Aperçu du modèle mémoire JAVA

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2020-2021

Version du 14 novembre 2020

Un problème de vue

```
public static void main(String[] args) {
  A a = new A() ;          // Création d'un objet a de la classe A
  a.start() ;              // Lancement du thread a
  a.valeur = 1 ;           // Modification de l'attribut valeur
  a.fin = true ;           // Modification de l'attribut fin
}

static class A extends Thread {
  public int valeur = 0 ;
  public boolean fin = false ;

  public void run() {
    while(! fin) {} ;      // Attente active
    System.out.println(valeur);
  }
}
```

Ce programme termine-t-il toujours ? Peut-il afficher 0 ?

À quoi sert le modèle mémoire Java ?

Le modèle mémoire Java définit la sémantique des programmes Java, c'est-à-dire l'ensemble des résultats possibles d'un programme donné.

Il s'agit en fait aujourd'hui d'apprendre pourquoi un programme Java peut sembler

① *cacher* momentanément les écritures réalisées en mémoire,

② ou bien ne pas exécuter les instructions dans *l'ordre du programme*.

En particulier, le modèle mémoire Java précise l'effet du mot-clef **`volatile`**. Il décrit aussi les effets des instructions **`start()`**, **`join()`**, **`wait()`**, etc. ce qui permet de comprendre à quels moments le modificateur **`volatile`** peut finalement être omis.

Ce cours peut vous aider dans un futur proche

- à répondre correctement à une question technique lors d'un entretien ;
- à comprendre pourquoi le code d'un collègue n'est pas si stupide que ça ;
- à expliquer à vos collègues pour quelles raisons votre code fonctionne bien ;
- et même à trouver pourquoi votre programme produit des résultats incohérents.

Ajout du mot-clef « volatile »

```
public static void main(String[] args) {
  A a = new A() ;              // Création d'un objet a de la classe A
  a.start() ;                  // Lancement du thread a
  a.valeur = 1 ;               // Modification de l'attribut valeur
  a.fin = true ;               // Modification de l'attribut fin
}

static class A extends Thread {
  public int valeur = 0 ;
  public volatile boolean fin = false ;

  public void run() {
    while(! fin) {} ;          // Attente active
    System.out.println(valeur);
  }
}
```

Ce programme termine-t-il toujours ? Peut-il afficher 0 ?

Exécutions légales

Le **modèle mémoire java** (JMM, pour « Java Memory Model ») définit la sémantique des programmes Java ; c'est en fait un *modèle d'exécution* : il détermine quelles sont les **exécutions légales** (et donc les résultats légaux) d'un programme donné.

C'est à la fois

① une *garantie* pour les développeurs :

Un programme Java ne peut pas faire n'importe quoi !

② une *contrainte* pour les fabricants de machines virtuelles Java :

Les JVM ne doivent pas faire n'importe quoi !

Il ne faut pas confondre !

Le *Java Language Specification* et le *Java Virtual Machine Specification* ont des objectifs bien distincts :

① le premier détermine ce que sont les **exécutions légales** d'un programme, à l'aide du *modèle mémoire* qui s'appuie sur des **ordres partiels**.

② le second guide les concepteurs de machines virtuelles en autorisant certaines **permutations d'instructions**.

À quoi sert le modèle mémoire Java ?

Les sources multiples du problème :

- Les compilateurs peuvent **réordonner** les instructions.
- Les **processeurs** peuvent aussi réordonner les instructions !
- Parfois les nouvelles valeurs des variables sont **retenues** dans les caches.

Le modèle mémoire d'un langage détermine (plus ou moins directement) le type de manipulations autorisées sur le code et sur les mises-à-jour des variables, et donc l'ensemble de *ses résultats légaux*.

✔ *Introduction*

☛ *La notion d'exécution (de manière simplifiée)*

La relation de précédence « a lieu avant »

Il est fondamental, lors de l'exécution d'une instruction, de pouvoir déterminer quelles autres instructions (sur le même thread, ou sur les autres threads) *doivent avoir eu lieu* précédemment, celles qui *peuvent avoir eu lieu*, et celles qui *ne le peuvent pas*.

Le JMM définit une *relation* « happens-before » entre les instructions d'une exécution, relation qui est un **ordre partiel**, noté $<_{\mathrm{HB}}$. Cette relation est donc **transitive** :

$\rightsquigarrow$ si x a lieu avant y et si y a lieu avant z, alors x a lieu avant z.

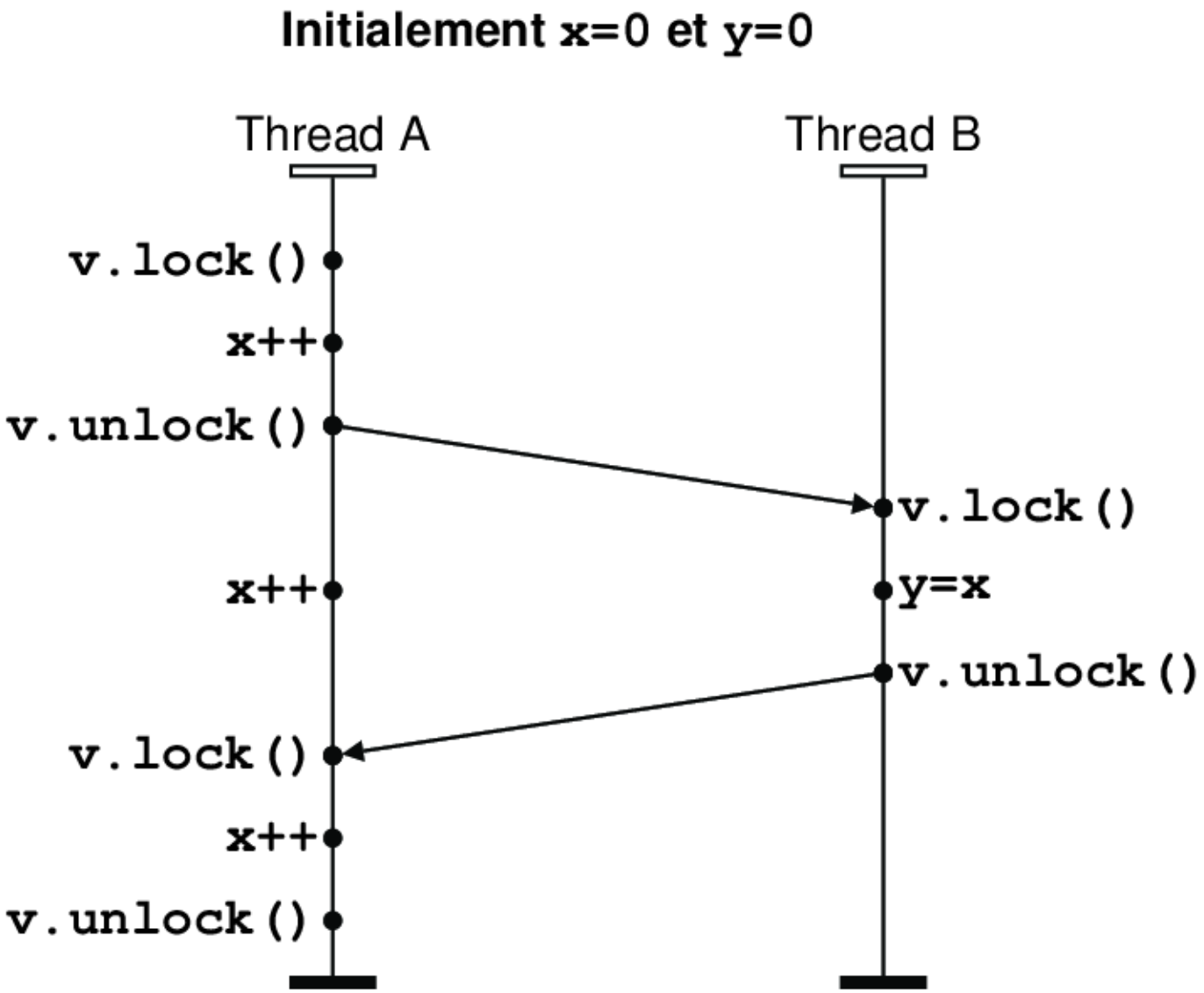
Cette relation est construite principalement par

① **l'ordre du programme** qui garantit une consistance locale à chaque thread ;

② certaines actions qui produisent des **synchronisations** entre threads ; par exemple, les opérations sur un verrou sont **totalement ordonnées** par la relation $<_{\mathrm{HB}}$.

Chaque instruction *voit* ce qui la précède selon $<_{\mathrm{HB}}$, mais potentiellement un peu plus !

Exemple d'exécution légale (avec deux synchronisations)

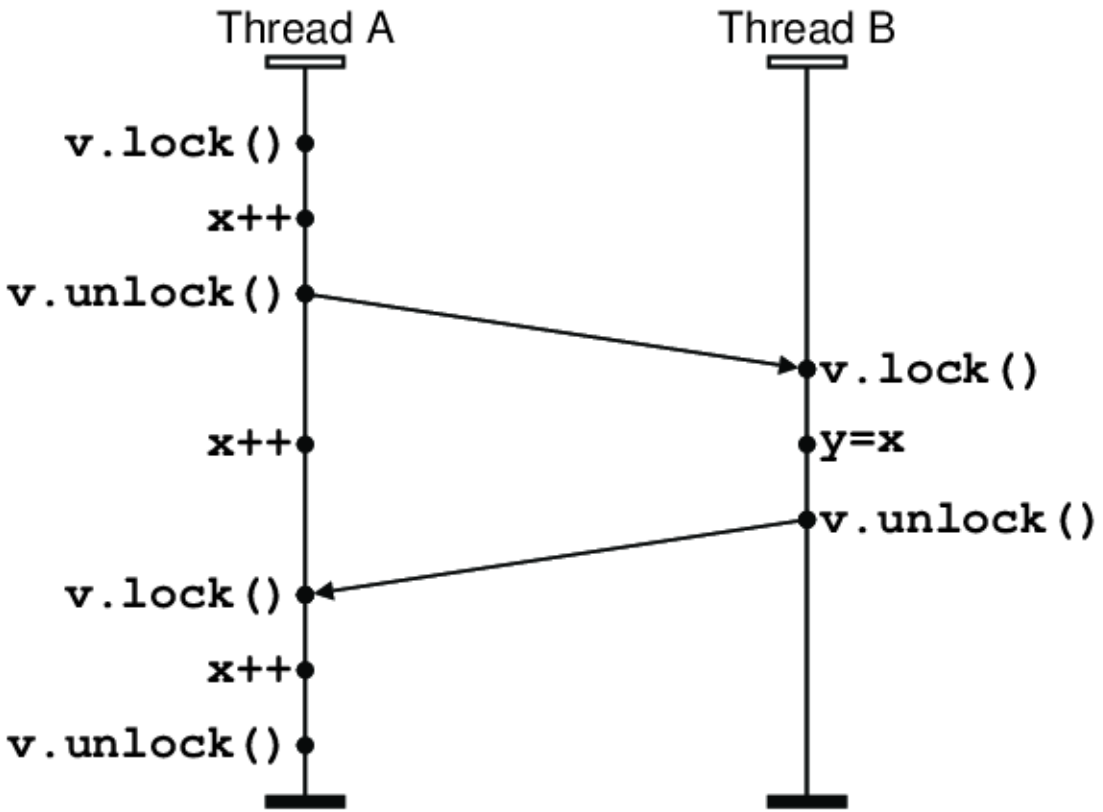


Que vaut `y` *à la fin de cette exécution ?*

- ✔ *Introduction*
- ✔ *La notion d'exécution (de manière simplifiée)*
- ☛ *Visibilité assurée par l'emploi d'un verrou*

Il faut distinguer la relation « a lieu avant » et l'observation effective

Initialement `x=0` et `y=0`



- ⇝ **`y=x`** doit observer la première incrémentation de **`x`**, qui a lieu avant.
- ⇝ **`y=x`** ne peut pas observer la troisième incrémentation de **`x`**, qui a lieu après.
- ⇝ **`y=x`** observera ou non la seconde incrémentation de **`x`**.
- ⇝ à la fin de cette exécution, **`y=1`** ou **`y=2`** (mais jamais **`y=3`**).

Première caractéristique des exécutions légales

Une exécution légale détermine l'ordre des opérations sur chaque verrou.

Les synchronisations associées à un verrou déterminent donc l'ordre d'acquisition du verrou par les différents threads, conformément à la notion de verrou :

- au plus un seul thread possède le verrou à chaque instant ;
- les verrous sont ré-entrants (sauf le verrou timbré) ;
- seul le thread qui possède le verrou peut le relâcher : chaque appel à **`v.lock()`** ≪ a lieu avant ≫ l'appel à **`v.unlock()`** associé, par le même thread, et donc *selon l'ordre des instructions du programme.*

Ainsi un verrou produit essentiellement des synchronisations qui vont d'un relâchement du verrou par un thread, lors d'un appel à **`v.unlock()`**, vers l'acquisition suivante du verrou par un autre thread, lors d'un appel ultérieur à la méthode **`v.lock()`**.

Visibilité et verrou : un exemple simple

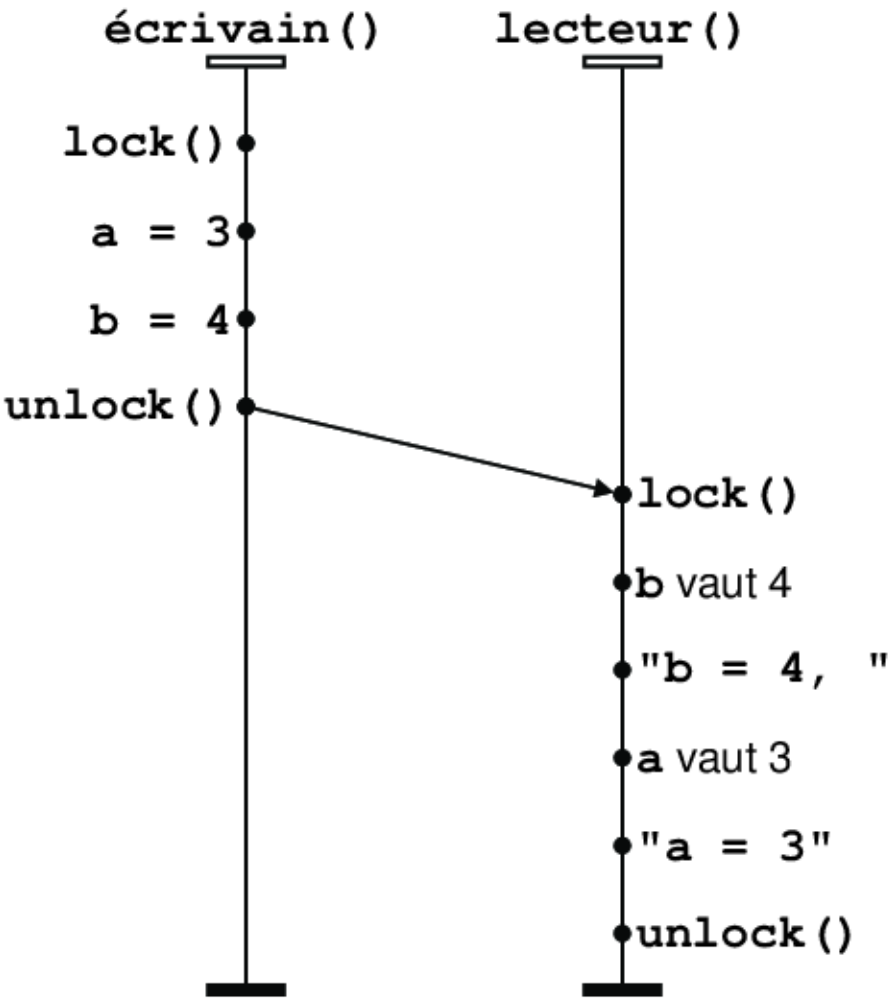
```
class SynchronizedNotSoSimple {
  private int a = 1, b = 2 ; // ne sont pas déclarées volatiles !
  void synchronized écrivain() {
    a = 3;
    b = 4;
  }
  void synchronized lecteur() {
    System.out.print("b␣=␣" + b + ",␣");
    System.out.println("a␣=␣" + a);
  }
}
```

Un thread va exécuter **`écrivain()`** et un autre **`lecteur()`**. Du fait de **`synchronized`**, les seules sorties légales sont :

- **`"b=2, a=1"`** si le lecteur prend le verrou en premier ;
- **`"b=4, a=3"`** si l'écrivain prend le verrou en premier.

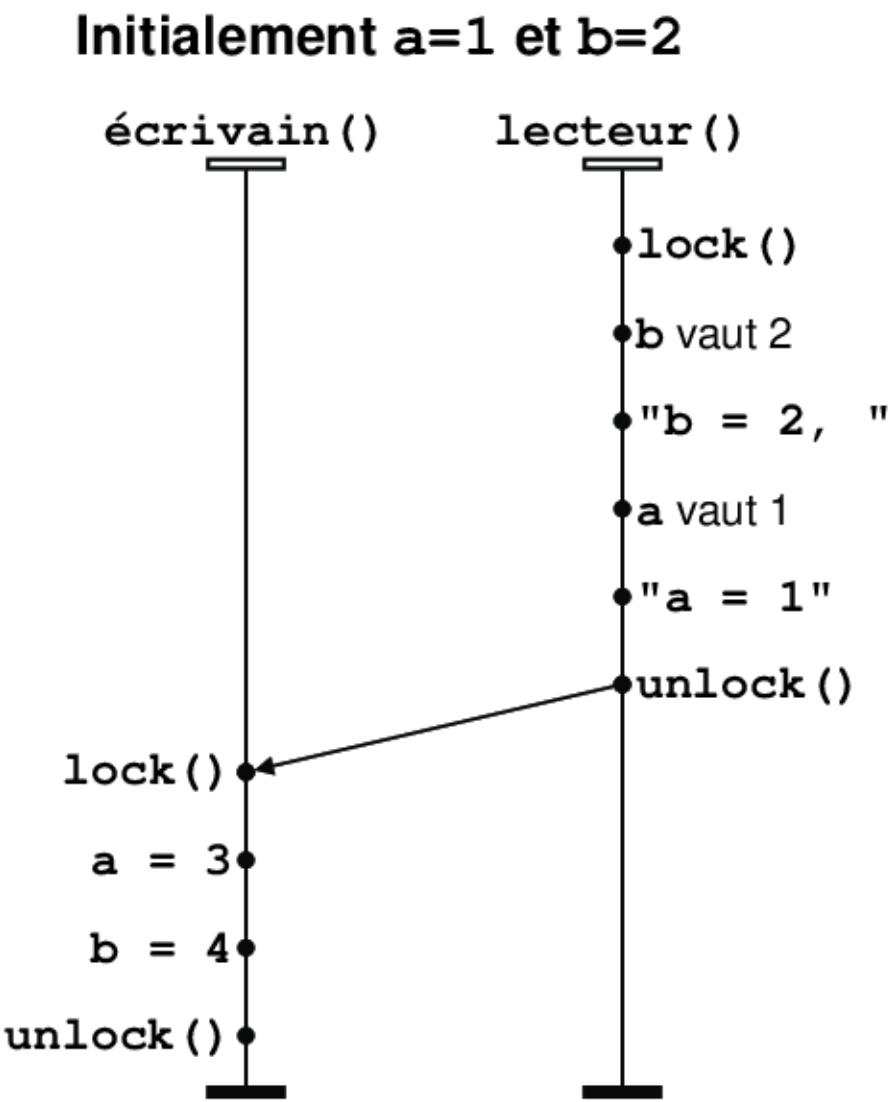
Une exécution légale

Initialement a=1 et b=2



Le verrou garantit la visibilité ! Donc ajouter `volatile` ne servirait à rien.

Une seconde exécution légale

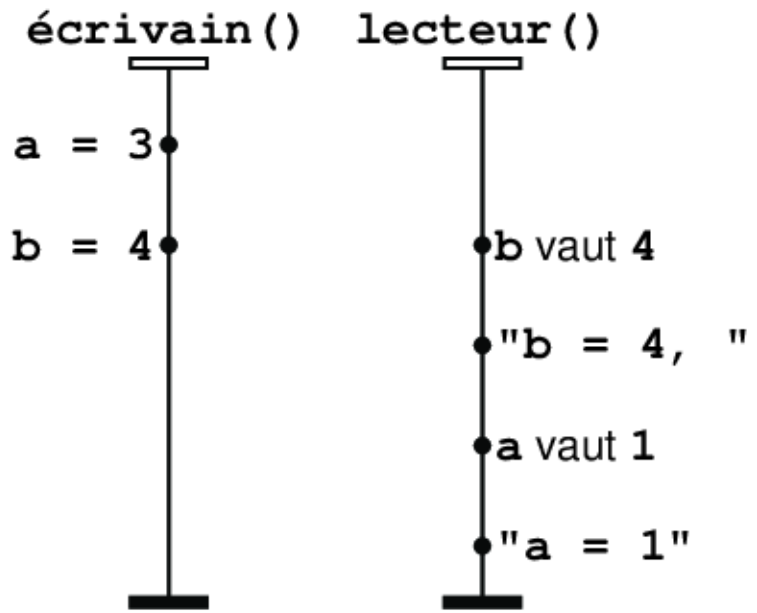


En supprimant synchronized, le résultat peut être très surprenant !

```
class NotSoSimple {
  int a = 1, b = 2;

  void écrivain() {
    a = 3;
    b = 4;
  }

  void lecteur() {
    System.out.print("b = " + b +", ");
    System.out.println("a = " + a);
  }
}
```



Il n'y a aucune synchronisation !

Selon le modèle mémoire Java, ce programme pourra afficher **"b = 4, a = 1"**...

Retour sur le problème de vue réglé avec un print("")

```
public static void main(String[] args) {
  A a = new A() ;              // Création d'un objet a de la classe A
  a.start() ;                  // Lancement du thread a
  a.valeur = 1 ;               // Modification de l'attribut valeur
  a.fin = true ;               // Modification de l'attribut fin
  System.out.println("Le main termine.") ;
}

static class A extends Thread {
  public int valeur = 0 ;
  public boolean fin = false ;
  public void run() {
    while(! fin) { System.out.print(""); } ;  // Attente active
    System.out.println(valeur) ;
  }
}
```

C'est vraiment très moche, mais ça dépanne...

Synchronisations opérées par les autres verrous de Java

La documentation de l'interface **`Lock`** indique :

≪ *All Lock implementations must enforce the same memory synchronization semantics as provided by the built-in monitor lock, as described in section 17.4 of The Java Language Specification :*

- *A successful lock operation has the same memory synchronization effects as a successful Lock action.*
- *A successful unlock operation has the same memory synchronization effects as a successful Unlock action.*

Unsuccessful locking and unlocking operations, and reentrant locking/unlocking operations, do not require any memory synchronization effects. ≫

Le ≪ *built-in monitor lock* ≫ évoqué ici est simplement le verrou intrinsèque de l'objet.

Synchronisations de l'instruction `wait()`

Un appel à **`v.wait()`** comporte de manière sous-jacente un **`v.unlock()`** au moment de l'endormissement ainsi qu'un **`v.lock()`** au moment du réveil.

Le modèle mémoire Java regarde effectivement l'instruction **`wait()`** comme une suite non atomique d'actions parmi lesquelles certaines induisent des synchronisations liées à l'usage du verrou intrinsèque de l'objet.

≪ *17.2.1 Wait*

Wait actions occur upon invocation of wait(), or the timed forms wait(long millisecs) and wait(long millisecs, int nanosecs).

...

The following sequence occurs :

1. *Thread* t *is added to the wait set of object* m*, and performs* n *unlock actions on* m*.*
2. *...*
3. *Thread* t *performs* n *lock actions on* m*.* ≫

Les verrous de lecture-écriture assurent une synchronisation

Selon la documentation de Java 12,

≪ *All ReadWriteLock implementations must guarantee that the memory synchronization effects of writeLock operations (as specified in the Lock interface) also hold with respect to the associated readLock.*

That is, a thread successfully acquiring the read lock will see all updates made upon previous release of the write lock. ≫

Le verrou timbré synchronise également !

Selon la documentation,

≪ *Methods with the effect of successfully locking in any mode have the same memory synchronization effects as a Lock action described in Chapter 17 of The Java Language Specification.*

Methods successfully unlocking in write mode have the same memory synchronization effects as an Unlock action.

In optimistic read usages, actions prior to the most recent write mode unlock action are guaranteed to happen-before those following a tryOptimisticRead only if a later validate returns true ; otherwise there is no guarantee that the reads between tryOptimisticRead and validate obtain a consistent snapshot. ≫

- ✔ *Introduction*
- ✔ *La notion d'exécution (de manière simplifiée)*
- ✔ *Visibilité assurée par l'emploi d'un verrou*
- ☛ *Le modificateur volatile*

Retour au premier exemple

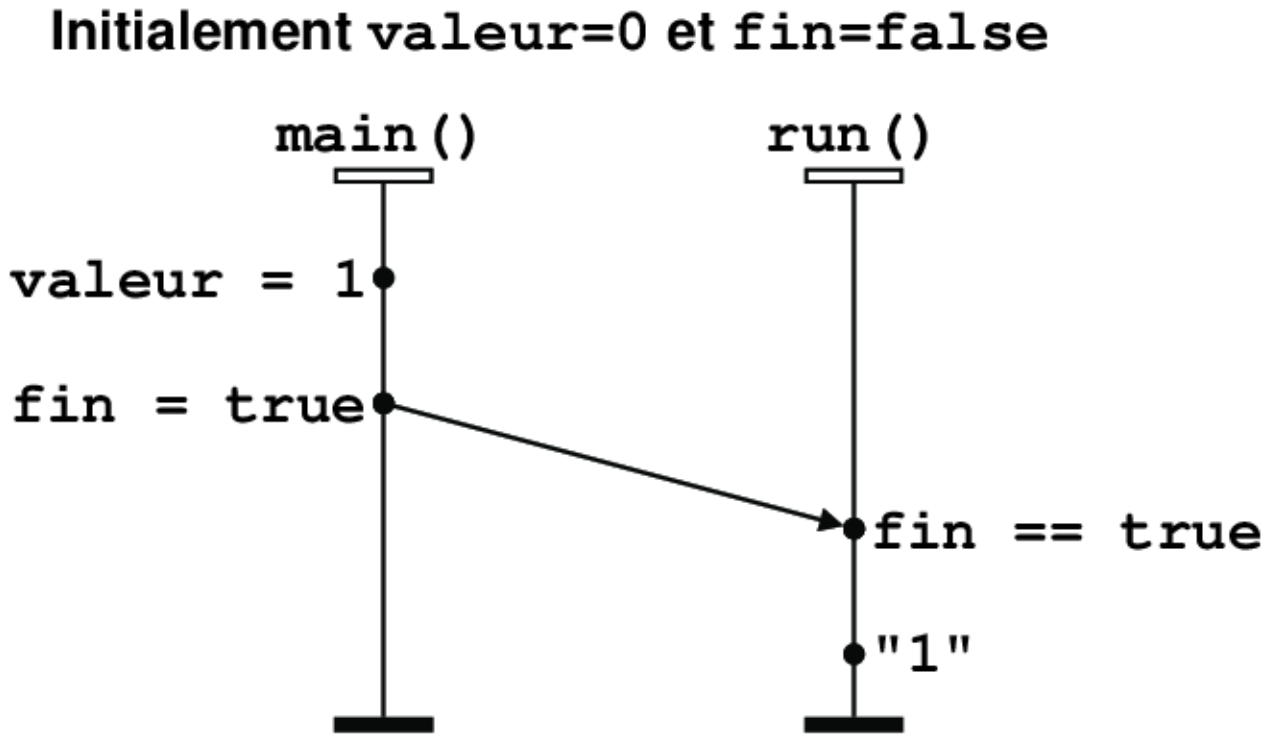
```
public static void main(String[] args) {
  A a = new A();          // Création d'un objet a de la classe A
  a.start();              // Lancement du thread a
  a.valeur = 1;           // Modification de l'attribut valeur
  a.fin = true;           // Modification de l'attribut fin
}
static class A extends Thread {
  public int valeur = 0 ;
  public volatile boolean fin = false ;

  public void run() {
    while(! fin) {} ;     // Attente active
    System.out.println(valeur) ;
  }
}
```

Ce programme termine-t-il toujours ? Peut-il afficher 0 ?

La variable `valeur` bénéficie de la volatilité de `fin`

Une exécution légale doit, pour chaque variable volatile **`f`**, comporter une **synchronisation** de chaque *écriture* dans **`f`** vers *les lectures et les écritures de* ***`f`*** *ultérieures.*



Ce programme terminera et affichera **`"1"`**, car la modification du booléen volatile **`fin`** sera vue par le second thread, de même que celle de **`valeur`** *par transitivité* de la relation « a lieu avant » et du fait de l'ordre des instructions du programme.

Suppression du mot-clef `volatile` en pratique

Si tous les accès à une variable sont protégés par un *verrou*, alors il est inutile de déclarer cette variable **`volatile`**.

C'est le cas lorsque l'on adopte le modèle des moniteurs. Vous pouvez donc retirer **`volatile`** sur plusieurs exemples du cours !

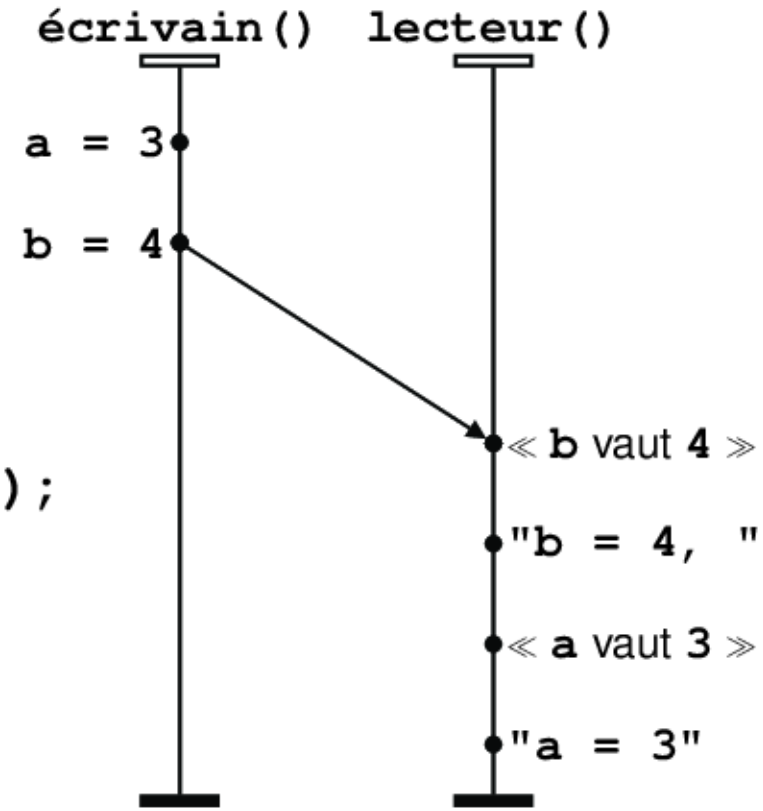
C'est utile à savoir car :

- ça évite de mettre **`volatile`** partout (ce qui peut nuire aux performances) ;
- il est parfois impossible d'assurer la visibilité des données à l'aide du mot-clef **`volatile`**.

En effet, si un objet est déclaré **`volatile`**, les modifications de cette référence seront certes visibles, mais pas nécessairement les modifications des attributs de cet objet.
N.B. Ça vaut aussi pour les éléments d'un tableau.

Autre illustration du sens de `volatile`

```
class NotSoSimple {
  int a = 1;
  int volatile b = 2;
  void écrivain() {
    a = 3;
    b = 4;
  }
  void lecteur() {
    System.out.print("b = " + b +", ");
    System.out.println("a = " + a);
  }
}
```

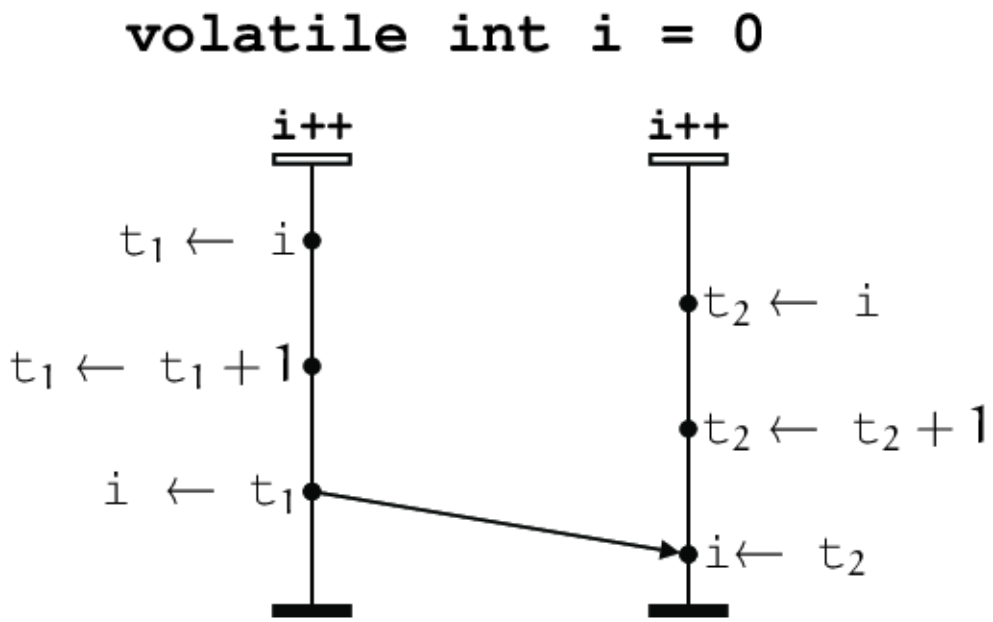


Ce programme ne pourra pas afficher **"b = 4, a = 1"** ; car si le second thread voit que b vaut 4, il voit nécessairement aussi que a vaut 3.

- ✔ *Introduction*
- ✔ *La notion d'exécution (de manière simplifiée)*
- ✔ *Visibilité assurée par l'emploi d'un verrou*
- ✔ *Le modificateur volatile*
- ☛ *Visibilité des objets atomiques*

Ultime rappel

L'incrémentation d'une variable volatile n'est pas atomique.



C'est l'une des raisons pour lesquelles quelques classes d'**objets atomiques** ont été introduites dans Java.

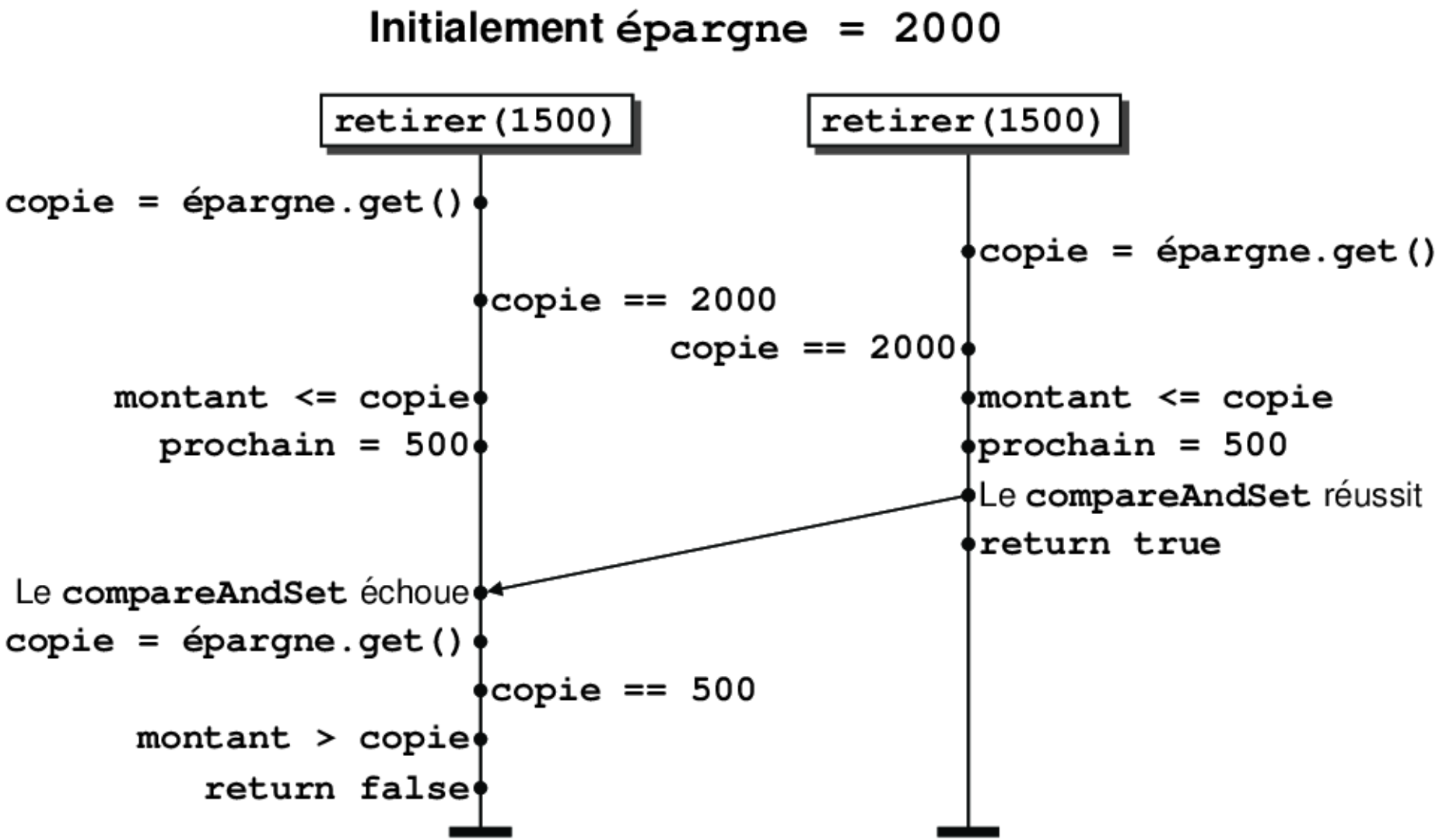
Exemple typique de codage « optimiste » (déjà vu)

```
public class CompteBancaire {
  private final AtomicLong épargne;

  ...

  public boolean retirer(long montant) {
  for (;;) {
    int copie = épargne.get();
    if (montant > copie) return false;
    int prochain = copie - montant;
    if (épargne.compareAndSet(copie, prochain)) return true;
  }
  ...
}
```

Exécution de la méthode `retirer()`



Les modifications du contenu d'un objet atomique sont toujours visibles !

Selon le site `docs.oracle.com`

The memory effects for accesses and updates of atomics generally follow the rules for volatiles, as stated in section 17.4 of « The Java Language Specification. »

- **`get()`** has the memory effects of reading a volatile variable.
- **`set()`** has the memory effects of writing (assigning) a volatile variable.
- **`compareAndSet()`** and all other read-and-update operations such as **`getAndIncrement()`** have the memory effects of both reading and writing volatile variables.

- ✔ *Introduction*
- ✔ *La notion d'exécution (de manière simplifiée)*
- ✔ *Visibilité assurée par l'emploi d'un verrou*
- ✔ *Le modificateur volatile*
- ✔ *Visibilité des objets atomiques*
- ☛ *Visibilité des éléments d'un tableau*

Tableaux formés d'objets `atomics`

Méfiance !

Il n'y a aucun moyen de rendre les **éléments d'un tableau** ≪ volatiles. ≫

En effet, si un tableau est déclaré **`volatile`**, c'est la référence du tableau qui bénéficiera de la volatilité : les modifications des éléments du tableau ne seront pas nécessairement vues.

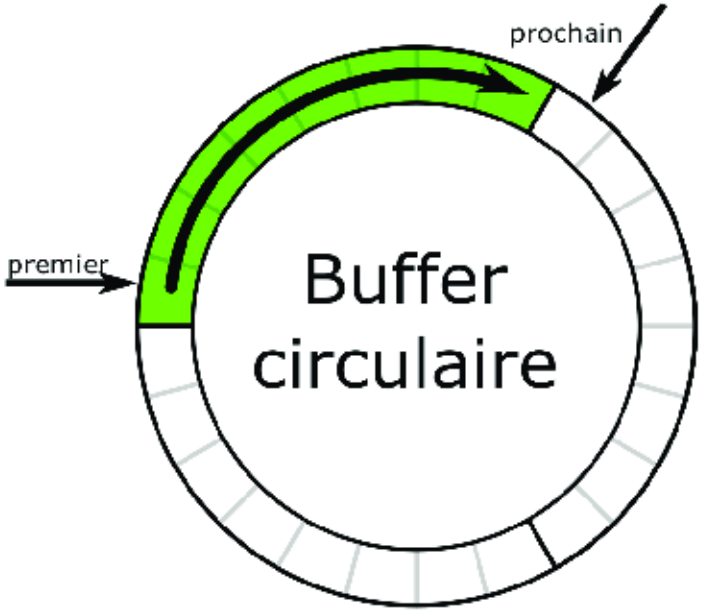
Les classes dédiées **`AtomicIntegerArray`**, **`AtomicLongArray`**, etc. permettent de disposer d'un tableau

- dont les éléments peuvent être manipulés comme des objets atomiques ;
- rassemblés dans un seul objet dont la visibilité est garantie.

Pour un tableau d'objets quelconques, on utilise un **`AtomicReferenceArray`**< E > qui fournit un tableau de références, manipulables atomiquement, vers des objets. Ces derniers seront de préférence, mais pas obligatoirement, des objets *immuables*.

La visibilité peut aussi être assurée par un verrou

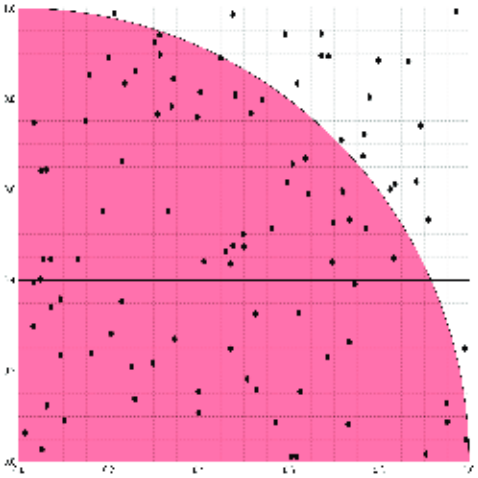
```
class Buffer {
  private final int taille;
  private final byte[] buffer;
  private int disponibles = 0;
  private int prochain = 0;
  private int premier = 0;
  Buffer(int taille) {
    this.taille = taille;
    this.buffer = new byte[taille];
  }
  synchronized void déposer(byte b) { ... }
  synchronized byte retirer() { ... }
}
```

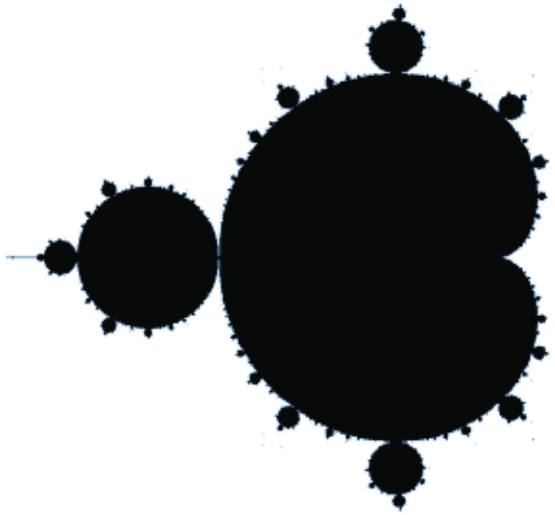


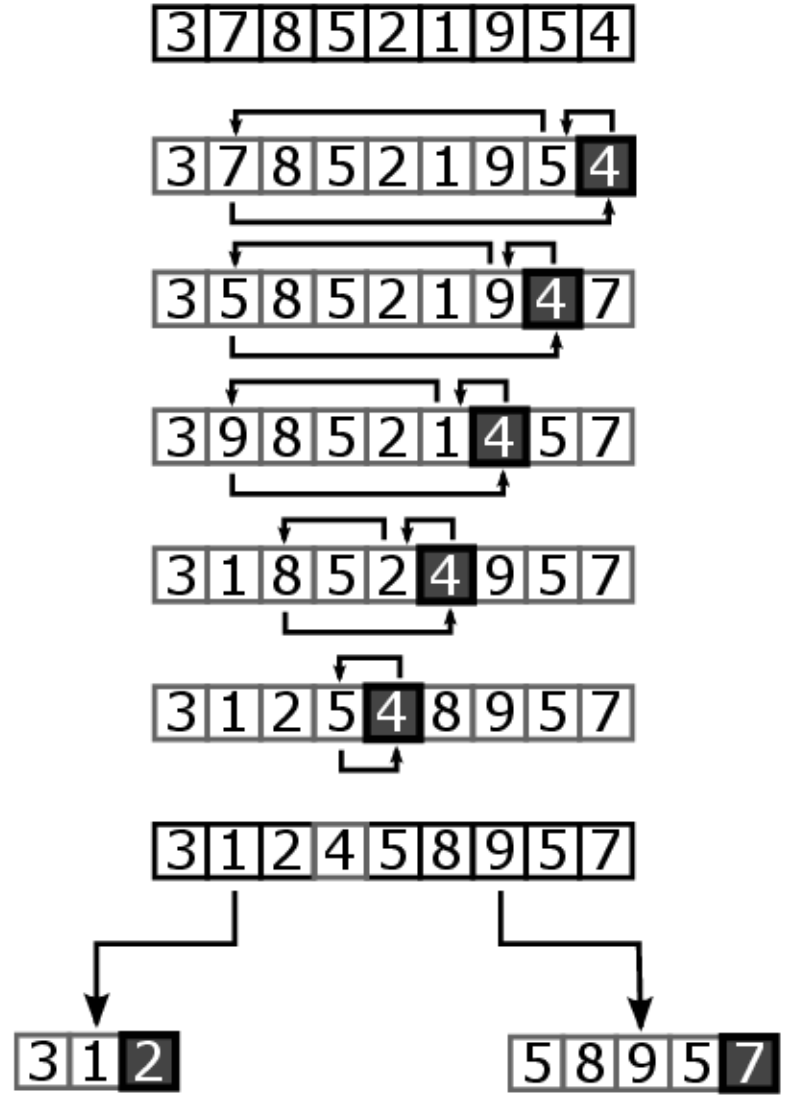
Le verrou assure la visibilité des modifications du contenu du tableau **`buffer`**.

- ✔ *Introduction*
- ✔ *La notion d'exécution (de manière simplifiée)*
- ✔ *Visibilité assurée par l'emploi d'un verrou*
- ✔ *Le modificateur volatile*
- ✔ *Visibilité des objets atomiques*
- ✔ *Visibilité des éléments d'un tableau*
- ☛ *Autres précisions utiles en pratique*

Exemples de négligences passées, mais sans conséquence







Les données partagées doivent être synchronisées pour être visibles.

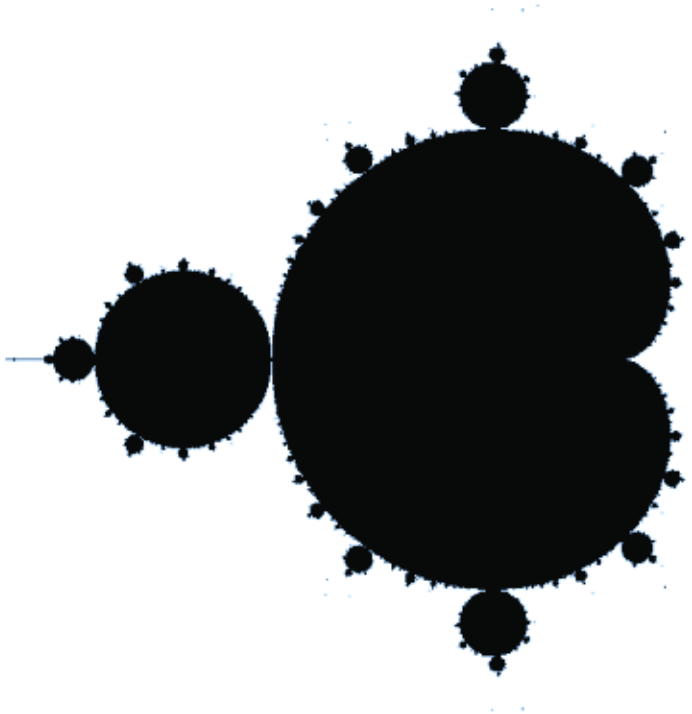
Visibilité des résultats obtenus à l'issue d'un partage

Outre les synchronisations exigées par les opérations sur les verrous ou les accès aux variables volatiles, le JMM impose certaines autres synchronisations assez naturelles :

① La dernière action d'un thread **`t`** *a lieu avant* les actions déclarées après l'instruction **`t.join()`** sur un autre thread.

② Les actions représentées par un objet **`Future`** *ont lieu avant* l'obtention du résultat produit par la méthode **`get()`** sur cet objet.

③ Les actions d'une tâche soumise à un service de complétion *ont lieu avant* l'occurrence de la méthode **`take()`** associée à cette tâche.

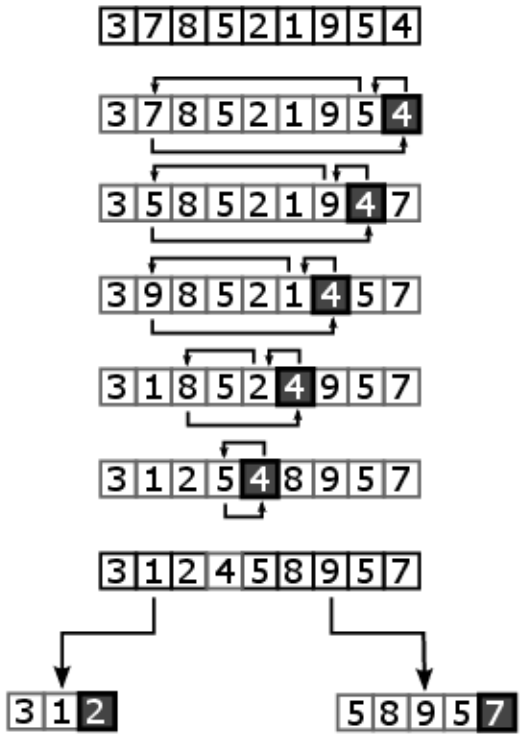


```
for (int i = 0 ; i < taille ; i++) {
  service.take() ;
} // Chaque ligne complétée sera nécessairement visible.
```

Visibilité des actions préalables à un partage

① Une action qui démarre un thread via la méthode **`start()`** *a lieu avant* toutes les actions exécutées par ce thread.

② La soumission d'une tâche **`Runnable`** ou **`Callable`** à un exécuteur *a lieu avant* l'exécution de celle-ci.




```
public Boolean call() {
   ...
   int p = partition(tableau, début, fin);
   triAGauche = new Triage(début, p-1);
   service.submit(triAGauche) ;
   ...
} // Le thread exécutant triAGauche verra la partition faite.
```

Collections, barrières, loquets, sémaphores, etc.

Tout comme les objets atomiques sont garantis de se comporter comme des variables volatiles, les autres outils introduits dans Java 5 ont des propriétés naturelles vis-à-vis de la relation « a lieu avant » et donc au niveau de la visibilité :

- L'ajout d'un objet à une collection concurrente *a lieu avant* l'accès en lecture ou le retrait de cet objet dans cette collection.
- Les actions déclarées avant chaque appel à **`loquet.countDown()`** *ont lieu avant* celles déclarées après les retours de la méthode **`loquet.await()`** correspondants.
- Les actions déclarées avant l'appel à **`barrière.await()`** *ont lieu avant* celles associées éventuellement à cette barrière, et celles-ci ont elles-même lieu avant celles déclarées après le retour de la méthode **`barrière.await()`**.
- etc.

Autres synchronisations garanties

① L'écriture de la valeur par défaut dans les champs d'un objet se synchronise avec toutes les accès à cet objet.

Par défaut, on lit la valeur par défaut !

② Si un thread **`t1`** interrompt un thread **`t2`**, l'interruption par **`t1`** se synchronise avec toutes les détections de l'interruption par **`t2`** : exception, appels aux méthodes **`interrupted()`** ou **`isInterrupted()`**.

Notion de consistance séquentielle

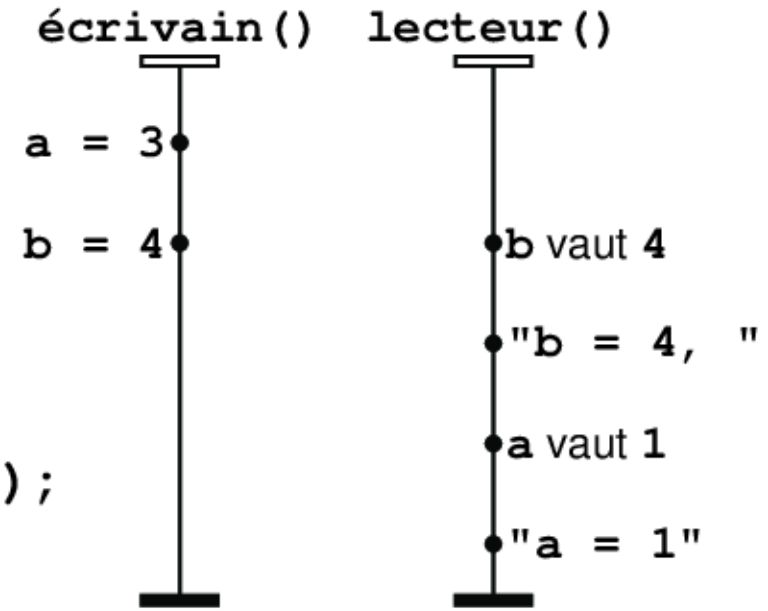
Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2020-2021

Version du 14 novembre 2020

Une exécution (déjà vue) qui n'est pas séquentiellement consistante

```
class NotSoSimple {
  int a = 1, b = 2;
  void écrivain() {
    a = 3;
    b = 4;
  }
  void lecteur() {
    System.out.print("b = " + b +", ");
    System.out.println("a = " + a);
  }
}
```



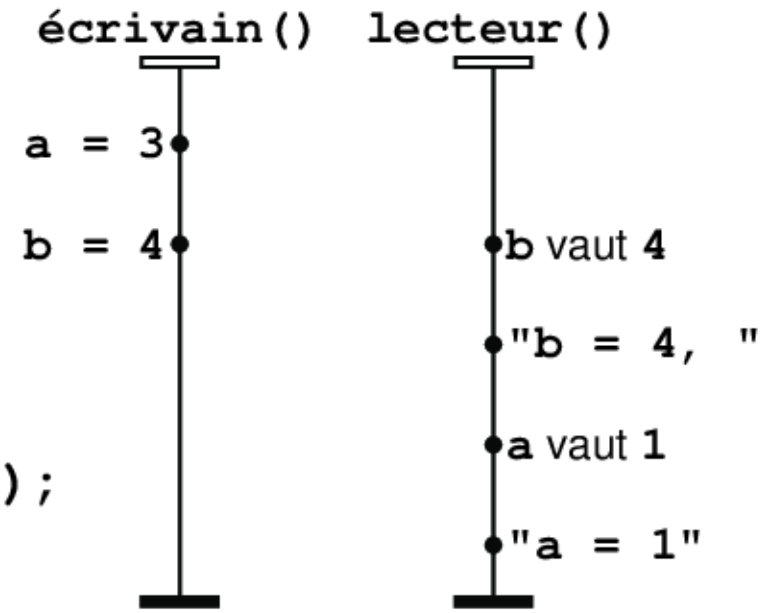
Ce programme peut légalement afficher **`"b = 4, a = 1"`**. Mais comment ?

① L'écriture en mémoire de la nouvelle valeur de **`a`** ne semble pas avoir été réalisée, *ou bien*

② Le programme ne semble pas s'exécuter dans l'ordre des instructions de **`écrivain()`**.

Une exécution qui n'est pas séquentiellement consistante : « explications »

```
class NotSoSimple {
  int a = 1, b = 2;
  void écrivain() {
    a = 3;
    b = 4;
  }
  void lecteur() {
    System.out.print("b = " + b +", ");
    System.out.println("a = " + a);
  }
}
```



① `a ← 3; b ← 4; "b=4, "; "a=1"`

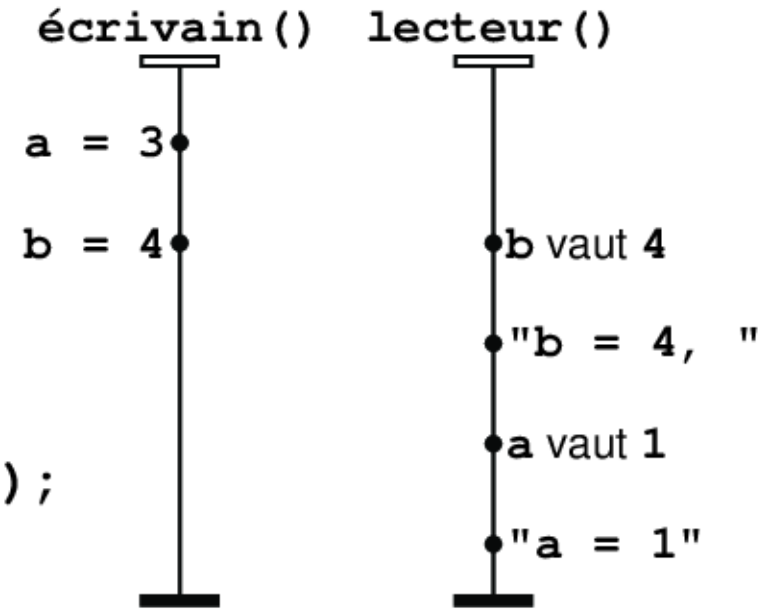
⇝ la modification de la valeur **a** en mémoire n'est pas vue par le lecteur.

② `b ← 4; "b=4, "; "a=1"; a ← 3`

⇝ l'ordre des instructions du programme n'est pas respecté par l'écrivain.

Qu'est-ce qu'une exécution séquentiellement consistante ?

```
class NotSoSimple {
  int a = 1, b = 2;
  void écrivain() {
    a = 3;
    b = 4;
  }
  void lecteur() {
    System.out.print("b = " + b +", ");
    System.out.println("a = " + a);
  }
}
```



Intuitivement, une exécution d'un programme parallèle est *séquentiellement consistante* si

① Chaque écriture en mémoire est ≪ instantanée ≫ et immédiatement visible ;

② Les instructions sont effectuées dans l'ordre du programme.

Modèle de consistance mémoire

Principe de la consistance séquentielle (par L. Lamport, prix Turing 2013)

≪ ... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. ≫

Leslie Lamport, IEEE Trans. Comput. C-28,9 (1979), 690-691,

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.

Autrement dit, il doit exister un **ordre total** sur l'ensemble des instructions exécutées par le programme (appelé une *trace d'exécution*) tel que

① Les instructions du code de chaque thread ne sont pas permutées !

② Chaque écriture en mémoire est visible par toutes les opérations ultérieures.

un peu comme si le programme était exécuté sur une machine monoprocesseur, sans mémoire cache, ni pipeline.

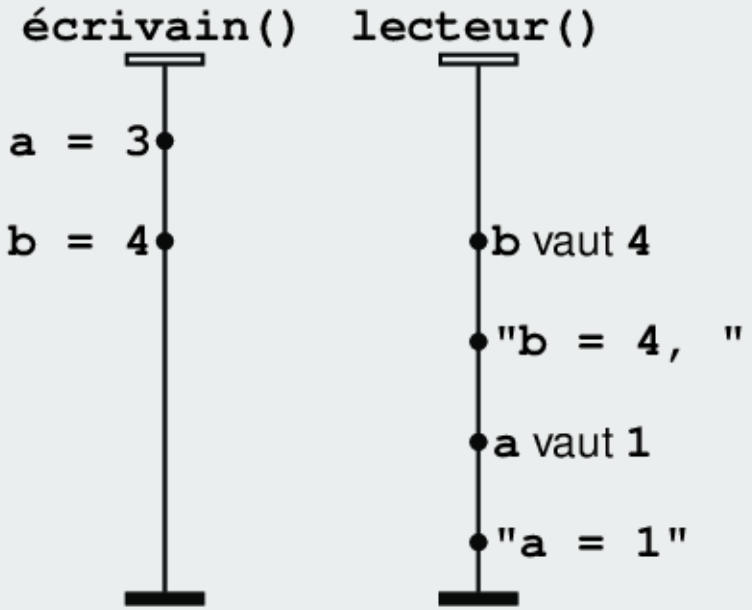
Mauvaise nouvelle (nous venons de le voir...) :

Le modèle mémoire Java *ne garantit pas* la consistance séquentielle de toutes les exécutions.

- ✔ *Qu'est-ce que la consistance séquentielle ?*
- ☛ *La notion de data-race*

Data-race et programmes bien synchronisés en Java

Une *exécution légale* d'un programme Java contient une **data-race** si elle comporte une première instruction de lecture ou d'écriture sur une variable *non volatile (ni atomique)* et une seconde instruction d'écriture *sur cette même variable* qui ne sont pas reliées par la relation ≪ happens before ≫.



Jargon :

On dit qu'un programme est ≪ **bien synchronisé** ≫ ou **DRF (pour ≪ data-race free ≫)** si aucune de ses exécutions légales ne contient de *data-race*.

Garantie des programmes bien synchronisés (en Java)

Bonne nouvelle :

≪ Le modèle mémoire Java garantit que les exécutions d'un programme **sans data-race** sont toutes **séquentiellement consistantes !** ≫

C'est bien ce que l'on veut !

Garantie des programmes bien synchronisés

Bonne nouvelle :

> « Le modèle mémoire Java garantit que les exécutions d'un programme **sans data-race** sont toutes **séquentiellement consistantes !** »

C'est bien ce que l'on veut !

Ce que ça veut dire en pratique.

Si la seule explication possible d'un résultat inattendu du programme est que

- les écritures n'ont pas été correctement réalisées en mémoire, ou
- les instructions du programme n'ont pas été réalisées dans l'ordre

c'est-à-dire que l'exécution observée n'est pas *séquentiellement consistante*, alors le programme contient **nécessairement** une *data-race*.

Ça peut aider pour débugger !

Au-delà de la garantie DRF

Un programme produira uniquement des exécutions **séquentiellement consistantes** s'il satisfait l'une des conditions suivantes :

① Chaque variable partagée est manipulée systématiquement avec un verrou donné.

② Le programme ne contient pas de data-race. **Pas très pratique !**

③ Chaque variable partagée est ou bien déclarée *volatile*, ou bien dans un objet ou un tableau *atomique*, ou bien protégée avec un *verrou*. **Plus simple !**

Néanmoins,

- pour démontrer ces affirmations, il faut être un expert du modèle mémoire Java...
- pour appliquer correctement la seconde règle, il faut aussi être un expert du modèle mémoire et pouvoir affirmer que l'on a considéré toutes les exécutions potentielles...
- l'absence de data-race peut aussi résulter simplement de l'emploi des méthodes `start()`, `join()`, `submit()`, ou encore `get()` appliquée à un objet `Future`.

En résumé

Le modèle mémoire Java formalise les exécutions légales d'un programme Java et détermine à quel moment les modifications effectuées par un thread seront nécessairement *vues* par un autre thread.

Il définit en particulier la sémantique du mot-clef **`volatile`** ; celui-ci induit une visibilité, comme les verrous.

Certaines méthodes des threads : **`start()`**, **`join()`**, **`interrupt()`**, etc. ou des tâches : **`submit()`**, **`call()`**, **`get()`**, etc. garantissent également des synchronisations de la mémoire.

Enfin, puisque Java garantit la *consistance séquentielle* des exécutions sans data-race, tout programme inconsistant résulte nécessairement d'un défaut de synchronisation.

Que veut dire le mot `final` ?

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2020-2021

Version du 14 novembre 2020

Une situation de data-race

```
class A {
  int f;
  public A() { f = 42 ; }
}
class B {
  A a;
  static void écrivain() { a = new A() ; }
  static void lecteur() { if ( a != null ) println(a.f) ; }
}
```

Deux threads appliquent simultanément et respectivement les deux méthodes **`écrivain()`** et **`lecteur()`** sur un même objet de la classe B.

Que peut afficher le second thread ?

Réponse qui fait peur

```
class A {
  int f;
  public A() { f = 42 ; }
}
class B {
  A a;   // n'est pas déclarée volatile! Il y a une data-race
  static void écrivain() { a = new A() ; }
  static void lecteur() { if ( a != null ) println(a.f) ; }
}
```

Le thread appliquant **`lecteur()`** pourra afficher :

- **`"42"`**, si le premier thread est suffisamment rapide pour construire **`a`** ;
- rien, si le second thread est trop rapide pour voir l'objet **`a`** construit ;
- **`"0"`**, car rien n'assure qu'il voit l'objet **`a`** complètement, même s'il est construit ;
- ou encore : **`NullPointerException`** car rien n'assure que la seconde lecture de la référence **`a`** ne renvoie pas **`null`**...

Code corrigé par un expert (mais avec encore une data-race)

```
class A {
  final int f;
  public A() { f = 42 ; }
}
class B {
  A a;    // n'est pas déclarée volatile! Il y a une data-race
  static void écrivain() { a = new A() ; }
  static void lecteur() {
    A copie = a ;
    if ( copie != null ) println(copie.f) ;
  }
}
```

Le thread appliquant **`lecteur()`** pourra maintenant uniquement afficher :

- rien, s'il est trop rapide pour observer que l'objet **`a`** a été construit ;
- ou **`"42"`**, si le premier thread est suffisamment rapide pour construire **`a`**.

Cas particulier des champs déclarés `final`

Un champ **`final`** ne peut voir sa valeur fixée qu'une seule fois, et ne peut plus voir sa valeur changée une fois totalement exécuté le constructeur de l'objet qui le porte.

Premier cas

```
public class MaClasse {
  private final int monChamp = 3;
...
```

Second cas

```
public class MaClasse {
  private final int monChamp;
  public MaClasse() {
    ...
    monChamp = 3;
    ...
```

Cas particulier des champs déclarés `final`

≪ *Un champ* ***`final`*** *ne peut voir sa valeur fixée qu'une seule fois, et ne peut plus voir sa valeur changée une fois totalement exécuté le constructeur de l'objet qui le porte.* ≫

De plus, la javadoc dit :

≪ Fields declared final are initialized once, but never changed under normal circumstances ≫.

Néanmoins il existe des circonstances anormales !

Que veut dire le mot `final` ?

`https://docs.oracle.com/javase/specs/...:`

≪ An object is considered to be completely initialized when its constructor finishes. A thread that can only see a reference to an object after that object has been completely initialized is guaranteed to see the correctly initialized values for that object's final fields. ≫

Selon le modèle mémoire, quand la construction d'un objet est terminée, la référence de cet objet devient disponible pour le thread qui crée l'objet. De plus, les valeurs (définitives) des champs déclarés **`final`** sont alors visibles par tout thread qui accède ultérieurement à cet objet.
Ainsi, *une fois l'objet construit*, le champ **`final`** peut être accédé par différents threads *sans synchronisation*.

On obtient la garantie d'une visibilité sans synchronisation !

Néanmoins, il est parfois possible de lire un champ **`final`** avant que le constructeur ait terminé, c'est-à-dire *pendant la construction de l'objet*. Il n'y a alors **aucune garantie** sur la valeur retournée.

Comparaison entre `final` et `volatile`

```
class A {
  int e ;
  final int f ;
  public A() { e = 21 ; f = 42 ; }
}
class B {
  A a;
  static void écrivain() { a = new A() ; }
  static void lecteur() {
    A copie = a ;
    if ( copie != null ) println(copie.f + "␣" + copie.e) ;
  }
}
```

Le thread appliquant **`lecteur()`** pourra afficher **`"42 0"`**, car seul **`f`** est déclaré **`final`** : aucune garantie n'est accordée à **`e`** même s'il semble initialisé *avant* **`f`**.

Très mauvaise pratique : publication prématurée de `this`

```
class A {
  final int f;
  public A(B b) { f = 42 ; b.a = this ; }
}
class B {
  A a;
  static void écrivain() { new A(this) ; }
  static void lecteur() {
    A copie = a ;
    if ( copie != null ) println(copie.f) ;
  }
}
```

Le thread appliquant **`lecteur()`** pourra afficher **`"0"`**, car la référence **`a`** est potentiellement disponible avant la fin de la construction de l'objet ! Le mot-clef **`final`** est alors sans effet.

Pour conclure

Vous avez appris dans cette UE :

- à utiliser les fonctionnalités de Java pour améliorer les performances d'un programme en répartissant la tâche globale sur plusieurs threads ou, mieux, sur un réservoir de threads ;
- à garantir l'atomicité des accès aux données partagées en utilisant des verrous, des objets atomiques ou des collections spécifiques ;
- à concevoir et à implémenter des objets « threadsafe » sous la forme de moniteurs ou même sans verrou ;
- à analyser et à éviter les interblocages et le problème ABA.

Plus précisément, puisqu'il s'agit de Java, vous savez à présent :

- utiliser **`synchronized`** et s'il le faut **`volatile`** ou **`final`** ;
- protéger chaque **`wait`** par un **`while`** approprié ;
- préférer en général **`notifyAll()`** plutôt que **`notify()`** ;
- détecter et corriger un programme mal synchronisé.