

À l'aide d'une utilisation raisonnée des verrous

- 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.
- $\rightarrow$ quand on écrit un programme concurrent à l'aide de verrous explicites, 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 aisé 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 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*

Est-ce la réalité ?

- Rappel : *thread* = abstraction
  - simulant la séquentialité dans le *thread* ;
  - permettant une communication instantanée inter-*thread* via une mémoire partagée ;
  - (et simulant le parallélisme parfait [1] entre *threads*).
- Est-ce vraiment la réalité ?

  → Divulgâchage : NON !

En réalité, paradigme idéal trop contraignant, empêchant les optimisations matérielles.

Modèle d'exécution réellement implémenté par la JVM : le **JMM** [2].

Seule garantie : sous condition, ce qu'on observe est indistinguable du paradigme idéal.

1. Hors synchronisation, évidemment.
2. Java Memory Model

Modèle de mémoire Java et optimisations

Cohérence de la mémoire

Réalité physique : chaque cœur de CPU dispose de son propre cache [1] de mémoire.

Interprétation : Chaque *thread* 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 visibilité des changements et de cohérence de la mémoire)

Solution naïve : répercuter immédiatement les changements dans tous les caches immédiatement.

Problème : cette opération est coûteuse et supprime le bénéfice du cache.

→ **le JMM :** ne garantit donc pas une cohérence parfaite (mais un minimum quand-même...)

1. Mémoire locale propre au cœur, plus proche physiquement et plus rapide que la mémoire centrale.

Modèle de mémoire Java et optimisations

Consistence de la mémoire

Par exemple, dans le programme suivant :

```
public class ThreadConsistence extends Thread{
    static boolean x = false, y = false;

    public void run(){
        if (x || !y) { x = true; y = true; } else System.out.println("Bug !");
        // Affiche "Bug !" si on trouve y vrai alors que x est faux
    }

    public static void test(int nthreads) throws InterruptedException {
        for (int i = 0; i < nthreads; i++) new ThreadConsistence().start();
    }
}
```

L'appel `test(100)` peut afficher « `Bug !` ».[1]

Par exemple : si un des *threads* finit d'exécuter « `x = true; y = true;` » et un autre reçoit, dans son cache, la modification sur `y` mais pas sur `x`.

1. Le JMM autorise cette possibilité théorique, mais probablement vous ne verrez jamais ce message !

Modèle de mémoire Java et optimisations

Réarrangement des instructions

Réalité physique : Les CPU sont dotés de mécanismes permettant de réordonner des instructions [1] qu'il sait devoir exécuter (afin de mieux occuper tous ses composants).

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.

Solution naïve : ajouter des barrières [2] partout dans le code compilé.

Problème : vitesse d'exécution sous-optimale (le CPU n'arrive plus à donner autant de travail à tous ses composants).

→ **le JMM :** ne garantit pas le respect exact de l'ordre du programme... mais promet que certaines choses importantes restent bien ordonnées.

1. *out-of-order execution*
2. Insruction spécifique prévue dans les CPU, justement pour empêcher le ré-ordonnancement.

Modèle de mémoire Java et optimisations

Réarrangement des instructions

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)
  $\rightarrow$ 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;`.

Modèle de mémoire Java et optimisations

Principe informel du JMM

Entre 2 points de synchronisation, toute optimisation est autorisée tant qu'elle ne change pas le comportement observable d'un *thread* qui s'exécuterait sans interférence d'un autre *thread*.

Donc

- *mono-thread* $\rightarrow$ aucune différence visible due à ces optimisations ;
- mais *multi-thread* $\rightarrow$ différences possibles si synchronisation insuffisante.

$\rightarrow$ 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 et ordre d'exécution

Définitions : 2 façons d'ordonner les évènements d'une exécution donnée [1]

- **Ordre arrivé-avant** :
  - ordre partiel sur les évènements 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.

Ordre arrivé-avant et ordre d'exécution

Relation entre les 2 ordres

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 prouvera 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. [1]

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 [2].

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

Exemples

Programme avec accès en compétition :

```
class Boite {
    int x;
}

public class Competition {
    public static void main(String args[]) {
        Boite b = new Boite();
        new Thread(() -> { b.x = 1; }).start();
        new Thread(() -> {
                System.out.println(b.x);
            }).start();
    }

}
```

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

Programme sans accès en compétition :

```
class BoiteSynchro {
    private int x;
    public synchronized int getX() {
        return x;
    }
    public synchronized void setX(int x) {
        this.x = x;
    }
}

public class PasCompetition {
    public static void main(String args[]) {
        BoiteSyncro b = new BoiteSynchro();
        new Thread(() -> {
                b.setX(1);
            }).start();
        new Thread(() -> {
                System.out.println(b.getX());
            }).start();
    }

}
```

Programme correctement synchronisé

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 réordonnancements 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 sequentiellement cohérente est sans accès en compétition.

Modèle d'exécution

La vérité, enfin !

L'ordre d'exécution exact d'un programme est imprévisible, mais ce qui suit est garanti :

> Si le programme est correctement synchronisé [1],
> alors son exécution est indiscernable d'une exécution séquentiellement cohérente.

Concrètement, pour que les optimisations ne provoquent pas d'incohérences, il « suffit » donc qu'il n'y ait **pas de compétition**.

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.

Éviter les problèmes, version courte

Si vous pouvez prouver que tout accès en compétition à vos variables partagées est impossible dans une exécution « normale »[a], alors vous serez pas importuné par les optimisations !

a. « Normale » = séquentiellement cohérente, non optimisée.

Supprimer les compétitions

Comment éviter les compétitions ?

- éviter de partager les variables quand ce n'est pas nécessaire → préférer les variables locales (jamais partagées) aux attributs ;
- quand ça suffit, privilégier les données partagées en lecture seule → privilégier les structures immuables (voir ci-après) ;
- sinon, renforcer la relation arrivé-avant :
  - utiliser les mécanismes de synchronisation déjà présentés,
  - marquer des attributs comme `final` ou `volatile` (voir ci-après) ;
- utiliser des classes déjà écrites et garanties « *thread-safe* ».
- Souvent, rien de tout ça ne convient : on peut avoir besoin d'attributs modifiables sans synchronisation ! Mais il faudra s'assurer qu'un seul *thread* peut y accéder.

Les mot-clés **volatile** et **final** (1)

Attributs volatils : un attribut déclaré avec **`volatile`** garantit [1] :

- que tout accès en lecture se produisant, chronologiquement, après un accès en écriture, arrive-après celui-ci.
  (concrètement : cet attribut n'est jamais mis dans le cache local d'un *thread*)
- que tout accès simple en lecture ou écriture est atomique (même pour **`long`** et **`double`**).

→ comme si cet attribut était accédé via des accesseurs **`synchronized`**.

Attributs finaux :

- déjà vu : un attribut **`final`** ne peut être affecté qu'une seule fois (lors de l'initialisation de la classe ou de l'objet).
- garantie supplémentaire : comme pour **`volatile`**, tout accès en lecture à un attribut **`final`** arrive-après son initialisation (unique, pour **`final`**).

1. Ceci ne concerne pas le contenu de l'éventuel objet référencé.

Les mot-clés **volatile** et **final** (2)

Technique infaillible : tous les attributs `volatile` (ou `final`) $\Longrightarrow$ accès en compétition impossible. Cependant pas idéale car :

- **non réaliste :** un programme utilise des classes faites par d'autres personnes ; [1]
- **non efficace :** `volatile` empêche les optimisations $\Rightarrow$ exécution plus lente. [2]

En plus, `volatile` ne permet pas de rendre les méthodes atomiques $\Rightarrow$ entrelacements toujours non maîtrisés $\Rightarrow$ `volatile` ne suffit pas toujours pour tout.

Exemple : avec `volatile int x = 0;`, si on exécute 2 fois en parallèle `x++`, on peut toujours obtenir 1 au lieu de 2.

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 `final`, 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.

Les objets immuables

- **Rappel : immuable** = 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* [1], mais en plus il ne peut pas y avoir d'incohérence de cache par rapport au contenu d'un objet immuable. [2]

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.

Les objets immuables

Faire des calculs en utilisant les objets immuables

- Typiquement, une étape de calcul consiste à créer un nouvel objet immuable à partir d'objets immuables existants (puisqu'on ne peut pas les modifier).
- Un tel calcul peut être réalisé à l'aide d'une **fonction pure** [1]
- **Inconvénient :** implique d'allouer un nouvel objet pour chaque étape de calcul. (coûteux, mais pas forcément excessif [2])
- Le résultat doit être correctement publié pour être utilisable par un autre *thread* :
  - grâce aux mécanismes (méthodes) de passage de message prévues 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 : `SharedRessources.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.

Variables atomiques

`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). [1]

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).

Autres objets *thread-safe*

Nombre de classes de l'API sont signalées comme *thread-safe*. En particulier, il peut être utile de rechercher la documentation des collections concurrentes (*package* `java.util.concurrent`).

Regardez les différentes implémentations de `BlockingQueue` et de `ConcurrentMap`, par exemple.

Les nombreux inconvénients de l'API *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 [1] aident à contourner ces écueuils.

→ 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

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 [1] ;
- 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. [2] [3]

1. Selon politique de l'exécuteur. Plusieurs possibles. Par exemple : nb. max. *threads* $\leq$ nb. cœurs.
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 $\rightarrow$ multi-tâche coopératif.

APIs pour la concurrence

Différents 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(f.join()); // appel bloquant avec récupération du résultat
}).fork(); // lancement de la tâche principale
```

  Dans le JDK : `Thread`, `Future` et `ForkJoinTask` suivent ce principe. [1]

1. Hors JDK, citons le principe des « fibres dans la bibliothèque Quasar (qui implémente aussi les « acteurs » en API bloquante).

APIs pour la concurrence

Différents styles d'APIs pour les tâches

- **API non bloquante :** une tâche qui dépend d'un résultat fourni par une autre est passée en tant que **fonction de rappel** (*callback*) [1]. 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`. [2]

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), ...

Interfaces `Executor` et `Runnable`

- En Java, les exécuteurs sont les instances de l'interface `Executor` :

```
public interface Executor { void execute(Runnable command); }
```

L'appel `unExecutor.execute(unRunnable)` exécute la méthode `run()` de `unRunnable`. Ainsi, dans ce cas, les tâches sont des instances de `Runnable`.

- **Un exemple :** (`ExecutorService` étend `Executor`)

```
// instanciation d'un exécuteur gérant un thread unique :
ExecutorService executor = Executors.newSingleThreadExecutor();

// lancement d'une tâche (décrite par la lambda expression, type inféré Runnable)
executor.execute(() -> { System.out.println("bla bla bla"); });

// on détruit les threads dès que tout est terminé
executor.shutdown();
```

Implémentations diverses, utilisant un ou plusieurs *threads* (***thread pool***).

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

- `ExecutorService` : étend `Executor` en y ajoutant :
  - `<T> Future<T> submit(Callable<T> task)` : programme une tâche.
  - Des méthodes pour demander et/ou attendre la terminaison des tâches en cours.
- `Callable<V>` est comme `Runnable`, mais sa méthode retourne un résultat :

```
public interface Callable<V> { V call(); }
```

- `Future<V>` = objets pour accéder à un résultat de type `V` promis dans le futur :

```
public interface Future<V> {
  boolean cancel(boolean mayInterruptIfRunning);
  V get() throws InterruptedException, ExecutionException;
  V get(long timeout, TimeUnit unit) throws InterruptedException,
      ExecutionException, TimeoutException;
  boolean isCancelled();
  boolean isDone();
}
```

Les méthodes `get` sont bloquantes jusqu'à disponibilité du résultat.

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

Usage (schéma général)


```
ExecutorService es = ...;
...
Callable<String> task = ...;
// task sera executé dans thread choisi par es :
Future<String> result = es.submit(task);
...
// le programme attend puis affiche le résultat :
System.out.println("Résultat : " + result.get());
```

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

Exemple plus complet


```java
public class TestCall implements Callable<Integer> {
    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<String> futur1 = executor.submit(new TestCall(1));
        Future<String> futur2 = executor.submit(new TestCall(2));
        try { System.out.println(futur1.get() + futur2.get()); } // affiche "3"
        catch(ExcecutionException | InterruptedException e) { ... }
        finally { executor.shutdown(); }
    }
}
```

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 exécuteurs

À l'aide de la classe `Executors` :

- = bibliothèque de fabriques statiques d'`ExecutorService`.
- `static ExecutorService newSingleThreadExecutor()` : crée un `ExecutorService` utilisant un *worker thread* unique.
- `static ExecutorService newCachedThreadPool()` : crée un exécuteur dont les *threads* du *pool* se créent, à la demande, sans limite, mais sont réutilisés s'ils redeviennent disponibles.
- `static ExecutorService newFixedThreadPool(int nThreads)` : même chose, mais avec limite fixée à `n` *threads* [1].

On peut aussi utiliser directement les constructeurs de `ThreadPoolExecutor` ou de `ScheduledThreadPoolExecutor` [2] (nombreuses options).

1. Choisir `n` en rapport avec le nombre d'unités de calcul/cœurs.
2. Implémente `ScheduledExecutorService`, permettant de programmer des tâches périodiques et/ou différées. Les futurs retournés implémentent `ScheduledFuture<V>`. Regardez la documentation.

Les limites de `ThreadPoolExecutor`

(et des fabriques `newCachedThreadPool()` et `newFixedThreadPool()`)

- But d'un *thread pool* = réduire le nombre de *threads* → petit nombre de *threads*.
- Or les *threads* bloqués (par appel bloquant, comme `f.get()`, au sein de la tâche) ne sont pas réattribuables à une autre tâche [1].
  → moins de *threads* disponibles dans le *pool* → ralentissement.
- **Cas extrême :** si grand nombre de tâches concurrentes avec interdépendances, il arrive que tout le *pool* soit bloqué par des tâches en attente de tâches bloquées ou non démarrées → rien ne viendra débloquer la situation.
  Cette situation s'appelle un ***thread starvation deadlock*** [2].

Comment concilier *pool* de taille bornée et garantie d'absence de bloquage ?

1. Il est impossible de « sortir » une tâche déjà en exécution sur un *thread* pour le libérer.
2. Du point de vue des *threads*, c'est bien un *deadlock* : dépendance cyclique entre *threads*.
Du point de vue des tâches, dépendance pas forcément cyclique, mais bloquage car multi-tâche non-préemptif s'exécutant sur un nombre limité d'unités d'exécution.

Work-stealing strategy

Decription

Solution : stratégie de vol de travail (***work stealing***). Principe :

- une file d'attente de tâches par *thread* au lieu d'une commune à tout le *pool* ;
- tâches générées par une autre tâche $\rightarrow$ ajoutées sur file du même *thread* ;
- quand un *thread* veut du travail, il prend une tâche en priorité dans sa file, sinon il en « vole » une dans celle d'un autre *thread* ;
- **le plus important :** si le résultat attendu n'est pas disponible, `get` (et `join`) exécute d'abord les tâches en file au lieu de bloquer le *thread* tout de suite.

  $\Rightarrow$ C'est là que se met en place la coopération entre tâches.

Le vol de travail assure que si tous les *threads* sont bloqués (par des tâches en attente d'une autre tâche), c'est qu'il n'y a plus de tâche à démarrer.

Cela veut dire que les tâches attendues sont déjà démarrées et non terminées. C'est donc qu'elles sont elles-mêmes en attente d'une tâche... aussi en attente.

$\Longrightarrow$ la seule possibilité c'est que les tâches s'attendant les unes les autres forment un (ou des) cycle de dépendances.

$\Longrightarrow$ si pas de dépendances cycliques, *thread starvation deadlock* impossible.

Work-stealing strategy

Indications et contre-indications

- Petites tâches à dépendances acycliques (notamment, algorithmes récursifs)
  → stratégie très intéressante (pas de *thread starvation deadlock*).
- Tâches sans dépendances
  → stratégie inutile et plus lourde que la stratégie « naïve »[1].
- Tâches à dépendances cycliques
  → *deadlocks* assurés (mais aucune stratégie ne peut fonctionner dans ce cas !).

1. Sans compter qu'en Java, l'implémentation de celle-ci (`ThreadPoolExecutor`) est plus configurable que l'implémentation de la *work-stealing strategy* (`ForkJoinPool`).

ForkJoinPool et ForkJoinTask

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

Ainsi Java propose :

- la classe `ForkJoinPool` : implémentation d'`Executor` utilisant cette stratégie
- la classe `ForkJoinTask` : tâches capables de générer des sous-tâches (« `fork()` ») et d'attendre leurs résultats pour les utiliser (« `join()` »).

`ForkJoinPool` est considéré suffisament efficace pour que le *thread pool* par défaut (utilisé implicitement par plusieurs API concurrentes) soit une instance de cette classe.

Obtenir le *thread pool* par défaut : `ForkJoinPool.commonPool()`.

ForkJoinPool et ForkJoinTask

- Méthodes de `ForkJoinTask` :
  - `ForkJoinTask<T> fork()` : demande l'exécution de `t` et rend la main. Le résultat du calcul peut être récupéré plus tard en interrogeant l'objet retourné.
  - `T invoke()` : pareil, mais attend que `t` soit finie et retourne le résultat.
  - `T join()` : attendre le résultat du calcul signifié par `t` (`T get()` existe aussi car `ForkJoinTask<T>` implémente `Future<T>`)

  `fork` et `invoke` exécutent la tâche dans le *pool* dans lequel elles sont appelées, si applicable, sinon dans le *pool* par défaut.
- Pour exécuter une tâche sur un `ForkJoinPool` précis, on appelle les méthodes suivantes (de la classe `ForkJoinPool`) sur le *pool* `p` cible :
  - `<T> T invoke(ForkJoinTask<T> task)` : demande l'exécution de `task` sur `p` et retourne le résultat dès qu'elle se termine (appel bloquant).
  - `<T> ForkJoinTask<T> submit(ForkJoinTask<T> task)` : idem, mais rend la main tout de suite en retournant un futur, permettant de récupérer le résultat plus tard.
  - `void execute(ForkJoinTask<?> task)` : idem, aussi non bloquant, mais on ne récupère pas le résultat.

ForkJoinPool et ForkJoinTask

Et ForkJoinTask ?

- Classe abstraite. Ses objets sont les tâches exécutables par `ForkJoinPool`.
- On préfère étendre une de ses sous classes (abstraites aussi) [1] :
  - `RecursiveAction` : tâche sans résultat (exemple : modifier les feuilles d'un arbre)
  - `RecursiveTask<V>` : tâche calculant un résultat de type `V` (exemple : compter les feuilles d'un abre)

  Dans les 2 cas, on étend la classe en définissant la méthode `compute()` qui décrit les actions effectuées par la tâche :
  - pour `RecursiveAction` : `protected void compute();`
  - pour `RecursiveTask<V>` : `protected V compute();`
- 3 fabriques statiques `ForkJoinTask.adapt(...)` [2] permettent de créer des tâches à partir de `Runnable` et de `Callable<V>`.

```
ForkJoinTask.adapt(() -> { /* instructions */ }).fork() // sympa avec les lambdas !
```

1. Ces deux classes servent juste à faciliter l'implémentation.
2. Le nom `adapt` provient du patron *adapter*.

ForkJoinPool et ForkJoinTask

Et car rien ne vaut un exemple...

La tâche récursive :

```
class Fibonacci extends RecursiveTask<Integer> {
    final int n;

    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 : `CompletableFuture`

- ```
  class CompletableFuture<T> implements Future<T>, CompletionStage<T>
  ```
- `CompletionStage<T>` propose des méthodes pour ajouter des *callbacks* à exécuter lors de la complétion de la tâche.

$\rightarrow$ changement de paradigme : plus d'appels bloquants dans les tâches élémentaires, mais dépendances maintenant décrites en dehors de celles-ci.

Le programme appelant compose ces tâches entre elles pour décrire une « recette » [1] qu'il soumet au *thread pool* (et donc n'effectue pas non plus d'appel bloquant).

1. Les savants parlent de « monade ».
  ```

Une alternative : CompletableFuture

Exemple : de ForkJoinTask à CompletableFuture


```
Rf rf = Boulot.f();
Future<Rg> frg = ForkJoinTask.adapt(() -> Boulot.g(rf)).fork();
Rh rh = Boulot.h(rf);
Ri ri = Boulot.i(rh, frg.join());
System.out.println("Résultat : " +ri);
```

Devient :

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

Les *reactive streams* [2] 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 » [1], 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* »).

1. 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.
2. C'est le nom de la spécification, cela n'a pas de rapport avec l'API *stream* de Java.

Les *reactive streams* avec `Flow`

Un exemple simple (programme)

On définit un `Subscriber` :

```
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) {
      subscription.request(1); // Je suis prêt à recevoir 1 autre message !
      System.out.println(this + " : " + item + " (thread " + 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 {
      try (var publisher = new SubmissionPublisher<String>()) { // SubmissionPublisher 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);
      }
  }
```

Les *reactive streams* avec `Flow`

Un exemple simple (affichage)

On observe alors sur la sortie standard :

```
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@1446b42: 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@1446b42: amet (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: amet (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@284682b2: C'est fini !
PrintSubscriber@1446b42: C'est fini !
```

Les *reactive streams* avec `Flow`

Les interfaces


```java
public class Flow {
  private Flow() {} // non instanciable

  public static interface Publisher<T> {
    public void subscribe(Subscriber<? super T> subscriber);
  }

  public static interface Subscriber<T> {
    public void onSubscribe(Subscription subscription);
    public void onNext(T item);
    public void onError(Throwable throwable);
    public void onComplete();
  }

  public static interface Subscription {
    public void request(long n);
    public void cancel();
  }

  public static interface Processor<T,R> extends Subscriber<T>, Publisher<R> {  }

  static final int DEFAULT_BUFFER_SIZE = 256;

  public static int defaultBufferSize() { return DEFAULT_BUFFER_SIZE; }
}
```

Les *reactive streams* avec `Flow`

Explication

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-pression** » (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 :

- utilse un *thread pool* (soit celui passé en paramètre, soit `ForkJoinPool.commonPool()`)
- a 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).

Les *reactive streams* avec `Flow`

Exemple de `Processor`

Pour décomposer un traitement en plusieurs étapes, on peut établir une chaîne : `Publisher` → `Processor` → ... → `Processor` → `Subscriber`.

Or `Processor` = `Subscriber` + `Publisher` et, malheureusement, `Publisher` n'a pas d'implémentation dans le JDK et elle est difficile à implémenter.

Solution : comme suggéré juste avant, réutiliser `SubmissionPublisher` :

```
public abstract class AbstractProcessor<T, U> implements Flow.Processor<T, U> {
  private final SubmissionPublisher<U> dispatcher; // préférons la composition à l'héritage !
  // constructeurs de même signature que ceux de SubmissionPublisher
  public AbstractProcessor() { this.dispatcher = new SubmissionPublisher<U>(); }
  public AbstractProcessor(Executor executor, int maxBufferCapacity) {
      this.dispatcher = new SubmissionPublisher<U>(executor, maxBufferCapacity);
  }
  public AbstractProcessor(Executor executor, int maxBufferCapacity, BiConsumer<? super
        Flow.Subscriber<? super U>, ? super Throwable> handler) {
      this.dispatcher = new SubmissionPublisher<U>(executor, maxBufferCapacity, handler);
  }
  // délégation des abonnements et de la soumission de nouveaux messages à dispatcher
  @Override public final void subscribe(Flow.Subscriber<? super U> subscriber) {
      dispatcher.subscribe(subscriber);
  }
  protected final int submit(U item) { return dispatcher.submit(item); }
}
```

Les *reactive streams* avec `Flow`

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) {
      subscription.request(1);
      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 concrétisation.
En version très courte on aurait pu avoir :

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

Choisir une API concurrente

Dans ce cour, 4 API haut niveau concurrentes vous ont été présentées :

- les *streams* concurrents
- *fork/join*
- `CompletableFuture`
- `Flow`

Toutes les 4 permettent d'éviter l'usage de variables partagées (et la synchronisation explicite).

Si certains programmes peuvent être réalisés indifféramment avec plusieurs APIs, certaines sont clairement plus adaptées à certains cas d'usage...

Par ailleurs, certaines API correspondront mieux à votre façon de penser.

Pour vous faire une idée, **expérimentez !**