

# CPS — Programmation par composants

© 2019- Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie  
Sorbonne Université

Jacques.Malenfant@lip6.fr

# Cours 5

## Composants concurrents en BCM

# Problématique de la concurrence

- Parallélisme et concurrence sont deux termes parfois utilisés *indifféremment* alors qu'ils recoupent des *perspectives différentes* :
  - Le *parallélisme* insiste sur l'exécution d'actions *en même temps* promettant une augmentation de la performance.
  - La *concurrence* insiste sur le fait que des actions *simultanées* accédant aux *mêmes données* peuvent provoquer de l'incohérence ou vont se gêner dans ces accès, d'où leur **concurrence**.
- Un parallélisme où la concurrence est maîtrisée résulte d'une bonne compréhension :
  - des mécanismes permettant de choisir entre *isoler* les données ou les *partager* entre plusieurs *threads* ;
  - des mécanismes de contrôle d'accès pour *assurer la cohérence* des données mutables partagées ;
  - des mécanismes restreignant le parallélisme pour assurer, par *exclusion mutuelle*, la cohérence des données et des calculs.

Principale source pour la préparation de cette séance : *Java Concurrency in Practice*, B. Goetz *et al.*, Addison-Wesley, 2006.

# Plan

- 1 Exclusion mutuelle et synchronisation
- 2 Gestion de la concurrence
- 3 La notion de sûreté de concurrence
- 4 Gestion du partage d'objets

# Objectifs de la séquence

## 1 Objectifs pédagogiques

- Comprendre la notion d'*entrelacement* pour analyser l'ordre *réel* dans lequel les instructions de différents *threads* parallèles sont exécutées en pratique.
- Comprendre le besoin d'*exclusion mutuelle* entre *threads* dans l'accès aux variables et aux ressources partagées.
- Comprendre comment ce besoin d'exclusion mutuelle est aussi lié à une autre notion importante : celle de *section critique*.

## 2 Compétences à acquérir

- Savoir identifier les sections critiques dans le code d'un composant.
- Savoir comment utiliser les mécanismes classiques de synchronisation, (sémaphore, moniteurs) et ceux offerts par Java (méthodes et blocs synchronisés, classes synchronisées) pour assurer l'exécution en exclusion mutuelle des sections critiques.

## Exemple classique : dépôt sur un compte bancaire

- Considérons l'opération de dépôt sur un compte bancaire :

```
public void depot(double montant) { this.solde += montant; }
```

- Après compilation, l'exécution de ce code se décompose en *opérations atomiques* (en première approximation<sup>1</sup>) :

```
temp1 = this.solde; temp2 = temp1+montant; this.solde = temp2;
```

- Maintenant, exécutons en parallèle deux appels `c.depot(2500.0)` et `c.depot(50.0)` ; n'importe quel entrelacement des opérations atomiques des deux *threads* devient possible, dont celui-ci :

| <code>this.solde</code> | <code>c.depot(2500.0)</code>         | <code>c.depot(50.0)</code>         |
|-------------------------|--------------------------------------|------------------------------------|
| 1000.0                  | <code>temp1 = this.solde;</code>     | —                                  |
| 1000.0                  | <code>temp2 = temp1 + 2500.0;</code> | —                                  |
| 1000.0                  | —                                    | <code>temp3 = this.solde;</code>   |
| 1000.0                  | —                                    | <code>temp4 = temp3 + 50.0;</code> |
| 3500.0                  | <code>this.solde = temp2;</code>     | —                                  |
| 1050.0                  | —                                    | <code>this.solde = temp4;</code>   |

À la fin, le solde n'a été augmenté que de 50.0...

<sup>1</sup> chacune exécutée en exclusion mutuelle avec toute autre, ce qui est le cas sur un processeur mono-cœur.

# Notion d'entrelacement des instructions

- Supposons deux *threads* A et B dont on numérote les opérations atomiques qu'ils exécutent A1, A2, ... et B1, B2, ...
- Alors un *entrelacement* est une séquence de ces opérations, par exemple A1, B1, B2, A2, B3, A3, A4, ...
  - En examinant l'effet de l'exécution d'un certain entrelacement des deux *threads* (comme dans l'exemple du dépôt), on peut vérifier si cet entrelacement produit un résultat cohérent ou non.
  - Pour analyser l'effet de l'exécution parallèle de A et B, on ne peut faire aucune hypothèse sur l'entrelacement qui sera exécuté ; de plus, l'entrelacement effectivement exécuté sera généralement différent *d'une exécution à l'autre* du programme.
- Pour garantir qu'un programme parallèle produise **toujours** un résultat cohérent, il faut s'assurer que **tous les entrelacements possibles** des opérations atomiques de ses *threads* parallèles vont produire un résultat cohérent (*i.e.*, programme *sûr*), et si possible le même (*i.e.*, programme *déterministe*).





# Mécanisme de base : l'instruction `testAndSet`

- A et B accèdent à la ressource sans réel contrôle parce que le test **et** l'affectation de `b` *ne forment pas une opération atomique* :
  - atomique : en exclusion mutuelle et non interruptible.
- La façon la plus simple pour résoudre ce problème est de fournir sur le processeur une instruction atomique `testAndSet` :
  - `testAndSet` : atomiquement vérifie la valeur d'une variable booléenne (en fait, un registre) *et* lui affecte faux si elle était vraie.
  - Assurer l'exclusion mutuelle par `testAndSet` oblige toutefois les *threads* à l'*attente active* :
    - **attente active** : tester répétitivement une condition jusqu'à ce qu'elle devienne vraie.
  - Les langages de programmation concurrente de haut niveau proposent des mécanismes plus évolués *bloquant* les *threads* jusqu'à ce qu'un autre *thread* qui, par exemple, a saisi la ressource, la libère avant de les débloquent, évitant l'attente active.
    - **attente passive** : se mettre en sommeil pour n'être réveillé que lorsqu'une condition devient vraie.

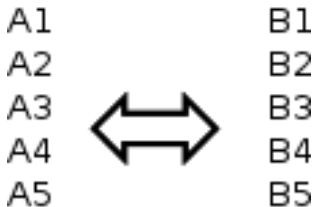
## Section critique et concept de sémaphore

- **Sections critiques** : séquences d'instructions qui doivent être exécutées en exclusion mutuelle les unes des autres, par rapport à une ressource qu'elles partagent.
- Pour implanter des sections critiques, il faut s'assurer que le *thread* qui acquiert le *droit d'y entrer* exclut qu'*aucun autre* puisse le faire avant qu'il ne quitte celle-ci.
- Un mécanisme simple inspiré des chemins de fer : le *sémaphore*.
  - Objet ayant deux opérations *atomiques* : `acquire` et `release`.
  - `acquire` tente d'acquérir le droit de passer et de fermer le sémaphore mais bloque le *thread* s'il est déjà fermé.
  - `release` rouvre le sémaphore, autorisant l'*un des threads* bloqués sur `acquire` ou le prochain à y parvenir, de l'exécuter (*i.e.*, acquérir le droit de passer et le refermer aussitôt).
- La classe Java `java.util.concurrent.Semaphore` permet de créer des sections critiques mutuellement exclusives par le partage d'un même objet sémaphore (voir la javadoc de cette classe).

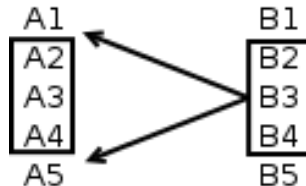
# Sections critiques et entrelacements possibles

- L'effet des sections critiques est de former des *blocs insécables* d'instructions ainsi que de *contraindre* les entrelacements *possibles* à voir ces blocs comme des opérations atomiques non modifiables.

Sans section critique



Avec sections critiques



# Sections critiques et entrelacements possibles

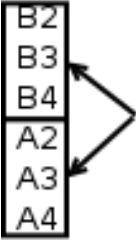
- L'effet des sections critiques est de former des *blocs insécables* d'instructions ainsi que de *contraindre* les entrelacements *possibles* à voir ces blocs comme des opérations atomiques non modifiables.

Sans section critique

A1  
B1  
B2  
A2  
B3  
B4  
A3  
B5  
A4  
A5

Avec sections critiques

A1  
B1  
B2  
B3  
B4  
A2  
A3  
A4  
A5  
B5



*sections critiques protégées*

# Un mécanisme plus structurant : les moniteurs

- **Moniteur** (*thread-safe monitor*) : mécanisme de synchronisation inventé pour protéger des types de données abstraits et qui ne permet qu'à un seul *thread* à la fois d'exécuter l'une de ses procédures ou fonctions.
  - Types de données abstraits : données masquées, accessibles uniquement via des procédures et fonctions publiques.
- Ainsi, les données n'étant pas accessibles autrement que par les fonctions ou procédures du moniteur, cela revient à assurer l'exclusion mutuelle entre les *threads* cherchant à les accéder à travers ses procédures ou fonctions.
  - En Java, l'équivalent d'un moniteur est obtenu en mettant toutes les méthodes d'une classe `synchronized` (y compris héritées).
  - En BCM, une autre façon de faire d'un composant un moniteur en n'attribuant qu'un seul *thread* à ce composant et en s'assurant de lui soumettre toutes les requêtes/tâches.
  - Un seul service (méthode) pouvant alors s'exécuter à la fois, il y aura *de facto* exclusion mutuelle ; c'est le premier mécanisme d'exclusion mutuelle de BCM, mais très limitatif.

# Quelles sont les sections critiques dans un composant ? I

- Certaines modifications *complexes* sur des variables *individuelles* de types primitifs (similaires à `testAndSet`, mais sur autre chose qu'un booléen) exigent un accès exclusif.
  - `java.util.concurrent.atomic` : classes d'objets atomiques de types primitifs (`AtomicBoolean`, `AtomicInteger`, ...).
    - Exemple : un compteur.

```
int compteur = 0;
compteur = compteur + 1; // opération non atomique
// utilisation de compteur
```

est remplacé par :

```
AtomicInteger compteur = new AtomicInteger(0);
int v = compteur.incrementAndGet(); // opération atomique
// utilisation de v
```
  - Les classes d'objets atomiques offrent des méthodes s'exécutant en exclusion mutuelle les unes avec les autres.
- Plus complexe à voir sont les *relations entre valeurs de plusieurs variables* qui doivent être maintenues lors de leur modification.

## Quelles sont les sections critiques dans un composant ? II

- Exemple : pile implantée avec un tableau et un indice donnant la position de son sommet.

```
Object[] pile = new Object[MAX_SIZE];  
int sommet = -1; // position du dernier élément empilé
```

- Cette représentation d'une pile suppose que dès qu'on empile ou on dépile une valeur, cette valeur soit ajoutée au ou retirée du tableau `pile` *ET* que la valeur de `sommet` soit mise à jour.

```
public void empiler(Object o) {  
    sommet++; pile[sommet] = o;  
}  
  
public Object depiler() {  
    Object o = pile[sommet]; sommet--;  
    return o;  
}
```

- De telles séquences de modifications assurant la cohérence d'une représentation doivent être faites en exclusion mutuelle avec tout autre code accédant à ces données.
- Elles forment donc un bon exemple de section critique.

# Généralisation : notions d'invariant I

- Toute représentation liant plusieurs données exige que des relations entre ces données soient maintenues dans la durée. C'est la notion *d'invariant*.
  - Littéralement, un invariant est une propriété (*e.g.*, expression logique) devant être *vraie* et *ne pas varier*.
- Avec objets et composants, on distingue deux types d'invariants :
  - des invariants en *boîte noire* dont l'expression n'utilise que l'interface externe (méthodes visibles, types abstraits) ;
  - des invariants *de représentation* portant aussi sur les variables internes, dépendant donc des choix d'implantation, et dont l'expression nécessite un accès à celles-ci (sémantique axiomatique)).
- Les invariants sont souvent présentés en programmation comme étant *tout le temps vrais*, mais c'est un peu simpliste.
  - Dans les langages comme Java, on ne peut modifier les variables qu'une par une par l'affectation.
  - Pour modifier des variables liées par un invariant, cet invariant *doit* être invalidé *pendant* qu'on leur affecte de nouvelles valeurs. ▶



# Généralisation : notions d'invariant II

- Dans ces conditions, quand un invariant doit-il être vrai ?
  - Réponse simpliste : tout le temps, sauf pendant l'exécution de l'une ou l'autre des méthodes de l'objet ou du composant.
    - Mais non, c'est plus complexe en cas d'appel en retour (*call back*) : une méthode m1 du composant A suspendue pour appeler le composant B qui rappelle une méthode m2 de A...
  - Réponse plus juste : il doit être vrai tout le temps, sauf quand on exécute du code dans l'objet ou le composant.
    - Mais, là encore, cette réponse est insuffisante si plusieurs *threads* peuvent s'exécuter en parallèle dans l'objet ou le composant et donc pourraient violer l'invariant en même temps.
  - Meilleure réponse : tout le temps, sauf pendant l'exécution de séquences d'instructions qui *violent l'invariant i.e.* celles qui forment des *sections critiques* :
    - à exécuter atomiquement et en exclusion mutuelle avec tout code accédant aux données liées par cet invariant ;
    - l'invariant concerné est vrai *avant et après* la section critique mais peut être faux *pendant* l'exécution de celle-ci.

# Plan

- 1 Exclusion mutuelle et synchronisation
- 2 Gestion de la concurrence**
- 3 La notion de sûreté de concurrence
- 4 Gestion du partage d'objets

# Objectifs de la séquence

## 1 Objectifs pédagogiques

- Comprendre la relation entre exclusion mutuelle et les mécanismes de synchronisation pour la mise en œuvre de la cohérence des données malgré des accès concurrents.
- Comprendre le rôle particulier des *verrous intrinsèques* de Java dans la cohérence des accès concurrents aux objets.

## 2 Compétences à acquérir

- Savoir utiliser les mécanismes standards de Java pour implanter des sections critiques dans les composants BCM : verrous, sémaphores, *etc.*
- Savoir utiliser à bon escient les verrous intrinsèques et les verrous explicites en Java et en BCM.
- Savoir partitionner les invariants, pré- et postconditions d'une entité logicielle pour réduire les contraintes de synchronisation tout en préservant la cohérence des données.

# Granularité de l'exclusion mutuelle

- Rappel : en programmation concurrente, l'exclusion mutuelle protège la cohérence des données exprimée par les invariants.
- Le principe même de la programmation par objets étant de concevoir les classes autour d'un invariant, gérer l'exclusion mutuelle se fait donc d'abord au *niveau de la classe* (composant).
  - Cette observation explique l'importance du mécanisme des méthodes `synchronized` en Java (voir ci-après).
  - De nombreuses classes de la bibliothèque standard de Java gère leurs accès concurrents selon cette approche.
- Mais si le niveau de *granularité* de la classe est souvent trop gros par rapport à l'exclusion mutuelle *nécessaire*.
  - On peut alors *partitionner* les structures de données par *sous-invariants indépendants* et les protéger par des synchroniseurs *distincts* que devront acquérir leurs sections critiques.
  - Lorsque les sous-invariants ne partitionnent pas les données (des chevauchements existent), chaque section critique touchant à plusieurs sous-invariants à la fois devra acquérir les synchroniseurs de chacun de ces derniers.

# Verrouillage et verrous explicites

- À la base, un verrou simple est un synchroniseur qui :
  - peut être saisi (*lock*), détenu, puis relâché (*unlock*) par un *thread* ;
  - à tout instant, ne peut être saisi que par au plus un *thread* ;
  - ne peut être relâché que par le *thread* qui l'a saisi ;
  - donc similaire à un sémaphore binaire sauf que le déverrouillage doit être fait par le *thread* qui l'a verrouillé.
- Verrouillage par *thread* versus par appel :
  - en verrouillage par *thread*, un verrou déjà saisi et encore détenu par un *thread* peut être « resaisi » (appel à `lock`) par ce même *thread* ;
  - en verrouillage par appel, un verrou saisi par un appel à `lock` ne peut être resaisi tant qu'il n'a pas été relâché, y compris par le *thread* qui le détient.
- Réentrant versus non réentrant : un verrou fonctionnant en verrouillage par *thread* est dit *réentrant*, alors que s'il fonctionne par appel, il est *non réentrant*.

# Protection d'invariants par verrous et sections critiques I

- 1 Toute variable mutable partagée par plus d'un *thread* doit voir *tous* ses accès protégés par *un même et unique verrou* pour tous les *threads*.
- 2 Pour chaque (sous-)invariant qui implique plus d'une variable mutable partagée, tous les accès en lecture *ET* en écriture à toutes ces variables doivent être protégés par *le même verrou* ; ce verrou doit être *clairement documenté*.
- 3 Toute section critique modifiant une ou plusieurs variables mutables partagées couvertes par plus d'un sous-invariant doit saisir *tous les verrous* protégeant *tous ces sous-invariants* à son entrée et ne les relâcher qu'à sa sortie.
- 4 Dans le choix des opérations à inclure dans les sections critiques, il faut établir un *compromis* entre garantir la simplicité et la sûreté d'une part et augmenter le parallélisme et la performance potentielle. Il ne faut *jamais* sacrifier la sûreté à la performance.

# Protection d'invariants par verrous et sections critiques II

- 5 Comme les fautes (exceptions) et les bloquages dans les sections critiques peuvent entraîner une détention d'une durée indéfinie de son verrou :
- les calculs longs et les opérations à suspension (entrées/sorties, appels distants synchrones, etc.) sont à *éviter* autant que possible dans les sections critiques ;
  - tout verrou acquis explicitement doit *toujours* être relâché même si une exception est lancée (*i.e.*, dans une clause `finally` d'un `try`).
- 6 En BCM4Java, un verrou ne doit *jamaïs* être visible à l'extérieur d'un composant (à moins que ce soit entre composite et sous-composants).

Les synchronisations inter-composants doivent donc :

- soit être *encapsulées* au sein du composant détenant les données mutables protégées derrière les appels de ses services (ex.: composant table de hachage concurrente),
- soit se faire via un *composant de synchronisation* partagé (ex.: composant sémaphore).

# Types de verrous explicites en Java

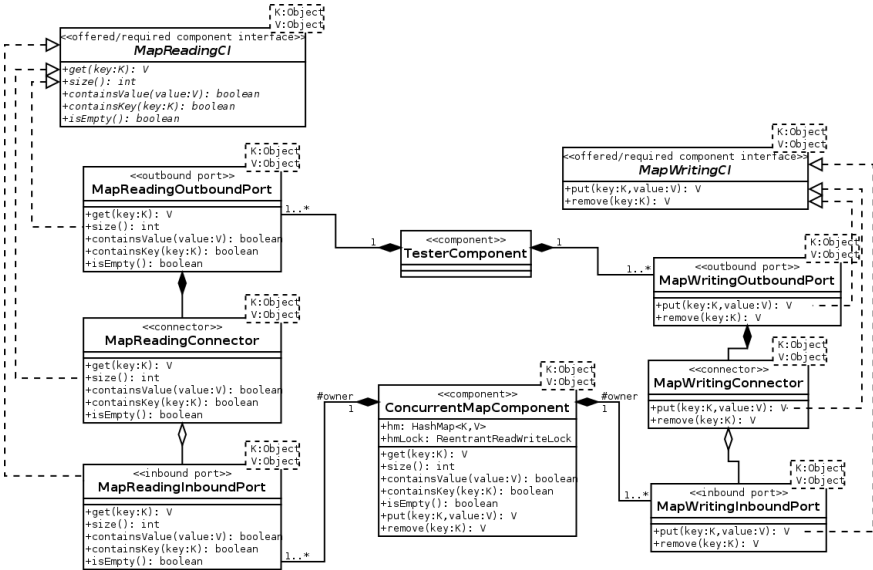
- Les principales classes de verrous de Java sont :
  - `ReentrantLock` : simple verrou réentrant.
  - `ReentrantReadWriteLock` : associe deux verrous pour gérer de manière croisée des accès en lecture et en écriture.
  - `StampedLock` : généralisation du « read/write lock » permettant de convertir un verrou déjà acquis (ex.: transformer dynamiquement un verrou acquis en lecture en verrou en écriture).
- Les verrous proposent souvent deux formes d'acquisition :
  - ❶ l'acquisition bloquante : (ex.: `lock`) si le verrou ne peut être acquis, le *thread* est bloqué jusqu'à ce que le verrou soit libéré ;
  - ❷ l'acquisition non bloquante : (ex.: `tryLock`) si le verrou ne peut être acquis, l'acquisition échoue mais le *thread* peut continuer son exécution.
  - Utiliser l'acquisition non bloquante pour retester en boucle est une *très très mauvaise* utilisation des ressources ! (attente active)
  - Le cas d'utilisation *acceptable* (unique) de l'acquisition non bloquante est de permettre d'exécuter *une action alternative* à la section critique en cas de non acquisition.



## Exemple : notre table de hachage concurrente

- Considérons une table de hachage, avec ce cahier des charges particulier :
  - les opérations en écriture (ajouts et retraits) s'exécutent en exclusion mutuelle entre elles et avec toute autre opération ;
  - les opérations en lecture seule s'exécutent en exclusion mutuelle avec celles en écriture ;
  - les opérations en lecture seule ne s'exécutent pas en exclusion mutuelle, un nombre quelconque de *threads* peuvent exécuter ces opérations en parallèle ;
  - on peut autoriser l'entrée d'autant de *threads* en lecture que souhaité mais cela ne doit jamais retarder indéfiniment le traitement des ajouts et des retraits.
- Pour répondre à ce cahier des charges, nous utilisons un `ReentrantReadWriteLock` de Java qui possède exactement les caractéristiques requises.

# Diagramme UML



# Mécanisme `synchronized` de Java I

- Des verrous intrinsèques de type `ReentrantLock` sont associés à chaque objet mais aussi à chaque classe.
  - Les méthodes d'instance `synchronized` appelées sur un objet donné utilisent le verrou intrinsèque de cet objet pour s'exécuter en exclusion mutuelle les unes par rapport aux autres.
  - Les méthodes statiques `synchronized` utilisent le verrou intrinsèque de leur classe de définition.
  - Lorsqu'une méthode `synchronized` est exécutée, le verrou correspondant est saisi automatiquement au début puis relâché à la fin de l'exécution de cette méthode.

- Bloc `synchronized` : séquence d'instructions avec accès exclusif par rapport à un objet désigné explicitement :

```
synchronized (this) { // utilise le verrou de l'objet courant (this)
    // séquence d'instructions protégée (section critique)
}
```

- Le verrou intrinsèque de l'objet référencé est saisi puis relâché automatiquement par le *thread* exécutant le bloc.

# Mécanisme `synchronized` de Java II

- Le bloc peut utiliser n'importe quel verrou intrinsèque, pas que celui de « `this` » ; gare à la confusion car le code protégé n'est pas nécessairement celui de l'objet dont on a saisi le verrou.
- Le mot-clé `synchronized` ajouté à la signature d'une méthode fait en sorte qu'elle soit considérée comme un bloc `synchronized` sur l'objet sur lequel elle s'exécute.
- Les verrous intrinsèques sont *réentrants*, ce qui permet les appels entre méthodes et les exécutions de blocs `synchronized` d'un et sur un même objet (y compris les appels récursifs).
- La facilité avec laquelle on peut se référer à l'objet exécutant un bout de code (`this`) (ou à l'instance de `Class<?>` pour le code statique) explique à la fois les choix de conception de Java et la grande utilisation des verrous intrinsèques dans les programmes.

# La notion d'équité

- Dans l'utilisation des mécanismes de synchronisation bloquant, comme les verrous, la notion d'équité (*fairness*) dans l'acquisition des verrous peut prendre une grande importance.
  - Équité : qualité d'un mécanisme de synchronisation consistant à assurer des chances égales d'acquisition à tous les *threads*.
  - Plusieurs algorithmes concurrents nécessitent l'équité pour être prouvés corrects et donner le bon résultat.
- Assurer l'équité n'est pas aussi simple qu'il n'y paraît.
  - Comment empêcher un *thread* qui tente en boucle une acquisition non bloquante de se montrer plus rapide que les autres *threads* ?
  - Il faut souvent alourdir les mécanismes et parfois interagir avec l'ordonnanceur pour assurer l'équité, ce qui est coûteux en termes de performance.
- Pour ces raisons, assurer ou non l'équité est souvent laissé au choix du programmeur, par des paramètres à la création.
- En Java, sans le garantir absolument, un verrou *équitable* cèdera au *thread* qui a attendu le plus longtemps lors d'un `unlock`.

# Interblocages entre *threads*

- Un grand problème fondamental de la synchronisation entre *threads* est l'*interblocage* (*deadlock*).
  - Nous avons vu une forme d'interblocage par famine de *threads*. Ici, nous allons voir une seconde forme d'interblocage dû aux mécanismes de synchronisation.
- Le prototype de cette forme d'interblocage se produit lorsque deux *threads* cherchent à se saisir des deux mêmes verrous et se bloquent l'un l'autre :

```
ReentrantLock l1 = new ReentrantLock();
```

```
ReentrantLock l2 = new ReentrantLock();
```

*Thread A*

```
l1.lock();
```

```
l2.lock(); // bloqué !
```

```
// code protégé
```

*Thread B*

```
l2.lock();
```

```
l1.lock(); // bloqué !
```

```
// code protégé
```

# Comment éviter cette forme d'interblocage ?

- Cet interblocage survient lorsque des *threads* distincts :
  - doivent acquérir plusieurs verrous ;
  - lorsqu'ils les acquièrent selon des ordres *différents*.
- Les deux moyens les plus simples pour éviter l'interblocage sont :
  - 1 Utiliser un seul verrou pour contrôler l'accès à toutes les ressources. C'est une solution intéressante mais peu modulaire et potentiellement peu performant.
  - 2 S'assurer que tous les *threads* acquièrent *toujours tous les verrous* exactement dans le même ordre (en anglais, cette technique est appelée *consistent lock acquisition ordering*).
    - Ainsi, le premier *thread* qui va acquérir un verrou va bloquer tous les autres sur ce verrou jusqu'à ce qu'il le relâche, les empêchant d'acquérir d'autres verrous dont il aurait besoin.

Cette solution n'est pas une garantie totale en Java car il est possible de construire des scénarios où les interactions sémantiques entre l'acquisition des verrous, l'ordonnancement des *threads* et les actions comme `wait` peuvent amener à des interblocages même avec une acquisition ordonnée des verrous. À l'intérieur d'un composant BCM, ces scénarios paraissent toutefois peu crédibles en général.

# Synchroniseurs en Java

- Outre les variables futures et les verrous, plusieurs autres formes de synchronisation entre *threads* peuvent s'avérer utiles.
  - Ex.: un algorithme parallèle lance plusieurs tâches parallèles dans une boucle, tâches devant toutes être synchronisées avant de faire le tour de boucle suivant.
- Autres synchroniseurs explicites de Java :
  - Loquets (*latch*) : il est créé dans un état initial puis évolue vers un état final où il libère les *threads* bloqués sur lui.
    - Ex.: `CountDownLatch` : créé avec une valeur initiale décrémentée à chaque appel à `countDown`; les *threads* bloqués en appelant `await` sont libérés lorsque le compte arrive à 0.
  - Barrières : elles créent des rendez-vous pour des *threads* qui s'y joignent et bloquent graduellement puis reprennent leurs exécutions lorsque tous sont arrivés (et peuvent recommencer).
    - Ex.: `CyclicBarrier` dont une instance est créée avec un nombre prédéfini de *threads* devant se joindre cycliquement au rendez-vous ; les *threads* signalent leur arrivée par `await`.



# Plan

- 1 Exclusion mutuelle et synchronisation
- 2 Gestion de la concurrence
- 3 La notion de sûreté de concurrence**
- 4 Gestion du partage d'objets

# Objectifs de la séquence

## 1 Objectifs pédagogiques

- Comprendre les notions de *sûreté de concurrence* et d'entité logicielle à *concurrence sûre*.
- Comprendre les principales *menaces* à la concurrence sûre et les principales techniques pour les contrer.

## 2 Compétences à acquérir

- Savoir appliquer une démarche systématique pour analyser les spécifications et insérer les actions nécessaires pour rendre sûre une entité logicielle.

# Qu'est-ce que la *sûreté de concurrence* ?

## Sûreté de concurrence (*thread safety*)

La science, la technologie et l'ingénierie logicielle permettant d'écrire des programmes parallèles qui gèrent de manière sûre les accès concurrents aux ressources mutables partagées.

## Entité logicielle à concurrence sûre (*thread safe*)

Entité logicielle pouvant s'exécuter avec du parallélisme en garantissant la sûreté de concurrence.

- Il s'agit ici d'étudier la sûreté de concurrence sous l'angle de la gestion explicite des *threads* dans le contexte de Java et de BCM4Java.

# Définitions plus opérationnelles et corollaires I

## Classe à concurrence sûre

Une classe (Java) est sûre du point de vue de la concurrence si elle se comporte conformément à sa spécification (correctement) lorsqu'elle est accédée par plusieurs *threads*, peu importe l'ordonnancement ou les entrelacements admissibles de leurs exécutions, et ce sans que le code appelant ait à avoir recours à des synchronisations supplémentaires ou à d'autres actions de coordination.

- **Corollaire 1** : une classe à concurrence sûre *encapsule* dans son code toutes les synchronisations requises par son utilisation, évitant ainsi que les clients aient à prévoir leurs propres mécanismes.
- **Corollaire 2** : la sûreté de concurrence étant liée aux accès concurrents à un état mutable, l'absence d'état mutable est *de facto* la meilleure garantie de sûreté de concurrence.

# Définitions plus opérationnelles et corollaires II

- **Corollaire 3** : la sûreté de concurrence visant à assurer la conformité à une spécification, elle vise à garantir le respect de ses axiomes (classiques et de représentation).
- **Corollaire 4** : sans une idée claire et précise (formelle ou informelle) des axiomes (préconditions, postconditions et invariants) qui doivent être préservés sur une entité logicielle, aucune sûreté de concurrence ne peut être réellement garantie.
- Principales menaces à la sûreté de concurrence :
  - les *conditions de course*, quand les vitesses relatives des *threads* produisent différents *entrelacements* possibles de leurs instructions qui font que l'ordre dans lequel ils accèdent aux ressources partagées non protégées ne peut être prédit et peut résulter dans des états incohérents.
- Principales techniques pour assurer la sûreté de concurrence :
  - l'atomicité des accès aux ressources partagées mutables ;
  - l'exclusion mutuelle produite par la protection des sections critiques grâce à des outils de synchronisation.

# Démarche standard pour assurer la sûreté de concurrence

- ➊ Définir les structures de données et leurs opérations.
- ➋ Spécifier les invariants, ainsi que les pré- et posts-conditions.
- ➌ Regrouper dans le code les séquences d'instructions qui modifient l'état des structures de données ayant un lien entre elles par les invariants, pré- et post-conditions.
- ➍ Encapsuler ces séquences dans des sections critiques.
- ➎ Choisir un mécanisme de synchronisation et s'assurer de partager la même instance de synchroniseur entre toutes les sections critiques travaillant sur les mêmes données.
- Principales difficultés :
  - arriver à identifier tous les accès possibles aux données, y compris indirects via des mécanismes cachés dans l'implantation du langage ou des structures de données ;
  - ne pas surséquentialiser l'exécution pour préserver autant de parallélisme que possible (bien que la sûreté ne doit jamais être compromise pour augmenter le parallélisme).

# Plan

- 1 Exclusion mutuelle et synchronisation
- 2 Gestion de la concurrence
- 3 La notion de sûreté de concurrence
- 4 Gestion du partage d'objets**

# Objectifs de la séquence

## 1 Objectifs pédagogiques

- Comprendre les notions de *visibilité* et d'accessibilité aux données partagées, ainsi que les principaux phénomènes temporels dans l'exécution des programmes parallèles qui les impactent.
- Comprendre la notion et les différentes formes de *confinement* des données pour mettre en œuvre la concurrence sûre.

## 2 Compétences à acquérir

- Savoir utiliser les mot-clés `volatile`, `protected` et `private` pour favoriser la sûreté de concurrence.
- Savoir rendre des données mutables accessibles de manière sûre à plusieurs *threads*.
- Savoir utiliser quelques moyens simples offerts par Java pour rendre des collections immutables et à concurrence sûre.



# Concurrence, partage et ordonnancement des instructions I

- Le problèmes de concurrence sont d'abord la conséquence du partage de variables et de structures de données entre *threads*.
  - La notion de partage découle de la *visibilité* (des variables).
  - La visibilité découle aussi de l'*accessibilité* et de sa *temporalité*.
  - L'accès aux variables et aux données, à la base, part de l'exécution d'instructions qui vont les lire et les écrire, et de l'ordre dans lequel elles vont le faire.
- Avant même d'introduire les *threads*, on constate qu'il devient de plus en plus difficile de raisonner sur l'ordre dans lequel les instructions sont exécutées et sur leur atomicité.
  - Depuis toujours, les compilateurs optimisent le code qu'ils produisent et ce, en modifiant l'ordre des instructions machine.
  - Depuis plusieurs années maintenant, les processeurs réordonnent aussi localement les instructions machine à l'exécution pour favoriser la performance (pileline, accès aux caches, etc.).
  - Raisonner sur la visibilité et l'accessibilité en se fiant sur l'ordre statique des instructions sources (Java) devient donc plus hasardeux car il peut être bouleversé à la compilation et à l'exécution.

# Concurrence, partage et ordonnancement des instructions II

- Pour limiter les effets de ces technologies, Java propose le mot-clé *volatile* pour informer le compilateur qu'une variable est partagée et donc qu'il faut éviter de réordonner ses instructions de lecture/écriture avec les autres instructions.
  - La déclaration *volatile* *n'est pas* un mécanisme de synchronisation, seulement un *facilitateur* de synchronisation.
  - Elle indique au compilateur de bien séparer les instructions à protéger des autres : les instructions les précédant restent avant et les suivantes restent après.
- D'autre part, certaines opérations de haut niveau des langages de programmation qu'on pourrait croire *atomiques* ne le sont pas nécessairement en réalité.
  - La lecture et l'écriture de valeurs représentées sur 64 bits (doubles et entiers longs) ne sont pas nécessairement atomiques !

# Contrôle de la visibilité et de l'accessibilité

- La gestion de la visibilité des variables est un point fondamental en programmation par objets et par composants.
  - Tous les mécanismes habituels pour limiter la portée des variables sont essentiels aussi pour maîtriser la concurrence.
  - Toutefois, les déclarations des variables (`private`, `protected`, ...) ont été d'abord pensées d'un point de vue statique, pour déterminer dans quelles parties du programme source (classes, packages, etc.) une variable peut être accédée.
  - Elles disent cependant peu de choses sur la *dynamique* et donc la *propagation* des références.
- En Java, de nombreuses autres constructions peuvent engendrer du partage sans qu'on en soit toujours conscient.
  - Classes internes : (sauf pour les statiques) leurs instances comportent une référence sur une instance de la classe englobante qu'elles peuvent donc accéder.
  - *Attention, cela inclut les classes anonymes*, dont par ailleurs les méthodes peuvent *capturer* des variables visibles à l'endroit de leur définition.

## Exemple : capture de variable privée

```
public class Toto {
    private volatile ArrayList<Integer> maListe;

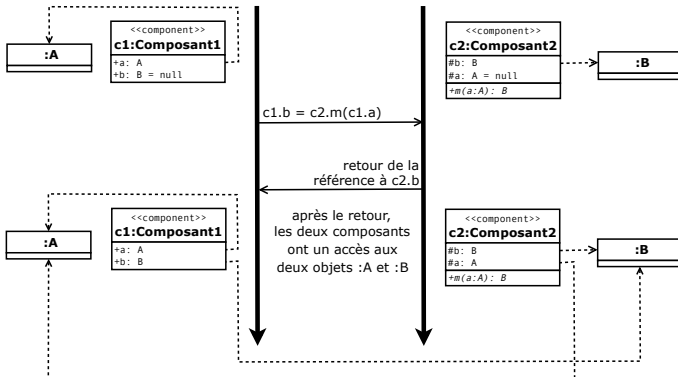
    public Toto() throws InterruptedException {
        Runnable titi = new Runnable() {           // création d'une classe anonyme
            @Override                               // interne puis instantiation
            public void run() {                     // capture de maListe de
                maListe.add(10);                    // l'instance englobante
            }
        };
        (new Thread(titi)).start();
        Thread.sleep(1L); // force l'ordonnancement du thread précédent
        this.maListe = new ArrayList<Integer>();
    }

    public static void main(String[] args) {
        try { Toto toto = new Toto();
        } catch (InterruptedException e) { throw new RuntimeException(e); }
    }
}
```

- L'exécution produit une exception `NullPointerException`.

# Sources de partage indésirable : fuites de références

```
public B m(A a) { this.a = a; return this.b; }
```



- Après les fuites de références, plusieurs *threads* possèdent des références aux mêmes variables/objets, on retrouve un problème d'accès concurrent potentiel.
- En BCM encore plus qu'en Java, il faut éviter les fuites de références hors des composants (les conséquences s'aggravant lors du passage aux déploiements répartis comme nous le verrons plus tard).

# Gestion de l'accessibilité des variables et données I

- Puisque la concurrence découle du partage des variables et données *mutables*, s'assurer de l'*immutabilité* des variables et données partagées évite bien des problèmes.
- Mais si le partage est nécessaire, limiter la portée de ce partage par le *confinement* réduit aussi les problèmes :
  - Confinement dans la pile d'exécution : données accessibles seulement depuis des variables locales aux méthodes.
    - Les données sont rangées dans des *variables locales uniquement*, en évitant de les insérer dans des structures de données partagées, de les passer en paramètres ou de les retourner en résultat.
  - Confinement au sein d'un objet/composant : données locales qui ne sont accessibles qu'au sein d'un certain objet/composant.
    - Les données sont rangées dans des variables d'instance *invisibles* de l'extérieur *i.e.*, seules les méthodes de l'objet/composant les accèdent ce qui permet d'encapsuler l'exclusion mutuelle.
    - Invisible de l'extérieur de l'*objet*  $\neq$  invisible de l'extérieur de la *classe de définition* (*private* versus *protected* versus *public*).

# Gestion de l'accessibilité des variables et données II

- Confinement au sein d'un *thread* : données locales aux *threads*.
  - Il faut s'assurer que ces données soient dans des variables *locales* au *thread* et jamais rendues accessibles à d'autres *threads* en les rangeant dans des structures de données partagées.
  - La classe `ThreadLocal<T>` permet de créer un réceptacle pour associer à une variable une valeur de type `T` par thread, mais cette technique est un peu complexe à utiliser avec les *pools* de *threads* donc moins utile en BCM4Java qu'en Java en général.
- Le confinement est le plus souvent obtenu par une *discipline de programmation stricte* à *documenter* par le programmeur.
- Partager une valeur ou un objet de manière sûre requiert :
  - si c'est un objet, de l'initialiser en exclusion mutuelle ;
  - d'initialiser la variable référente en exclusion mutuelle ;
  - de rendre cette variable volatile et
    - soit finale et passant par une référence atomique (e.g., `AtomicReference<T>`),
    - soit en assurer l'accès en exclusion mutuelle en la protégeant synchroniseur.

## « *Poor man's* » immutabilité et synchronisation

- Les collections standards étant très utilisées, Java fournit des décorateurs pour les rendre immutables ou synchronisés.
- La classe `Collections` définit des méthodes statiques :
  - `unmodifiableXXX` : décore une collection pour masquer toutes les méthodes en écriture.  
Ex.: `<T> List<T> unmodifiableList(List<? extends T> c)`
  - `synchronizedXXX` : décore une collection pour rendre toutes ses méthodes synchronisées.  
Ex.: `<T> List<T> synchronizedList(List<? extends T> c)`
- Ces décorateurs offrent une assez bonne protection sur les collections classiques de Java, mais de manière un peu grossière. Quelques précautions sont de mise :
  - Les collections paraissent immutables, mais les objets qu'elles contiennent ne le sont pas  $\Rightarrow$  immutabilité superficielle.
  - Les collections synchronisées le sont globalement, ce qui restreint fortement le parallélisme en sérialisant tous les accès.



# Les collections concurrentes de Java

- Les collections Java ont généralement été conçues pour la programmation séquentielle.
  - Les collections synchronisées (`Vector<T>`, ...) de même que les décorateurs synchronisés (`synchronizedList()`, ...) assurent l'exclusion mutuelle, mais ne sont pas pensés pour favoriser le parallélisme mais plutôt le restreindre.
- Java a donc introduit des versions dites *concurrentes* des collections standards et en a ajouté de nouvelles pour ce faire.
- Trois principales modifications sémantiques sont généralement impliquées :
  - ➊ Ajout d'opérations complexes *atomiques*.  
Ex.: `putIfAbsent` : test si le paramètre apparaît dans la collection et l'ajoute s'il n'y était pas.
  - ➋ *Affaiblissement* de la sémantique de certaines méthodes pour augmenter le parallélisme dans l'accès à la collection (ex.: `size`).
  - ➌ Ré-interprétation de la *sémantique des pré-conditions*.

# Affaiblissement de la sémantique

- Synchroniser une collection globalement sérialise toutes les opérations sur cette dernière.
- Les versions concurrentes utilisent des stratégies d'exclusion mutuelle plus fines, autorisant plusieurs *threads* à lire, parcourir voire modifier la collection en parallèle.
  - `ConcurrentHashMap<T>` autorise un nombre illimité d'accès en lecture, des accès en lecture en parallèle avec des accès en écriture et un nombre limité d'accès en écriture en parallèle.
  - Il devient alors possible d'itérer sur une collection en parallèle avec sa modification, l'itération se faisant sur une vue immuable.
  - En rendant cela possible, l'itération peut encore provoquer une `ConcurrentModificationException` et les opérations globales comme `size` ont une sémantique plus faible (la taille peut avoir changé au moment où le résultat est retourné).
- On peut généralement augmenter le parallélisme en remplaçant les collections séquentielles ou synchronisées par leur équivalent concurrente à relativement peu de risque d'incohérence.

# Ré-interprétation des pré-conditions

- Préserver l'invariant et garantir les post-conditions requiert l'atomicité et l'exclusion mutuelle dans les accès aux variables impliquées.
- Pour les pré-conditions, les choses sont moins claires.
- Certaines opérations ont des pré-conditions fondées sur l'état d'un objet et la violation d'une précondition ne peut mener qu'à une erreur en programmation *séquentielle*.
  - Ex.: pour dépiler un objet, une pile ne peut être vide.
- En programmation *parallèle*, une autre interprétation est possible : attendre que la précondition devienne vraie par l'action d'un autre *thread*.
- Cette interprétation justifie en Java de proposer des variantes « concurrentes » de certaines classes standards :
  - `BlockingQueue<T>` par rapport à `Queue<T>` : les méthodes `put` et `take` bloquent lorsque la file est pleine ou vide.

# Tâche asynchrone avec résultat futur : `FutureTask<T>`

- Une des limitations des variables futures de Java est qu'il faut généralement lancer l'exécution d'une tâche pour obtenir la variable future représentant son résultat.
- Pour surmonter cette difficulté, la classe `FutureTask<T>` associe une tâche et la variable future représentant son résultat en un seul objet :

```
Callable<Double> computation =  
    new Callable<Double>() {  
        @Override  
        public Double call() throws Exception { return ...; }  
    };  
  
FutureTask<Double> ft = new FutureTask<Double>(computation);  
ft.run(); // démarrage indépendant de la création !  
double result = ft.get();
```

- Ces `FutureTask<T>` permettent un partage sûr des tâches elles-mêmes plutôt que simplement de leurs résultats, comme nous allons le voir dans l'exemple suivant.

## Exemple : Composant avec cache I

- Objectif : utiliser un cache de données pour conserver et réutiliser les résultats de calculs particulièrement longs
  - Supposons un composant proposant un service `compute` prenant un argument de type `A` et retournant un résultat `V`.
  - Ce composant utilise une `ConcurrentHashMap<A, V>` comme cache pour ne pas recalculer plusieurs fois pour un même argument.
- L'objectif de ce composant est de permettre plusieurs calculs en parallèle grâce à un *pool* de *threads*.
- Le défi est de s'assurer que le résultat pour un argument donné ne soit jamais calculé deux fois, *même* si deux *threads* appellent le composant en même temps pour le même argument.
  - Utiliser les variables futures forcerait à lancer une tâche à chaque appel avant de pouvoir se rendre compte si une autre avait déjà été lancée, ce qui demanderait d'interrompre ensuite les tâches lancées après la première : *raté* !

## Exemple : Composant avec cache II

- Pour éviter ce problème, on utilise plutôt une `FutureTask` :

```
FutureTask<V> ft = new FutureTask<V>(computation);  
f = cache.putIfAbsent(arg, ft);  
if (f == null) { f = ft; ft.run(); }  
V ret = f.get();
```

- Au premier passage avec un argument donné, `putIfAbsent` retournera `null` et la tâche insérée sera lancée.
  - Aux passages suivants, une nouvelle tâche est créée mais, comme `putIfAbsent` retourne une valeur non `null`, on se rend compte qu'une autre tâche avait déjà été lancée avant de lancer la nouvelle qui peut simplement être détruite sans jamais avoir débuté le calcul.
- Plutôt que directement les résultats futurs, le cache contient `FutureTask` ce qui permet de partager d'abord des calculs (tâches) puis leurs résultats quand les calculs sont terminés.

# Prolongement : composant avec cache et parallélisme

- À la suite de l'exemple précédente, il s'agit ici d'ajouter une méthode `computeAll` prenant en paramètre un tableau de valeurs et retournant un tableau de résultats.
  - On lance en parallèle les calculs sur le plus possible des valeurs reçues en argument par la méthode `invokeAll`.
  - Puis on récupère les résultats en parcourant la liste des obtenue d'`invokeAll` par une boucle `for` pour faire les `get` sur chacune des variables futures.
  - Le `get` est bloquant si le résultat n'est pas encore disponible. Ce code montre donc les limites des variables futures :
    - la boucle `for` parcourt ces variables dans un ordre qui ne correspond pas nécessairement à l'ordre de production des résultats ;
    - en pire cas, si le premier résultat est le dernier produit, la boucle attendra pour ce résultat puis récupérera tous les autres sans attendre ;
    - nous avons vu `ExecutorCompletionServices` pour contourner ce problème, mais il n'utilise pas les `FutureTask`...

# Activités à réaliser avant le prochain TME

## Rappel

***La lecture de code en CPS est une activité aussi importante pour acquérir les concepts et les techniques de programmation présentées que la préparation des examens dans d'autres UE. N'oubliez pas que l'évaluation du projet porte entre autres choses sur la qualité de votre code.***

- ➊ Récupérer, lire attentivement puis essayer les exemples proposés dans le cours.
- ➋ (Re)Lire la seconde partie du cahier des charges du projet pour faire le lien avec les concepts introduits dans ce cours.