Consistance séquentielle en Java

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2022-2023

Version du 26 novembre 2022

Un problème de vue constaté en Travaux Pratiques

```
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 de la fin
    System.out.println(valeur);
  }
}
```

Ce programme termine-t-il toujours ?

Notion de trace d'exécution

Une *trace d'exécution* d'un programme est une séquence d'instructions élémentaires, effectuant des opérations de lecture et d'écriture sur les données, qui représente l'ensemble des actions réalisées au cours de l'exécution.

Les opérations réalisées par chaque thread doivent respecter l'**ordre du programme**.

Exemples :

$$W_{\texttt{main}}(\text{valeur}=1)\,;\, W_{\texttt{main}}(\text{fin}=\text{true})\,;\, R_{\texttt{a}}(\text{fin}=\text{true})\,;\, R_{\texttt{a}}(\text{valeur}=1)\,;$$

$$W_{\text{main}}(\text{valeur}=1)\,;\, W_{\text{main}}(\text{fin}=\text{true})\,;\, R_{\text{a}}(\text{fin}=\text{false})\,;\, R_{\text{a}}(\text{fin}=\text{false})\,;\, R_{\text{a}}(\text{fin}=\text{false})\,;\ldots$$

La spécification du langage Java détermine l'ensemble des *traces d'exécution légales* d'un programme.

Un peu d'histoire

La question de savoir ce que signifie l'exécution correcte d'un programme multithread par une machine multiprocesseur s'avère relativement ancienne.

> ≪*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*.

On retrouve déjà à l'époque chez Lamport l'idée selon laquelle une exécution d'un programme multithread sur une machine multiprocesseur doit pouvoir être regardée comme un **entrelacement des opérations réalisées par chaque thread** et que cette séquence d'actions doit respecter **l'ordre des instructions** indiqué par le programme.

La notion de consistance séquentielle selon Java

Une trace d'exécution est *séquentiellement consistante* si chaque lecture d'une variable voit la valeur déterminée par l'écriture la plus récente dans cette variable.

> ≪ *Sequential consistency is a very strong guarantee that is made about visibility and ordering in an execution of a program. Within a sequentially consistent execution, there is a total order over all individual actions (such as reads and writes) which is consistent with the order of the program, and each individual action is atomic and is immediately visible to every thread.* ≫
> *The Java Language Specification, Third edition, p. 560*.

Exemple :

$$W_{main}(\text{valeur}=1)\ ;\ W_{main}(\text{fin}=\text{true})\ ;\ R_a(\text{fin}=\text{false})\ ;\ R_a(\text{fin}=\text{false})\ ;\ R_a(\text{fin}=\text{false})\ ;\ldots$$

Cette trace d'exécution est légale ; mais elle n'est pas séquentiellement consistante, puisque les lectures de la variable **`fin`** par le thread **`a`** ne voient pas la valeur **`true`** écrite dans cette variable par le thread **`main`**.

Intuition sur l'origine des inconsistances séquentielles

Nous verrons aujourd'hui qu'en Java les exécutions non séquentiellement consistantes résultent toujours d'un défaut de synchronisation du programme.

Autrement dit, seuls les programmes « insuffisamment synchronisés » présentent des exécutions inconsistantes séquentiellement.

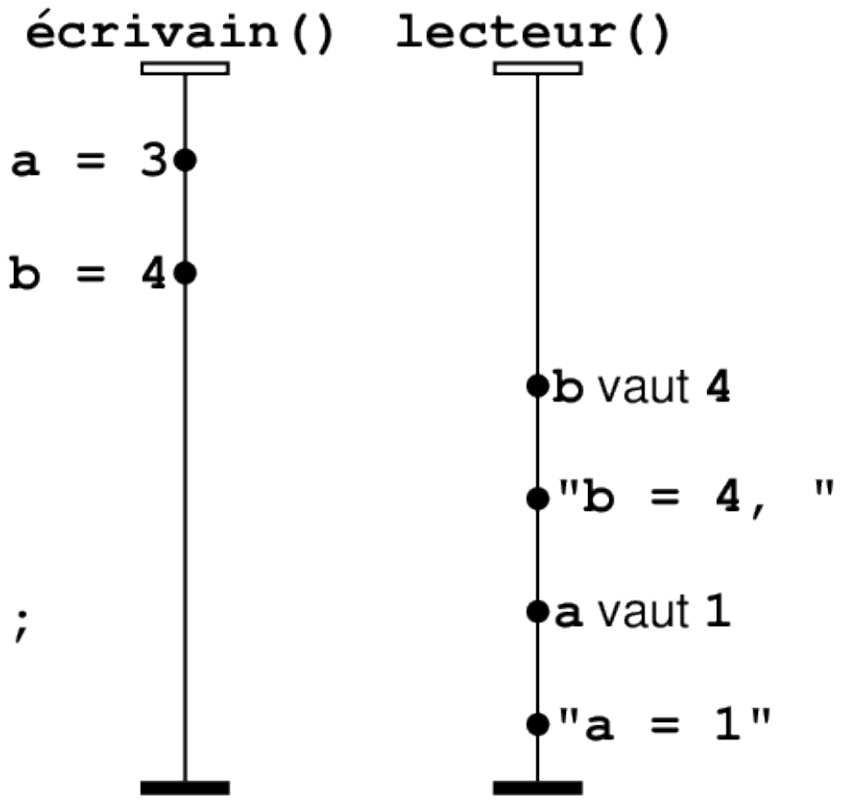
Celles-ci correspondent intuitivement à la lecture de variables placées dans la mémoire cache des processeurs, en l'absence de mise-à-jour adéquate des données par le protocole de cohérence mis en œuvre par la machine.

En réalité, la difficulté sous-jacente a également d'autres origines :

- Les compilateurs peuvent **supprimer** des instructions.
- Les compilateurs peuvent **réordonner** les instructions.
- Les **processeurs** peuvent aussi réordonner les instructions !

Second exemple de programme insuffisamment synchronisé

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



Si deux threads appliquent respectivement les méthodes **`écrivain()`** et **`lecteur()`**, le lecteur peut légalement afficher à l'écran **`"b = 4, a = 1"`**.

Dans cette exécution, seule la modification de la valeur de **`b`** par l'écrivain est vue par le lecteur.

Troisième exemple de programme insuffisamment synchronisé

```
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 les méthodes **`écrivain()`** et **`lecteur()`** sur un même objet de la classe **`B`**.

Que peut afficher le thread lecteur ?

✔ *Consistance séquentielle en Java*

☛ *Solutions empiriques*

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) { System.out.print(""); } ;  // Attente active
    System.out.println(valeur);
  }
}
```

Ce programme termine-t-il toujours ?

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) { yield(); } ;     // Attente de la fin
    System.out.println(valeur);
  }
}
```

Ce programme termine-t-il toujours ?

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) { sleep(10); } ;     // Attente de la fin
    System.out.println(valeur);
  }
}
```

Ce programme termine-t-il toujours ?

Conclusions pratiques

L'instruction `System.out.print("");` peut avoir un effet sur la visibilité des modifications de la valeur d'une donnée partagée non volatile.

En conséquence, un programme qui semble fonctionner correctement pourra ne plus fonctionner correctement en supprimant des affichages à l'écran !

De même, les instructions `yield()` et `sleep()` n'ont pas vocation à modifier la visibilité des données partagées ; mais en pratique, elles peuvent pallier à un défaut de synchronisation.

Ces techniques empiriques n'offrent aucune garantie !

Aperçu du modèle mémoire Java

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2022-2023

Version du 26 novembre 2022

Retour au problème de vue initial

```
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 peut-il afficher 0 ?

À quoi sert le modèle mémoire Java ?

Le modèle mémoire Java définit la sémantique des programmes Java multithread, c'est-à-dire l'ensemble des traces d'exécutions légales d'un programme donné.

En particulier, le modèle mémoire Java détermine l'effet du mot-clef **`volatile`**, de l'emploi d'objets atomiques et des opérations sur les verrous.

Il décrit aussi les effets d'instructions telles que **`start()`**, **`join()`**, **`wait()`**, etc.

✔ *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 ;

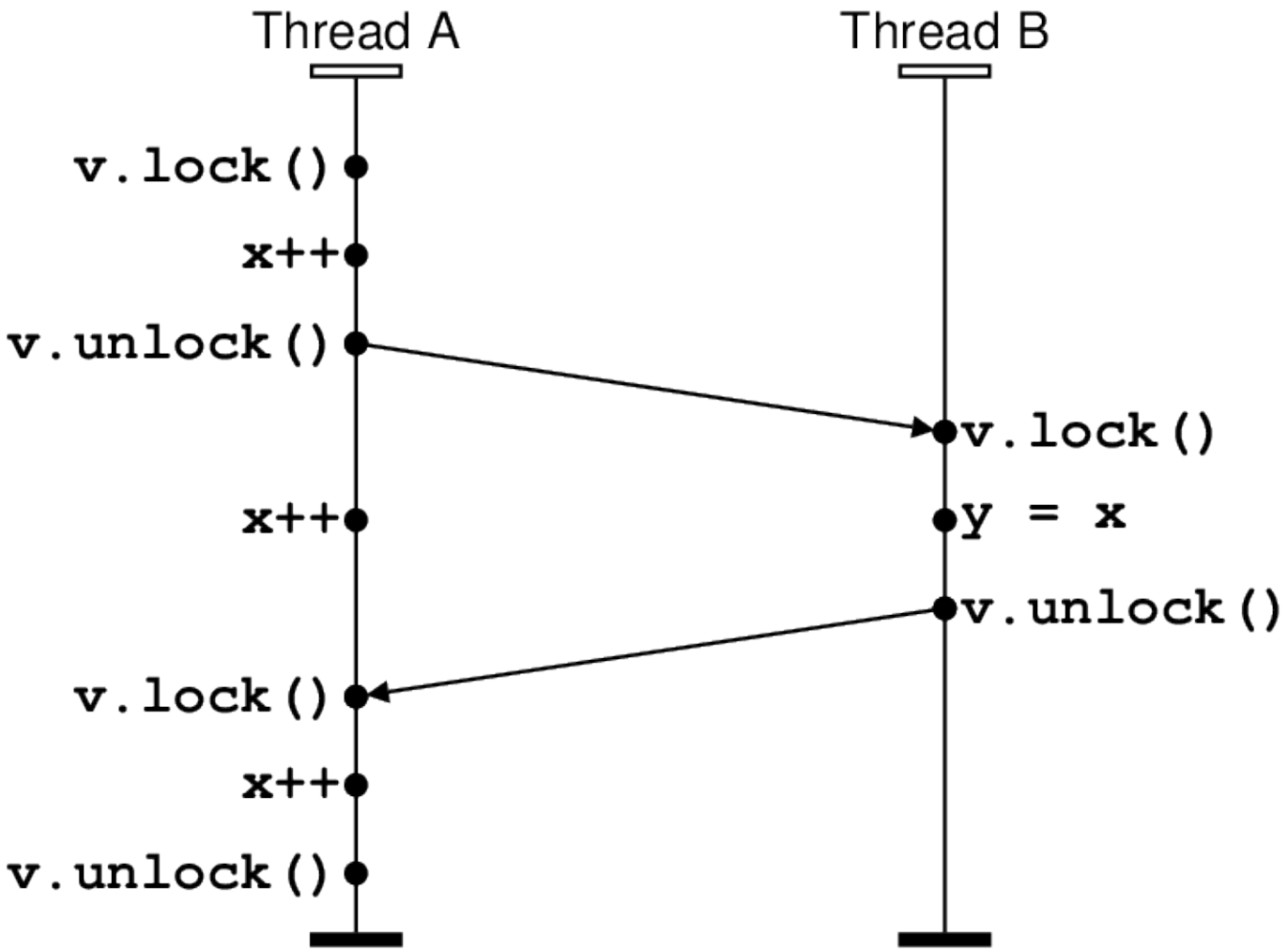
② certaines actions qui produisent des **synchronisations** entre threads.

Par exemple, les opérations sur un verrou sont **totalement ordonnées** par $<_{\mathrm{HB}}$.

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

Exemple d'exécution légale (avec deux synchronisations)

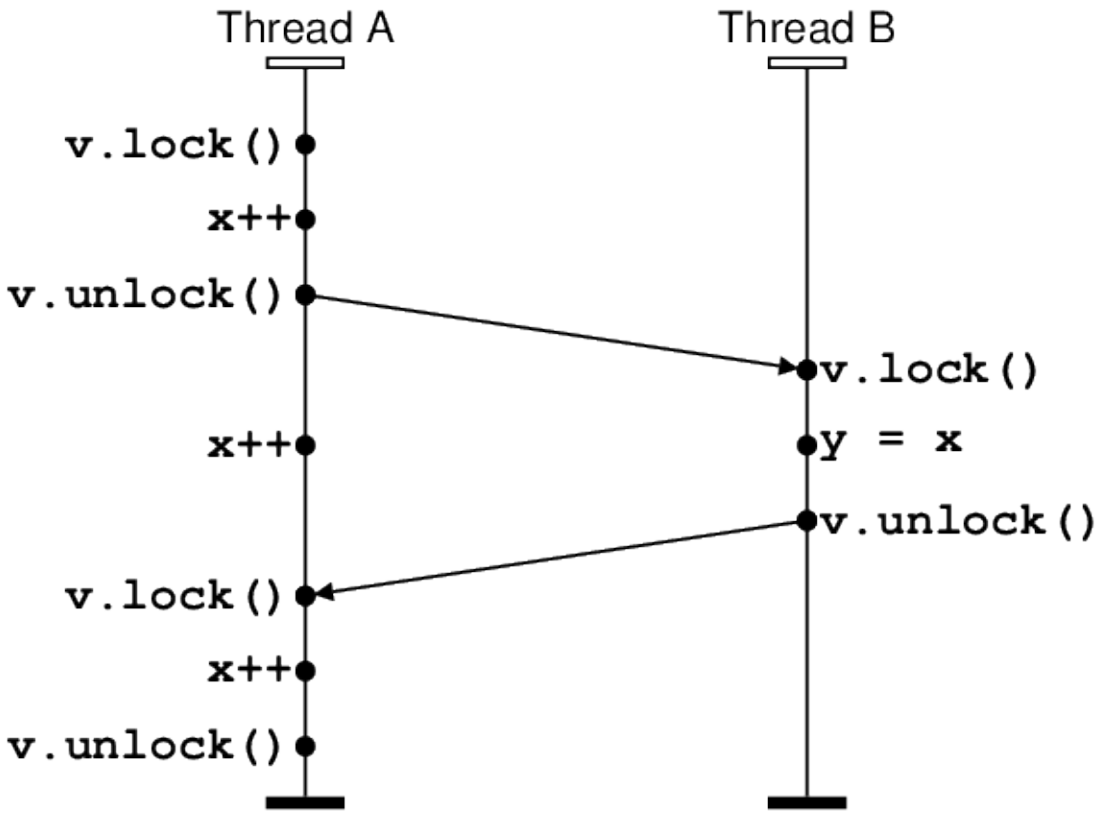
Initialement `x=0` et `y=0`



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

Il faut distinguer la relation ≪ a lieu avant ≫ et l'observation effective

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



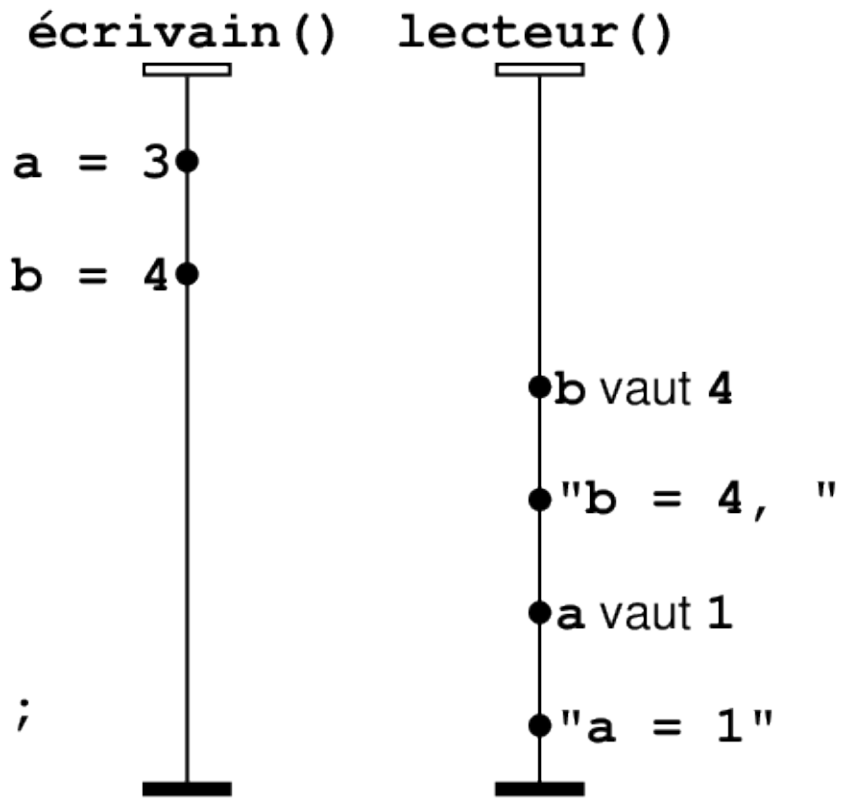
- ⇝ "`y = x`" doit observer la première incrémentation de `x`, qui a lieu ≪ avant. ≫
- ⇝ "`y = x`" ne peut pas observer la 3ième incrémentation de `x`, car elle a lieu ≪ après. ≫
- ⇝ "`y = x`" observera ou non la seconde incrémentation de `x`.
- ⇝ à la fin de cette exécution, `y=1` ou `y=2` (mais jamais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

Retour au second exemple de programme

```
class Simple {
  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 afficher `"b = 4, a = 1"` *!*

Retour au problème de vue initial

```
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 peut afficher 0 !

- ✔ *Introduction*
- ✔ *La notion d'exécution (de manière simplifiée)*
- ☛ *Ordre total des actions sur chaque verrou*

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()`**.

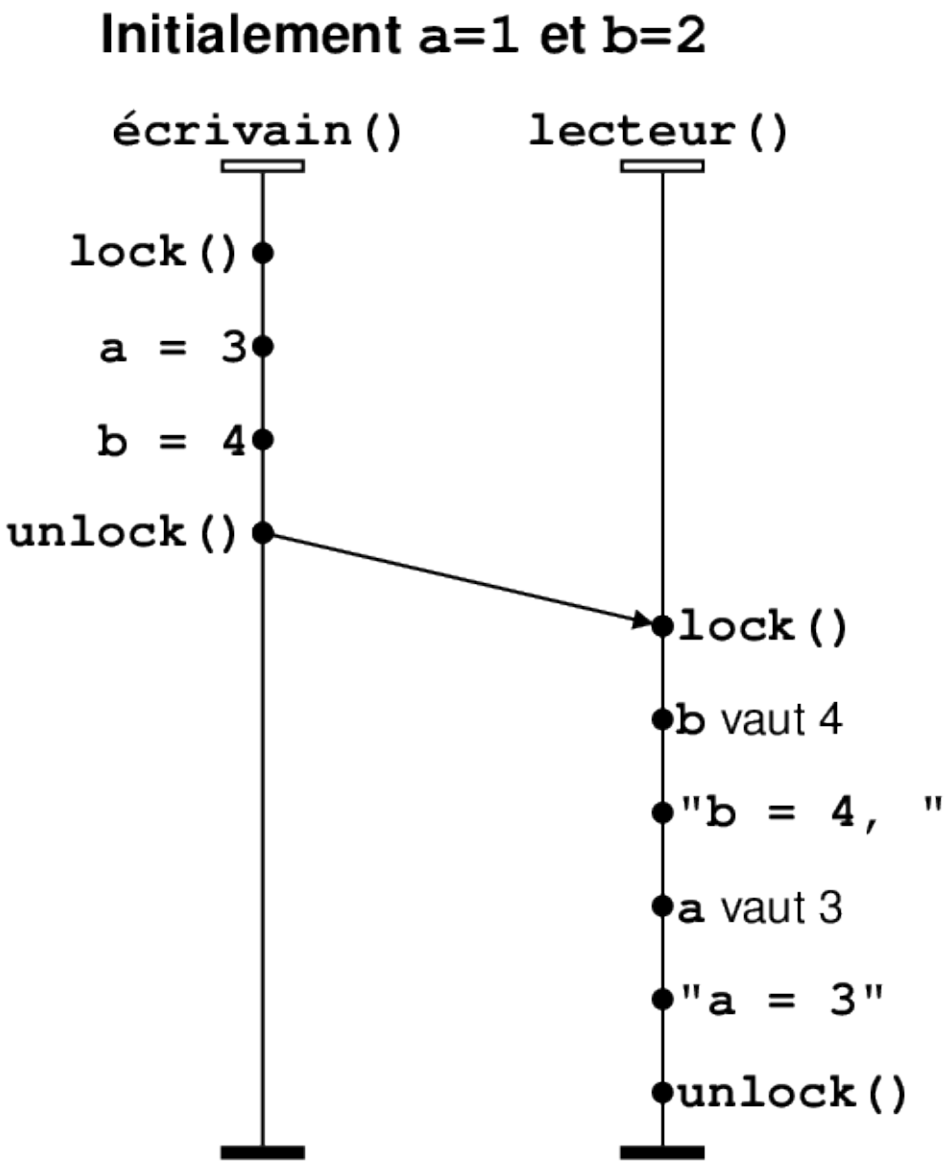
Visibilité et verrou : un exemple simple

```
class SynchronizedSimple {
  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 :

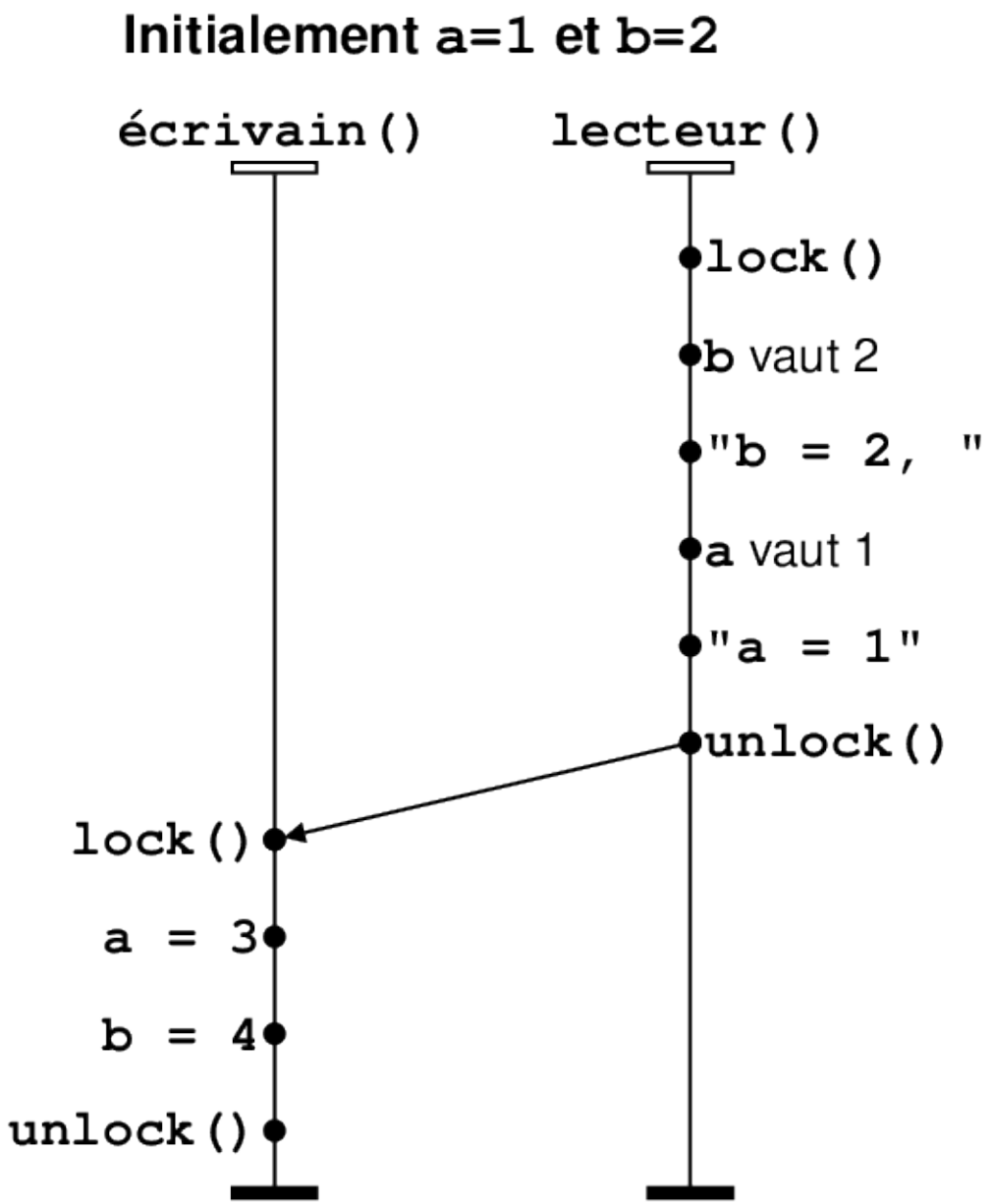
- **`"b=2, a=1"`** si le lecteur prend le verrou en premier ;
- **`"b=4, a=3"`** si l'écrivain prend le verrou en premier.

Une exécution légale



Grâce au verrou, chaque lecture voit la dernière écriture réalisée.

Une seconde exécution légale



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.* ≫

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

Les verrous de lecture-écriture assurent une synchronisation

Sans surprise, les opérations sur le verrou d'écriture sont totalement ordonnées, comme pour **`synchronized`**.

En outre, la documentation de Java indique que l'acquisition du *verrou de lecture* implique la visibilité des opérations réalisées préalablement avec le *verrou d'écriture*.

> ≪ *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, les opérations sur les timbres d'écriture ou de lecture ont le même effet que celles réalisées avec les verrous de lecture/écriture :

> ≪ *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.* ≫

En revanche, une opération de lecture optimiste n'offre une garantie sur la visibilité des opérations en mémoire précédentes que si elle est validée postérieurement.

> ≪ *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.* ≫

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*.* ≫

- ✔ *Introduction*
- ✔ *La notion d'exécution (de manière simplifiée)*
- ✔ *Ordre total des actions sur chaque verrou*
- ☛ *Le modificateur volatile assure la visibilité*

Retour au premier exemple avec ajout de « volatile »

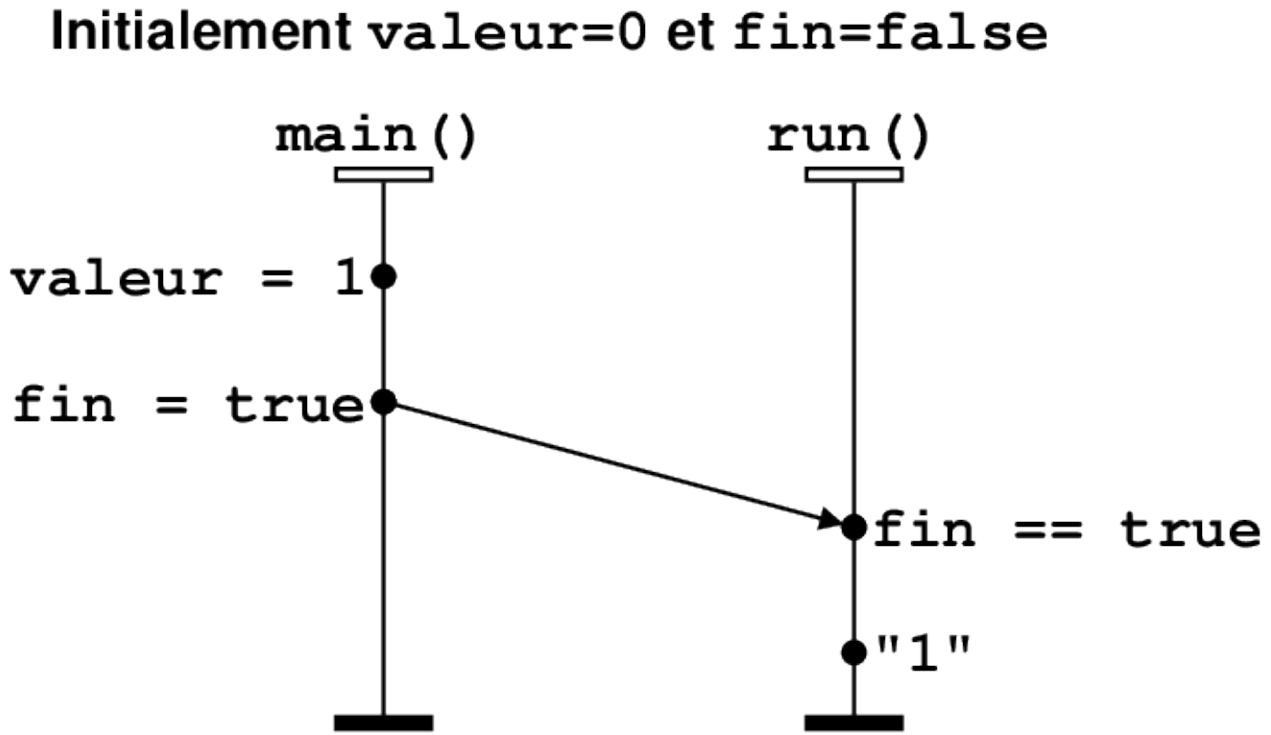
```
public static void main(String[] args) {
  A a = new A();          // Création d'un objet a de la classe A
  a.start();              // Lancement du thread a
  a.valeur = 1;           // Modification de l'attribut valeur
  a.fin = true;           // Modification de l'attribut fin
}
static class A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

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

Ce programme peut-il afficher 0 ?

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

Une exécution doit, pour chaque variable volatile, comporter une **synchronisation** de chaque *écriture* vers toutes les *lectures 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.

- ✔ *Introduction*
- ✔ *La notion d'exécution (de manière simplifiée)*
- ✔ *Ordre total des actions sur chaque verrou*
- ✔ *Le modificateur volatile assure la visibilité*
- ☛ *Les valeurs des atomiques sont volatiles*

Volatilité des objets atomiques de Java

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.

Autrement dit,

les données conservées dans un objet atomique sont volatiles.

Le problème des données dans les tableaux

Méfiance !

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

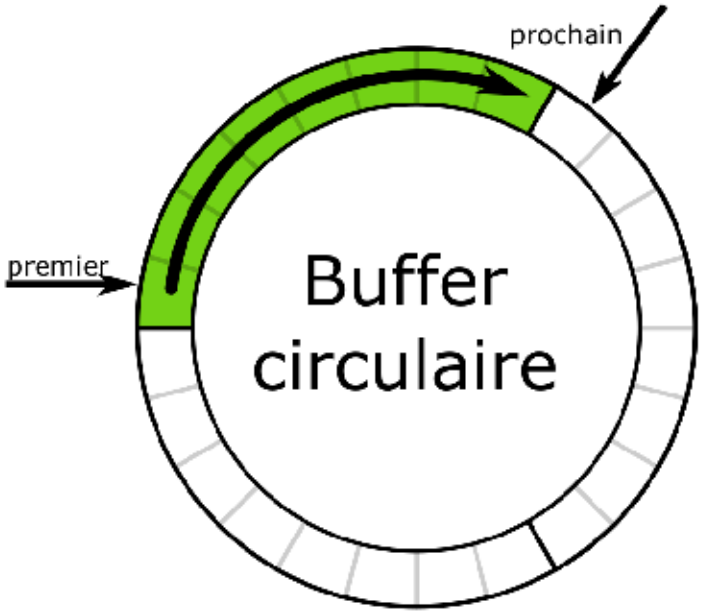
En effet, si un tableau est déclaré **`volatile`**, c'est la référence du tableau qui bénéficiera de la volatilité : les modifications des éléments du tableau par un thread ne seront pas nécessairement vues par les autres threads accédant au tableau.

Les classes dédiées **`AtomicIntegerArray`**, **`AtomicLongArray`**, etc. de Java permettent de disposer d'un tableau dont les données possèdent le caractère volatile et peuvent être manipulées comme des objets atomiques.

Pour des objets quelconques, on utilisera un **`AtomicReferenceArray`**$<$ E $>$ qui fournit un tableau de références volatiles, manipulables atomiquement.

La visibilité des éléments d'un tableau peut aussi être assurée par un verrou

```
class BufferSynchronisé {
  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() { ... }
}
```

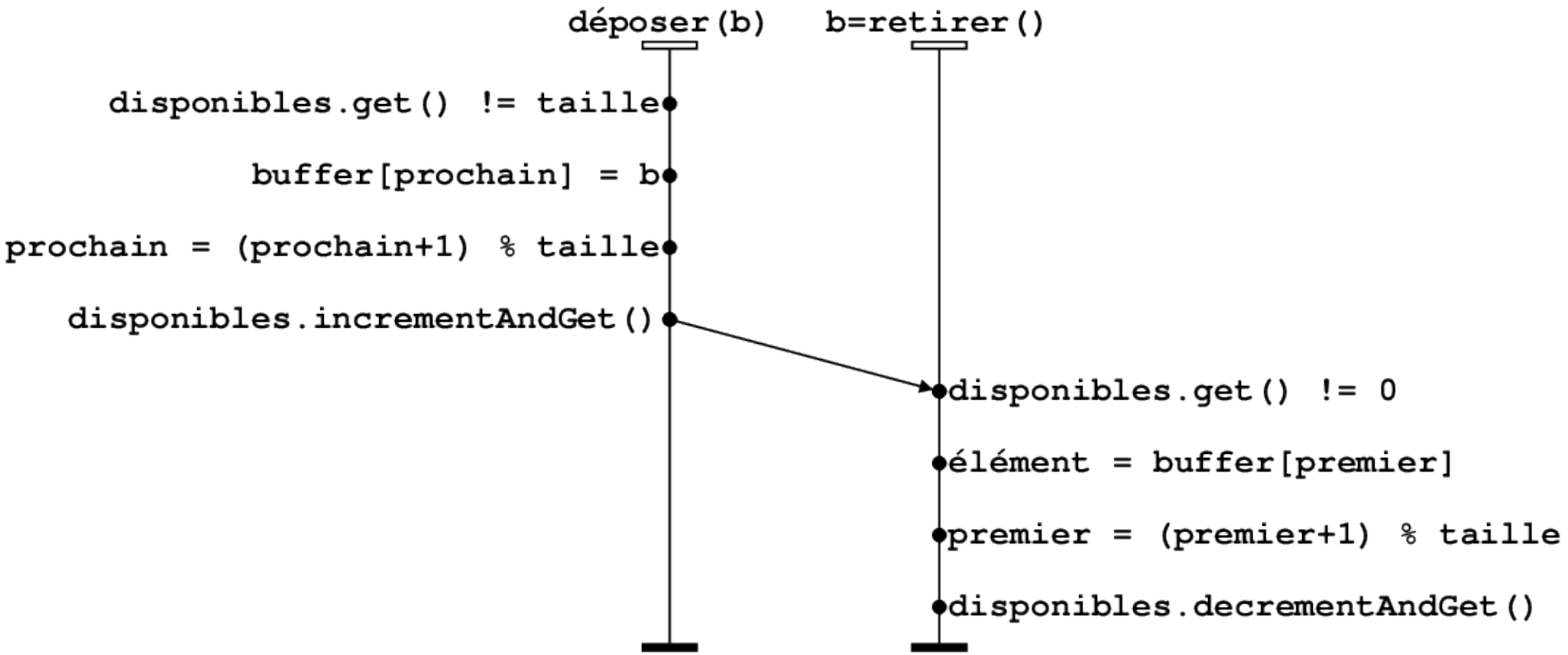


Le verrou assure la visibilité des modifications du contenu du tableau **`buffer`**.

Cas du buffer concurrent

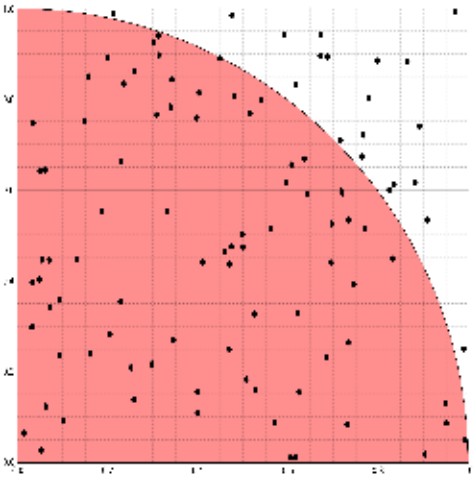
```
void déposer(byte b) {
  synchronized(dépôt){
    while (disponibles.get() == taille) Thread.yield() ;
    buffer[prochain] = b ;
    prochain = (prochain + 1) % taille ;
    disponibles.incrementAndGet() ;
  }
}
byte retirer() {
  synchronized(retrait){
    while (disponibles.get() == 0) Thread.yield() ;
    byte élément = buffer[premier] ;
    premier = (premier + 1) % taille ;
    disponibles.decrementAndGet() ;
    return élément;
  }
}
```

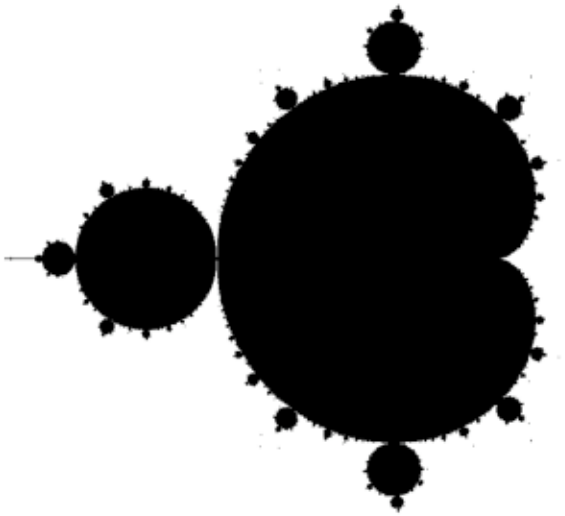
Communication d'un octet entre le producteur et le consommateur

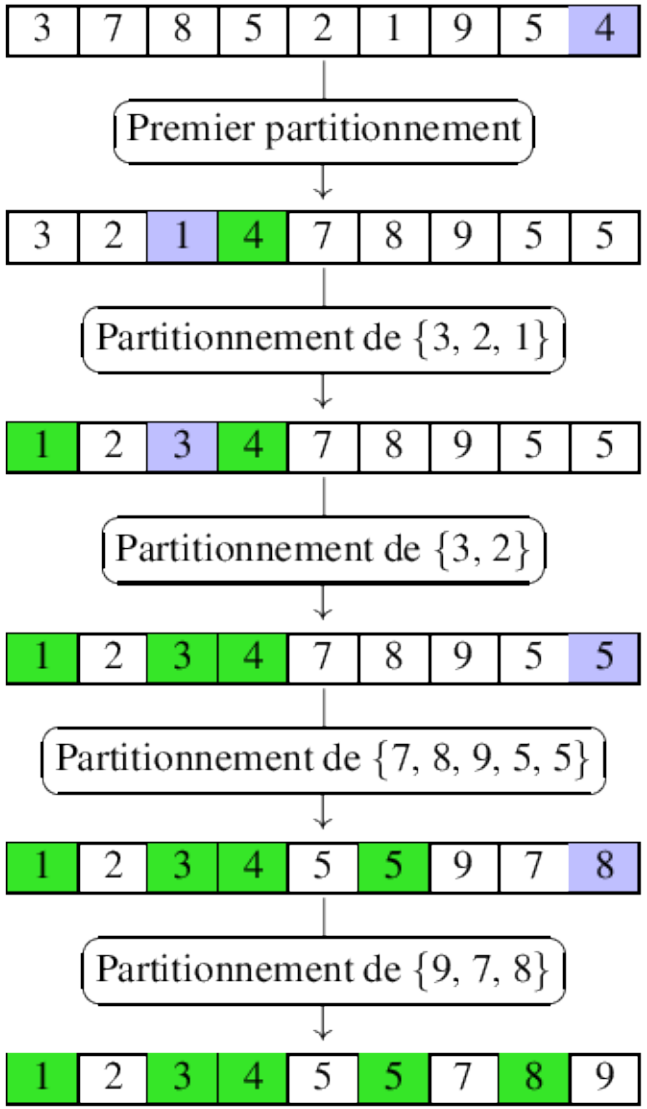


- ✔ *Introduction*
- ✔ *La notion d'exécution (de manière simplifiée)*
- ✔ *Ordre total des actions sur chaque verrou*
- ✔ *Le modificateur volatile assure la visibilité*
- ✔ *Les valeurs des atomiques sont volatiles*
- ☛ *Autres précisions utiles en pratique*

Exemples de négligences passées, mais sans conséquence pratique







Les données partagées doivent être ≪ synchronisées ≫ pour être visibles.

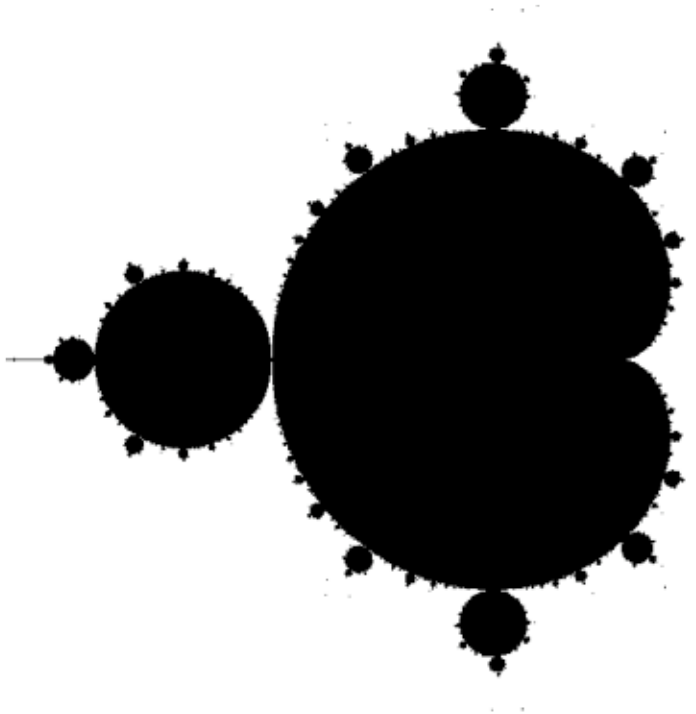
Visibilité des résultats obtenus à l'issue d'une tâche

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.

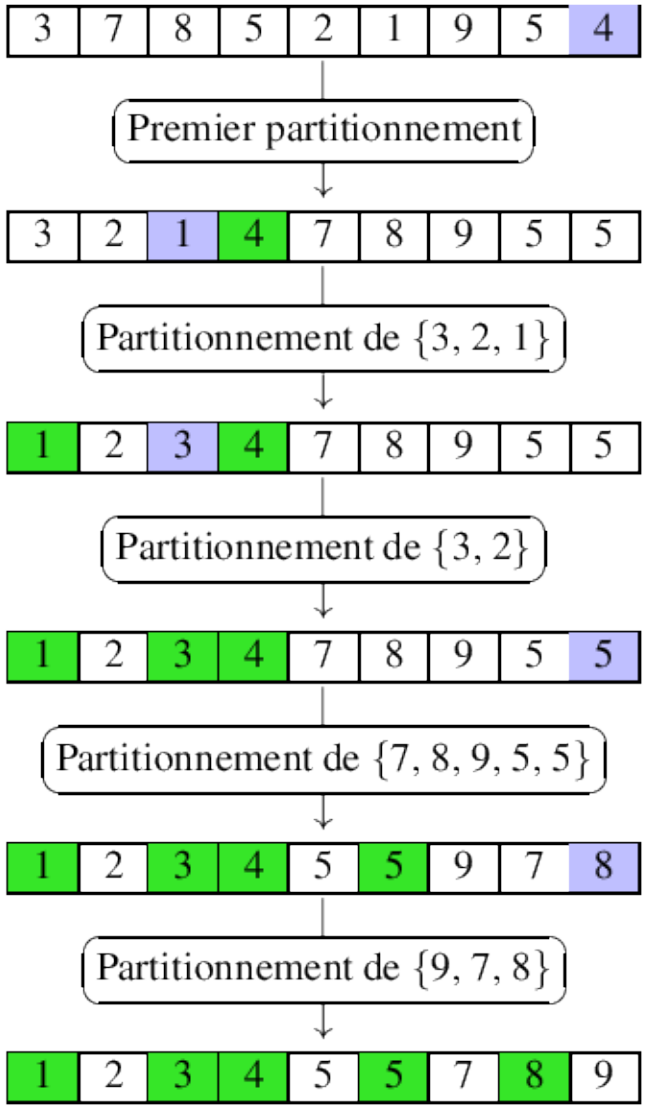


```
for (int i = 0 ; i < taille ; i++) {
  service.take() ;
} // Chaque ligne complétée sera nécessairement visible.
```

Visibilité des actions préalables à une tâche

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 tous les accès à cet objet.

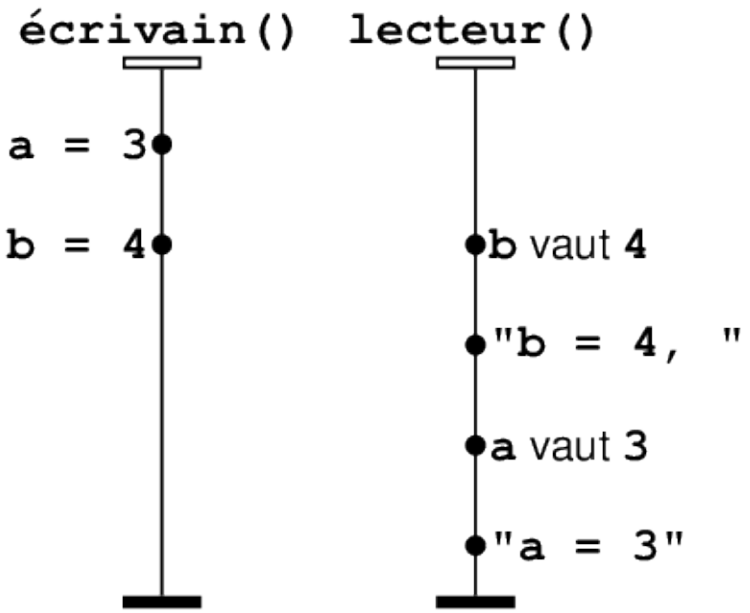
Par défaut, on lit la valeur par défaut !

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

- ✔ *Introduction*
- ✔ *La notion d'exécution (de manière simplifiée)*
- ✔ *Ordre total des actions sur chaque verrou*
- ✔ *Le modificateur volatile assure la visibilité*
- ✔ *Les valeurs des atomiques sont volatiles*
- ✔ *Autres précisions utiles en pratique*
- ☛ *Notion de data-race en Java*

La notion de data-race en Java

Une exécution séquentiellement consistante d'un programme contient une *data-race* si elle comporte une écriture sur une donnée et une autre action de lecture ou d'écriture sur cette donnée *non reliées par la relation ≪ happens before ≫.*



Rmq. 1. Les deux accès à la donnée ne sont pas *indépendants*.

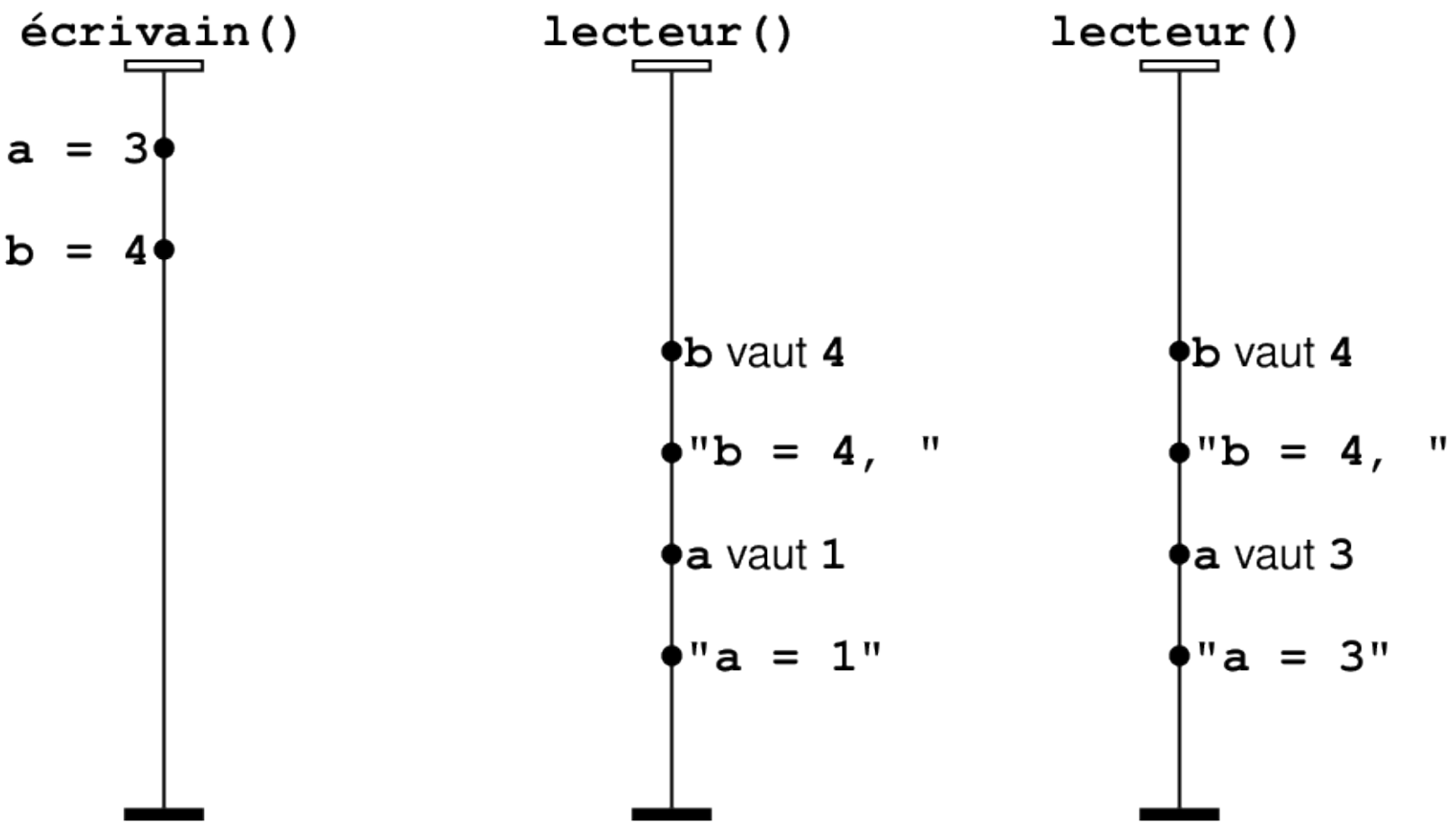
Rmq. 2. La donnée en cause est nécessairement *partagée* par deux threads.

Rmq. 3. Un programme dans lequel chaque variable partagée est correctement protégée par un **verrou** ne présente (évidemment) aucune data-race.

Rmq. 4. Il a été prouvé qu'un programme sans data-race aura uniquement des exécutions séquentiellement consistantes.

Un exemple d'exécution non séquentiellement consistante

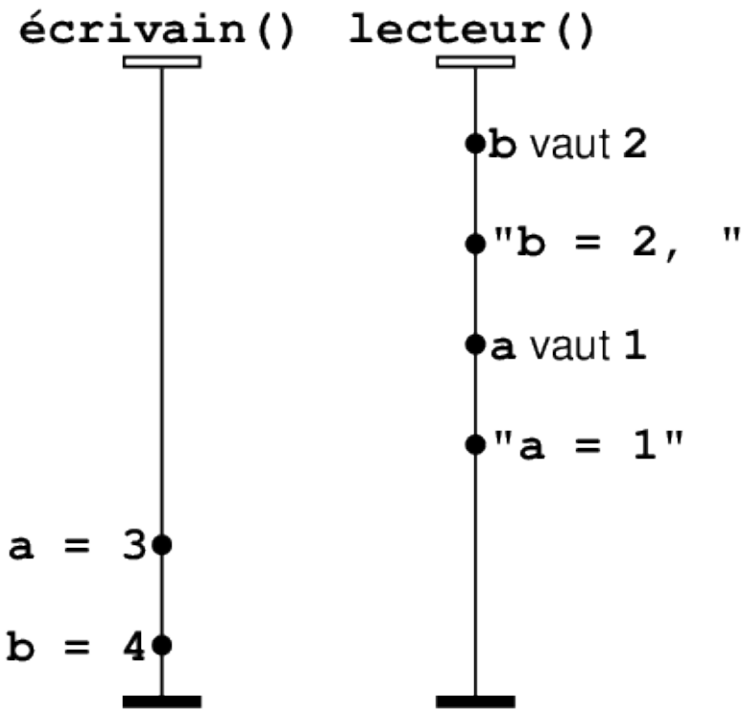
```
private int a = 1, b = 2 ; // ne sont pas déclarées volatiles !
```



Dans un programme avec data-race, certaines exécutions ne sont pas séquentiellement consistantes : les écritures ne sont nécessairement pas vues « **immédiatement** » ni même **dans le même ordre** par tous les threads...

Beaucoup de programmes simples ont des data-races !

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



Il ne suffit pas de déclarer « volatile » chaque variable partagée pour obtenir un programme sans data-race.

En résumé

Le modèle mémoire Java formalise les exécutions légales d'un programme Java ; il détermine quelles modifications des données effectuées par un thread seront nécessairement *vues* (tôt ou tard) par un autre thread.

C'est à la fois

① une *garantie* pour les développeurs ;

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

Il définit en particulier la sémantique du mot-clef **`volatile`**, des verrous (notamment **`synchronized`**) et des opérations sur les objets atomiques.

Certaines méthodes des threads : **`start()`**, **`join()`**, **`interrupt()`**, etc. ou relatives aux tâches : **`submit()`**, **`call()`**, **`get()`**, **`take()`**, etc. assurent également des synchronisations, c'est-à-dire une visibilité des données modifiées.

Il ne faut pas confondre !

Notez que le *Java Language Specification* et le *Java Virtual Machine Specification* sont des documents avec des objectifs bien distincts :

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

② le second guide les concepteurs de machines virtuelles en autorisant certaines **optimisations de code** plus ou moins aggressives, en particulier certaines **permutations d'instructions**.

Intérêt du modificateur `final`

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2022-2023

Version du 26 novembre 2022

Retour au troisième programme mal synchronisé

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

Deux threads appliquent simultanément les 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!
  void écrivain() { a = new A() ; }
  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 encore avec une data-race)

```
class A {
  final int f;
  public A() { f = 42 ; }
}
class B {
  A a;    // n'est pas déclarée volatile!
  void écrivain() { a = new A() ; }
  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`**.

Intérêt du 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.* ≫

Quand la construction d'un objet est terminée, la référence de cet objet devient disponible pour le thread qui crée l'objet. Ce thread voit correctement tous les champs de l'objet.

Mais la référence de l'objet créé peut être accessible par un autre thread qui, quant à lui, verra correctement les valeurs (définitives) des champs déclarés **`final`**. Les autres champs pourront en revanche sembler contenir la valeur par défaut...

N.B. Un champ déclaré **`final`** ne peut être aussi déclaré **`volatile`**.

Comparaison entre `final` et `volatile`

```
class A {
  int e ;
  final int f ;
  public A() { e = 21 ; f = 42 ; }
}
class B {
  A a;
  void écrivain() { a = new A() ; }
  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;
  void écrivain() { new A(this) ; }
  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, en soumettant des tâches à un réservoir de threads (threadpool) ;
- à veiller à l'**atomicité** des accès aux données partagées en utilisant des verrous, des objets atomiques ou des collections adéquates ;
- à concevoir et à implémenter des classes « threadsafe » sous la forme de **moniteurs** et même à **programmer sans verrou** ;
- à analyser et à éviter les **interblocages** ainsi que le problème **ABA** ;
- à garantir sur la **visibilité** des données partagées entre threads.

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

- utiliser correctement `synchronized` et parfois `volatile` ou `final` ;
- protéger chaque `wait` par une boucle `while` appropriée ;
- préférer en général `notifyAll()` plutôt que `notify()`.