



Conception & Prog. par objets 2

Chouki Tibermacine

Chouki.Tibermacine@umontpellier.fr

Adapation du cours de José Paumard, DevRel @ Orcale



Plan de l'ECUE

1. Programmation par objets en Python (Vincent Berry)
2. Généricité paramétrique en Java (Marianne Huchard)
3. Introspection et annotations en Java (Marianne Huchard)
4. Conception d'architectures logicielles
5. Construction de projets Java avec Maven
6. Modularisation des applications Java avec les modules et services Java 9+
7. Développement d'applications multi-threadées en Java
8. Interaction avec des sources de données avec JDBC et JPA
9. Développement d'applications à arch. distribuées avec *Java Enterprise*
10. Amélioration de la conception d'applications grâce au refactoring
11. Introduction à la programmation avec le langage Kotlin
12. Programmation par objets en JavaScript

Plan du cours

1. Notions de base sur la concurrence
2. Gérer les problèmes de synchronisation
 - 2.1 Problème du producteur/consommateur
 - 2.2 Utiliser le patron Wait & Notify
 - 2.3 Concurrence dans les processeurs multi-coeurs
 - 2.4 Étude de cas : le patron Singleton
3. Autres patrons pour la concurrence
 - 3.1 Executor, Future & Callable
 - 3.2 Verrous et conditions
4. Collections concurrentes

Concurrence

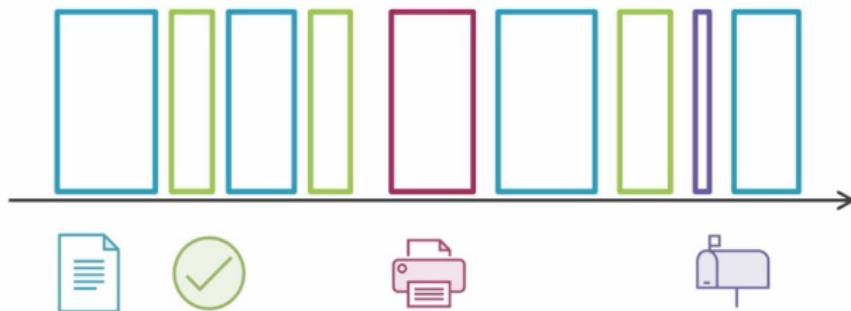
- Concurrence : exécuter plusieurs tâches en parallèle
- Se posent des problèmes de “situations de compétition” (*race conditions* en anglais) liés à l'accès aux ressources partagées par plusieurs tâches concurrentes
- Objectif : voir comment on écrit du code correct en présence de la concurrence
- Autre objectif : tirer profit des processeurs multi-coeurs actuels

Concept de *Thread* ou processus léger

- Concept déjà vu dans le cours de système (FAR en IG3)
- Concept qui existe au niveau du système d'exploitation (ce n'est pas propre au langage de programmation)
- Thread = ensemble d'instructions (une tâche) exécutée d'une façon particulière
- Une application peut être composée de plusieurs threads
- Plusieurs threads peuvent être exécutés "en même temps"

Application multi-threads et son exécution

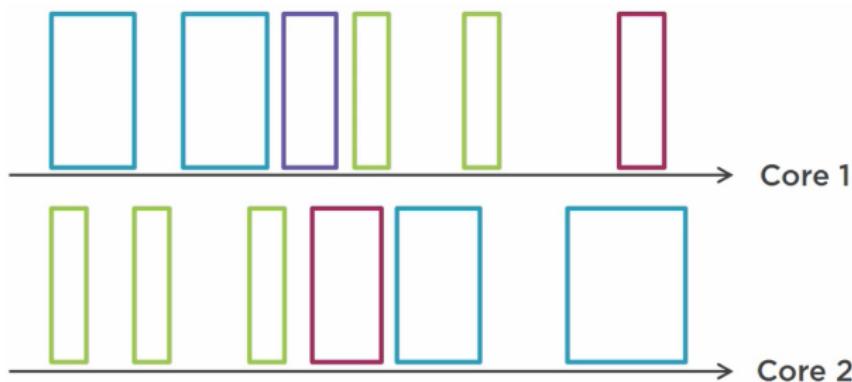
- Différentes tâches s'exécutent “en même temps” : rédiger un email, le spellchecker vérifie votre email, la liste des emails est mise à jour quand un nouvel email arrive, imprimer un email, ...
- Sur un processeur à un seul cœur, “en même temps” ne veut pas dire vraiment en parallèle



- On a juste l'impression que c'est "en même temps" parce que le processeur exécute les instructions très rapidement

Application multi-threads et son exécution -suite-

- Sur un processeur multi-cœurs (la plupart des processeurs modernes), un vrai parallélisme (exécution des threads vraiment en même temps)



Qui gère le partage du processeur ?

- Un module de l'OS : l'ordonnanceur (*scheduler*, en anglais)
- Raisons qui poussent l'ordonnanceur à mettre en pause un thread :
 - Le processeur doit être partagé de façon égale entre les threads
 - Le thread est en train d'attendre une entrée/sortie (les E/S sont des opérations trop lentes comparées à la vitesse du processeur)
 - Le thread attend qu'un autre thread fasse quelque chose (libérer une ressource, par exemple)

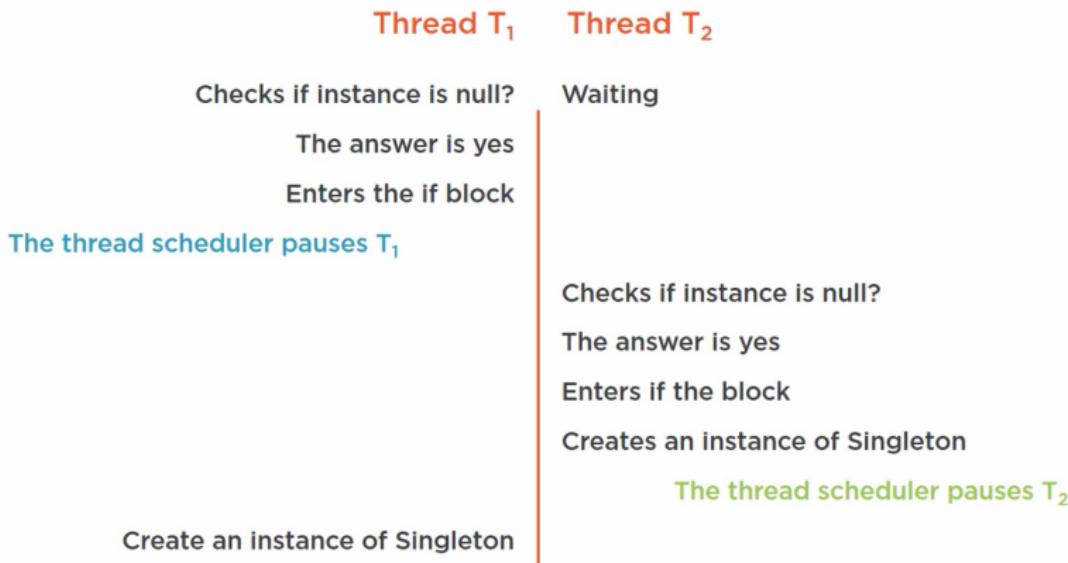
Situation de compétition (*race condition*)

- Deux threads différents qui **lisent et écrivent** sur la **même** variable (ou le **même** attribut d'une classe) en **même temps**
- Exemple : le patron *Singleton*

```
public class Singleton {  
  
    private static Singleton instance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if(instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

Situation de compétition (*race condition*) -suite-

Que se passe-t-il si deux threads invoquent getInstance() ?



⇒ Deux instances sont créées

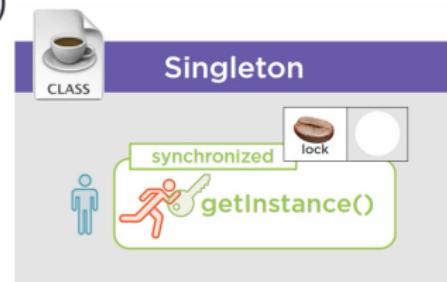
Solution du problème : la synchronisation

- La **synchronisation** prévient qu'un bloc de code s'exécute par plus d'un thread à la fois
- Ceci peut se faire en Java avec le mot clé **synchronized** :

```
public class Singleton {  
  
    private static Singleton instance;  
  
    private Singleton() {}  
  
    public static synchronized Singleton getInstance() {  
        if(instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

Comment cela fonctionne ?

- Le fait de marquer la méthode `getInstance()` comme `synchronized` va protéger les instructions de cette méthode d'être exécutées par plusieurs threads en même temps
- En réalité, la machine virtuelle Java (JVM) associe à cette méthode un objet **verrou** (*lock* en anglais)
- Cet objet possède une *clé* (un moniteur), qui est demandée par un thread à chaque fois qu'il veut exécuter la méthode synchronisée
- Si la clé est disponible, elle est donnée au thread, qui peut alors exécuter la méthode. Sinon, le thread doit attendre



Objet verrou

- Dans l'exemple précédent, on avait une méthode synchronisée qui était statique. L'objet jouant le rôle de verrou est alors l'objet réifiant la classe Singleton (l'objet Class de Singleton)
- Si c'était une méthode non-statique, l'objet verrou serait l'instance de la classe (qui fournit la méthode)
- Mais l'objet verrou peut être joué par n'importe quel objet Java

Objet verrou -suite-

- Objet verrou dédié :

```
public class Personne {  
    private final Object cle = new Object();  
  
    public void init() {  
        synchronized (cle) {  
            // Faire quelque chose ici  
        }  
    }  
}
```

Ici, on a un **bloc** synchronisé et non une méthode

- C'est une meilleure pratique, surtout quand la clé est un attribut privé

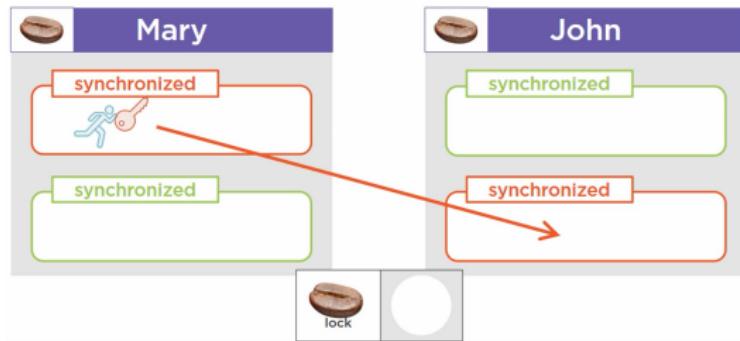
Synchroniser plusieurs méthodes

- Lorsqu'une méthode est synchronisée, elle va partager la clé avec les autres méthodes synchronisées de la classe (le verrou est l'objet Class ou l'instance de la classe)
- Un thread qui exécute une méthode synchronisée m1 définie dans la classe va empêcher d'autres threads d'exécuter une autre méthode synchronisée m2 de cette même classe
- Parfois, ceci n'est pas ce que l'on veut dans une application
- Pour synchroniser des threads exécutant la même méthode dans plusieurs instances d'une même classe (deux instances différentes de la classe Personne, par exemple), il faudrait utiliser un objet verrou stocké dans un attribut **statique** (de la classe Personne, par ex), sinon pas de synchronisation

Reentrant Locks

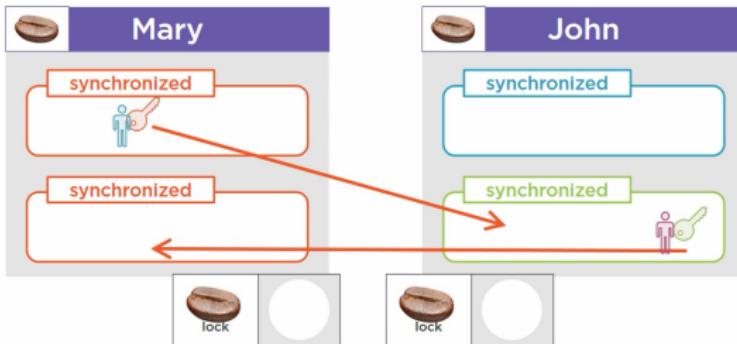
- Dans Java, les verrous sont ré-entrants. ça veut dire quoi?
- Un thread possédant la clé d'un verrou (exécutant une méthode synchronisée) peut exécuter n'importe quelle autre méthode synchronisée avec le même verrou

l'absence de clé pour entrer dans la deuxième méthode n'a pas d'impact sur lui, puisque il possède déjà la clé



Situations d'interblocage Deadlocks

- Deux threads T1 (bleu dans la figure) et T2 (violet) : T1 possède une clé C1 (rouge) et T2 possède une autre clé C2 (verte)
- T1 veut exécuter la méthode synchronisée protégée par C2 et T2 veut exécuter la méthode synchronisée protégée par C1
- Les deux threads sont mutuellement bloqués
- La JVM peut détecter cela et fournit des messages de logs (traces des piles d'appels), mais on est obligé de corriger le code



Lancer des threads avec le patron Runnable

La solution la plus simple pour créer des threads en Java :

1. Créer une instance de type Runnable (la tâche que l'on veut exécuter dans un nouveau thread)
2. Passer cette instance au constructeur de la classe Thread
3. Invoquer la méthode start de l'objet Thread pour lancer l'exécution de la tâche

Créer une instance de type Runnable

- Dans Java 7 et avant (on crée une classe anonyme et on l'instancie) :

```
Runnable task = new Runnable() {
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println("I am running in Thread: " +
            name);
    }
};
```

- Runnable est une interface possédant une seule méthode run()

Créer une instance de type Runnable -suite-

- Depuis Java 8, Runnable est une interface fonctionnelle
- Du coup, on peut simplement écrire une *lambda expression*, qui implémente la seule méthode de cette interface (`run()`):

```
Runnable task = () -> {  
    String name = Thread.currentThread().getName();  
    System.out.println("I am running in Thread: "+  
        name);  
};
```

Lancer l'exécution de la tâche (Runnable) dans un nouveau thread

- D'abord créer une instance de Thread et lui passer l'instance Runnable
- Ensuite lancer l'exécution de la tâche

```
Thread thread = new Thread(task);  
thread.start();
```

Ne jamais invoquer la méthode run() de l'objet thread.
Celle-ci ne démarre pas un nouveau thread mais exécute la tâche dans le thread courant (main par exemple si le code ci-dessus est placé dans la méthode main())

A vos claviers

- Créer une classe avec la méthode main(String... args)
- Y Ajouter le code des diapos précédentes
- Afficher également le nom du thread courant à la fin de la méthode main
- Exécuter plusieurs fois. Que remarquez-vous ?
- Pour exécuter le code à la fin de la méthode main toujours après la fin de l'exécution du thread que l'on crée, ajouter après l'invocation à start() :

```
thread.join();
```

A vos claviers -suite-

- Créer une classe IntWrapper qui encapsule un nombre entier et qui fournit deux méthodes : un getter et une méthode pour incrémenter le nombre
- Créer ensuite une deuxième classe (RaceCondition) avec un main dans laquelle vous créez une tâche Runnable qui incrémentera 1000 fois le nombre, et exécuter cette tâche dans un nouveau thread
- Ajouter à la fin du main un join puis un print du nombre
- Tester le programme. Qu'est-ce que ça donne ?

A vos claviers -suite-

- Modifier le main pour créer 1000 threads que vous stockez dans un tableau :

```
Thread[] threads = new Thread[1000];
for(int i = 0; i < threads.length ; i++) {
    threads[i] = new Thread(task);
    threads[i].start();
}
for(int i = 0; i < threads.length ; i++) {
    threads[i].join();
}
```

- Avant d'exécuter le programme, essayez de deviner quelle valeur sera affichée par le main après l'exécution de tous les threads ?
- Exécuter le programme pour vérifier. Expliquer pourquoi ce résultat

Problème de situation de compétition sur l'accès à l'attribut

- Le problème précédent provient du code de la méthode d'incrémentation du nombre dans la classe IntWrapper :
`value = value + 1` (ou `value += 1` ou `value++`), `value` étant l'attribut de la classe dans lequel est stocké le nombre entier enveloppé
- Cette instruction est constituée d'une opération CPU de lecture de la valeur de l'attribut puis d'une opération d'écriture de sa nouvelle valeur (après l'avoir incrémentée)
- Des interruptions de threads (faites par le *scheduler*) ont eu lieu entre les deux opérations, ce qui explique la valeur affichée
- Problème de *race condition*

A vos claviers

Pour résoudre ce problème :

- Déclarer un attribut de type Object dans la classe IntWrapper :

```
private Object lock = new Object();
```

Cet objet peut être de n'importe quel type objet (String, ...)

- Utiliser cet objet comme verrou :

```
public void increment() {  
    synchronized (lock) {  
        value = value + 1;  
    }  
}
```

- Exécuter le programme principal pour vérifier

Reproduire une situation d'interblocage

- Récupérer les classes A et RunningA depuis la page Web Moodle du cours
- Lire le code dans ces deux classes pour comprendre comment la situation d'interblocage est provoquée (dans la classe A)
- Exécuter la classe RunningA pour vérifier cela
- Pour arrêter le programme, tuer le processus de la JVM avec le bouton Stop (carré rouge) sur votre IDE
- Exécuter RunningA avec le debugger d'IntelliJ (click bouton droit sur la méthode main, puis Debug)
- Dans la partie basse gauche, appuyer sur le bouton pause, puis aller dans l'onglet Debugger de la tool-window Debug
- On peut voir les 3 frames main, Thread-0 et Thread-1 et quels verrous sont attendus (position watch) par chacun

Plan du cours

1. Notions de base sur la concurrence
2. Gérer les problèmes de synchronisation
 - 2.1 Problème du producteur/consommateur
 - 2.2 Utiliser le patron Wait & Notify
 - 2.3 Concurrence dans les processeurs multi-coeurs
 - 2.4 Étude de cas : le patron Singleton
3. Autres patrons pour la concurrence
 - 3.1 Executor, Future & Callable
 - 3.2 Verrous et conditions
4. Collections concurrentes

Plan du cours

1. Notions de base sur la concurrence
2. Gérer les problèmes de synchronisation
 - 2.1 Problème du producteur/consommateur
 - 2.2 Utiliser le patron Wait & Notify
 - 2.3 Concurrence dans les processeurs multi-coeurs
 - 2.4 Étude de cas : le patron Singleton
3. Autres patrons pour la concurrence
 - 3.1 Executor, Future & Callable
 - 3.2 Verrous et conditions
4. Collections concurrentes

Producteur / Consommateur

- Un producteur produit des valeurs dans un buffer
- Un consommateur utilise les valeurs depuis le buffer
- Plusieurs producteurs et consommateurs possibles
- Le buffer peut être vide ou plein
- Problème de *race condition* si l'on ne synchronise pas bien les opérations de lecture et d'écriture dans le buffer
- Récupérer le fichier ProducerConsumer.java sur la page Moodle du cours

Problème du Producteur / Consommateur

- Quel est le problème dans ce code ?
- Situation de compétition (race condition) sur les attributs count et buffer
- Pour corriger cela, il faudra synchroniser l'accès au buffer/count
- Nous pouvons utiliser un verrou, mais il faut qu'il soit le même (partagé) pour tous les producteurs et tous les consommateurs
- Ajouter ce verrou (attribut statique et privé dans la classe englobante) et synchroniser le corps des méthodes produce et consume
- Écrire dans une autre classe la méthode main pour exécuter un producteur et un consommateur dans des threads séparés (une tâche chacun pour produire et consommer plusieurs centaines de messages)

Problème du Producteur / Consommateur

- Est-ce que le code précédent marche ?
- Que se passe-t-il si le buffer est vide et que c'est le tour du consommateur de s'exécuter ?
- Le consommateur détient le verrou, et est bloqué dans la boucle while (`isEmpty()`)
- Le producteur est bloqué dans l'attente de la libération du verrou par le consommateur
- Situation d'interblocage
- Solution : faire attendre le consommateur (tout en le faisant libérer le verrou) jusqu'à ce que le producteur produise des données
- Ceci est nommé le patron *wait/notify*

Plan du cours

- 1. Notions de base sur la concurrence**
- 2. Gérer les problèmes de synchronisation**
 - 2.1 Problème du producteur/consommateur**
 - 2.2 Utiliser le patron Wait & Notify**
 - 2.3 Concurrence dans les processeurs multi-coeurs**
 - 2.4 Étude de cas : le patron Singleton**
- 3. Autres patrons pour la concurrence**
 - 3.1 Executor, Future & Callable**
 - 3.2 Verrous et conditions**
- 4. Collections concurrentes**

Patron *wait/notify*

- `wait()` et `notify()` sont des méthodes de la classe `Object`
- Règle : le thread qui invoke la méthode doit posséder la clé (le verrou) de l'objet
- `wait()` et `notify()` doivent donc être invoquées dans un bloc/méthode synchronisé(e)
- Invoquer `wait()` libère le verrou, puis met le thread dans un état WAIT
- Pour libérer le thread, il faut invoquer `notify()` sur l'objet verrou que ce thread utilise
- Invoquer `notify()` libère un thread qui est dans l'état WAIT (choisi au hasard) et le met dans un état Runnable
- `notifyAll()` libère tous les threads

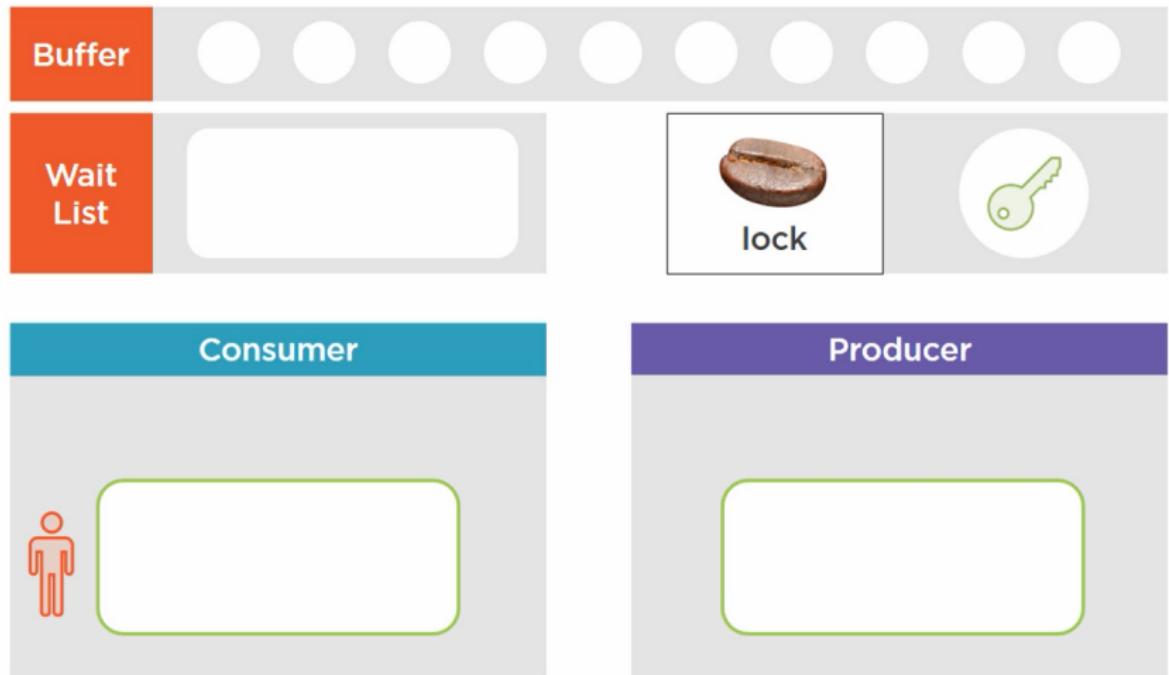
Correction du code précédent

```
public void produce() throws InterruptedException {  
    synchronized (lock) {  
        if (isFull()) lock.wait();  
        buffer[count++] = 1;  
        lock.notifyAll();  
    }  
}
```

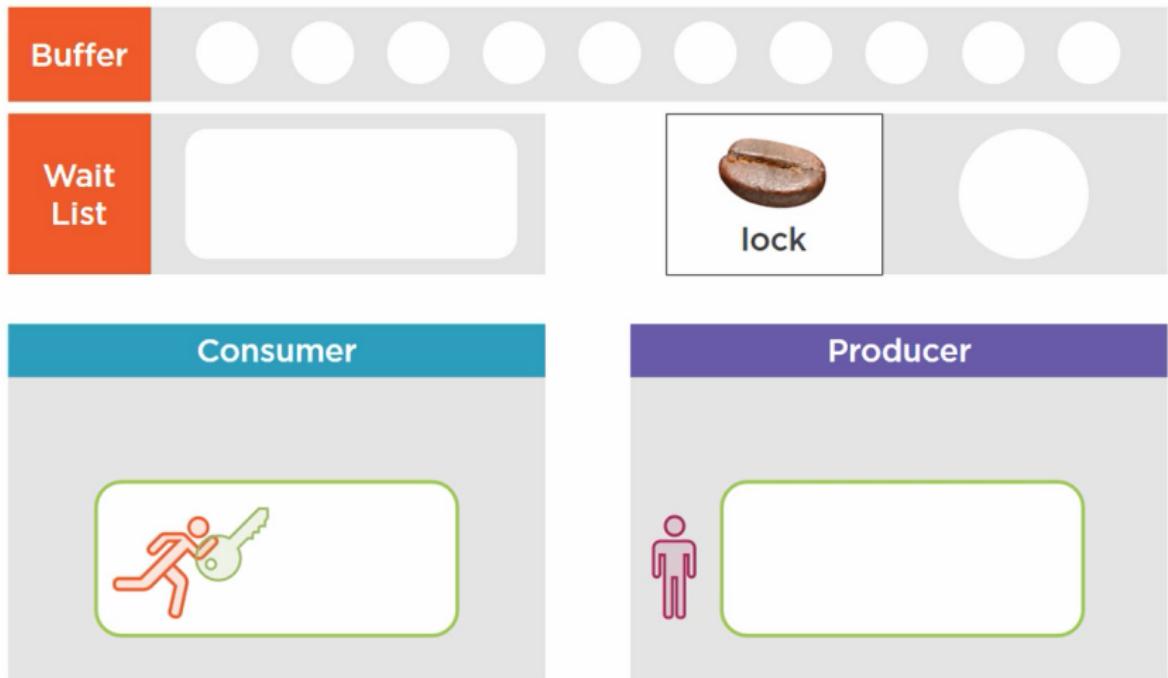
```
public void consume() throws InterruptedException {  
    synchronized (lock) {  
        if (isEmpty()) lock.wait();  
        buffer[--count] = 0;  
        lock.notifyAll();  
    }  
}
```

Tester ce code et ajouter un print de count à la fin du main

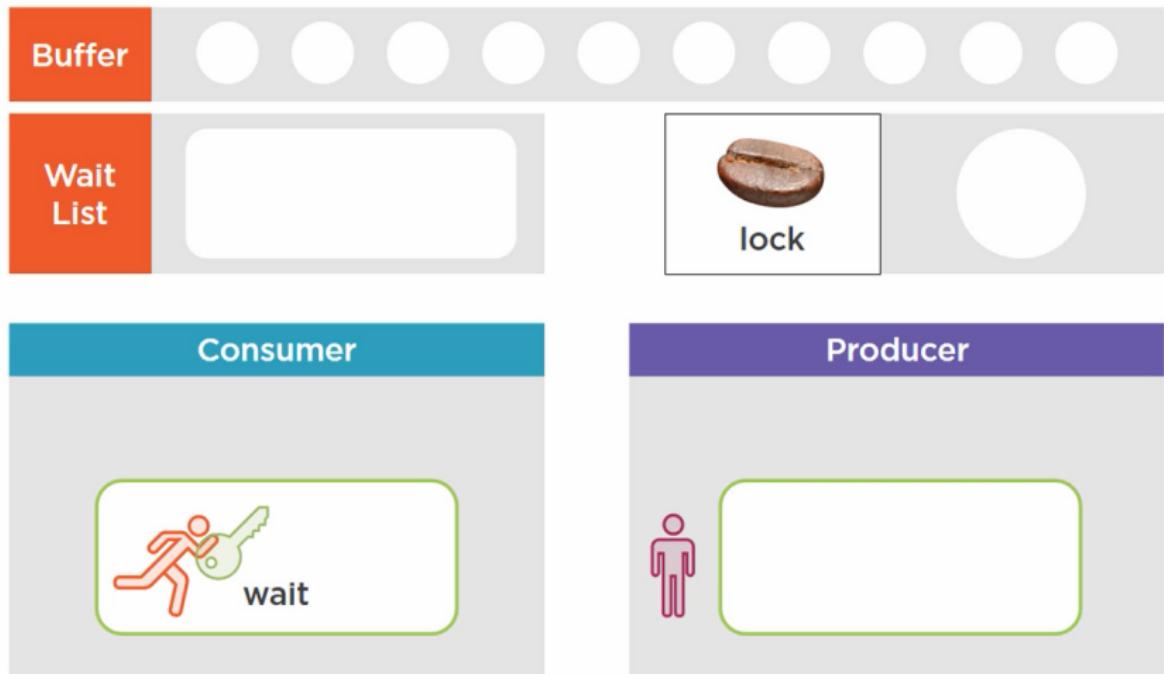
Fonctionnement du patron



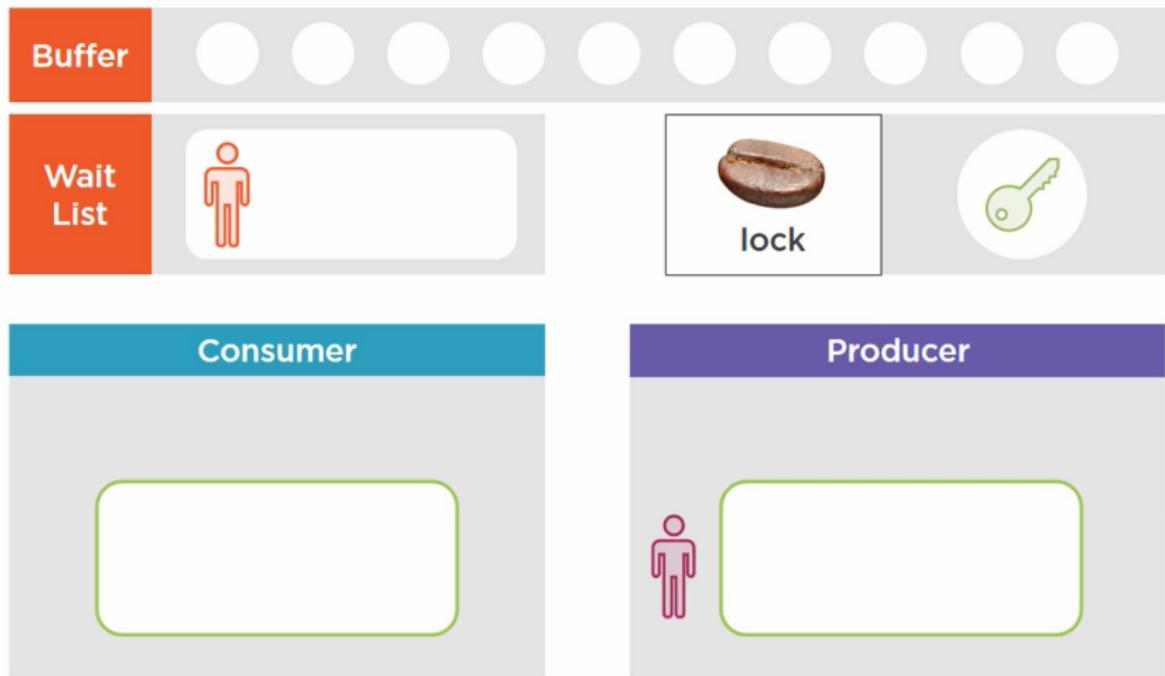
Fonctionnement du patron -suite-



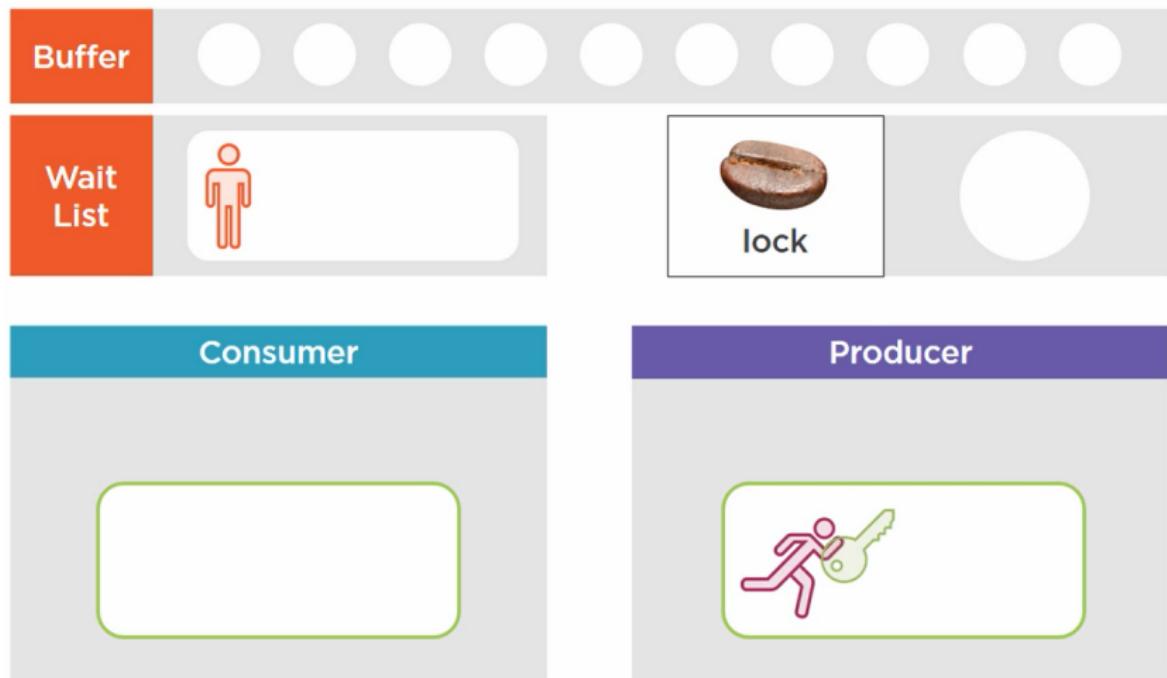
Fonctionnement du patron -suite-



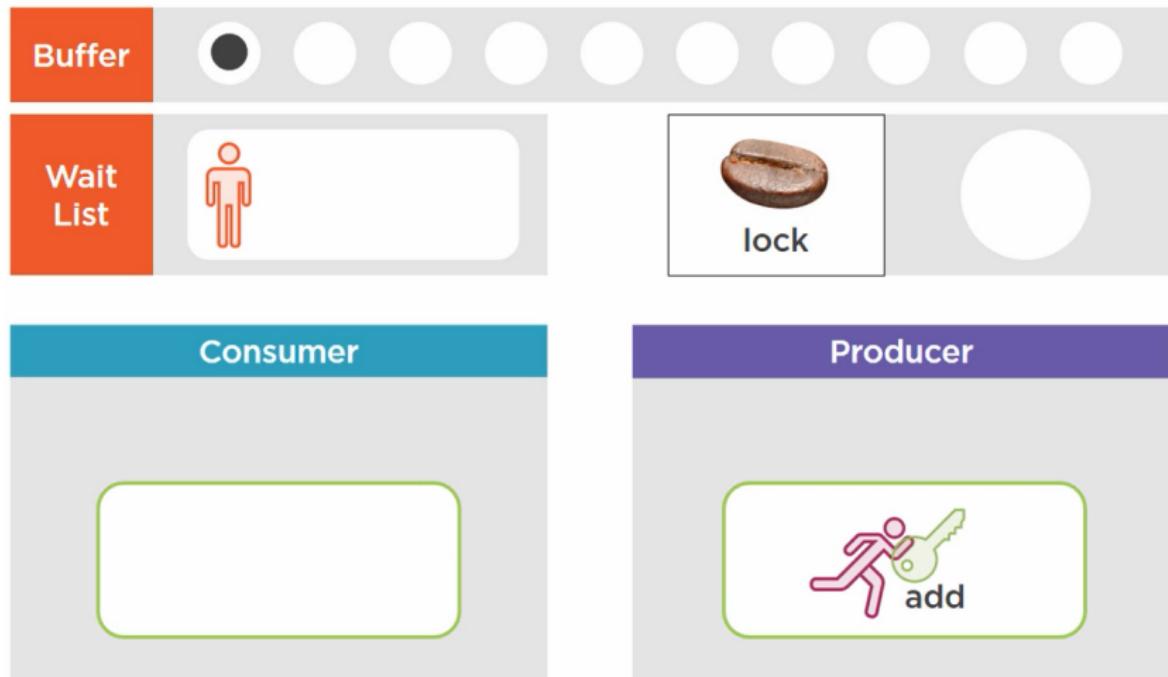
Fonctionnement du patron -suite-



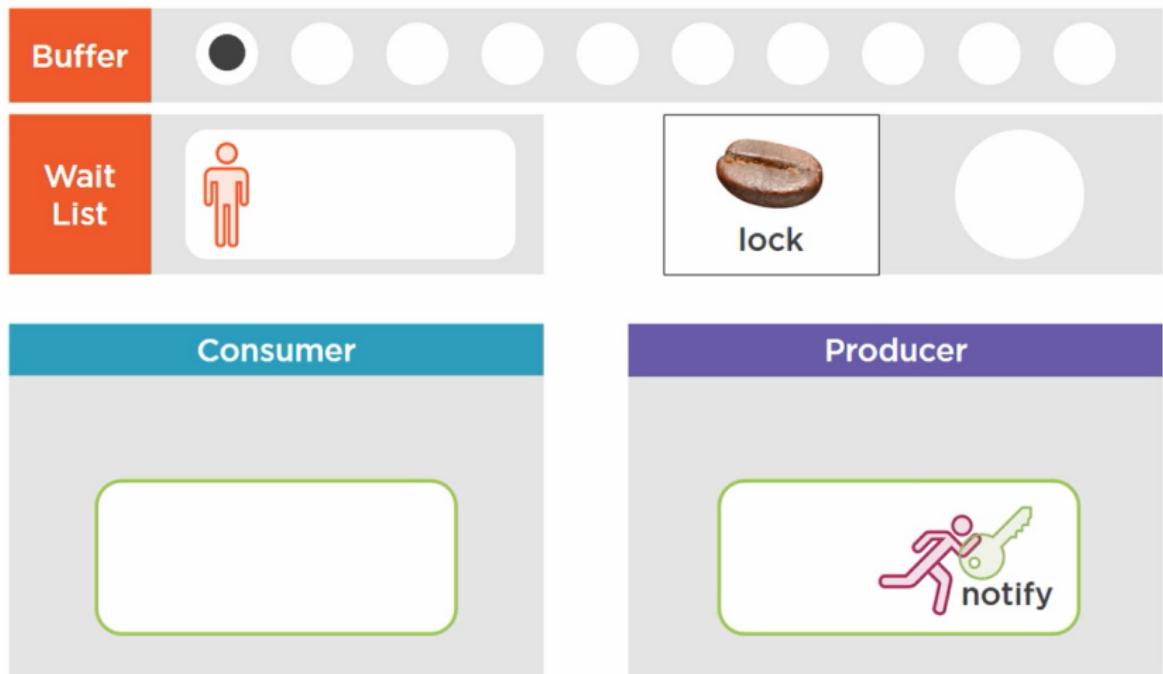
Fonctionnement du patron -suite-



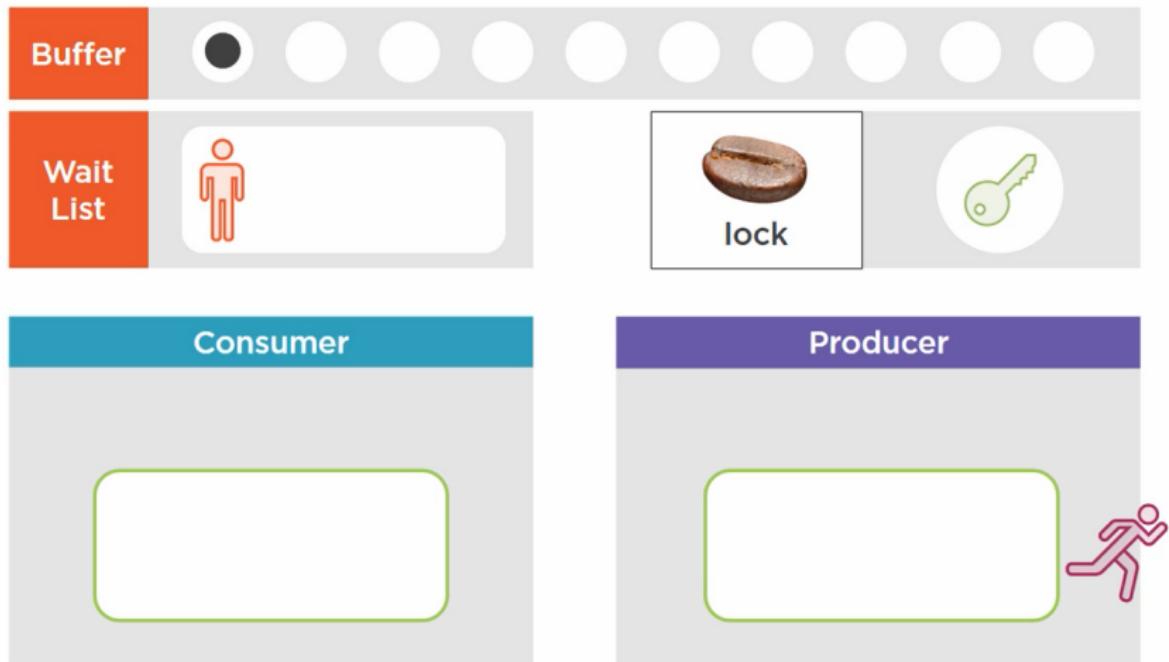
Fonctionnement du patron -suite-



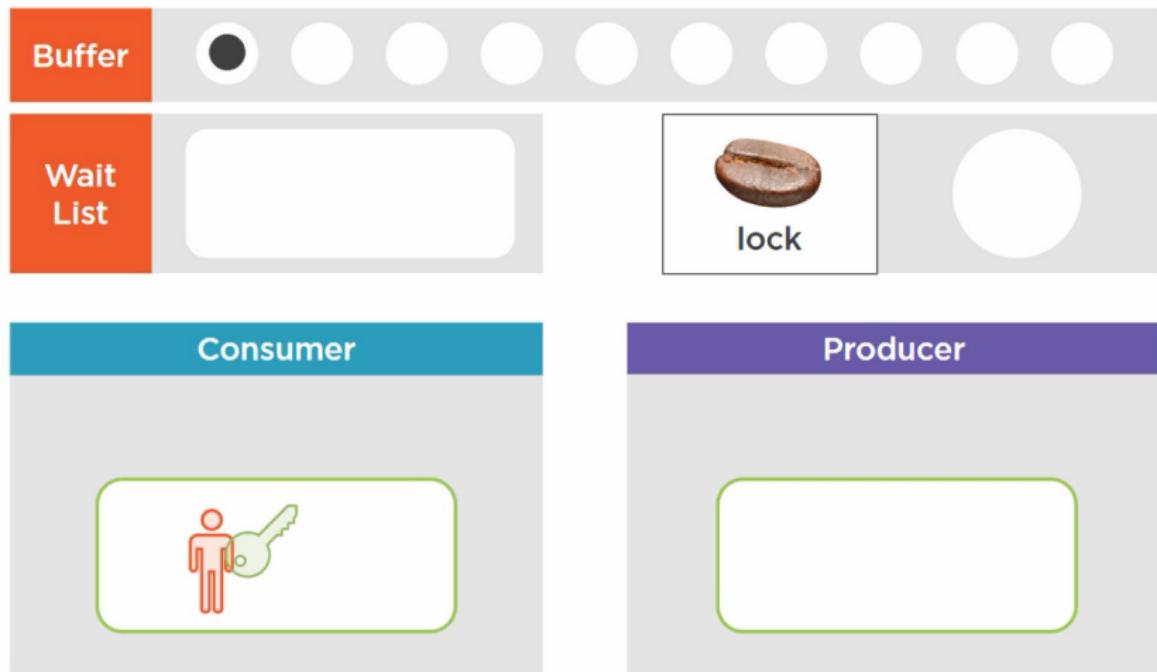
Fonctionnement du patron -suite-



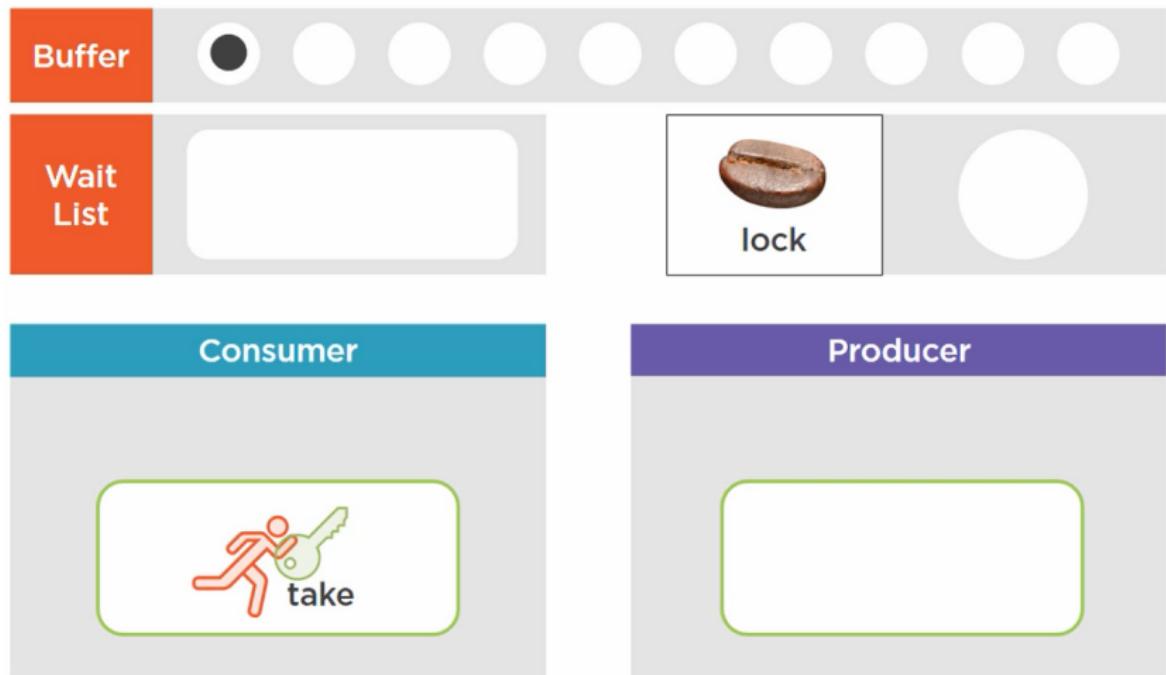
Fonctionnement du patron -suite-



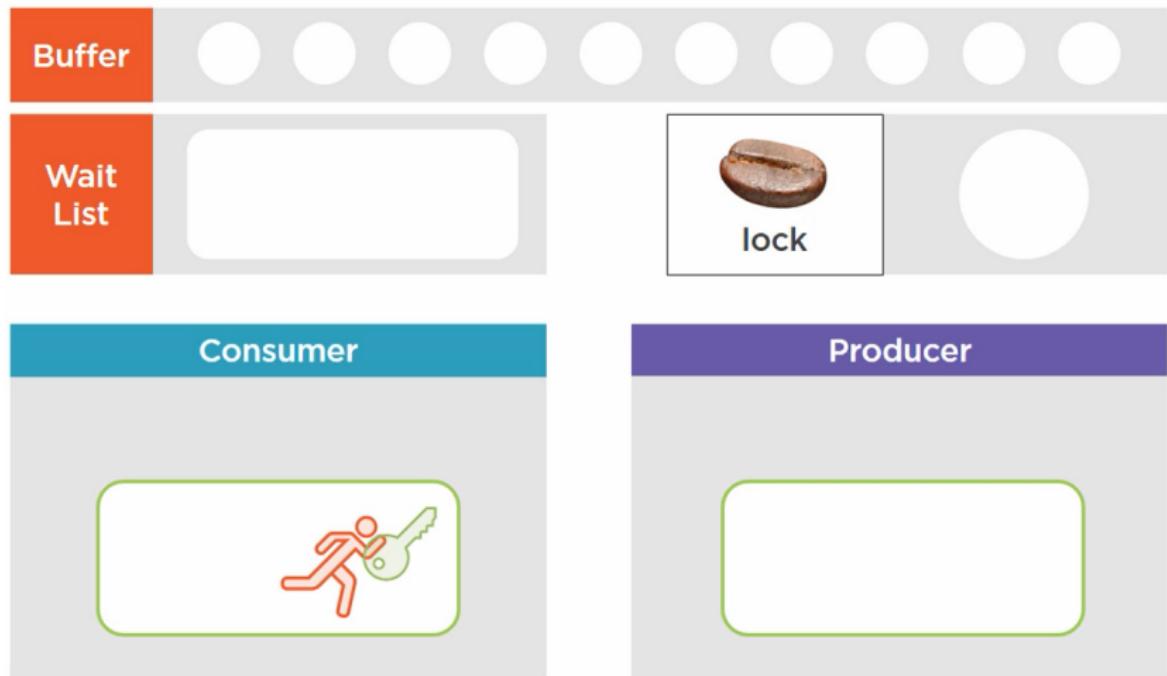
Fonctionnement du patron -suite-



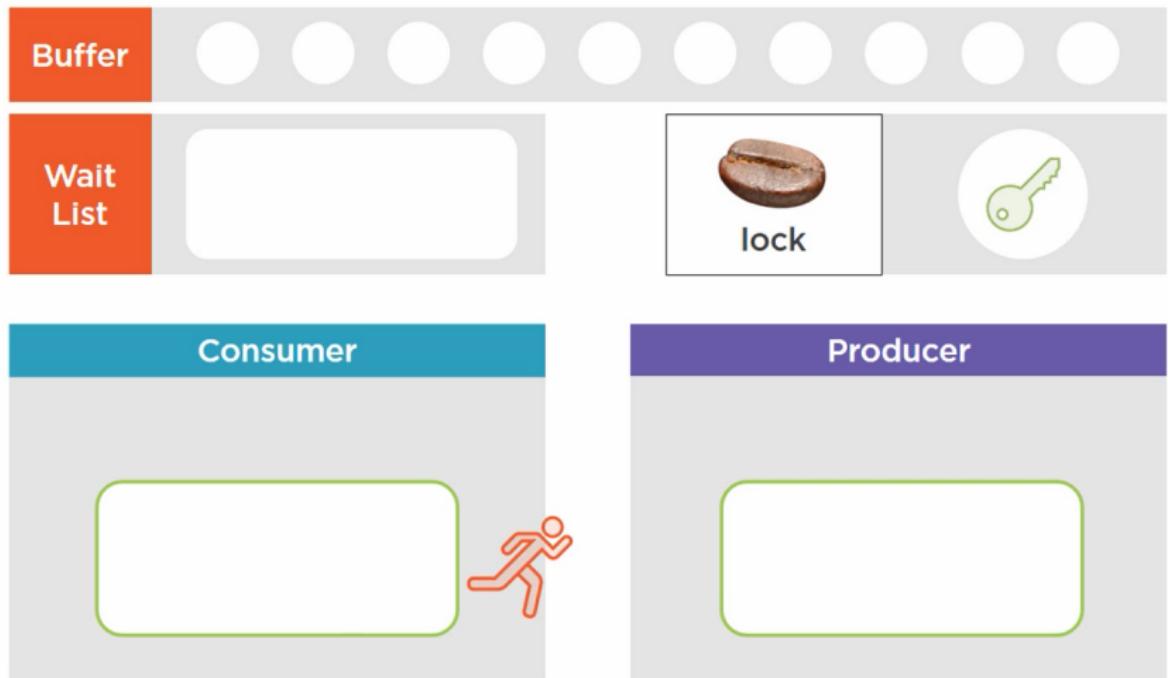
Fonctionnement du patron -suite-



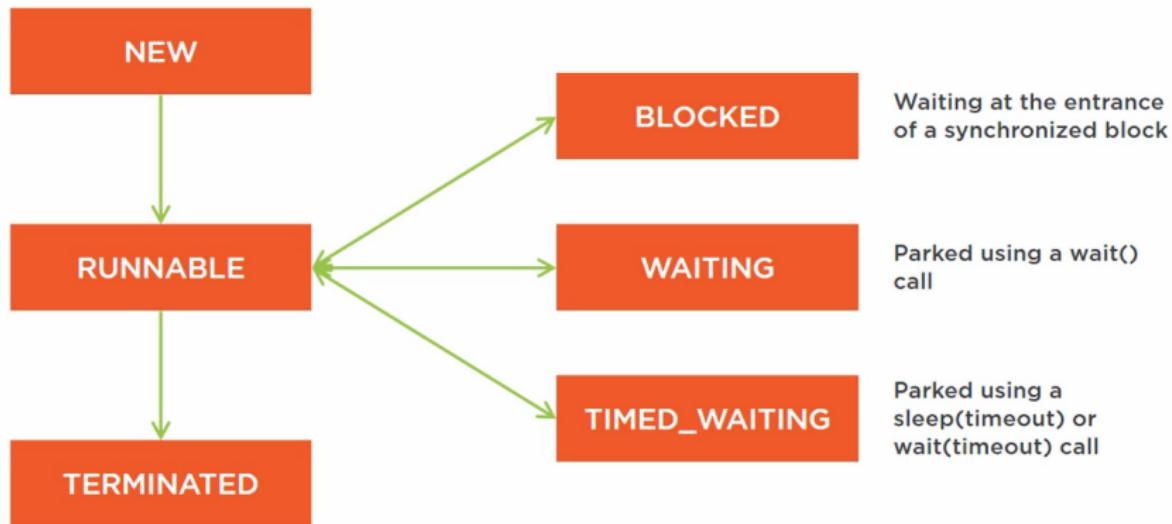
Fonctionnement du patron -suite-



Fonctionnement du patron -suite-



États des threads



Le scheduler ne sélectionne que les threads dans l'état Runnable pour leur attribuer du temps-processeur

Plan du cours

1. Notions de base sur la concurrence
2. Gérer les problèmes de synchronisation
 - 2.1 Problème du producteur/consommateur
 - 2.2 Utiliser le patron Wait & Notify
 - 2.3 Concurrence dans les processeurs multi-coeurs
 - 2.4 Étude de cas : le patron Singleton
3. Autres patrons pour la concurrence
 - 3.1 Executor, Future & Callable
 - 3.2 Verrous et conditions
4. Collections concurrentes

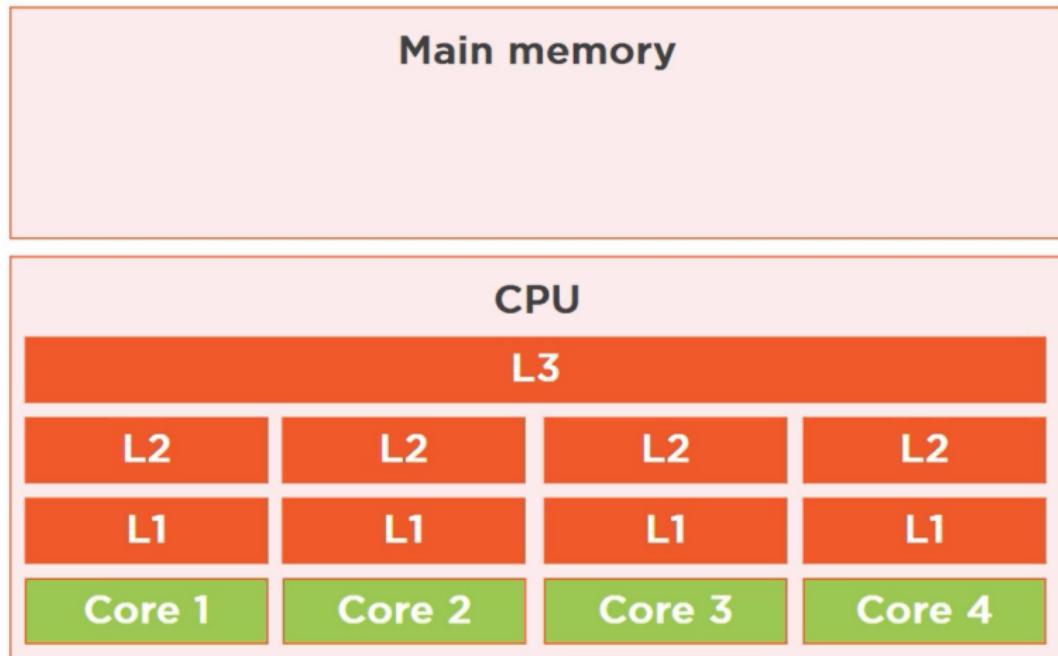
Limites de la synchronisation

- La synchronisation protège un bloc de code et garantit que ce code ne s'exécute que par un thread à la fois (permet de prévenir des *race conditions*)
- Dans certains cas ceci n'est pas suffisant, notamment dans des processeurs multi-coeurs
- Pour expliquer cela, reprenons l'exemple du producteur/consommateur
- Dans le consommateur, la méthode `consume()` :
 1. lit la valeur de l'attribut `count` depuis la mémoire
 2. le décrémente
 3. écrit la valeur de `count` vers la mémoire
- Le même schéma côté producteur : 1) lecture depuis la mémoire, 2) mise à jour puis 3) écriture vers la mémoire

Accès mémoire

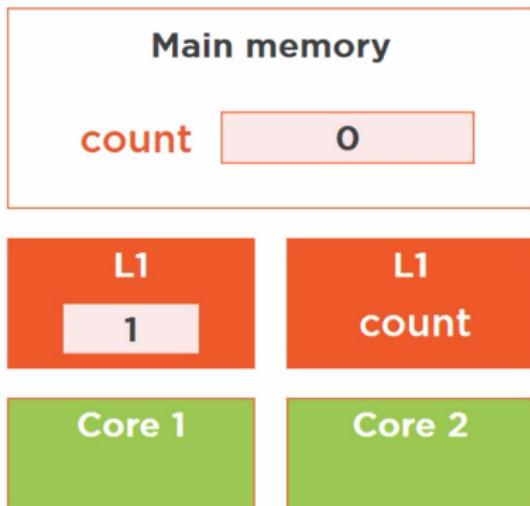
- 20 ans plus tôt, les processeurs n'avaient pas de caches
- Le code précédent fonctionne parfaitement bien
- Le processeur ne lit pas une variable depuis la mémoire (opération trop lente), il lit depuis son cache (mémoire interne)
- L'accès aux données est bien plus rapide, de l'ordre de fractions de nanosecondes, comparé aux centaines de ns pour la mémoire centrale
- Mais la mémoire centrale est bien plus importante en taille (plusieurs Go), contrairement aux caches des processeurs (32-256 Ko)

Architecture simplifiée classique d'un processeur



Le processeur possède plusieurs couches (*Layers*) de cache avec certaines couches spécifiques aux différents coeurs

Problème de visibilité



Core 1 needs count

- 1) The variable is copied in L1
 - 2) Core 1 can modify it
 - 3) Core 2 also needs count
 - 4) It should get the value 1, not 0
- This is visibility!

Visibilité

- Une variable est dite visible si les opérations d'écriture sur cette variable sont visibles
- Les opérations de lecture retourneront donc une valeur correcte
- Toutes les opérations d'écriture faite depuis un bloc synchronisé sont *visibles*

Le lien “happens-before” de Java et définition de la visibilité

- La notion de “happens-before” est une abstraction définie dans la spécification du langage Java (on y fait référence dans pas mal de javadocs)
- Elle permet de mieux comprendre l'ordre des opérations dans un processeur multi-coeurs
- **Visibilité** : une opération de lecture doit retourner la valeur mise en place par la **dernière** opération d'écriture
- Que veut dire le mot “**dernière** écriture” dans un processeur multi-coeurs, où le code s'exécute vraiment en parallèle (en même temps) ? on va voir cela sur une ligne de temps des opérations

Le modèle de mémoire de Java

- Supposons que le thread T1 écrit la valeur 1 dans un attribut x
- Supposons que le thread T2 lit la valeur de x et la stocke dans une variable r
- Que vaut r ?
- Le modèle de mémoire de Java (*Java Memory Model*, une partie de la spec du langage) répond de façon précise à cette question
- S'il n'y a pas de lien “happens-before” entre les deux opérations, alors la valeur de r est inconnue (ça peut être 1, ou bien 0 qui est la valeur par défaut d'un attribut de type nombre)
- Si ce lien existe entre les deux opérations, alors r vaut 1

Comment mettre en place des liens “happens-before” ?

- Pas de mot clé en Java
- **Règle** : un lien “happens-before” existe entre toutes les opérations d’écriture synchronisées ou *volatile*s et toutes les opérations de lecture synchronisées ou *volatile*s qui leur succèdent

Exemple d'opérations avec/sans liens

```
int index;  
  
void increment() {  
    index++;  
}  
  
void print() {  
    System.out.println(index);  
}
```

Two operations:

- 1) a write operation
- 2) a read operation

- Qu'affiche le print dans une app multi-threadée ?
- Sans synchronisation, ni attribut volatile, la réponse est impossible

Exemple d'opérations avec/sans liens -suite-

```
int index;  
  
void synchronized increment() {  
    index++;  
}  
  
void synchronized print() {  
    System.out.println(index);  
}
```

Two operations:

- 1) a write operation
- 2) a read operation

- Après avoir synchronisé les deux méthodes
- Il y a maintenant un lien “happens-before” entre l’écriture et la lecture
- La valeur correcte sera toujours affichée

Exemple d'opérations avec/sans liens -suite-

```
volatile int index;  
  
void increment() {  
    index++;  
}  
  
void print() {  
    System.out.println(index);  
}
```

Two operations:

- 1) a write operation
- 2) a read operation

- L'attribut est déclaré volatile (sa valeur doit être lue et écrite toujours en mémoire centrale)
- Il y a maintenant un lien “happens-before” entre l'écriture et la lecture
- La valeur correcte sera toujours affichée

Autre exemple plus complexe

```
int x, y, r1, r2;  
Object lock = new Object();  
  
void firstMethod() {  
    x = 1;  
    synchronized(lock) {  
        y = 1;  
    }  
}  
  
void secondMethod() {  
    synchronized(lock) {  
        r1 = y;  
    }  
    r2 = x;  
}
```

firstMethod() is writing x and y
secondMethod() is reading them

They are executed in
threads T₁ and T₂

Question: what is the value of r2?

- Il y a un lien HB entre x=1 et y=1, et entre r1=y et r2=x (ordre des instructions dans un même thread)

Autre exemple plus complexe -suite-

```
int x, y, r1, r2;  
Object lock = new Object();  
  
void firstMethod() {  
    x = 1;  
    synchronized(lock) {  
        y = 1;  
    }  
}  
  
void secondMethod() {  
    synchronized(lock) {  
        r1 = y;  
    }  
    r2 = x;  
}
```

If T₁ is the first to enter the synchronized block, then the execution is in this order:

- x = 1
- y = 1 Happens-before link between a synchronized write and a synchronized read
- r1 = y
- r2 = x

The value of r2 is 1

Autre exemple plus complexe -suite-

```
int x, y, r1, r2;  
Object lock = new Object();
```

```
void firstMethod() {  
    x = 1;  
    synchronized(lock) {  
        y = 1;  
    }  
}
```

```
void secondMethod() {  
    synchronized(lock) {  
        r1 = y;  
    }  
    r2 = x;  
}
```

If T_2 is the first to enter the synchronized block, then the execution is in this order:

- $r1 = y$

- $r2 = x$ or $x = 1?$

- $y = 1$

No happens-before link
between $r2 = x$ and $x = 1$

The value of $r2$ may be 0 or 1

Synchronisation vs Visibilité

Synchronisation

Elle garantit l'exécution exclusive d'un bloc de code

Visibilité

Elle garantit la cohérence des variables

Une variable visible \Rightarrow une opération de lecture va lire la valeur correcte de la variable

Règle d'or

Variables partagées

Toute variable partagée par plusieurs threads doit être accédée de façon synchronisée ou volatile

Le non-respect de cette règle conduit à un code qui comporte des bugs

False Sharing

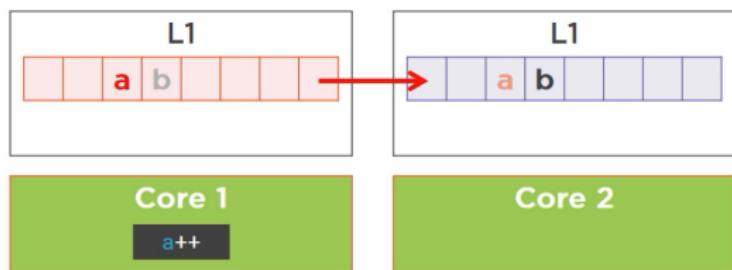
- L'architecture des processeurs avec caches apporte un défaut connu sous le nom de *False Sharing*
- Celui-ci peut avoir des effets négatifs sur les performances
- C'est quoi le *false sharing* ?
- Le cache est organisé en lignes de données et chaque ligne regroupe plusieurs octets de données (jusqu'à 64)
- Les variables se retrouvent groupées dans ces lignes et partagées par plusieurs caches de cœurs différents
- Une ligne est considérée comme "sale" (*dirty*) si une variable a été modifiée par l'un des cœurs
- La lecture d'une ligne sale déclenche le rafraîchissement de la ligne

False Sharing -suite-

```
volatile long a, b;
```

```
void firstMethod() {  
    a++;  
}
```

```
void secondMethod() {  
    b++;  
}
```



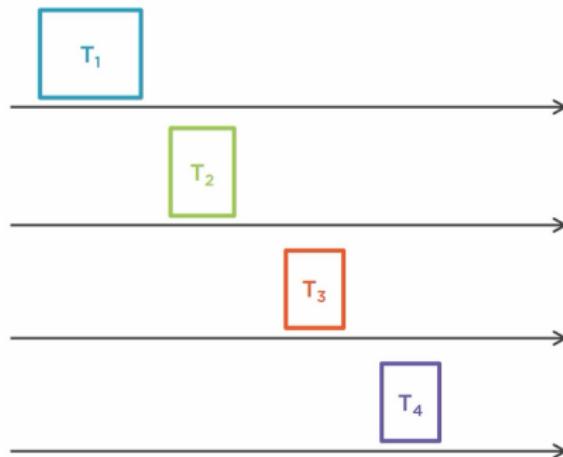
- Si dans le cœur 1 `firstMethod()` s'exécute (modifie donc a)
- Toute la ligne doit être rafraîchie dans le cache du cœur 2 (lors de la lecture de b, si l'on considère que dans le cœur 2 `secondMethod()` s'exécute), alors que ce n'est pas utile
- Des solutions à ce problème sont possibles par programmation (en déclarant, par ex, des variables supplémentaires autour de la variable juste pour compléter les lignes de cache)

Plan du cours

1. Notions de base sur la concurrence
2. Gérer les problèmes de synchronisation
 - 2.1 Problème du producteur/consommateur
 - 2.2 Utiliser le patron Wait & Notify
 - 2.3 Concurrence dans les processeurs multi-coeurs
 - 2.4 Étude de cas : le patron Singleton
3. Autres patrons pour la concurrence
 - 3.1 Executor, Future & Callable
 - 3.2 Verrous et conditions
4. Collections concurrentes

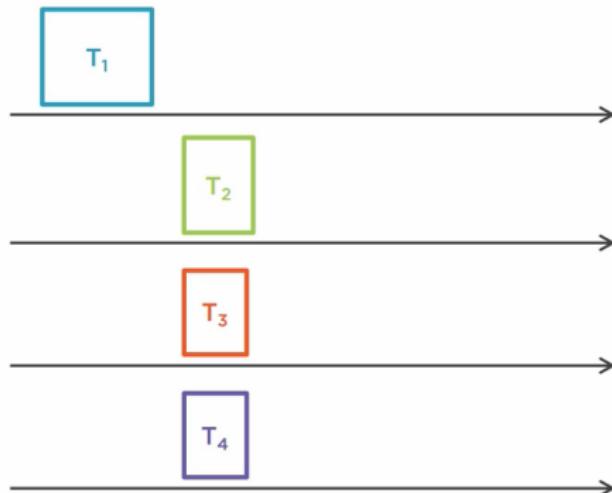
Patron Singleton correct (*thread-safe*) et efficace

- L'exemple vu en début du cours n'est pas encore au point



- Si on a 4 cœurs exécutant 4 threads qui invoquent tous getINSTANCE()
- Le premier qui s'exécute va créer l'instance et les autres vont tous devoir s'attendre mutuellement pour récupérer l'instance

Patron Singleton correct (*thread-safe*) et efficace



- Après la création de l'instance, on doit pouvoir faire les opérations de lecture en parallèle

Solution : Double Check Pattern

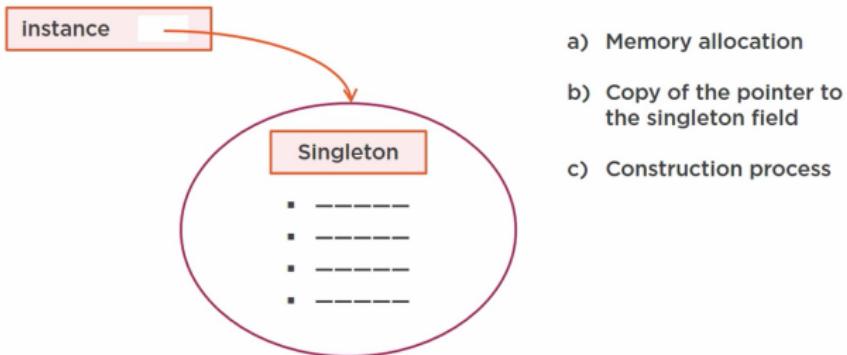
- Une double vérification de la présence de l'instance :

```
private static Object lock = new Object();
public static Singleton getInstance() {
    if(instance != null) {
        return instance;
    }
    synchronized(lock) {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Problème avec cette solution

- Nous avons une opération d'écriture synchronisée
- Mais l'opération de lecture (dans le premier if) n'est pas synchronisée
- L'attribut `instance` n'est pas volatile non plus
- A-t-on la garantie que l'opération de lecture récupère la référence mise en place par l'opération d'écriture ?
- Réponse : non, nous n'avons pas de lien “happens-before” entre les deux
- Ce code comporte un bug (visible uniquement dans les processeurs multi-cœurs)

Problème avec cette solution -suite-



- On n'est pas sûr que les étapes b) et c) s'exécutent dans cet ordre
- Du coup, si b) puis c) et si le thread qui crée l'instance est interrompu après b) et avant c) et si un autre thread récupère le pointeur (référence), il aura accès à une instance qui n'est pas encore construite

Encore une autre solution

- Rendre l'attribut volatile

```
public class Singleton {  
    private static volatile Singleton instance;  
    private Singleton() {}  
    private static Object lock = new Object();  
    public static Singleton getInstance() {  
        if(instance != null) {return instance;}  
        synchronized(lock) {  
            if (instance == null) {  
                instance = new Singleton();  
            }  
            return instance;  
        }  
    }  
}
```

Mais, ce n'est pas encore ça

- La solution précédente souffre de problèmes de performances comme vu précédemment
- En plus, on a des problèmes avec la serialisation et l'introspection avec Reflect (possibilité d'avoir plusieurs instances) :

<https://dzone.com/articles/java-singletons-using-enum>

- La solution est très simple : utiliser un enum comme décrit dans le tuto ci-dessus

Plan du cours

1. Notions de base sur la concurrence
2. Gérer les problèmes de synchronisation
 - 2.1 Problème du producteur/consommateur
 - 2.2 Utiliser le patron Wait & Notify
 - 2.3 Concurrence dans les processeurs multi-coeurs
 - 2.4 Étude de cas : le patron Singleton
3. Autres patrons pour la concurrence
 - 3.1 Executor, Future & Callable
 - 3.2 Verrous et conditions
4. Collections concurrentes

Plan du cours

1. Notions de base sur la concurrence
2. Gérer les problèmes de synchronisation
 - 2.1 Problème du producteur/consommateur
 - 2.2 Utiliser le patron Wait & Notify
 - 2.3 Concurrence dans les processeurs multi-coeurs
 - 2.4 Étude de cas : le patron Singleton
3. Autres patrons pour la concurrence
 - 3.1 Executor, Future & Callable
 - 3.2 Verrous et conditions
4. Collections concurrentes

Limites du patron Runnable/Thread

- Les threads sont créés à la demande, par l'utilisateur (développeur)
- On peut se retrouver dans certains applications avec plusieurs dizaines ou centaines de threads, et ceci alourdi grandement l'exécution
- Les threads dans ce patron sont tués à la fin de l'exécution de la tâche
- Les threads peuvent être des ressources chères à créer et à tuer à chaque fois par l'OS
- Ceci n'est pas le patron à utiliser dans une vraie application en production

Patron Executor

Amélioration de ce patron :

1. Créer un vivier (pool) de threads prêts à être utilisés et utiliser ce vivier
2. Passer une tâche au vivier de threads qui va se charger de l'exécuter

Besoins : 1) pouvoir créer un vivier de threads, 2) passer une tâche à ce vivier pour l'exécuter

Patron Executor -suite-

- Dans Java, le vivier de threads est une instance de type Executor :

```
package java.util.concurrent
public interface Executor {
    void execute(Runnable task);
}
```

- Plusieurs implémentations de cette interface sont fournies par le JDK
- Il existe une autre interface plus riche, ExecutorService, qui étend Executor avec des méthodes pour gérer plus finement le cycle de vie des tâches du vivier (ordonner leur arrêt, ...)

Créer un vivier de threads

- Utiliser la classe factory Executors (avec un 's' à la fin)
- Cette classe fournit environ 20 méthodes factory pour créer des viviers
- Exemple : crée un executor pour un seul thread

```
ExecutorService singleThreadExecutor =  
    Executors.newSingleThreadExecutor();
```

- Ce thread reste actif tant que l'executor est vivant
- Ensuite, on peut passer une tâche à cet executor, celle-ci sera exécutée
- Quand la tâche se termine, le thread n'est pas tué (il peut être réutilisé pour exécuter d'autres tâches)

Créer un vivier de threads -suite-

- Comment libérer les threads de ce vivier ?
- On éteint l'executor service avec la méthode shutdown()
- Les deux méthodes factory les plus utilisées sont :

```
ExecutorService singleThreadExecutor =  
    Executors.newSingleThreadExecutor();
```

- Pour créer un executor d'un seul thread
- Très utile pour la programmation réactive (tâches asynchrones non-bloquantes)

```
ExecutorService multipleThreadsExecutor =  
    Executors.newFixedThreadPool(10);
```

- Pour créer un vivier de n (10) threads

Autres types d'executor

- CachedThreadPool permet de créer des threads à la demande et les réutilise en les gardant pendant 60 sec. S'ils ne sont pas utilisés, il les détruit
 - Utile lorsque l'on a beaucoup de threads à exécuter pendant un temps assez court
- ScheduledThreadPool permet de programmer l'exécution de tâches dans le pool de threads dans le future :
 - méthode `schedule(task, delay)`
 - méthode `scheduleAtFixedRate(task, delay, period)`
exécute une tâche immédiatement après l'invocation puis après chaque `period`, sans s'arrêter
 - méthode
`scheduleWithFixedDelay(task, initialDelay, delay)`
exécute une tâche avec un délai au départ, ...

Exécuter une tâche dans le vivier

- Créer d'abord une tâche (Runnable) puis invoquer la méthode `execute()` :

```
ExecutorService singleThreadExecutor =  
    Executors.newSingleThreadExecutor();  
Runnable task =  
    () -> System.out.println("I run in a pool");  
singleThreadExecutor.execute(task);
```

- La tâche passée en argument est exécutée de manière asynchrone (la méthode `execute` fait un return)
- La tâche s'exécute dans le thread de l'executor en parallèle
- Le thread reste actif pour recevoir d'autres tâches à exécuter
- Si vous testez ce code, vous verrez que la JVM ne se termine pas (il faut invoquer `shutdown` à la fin)

Une file d'attente par vivier

- Chaque executor/vivier a une file d'attente
- Si vous exécutez plusieurs tâches :

```
ExecutorService singleThreadExecutor =  
    Executors.newSingleThreadExecutor();  
Runnable task1 = () ->  
    System.out.println("Task 1. I run in a pool");  
Runnable task2 = () ->  
    System.out.println("Task 2. I run in a pool");  
singleThreadExecutor.execute(task1);  
singleThreadExecutor.execute(task2);
```

Celles-ci vont s'exécuter dans l'ordre

- La 2ème tâche sera placée dans la file d'attente en attendant que le thread soit libéré de l'exécution de la 1ère tâche

Interroger sur l'état d'une tâche

- Est-il possible de savoir si une tâche s'est terminée ?
- Non, du moins avec Runnable
- Est-ce possible d'annuler une tâche ?
- Oui, si la tâche est dans la file d'attente

Limites des tâches Runnable

- Les tâches ne peuvent rien retourner
- Les tâches ne peuvent rien signaler comme exception
- Si la tâche implique des traitements susceptibles de lever des exceptions, comme des interactions avec une base de données, on doit capturer toutes les exceptions (try-catch) dans la tâche
- Parfois, on ne peut pas capturer les exceptions à ce niveau de l'application
- Il n'y a également aucun moyen de savoir si une tâche s'est terminée ou pas

Passer de Runnable à Callable

- Un nouveau modèle pour les tâches
- Des tâches qui peuvent exécuter du code qui :
 1. retourne un résultat
 2. peut lever une exception
 3. peut échanger des valeurs avec d'autres tâches
- Un objet qui agit comme un pont entre le thread principal et les threads du vivier/executor

Objets Callable

- Les besoins précédents sont satisfaits par les objets Callable :

```
package java.util.concurrent;
@FunctionalInterface
public interface Callable <V> {
    V call() throws Exception;
}
```

- L'interface Executor ne gère que des objets Runnable
- L'interface ExecutorService possède une méthode nommée submit qui prend en paramètre un objet Callable :

```
<T> Future <T> submit(Callable <T> task);
```

Cette méthode retourne un objet de type Future qui enveloppe le résultat

Les objets Future

- Une fois la tâche Callable est exécutée dans un thread de l'executor ou du vivier, le résultat (ou l'exception) est enveloppé dans un objet Future qui est renvoyé au thread appelant (main, par ex)

```
// Dans le thread principal (main) :  
Callable<String> task = () -> {  
    System.out.println("I'm a callable task");  
    return "Success";  
};  
Future<String> future =  
    singleThreadExecutor.submit(task);
```

submit renvoie immédiatement après un objet Future

- L'objet future contiendra le résultat lorsque la tâche aura terminé son exécution

Les objets Future -suite-

- On peut obtenir le résultat retourné par la tâche Callable en invoquant la méthode get()

```
System.out.println(future.get());
```

L'invocation de get() est bloquante

- Deux exceptions peuvent être levées ici :
 - InterruptedException si le thread de l'executor est interrompu (en invoquant shutdown sur l'executor)
 - ExecutionException si la tâche lève une exception (applicative). Cette dernière est enveloppée par l'objet ExecutionException par la méthode get()
- On peut passer en argument à get() une durée (timeout) au delà de laquelle une exception TimeoutException est levée

A vos claviers

- Tester l'utilisation d'une tâche callable qui afficher le message :
“I am a callable task that runs on Thread :
<current-thread-name>”
- Créer plusieurs tâches et vérifier que c'est le même thread qui les exécute
- Utiliser maintenant un FixedThreadPool de 5 threads et exécuter 10 tâches callable en utilisant ce vivier
- Que remarquez vous dans les noms des threads affichés ?

A vos claviers -suite-

- Tester la méthode get() en invocation simple, puis en invocation avec *timeout* (1 nano-secondes)
- Faire en sorte de provoquer un timeout
- La JVM dans ce dernier cas ne s'arrête pas proprement. Que faudra-t-il faire ? (ajouter un bloc try-finally autour de l'invocation de get())

Toujours s'assurer dans une application multi-threadée que la JVM se termine proprement à la fin de l'exécution

- Écrire maintenant une tâche qui lève une exception (IllegalStateException ou autre), l'exécuter avec un executor, puis faire un get sur l'objet future
- Expliquer ce qui se passe. Combien d'exceptions ? Décrire la stack trace

Éteindre l'executor

- Il est important d'éteindre un executor pour libérer proprement les ressources systèmes
- Trois façons de faire :
 1. méthode shutdown() vue précédemment pour éteindre l'executor une fois toutes les tâches (même celles dans la file d'attente) se sont exécutées/terminées (ne prend pas de nouvelles tâches à exécuter)
 2. méthode shutdownNow() arrête tout ce qui s'exécute et n'exécute pas les tâches en attente
 3. méthode awaitTermination(timeout) fait un shutdown() (c.à.d ne prend pas de nouvelles tâches à exécuter) et ensuite accorde un timeout aux tâches en cours d'exécution et celles en attente de s'exécuter

Plan du cours

1. Notions de base sur la concurrence
2. Gérer les problèmes de synchronisation
 - 2.1 Problème du producteur/consommateur
 - 2.2 Utiliser le patron Wait & Notify
 - 2.3 Concurrence dans les processeurs multi-coeurs
 - 2.4 Étude de cas : le patron Singleton
3. Autres patrons pour la concurrence
 - 3.1 Executor, Future & Callable
 - 3.2 Verrous et conditions
4. Collections concurrentes

Intrinsic & Explicit Locking

- Dans la section 2, on a vu comme utiliser synchronized et volatile pour synchroniser des threads
- Ceci est connu sous le nom de *Intrinsic Locking*
- Problème avec ce patron : interblocage (un thread bloqué dans un bloc synchronisé qui bloque d'autres threads)
- D'autres moyens sont possibles avec l'*Explicit Locking* en utilisant des objet de type Lock
- L'interface Lock offre une API riche pour gérer le problème d'inter-blocage

Exemple avec Lock

- Au lieu d'écrire cela :

```
Object lock = new Object();
synchronized (lock) {
    // Do something here ...
}
```

- On écrit ça :

```
Lock lock = new ReentrantLock();
try {
    lock.lock();
    // Do something here ...
}
finally {
    lock.unlock();
}
```

Interface Lock

- Interface introduite dans Java 5 (2004)
- Elle possède plusieurs implémentations, comme ReentrantLock
- Elle offre les mêmes garanties : exécution exclusive, et ordre des lectures/écritures
- Elle propose plus de fonctionnalités :
 - poser un verrou et être capable d'être interrompu par d'autres threads,
 - faire un essai d'obtenir le verrou : si un autre thread est en train d'exécuter le bloc, le thread courant n'est pas bloqué (if(lock.tryLock()) ... else ...)
 - demander l'obtention d'un verrou avec timeout
 - créer un verrou "juste" (le choix du thread bloqué à exécuter n'est pas fait par hasard, mais par ordre FIFO) : new ReentrantLock(true)

Producteur/Consommateur avec des locks

```
Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
Condition notEmpty = lock.newCondition();

class Producer {
    public void produce() {
        try {
            lock.lock();
            while (isFull(buffer))
                notFull.await();
            buffer[count++] = 1;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}

class Consumer {
    public void consume() {
        try {
            lock.lock();
            while (isEmpty(buffer))
                notEmpty.await();
            buffer[--count] = 0;
            notFull.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

Producteur/Consommateur avec des locks -suite-

- Dans l'exemple précédent, on a utilisé un autre type d'objets pour remplacer wait/notify : objets de type Condition
- Ces objets ont été utilisés pour mettre en pause des threads (méthode await()) et les réveiller (méthode signal())
- Ces objets fournissent des méthodes pour mettre en pause un thread : pendant un certain temps, jusqu'à une certaine date, ...

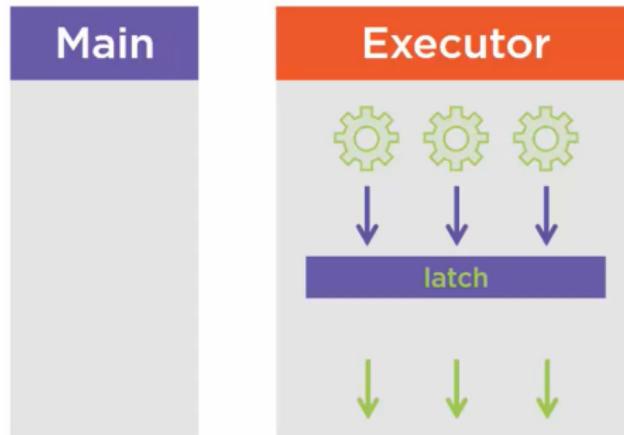
Nous pouvons également utiliser des verrous d'écriture et de lecture (readLock() & writeLock()) pour empêcher par exemple les écritures parallèles, mais pas les lectures parallèles

<https://www.baeldung.com/java-thread-safety#12-readwrite-locks>

Possibilité également d'utiliser des objets de type Semaphore (API équivalente à Lock mais avec n permissions possibles)

Autres objets pour gérer la concurrence

- Barrier (CyclicBarrier) : pour synchroniser des exécutions parallèles de plusieurs sous-tâches, attendre leur fin pour fusionner des données d'un grand dataset par exemple
- Latch (CountdownLatch) : une sorte de barrière mais qui ne se referme pas



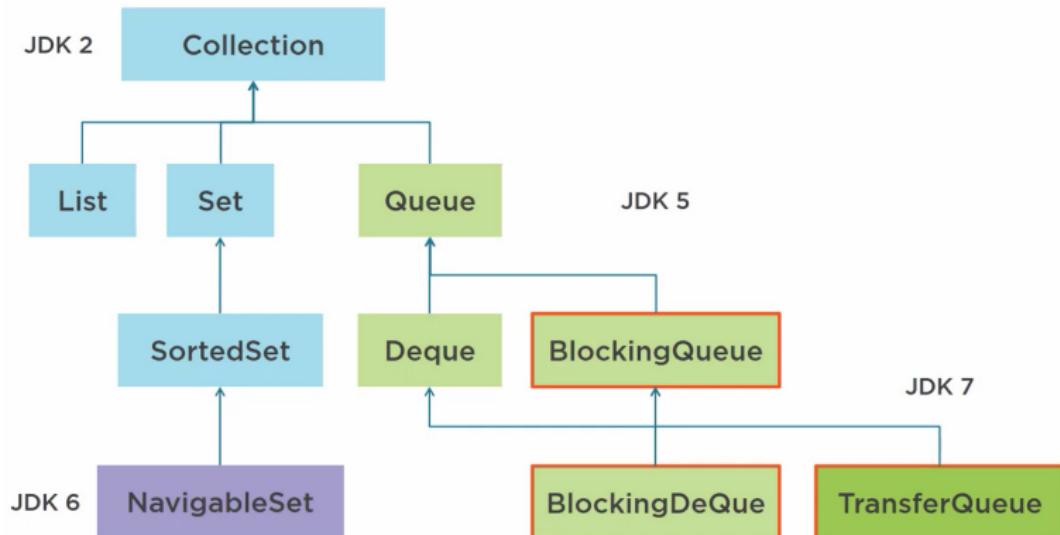
Plan du cours

1. Notions de base sur la concurrence
2. Gérer les problèmes de synchronisation
 - 2.1 Problème du producteur/consommateur
 - 2.2 Utiliser le patron Wait & Notify
 - 2.3 Concurrence dans les processeurs multi-coeurs
 - 2.4 Étude de cas : le patron Singleton
3. Autres patrons pour la concurrence
 - 3.1 Executor, Future & Callable
 - 3.2 Verrous et conditions
4. Collections concurrentes

Collections concurrentes du JDK

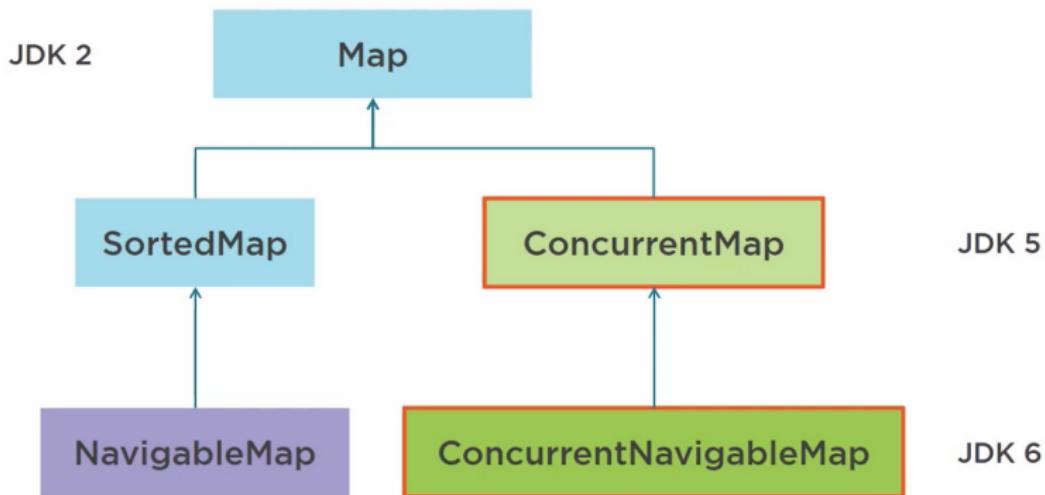
- Dans certaines applications multi-threadées, au lieu de gérer la synchronisation au niveau de l'application, on peut s'appuyer sur des collections "synchronisées" ou "concurrentes"
- Dans le JDK, il existe des moyens de créer des collections synchronisées :
`Collections.synchronizedCollection(uneCollection)`
- Mais, synchro de toute la collection ⇒ performances dégradées
- Utiliser plutôt des structures de données concurrentes (constitués de segments synchronisés)
- Deux sortes de structures de données concurrentes :
 - les collections
 - les maps

Les collections



- Les collections concurrentes sont à droite

Les maps



- Les collections concurrentes sont à droite

API concurrente pour ces structures de données

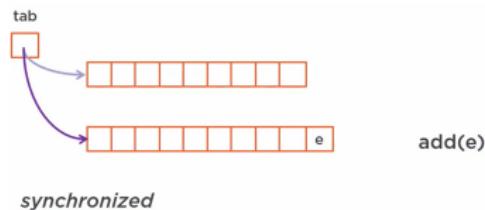
- Que veut dire API concurrente ?
- Elle définit un contrat garantissant un certain nombre de conditions dans un environnement concurrent
- Cela veut dire que les méthodes définies dans cette API sont *thread-safe* (garantissent un accès concurrent avec un comportement correct)
- Plusieurs implémentations sont fournies par le JDK
- Certaines sont capables de supporter un grand nombre de threads et d'autres non

Les listes concurrentes

- Vector et Stacks sont thread-safe
- Elles sont très mal implémentées (méthodes synchronisées de façon basique, avec synchronized) et peu efficaces
- A ne pas utiliser
- Dans du code légataire, il faut les remplacer

Les listes concurrentes -suite-

- Les structures copy-on-write list et set autorisent les lectures parallèles
- Pour une opération d'écriture, la liste/set est dupliquée et ensuite le pointeur est changé dans un bloc synchronisé
- CopyOnWriteArrayList :



- A utiliser quand il y a beaucoup de lectures et très peu d'écritures

Les files (*queues*) concurrentes

- Les interfaces queue (FIFO) et deque (*double-ended queue* ou files à double entrées, accessible depuis le début de la file (head) et la fin (tail)) servent à gérer des structures FIFO et LIFO
- Structures concurrentes :
 - `ArrayBlockingQueue` : une file bornée (nombre fixe de cellules) bloquante enveloppant un tableau
 - `ConcurrentLinkedQueue` : une file non-bornée bloquante

Les files (queues) concurrentes -suite-

- Une ArrayBlockingQueue : (ex file pleine)



```
boolean add(E e); // fail: IllegalArgumentException  
  
// fail: return false  
boolean offer(E e); boolean offer(E e, timeOut, timeUnit);  
  
// blocks until a cell becomes available  
void put(E e);
```

- 3 comportements possibles : exception levée, return false ou blocage jusqu'à la libération d'une cellule dans la file

Les files (queues) concurrentes -suite-

- Méthodes pour extraire un élément ou juste le lire :
 - poll() et peek() retournent null si la file est vide
 - remove() et element() lèvent des exceptions
 - take() de BlockingQueue est bloquante
- L'interface BlockingDeque : accepter des éléments dans le head et le tail
- addFirst(), offerFirst() non-bloquantes et putFirst() bloquante
- Méthodes pour extraire un élément ou juste le lire :
 - pollLast() et peekLast() retournent null si la file est vide
 - removeLast() et getLast() lèvent des exceptions
 - takeLast() de BlockingDeque est bloquante

Les maps concurrentes

- Deux implémentations : ConcurrentHashMap (JDK7 & redéfinie dans JDK8), ConcurrentSkipListMap (JDK6, sans synchronisation)
- Certaines méthodes sont atomiques :
 - `putIfAbsent(key,value)` test et ajout sans interruption
 - `remove(key,value)` test et suppression sans interruption
 - `replace(key,value)` et
`replace(key,existingValue,newValue)`

Les maps concurrentes -suite-

- Organisation spéciale de la map dans ces structures concurrentes : Dans JDK7, une hashmap concurrente était structurée en petits segments synchronisés pour autoriser les opérations parallèles (au lieu de bloquer l'accès à toute la map par une opération en cours)
- Nombre de segments = concurrency level (16, par défaut → 64k)
- Nombre de données doit être bien supérieur au nombre de segments

Les maps concurrentes -suite-

- Dans JDK8, la classe ConcurrentHashMap a été ré-implémentée pour qu'elle supporte des accès depuis des milliers de threads et des millions de paires clé-valeur
- Méthodes parallèles ajoutées, comme la recherche parallèle :

```
ConcurrentHashMap<Long , String > map =  
    new ConcurrentHashMap<>();  
String result =  
    map. search(10000,  
        (key , value)-> value . startsWith ("a")? "a" : null  
    );
```

- Le premier paramètre est le seuil de parallélisme (si plus de 10 000 paires clé/valeur, alors recherche parallèle)
- Le second paramètre est la méthode appliquée
- Autres méthodes de recherche : searchKeys(), searchValues(), ...

Opération parallèle de map/reduce

- Toujours dans ConcurrentHashMap, nous avons une méthode de réduction parallèle :

```
ConcurrentHashMap<Long, List <String>> map =  
    new ConcurrentHashMap<>();  
map. reduce(10000,  
    (key , value )->value . size () ,  
    (value1 , value2 )->Math . max( value1 , value2 )) ;
```

- Le second paramètre est une bifunction qui map les clés et valeurs
- Le troisième paramètre est une opération de réduction

Opération parallèle de parcours

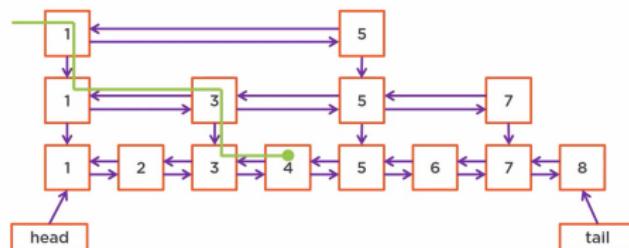
- Toujours dans ConcurrentHashMap, nous avons une méthode forEach :

```
ConcurrentHashMap<Long, List <String>> map =  
    new ConcurrentHashMap <>();  
map.forEach(10000,  
    (key, value) -> value.removeIf(s -> s.length() >  
        20));
```

- Le second paramètre est une biconsumer qui réalise une opération sur chaque paire clé-valeur
- Autres méthodes forEachKey, forEachValue, ...

Dernière structure concurrente

- ConcurrentSkipListMap (JDK6) basée sur une structure de données intelligente, une *skip list* (utilisée pour créer des listes chaînées efficaces, recherche en $O(\log(N))$ au lieu de $O(N)$)
- Une skip list : des couches de pointeurs au dessus de la liste chaînée donnant des accès rapides à des niveaux intermédiaires
- Exemple : rechercher 4



Dernière structure concurrente -suite-

- ConcurrentSkipListMap utilise une skiplist dans son implémentation
- Opérations thread-safe efficaces (environnements concurrents extrêmes)
- Il existe une autre structure dans le JDK implémentée avec une skiplist : ConcurrentSkipListSet
- Dans ce genre de structures concurrentes (utilisables par un très grand nombre de threads), il est recommandé de ne jamais appeler certaines opérations de consultation, comme size(), pour réaliser une opération d'E/S après

Exercice : Producteur/Consommateur

- Ré-écrire l'exemple avec une BlockingQueue (plus besoin de verrous pour synchroniser ou des wait/notify ou await/signal) :
 - Utiliser une ArrayBlockingQueue
 - Invoquer les méthodes bloquantes take() et put()
 - Les producteurs et consommateurs doivent être des classes qui implémentent l'interface Callable<String> : implémentent une méthode call()
 - Utiliser un FixedThreadPool de 5 threads
 - Créer plusieurs objets producteur et consommateur, que vous placerez dans une List<Callable<String> >
 - Exécuter cette liste dans le pool en invoquant la méthode invokeAll et en passant en argument la liste
 - Vous obtenez en retour une List<Future<String>> futures sur laquelle vous invoquer forEach() et pour chaque objet Future invoquer get()

Dernier exercice : ConcurrentHashMap

- Récupérer le fichier `movies-mpaa.txt` sur Moodle et le placer dans le dossier `resources` de votre projet IntelliJ
- Ce fichier recense une dizaine de milliers de films et leurs acteurs
- Récupérer les fichiers `.java` : `Actor`, `Movie` et `MovieReader`
- Dans la méthode `main()` de `MovieReader`, j'ai déclaré et initialisé une `ConcurrentHashMap` dans laquelle sont stockées des paires d'acteurs et leurs films
- En utilisant les méthodes parallèles précédentes, compléter la méthode `main` en affichant (seuil de parallélisme : 10) :
 - le nombre maximal de films joués par un acteur
 - le nom de cet acteur (avec le nombre maximal de films)
 - le nombre moyen de films dans lesquels tous les acteurs ont joué : on doit calculer le nombre total de films qu'on divise par la taille de la map (`map.size()`)

