

Outils de synchronisation en Java

Module
Systèmes Communicants et Synchronisés
Master Informatique
1ère année



L. Philippe & V. Felea

Gestion des synchronisations

- partage de ressources et du travail – concurrence avec les threads
- interférences entre threads
- concurrence d'accès → synchronisation compétitive
 - ▶ accès atomiques
- coordination d'accès → synchronisation coopérative

Interférence entre threads

● partage d'un objet

```
class Counter {  
    private int c = 0;  
    public void increment() {  
        c++;  
    }  
    public void decrement() {  
        c--;  
    }  
    public int value() {  
        return c;  
    }  
}
```

Classe

JVM

Thread1
récupère la valeur de c
incrémente la valeur obtenue
enregistre la valeur dans c

Thread 2
récupère la valeur de c
décrémente la valeur obtenue
enregistre la valeur dans c

Exécution

Thread 1: récupère c.

Thread 2: récupère c.

Thread 1: incrémente la valeur ; résultat 1.

Thread 2: décrémente la valeur ; résultat -1.

Thread 1: enregistre dans c ; c est à 1.

Thread 2: enregistre dans c ; c est à -1.

Accès cache

- processeurs multicœurs
- 1 cache/cœur
- accès aux données du cache plutôt que mémoire principale

```
$ java CounterMultiCore
a= 18968
$ java CounterMultiCore
a= 19981
$ java CounterMultiCore
a= 20000
.....
```

```
class MaDonnee {
    public int a;
    public MaDonnee() { a = 0; }
}

class MonThread extends Thread {
    MaDonnee md;
    int nbIter = 10000;

    public MonThread(MaDonnee ref) { md=ref; }
    public void run() {
        for (int i=0; i<nbIter; i++) md.a++; }
}

public class CounterMultiCore {

    public static void main(String args[]) {
        try {
            MaDonnee md = new MaDonnee();
            MonThread mt1 = new MonThread(md);
            MonThread mt2 = new MonThread(md);
            mt1.start();
            mt2.start();
            mt1.join();
            mt2.join();
            System.out.println(" a= "+md.a);
        } catch (Exception ex) {
            System.out.println(ex);
        }
    }
}
```

Cohérence données

- partage
 - ▶ mémoire
 - ▶ ressources
- ordonnancement non maîtrisé
- zones critiques :
cohérence $b = a + 10$?

```
$ java Concurrency
```

```
Thread-0 a = 20 b = 30
```

```
Thread-1 a = 20 b = 30
```

```
Thread-0 a = 1 b = 11
```

```
Thread-1 a = 21 b = 31
```

```
Thread-0 a = 2 b = 12
```

```
Thread-1 a = 22 b = 32
```

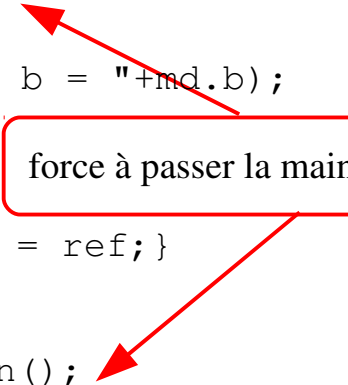
```
Thread-0 a = 23 b = 13
```

```
class MaDonnee {
    int a, b;
    public void setA(int sa) { a = sa ; }
    public void setB(int sb) { b = sb ; }
}

class MonThread1 extends Thread {
    MaDonnee md;
    public MonThread1( MaDonnee ref ) {md = ref;}
    public void run() {
        for (int i=0; i<100; i++) {
            md.setA(i) ; System.out.println();
            md.setB(10+i);
            S.o.p(this.getName()+" a = "+md.a+" b = "+md.b);
        } }
}

class MonThread2 extends Thread {
    MaDonnee md;
    public MonThread2( MaDonnee ref ) {md = ref;}
    public void run() {
        for (int i=0; i<100; i++){
            md.setA(20+i); System.out.println();
            md.setB(30+i);
            S.o.p(this.getName()+" a = "+md.a+" b = "+md.b);
        } }
}

public class Concurrency {
    public static void main(String args[]) {
        try {
            MaDonnee md = new MaDonnee();
            MonThread1 mt1 = new MonThread1(md);
            MonThread2 mt2 = new MonThread2(md);
            mt1.start() ; mt2.start();
            mt1.join() ; mt2.join();
        } catch(Exception ex) { System.out.println(ex); }
    }
}
```



force à passer la main

Cohérence données

- partage

 - ▶ mémoire

 - ▶ ressources

- ordonnancement

non maîtrisé

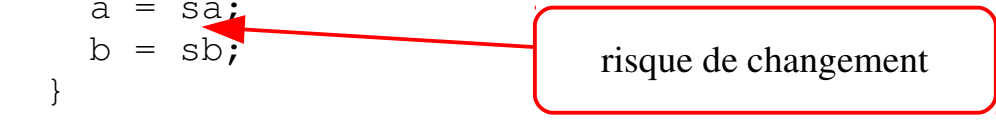
- zones critiques :
cohérence $b = a + 10$?

```
class MaDonnee {
    public int a, b;
    public void set(int sa, int sb) {
        a = sa;
        b = sb;
    }
}

class MonThread1 extends Thread {
    MaDonnee md;
    public MonThread1(MaDonnee ref) { md = ref; }
    public void run() {
        for (int i=0; i<10; i++){
            md.set(i, 10+i);
            S.o.p(this.getName()+" a= "+md.a+" b= "+md.b);
        }
    }
}

class MonThread2 extends Thread {
    MaDonnee md;
    public MonThread2(MaDonnee ref) { md = ref; }
    public void run() {
        for (int i=0; i<10; i++) {
            md.set(20+i, 30+i);
            S.o.p(this.getName()+" a= "+md.a+" b= "+md.b);
        }
    }
}

public class Concurrency {
    public static void main(String args[]) {
        MaDonnee md = new MaDonnee();
        MonThread1 mt1 = new MonThread1(md);
        MonThread2 mt2 = new MonThread2(md);
        mt1.start();      mt2.start();
        mt1.join();       mt2.join();
    }
}
```



Problématique et solutions

- exécution correcte quel que soit le nombre de threads (*thread-safe*)

- ◆ éviter les données partagées

- références (objets)

- données statiques

- ◆ sans risque : données non partagées

- données de l'objet (non-statiques)

- données de la pile

- paramètres et variables locales

- attention au débordement

- protéger les accès

- ◆ opérations atomiques

- ◆ section critique

- ◆ verrou

Atomicité

- opérations atomiques
 - ◆ lectures / écritures de références et variables primitives, sauf long et double
 - ◆ lectures / écritures de *toutes* les variables décrites comme volatile
 - ◆ volatile long a
 - ◆ multi-processeurs
- opérations non-atomiques : les autres
 - ◆ opérations
 - ◆ conditions
 - ◆ boucles
 - ◆ ...

Synchronized

- Java
 - ▶ 1 verrou par objet = moniteur / lock
 - ▶ pas lié à la classe Thread mais à l'instance
- *synchronized* : accès protégé
- gestion de la synchronisation compétitive (l'exclusion mutuelle)
- gestion de la synchronisation coopérative (barrière de synchronisation)
- pas partie du prototype de méthode : pas hérité

Synchronized

- un objet

```
synchronized(monObject) { ... }
```

- une ou plusieurs méthodes d'un objet

```
synchronized TypeRetour uneMeth1 (...) { ... }
```

```
synchronized void uneMeth2 (...) { ... }
```

- une ou plusieurs méthodes de classe

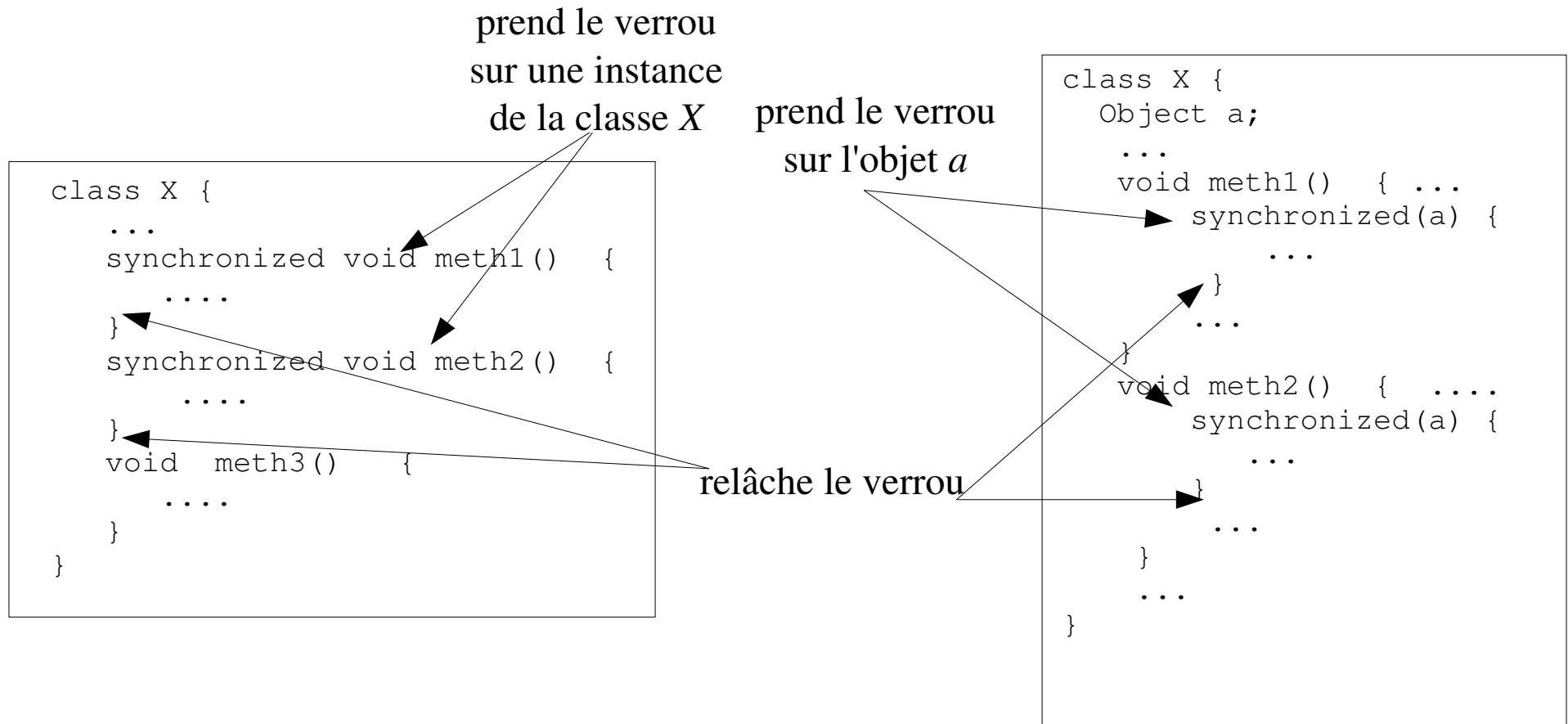
```
class MaClasse {
```

```
    static synchronized TypeRetour uneMeth1 (...) { ... }
```

```
    static synchronized void uneMeth2 (...) { ... }
```

```
}
```

Méthodes / blocs de code synchronisés



Concurrence d'accès

- protection de la méthode *set* par verrou
- résultat cohérent

```
$ java Concurrency
Thread-0 a= 20 b= 30
Thread-1 a= 20 b= 30
Thread-0 a= 1 b= 11
Thread-1 a= 21 b= 31
.....
```

```
class MaDonnee {
    public int a, b;
    public synchronized void set(int sa, int sb){
        a=sa;
        b=sb;
    }
}

class MonThread1 extends Thread {
    MaDonnee md;
    public MonThread1(MaDonnee ref){ md = ref;}
    public void run(){
        for (int i=0; i<10; i++){
            md.set(i, 10+i);
            S.o.p(this.getName()+" a= "+md.a+" b= "+md.b);
        }
    }
}

class MonThread2 extends Thread {
    MaDonnee md;
    public MonThread2(MaDonnee ref) { md = ref; }
    public void run(){
        for (int i=0; i<10; i++){
            md.set(20+i, 30+i);
            S.o.p(this.getName()+" a= "+md.a+" b= "+md.b);
        }
    }
}

public class Concurrency {
    public static void main(String args[]){
        MaDonnee md = new MaDonnee();
        MonThread1 mt1 = new MonThread1(md);
        MonThread2 mt2 = new MonThread2(md);
        mt1.start();      mt2.start();
        mt1.join();       mt2.join();
    }
}
```

Concurrence d'accès

- protection de la méthode *set* par verrou
- résultat cohérent
?????
- *System.out.println* non-atomique

```
$ java Concurrency
Thread-0 a= 20 b= 30
Thread-1 a= 20 b= 30
Thread-0 a= 1 b= 11
Thread-1 a= 21 b= 12
.....
```

```
class MaDonnee {
    public int a, b;
    public synchronized void set(int sa, int sb){
        a=sa;
        b=sb;
    }
}

class MonThread1 extends Thread {
    MaDonnee md;
    public MonThread1(MaDonnee ref) { md = ref; }
    public void run() {
        for (int i=0; i<10; i++){
            md.set(i, 10+i);
            S.o.p(this.getName()+" a= "+md.a+" b= "+md.b);
        }
    }
}

class MonThread2 extends Thread {
    MaDonnee md;
    public MonThread2(MaDonnee ref) { md = ref; }
    public void run(){
        for (int i=0; i<10; i++){
            md.set(20+i, 30+i);
            S.o.p(this.getName()+" a= "+md.a+" b= "+md.b);
        }
    }
}

public class Concurrency {
    public static void main(String args[]){
        MaDonnee md = new MaDonnee();
        MonThread1 mt1 = new MonThread1(md);
        MonThread2 mt2 = new MonThread2(md);
        mt1.start();      mt2.start();
        mt1.join();       mt2.join();
    }
}
```

Concurrence d'accès

- *System.out.println*
non-atomique
- opération d'affichage
protégée par verrou

```
$ java Concurrency
```

```
Thread-0 a = 20 b = 30
```

```
Thread-1 a = 20 b = 30
```

```
Thread-0 a = 1 b = 11
```

```
Thread-1 a = 21 b = 31
```

```
.....
```

```
class MaDonnee {
    public int a, b;
    public synchronized void set(int sa, int sb){
        a=sa; b = sb; }
    public synchronized void affich() {
        S.o.p(Thread.currentThread().getName()+
            " a = "+a+" b = "+b);
    }
}

class MonThread1 extends Thread {
    MaDonnee md;
    public MonThread1(MaDonnee ref) { md = ref; }
    public void run(){
        for (int i=0; i<10; i++) {
            md.set(i, 10+i); md.affich();
        }
    }
}

class MonThread2 extends Thread {
    MaDonnee md;
    public MonThread2(MaDonnee ref) { md = ref; }
    public void run(){
        for (int i=0; i<10; i++) {
            md.set(20+i, 30+i); md.affich();
        }
    }
}

public class Concurrency {
    public static void main(String args[]){
        MaDonnee md = new MaDonnee();
        MonThread1 mt1 = new MonThread1(md);
        MonThread2 mt2 = new MonThread2(md);
        mt1.start();      mt2.start();
        mt1.join();       mt2.join();
    }
}
```

Concurrence d'accès

- résultat cohérent

?????

- **set + affich** non-atomique

```
$ java Concurrency
```

```
Thread-0 a = 20 b = 30
```

```
Thread-1 a = 1 b = 11
```

```
Thread-0 a = 21 b = 31
```

```
Thread-1 a = 2 b = 12
```

```
.....
```

```
class MaDonnee {
    public int a, b;
    public synchronized void set(int sa, int sb){
        a=sa; b=sb;
    }
    public synchronized void affich() {
        S.o.p(Thread....getName()+" a = "+a+" b = "+b);
    }
}
class MonThread1 extends Thread {
    MaDonnee md;
    public MonThread1(MaDonnee ref) { md = ref; }
    public void run(){
        for (int i=0; i<10; i++){
            md.set(i, 10+i);
            md.affich();
        }
    }
}
class MonThread2 extends Thread {
    MaDonnee md;
    public MonThread2(MaDonnee ref) { md = ref; }
    public void run(){
        for (int i=0; i<10; i++){
            md.set(20+i, 30+i);
            md.affich();
        }
    }
}
public class Concurrency {
    public static void main(String args[]){
        MaDonnee md = new MaDonnee();
        MonThread1 mt1 = new MonThread1(md);
        MonThread2 mt2 = new MonThread2(md);
        mt1.start();      mt2.start();
        mt1.join();       mt2.join();
    }
}
```

Concurrence d'accès

- **set + affich** non-atomique
- protection par verrou
- pris deux fois :
moniteurs Java sont réentrants

```
$ java Concurrency
```

```
Thread-0 a= 0 b= 10
```

```
Thread-1 a= 20 b= 30
```

```
Thread-0 a= 1 b= 11
```

```
Thread-1 a= 21 b= 31
```

```
.....
```

```
class MaDonnee {
    public int a, b;
    public synchronized void set(int sa, int sb){
        a=sa; b=sb;
    }
    public synchronized void affich(){
        S.o.p(Thread.currentThread().getName()+
            " a = "+a+" b = "+b);
    }
}

class MonThread1 extends Thread {
    MaDonnee md;
    public MonThread1(MaDonnee ref) { md = ref; }
    public void run(){
        for (int i=0; i<10; i++){
            synchronized(md) {
                md.set(i, 10+i); md.affich();
            }
        }
    }
}

class MonThread2 extends Thread {
    MaDonnee md;
    public MonThread2(MaDonnee ref) { md = ref; }
    public void run(){
        for (int i=0; i<10; i++){
            synchronized(md) {
                md.set(20+i, 30+i); md.affich();
            }
        }
    }
}

public class Concurrency {
    public static void main(String args[]){
        MaDonnee md = new MaDonnee();
        MonThread1 mt1 = new MonThread1(md);
        MonThread2 mt2 = new MonThread2(md);
        mt1.start();      mt2.start();
        mt1.join();       mt2.join();
    }
}
```


Gérer la synchronisation coopérative entre threads (1)

- mécanisme de moniteur (technique de mettre un thread en attente et de le réveiller) – méthodes d'instance *Object*
- attente explicite : *wait([long mili])*
 - ▶ relâche le verrou, il faut donc le posséder (*IllegalMonitorStateException*)
 - ▶ bloque le thread appelant
- réveille : *notify()* / *notifyAll()*
 - ▶ thread débloqué doit reprendre le verrou (si *synchronized*) dès qu'il le pourra
 - ▶ lié à l'objet

Synchronisation coopérative entre threads – wait/notify (1)

```
class Counter{
    private int val;
    public synchronized void setCounter(int val){
        this.val = val;
        notify();
    }
    public synchronized int getCounter(){
        try{
            wait();
        }catch (InterruptedException e){
            e.printStackTrace();
        }
        return val;
    }
}

class WaitClass extends Thread{
    Counter c;
    public WaitClass(Counter c){this.c=c;}
    public void run(){
        S.o.p(getName()+ " Attente du compteur");
        S.o.p(getName()+ " Le compteur attendu "
              + c.getCounter());
    }
}
```

```
class NotifyClass extends Thread {
    Counter c;
    public NotifyClass(Counter c){this.c=c;}
    public void run(){
        try{
            sleep(1000);
        }catch (InterruptedException e){
            e.printStackTrace();
        }
        int alea = (int) (Math.random()*10);
        S.o.p(getName()+
              " Valeur générée " + alea);
        c.setCounter(alea);
    }
}

public class CoordThreadsM{
    public static void main(String[] args){
        Counter c = new Counter();
        new WaitClass(c).start();
        new NotifyClass(c).start();
    }
}
```

Thread-0 Attente du compteur

Thread-1 Valeur générée 0

Thread-0 Le compteur attendu 0 Master Informatique

Synchronisation coopérative entre threads – wait/notify (2)

```
class Counter{
    private int val;
    public void setCounter(int val){
        this.val = val;
    }
    public int getCounter(){ return val;}
}

class WaitClass extends Thread {
    Counter c;
    public WaitClass(Counter c){this.c=c;}
    public void run(){
        synchronized(c){
            try{
                c.wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
        S.o.p(getName()+" Le compteur attendu "+
            c.getCounter());
    }
}
```

```
class NotifyClass extends Thread {
    Counter c;
    public NotifyClass(Counter c){this.c=c;}
    public void run(){
        try{
            sleep(1000);
        }catch(InterruptedException e){
            e.printStackTrace();
        }
        synchronized(c){
            int val = (int) (Math.random()*10);
            S.o.p(getName()+
                " Valeur générée "+val);
            c.setCounter(val);
            c.notify();
        }
    }
}

public class CoordThreads {
    public static void main(String[] args){
        Counter c = new Counter();
        new WaitClass(c).start();
        new NotifyClass(c).start();
    }
}
```

Gérer la synchronisation coopérative entre threads (2)

- test (classe *Thread*) : *static boolean holdsLock(Object o)*
- *yield, sleep* : ne relâchent pas le verrou
- synchronisation de terminaison

▶ *join ([long millis])*

```
class Thread {  
    public final synchronized void join  
        (long millis)  
        throws InterruptedException {  
        ...  
        if (millis == 0){  
            while (isAlive())  
                wait(0);  
        }  
        ...  
    }  
}
```

Sémaphore binaire en Java

```
class Semaphore {
    private int valeur;
    public Semaphore(){
        valeur = 1;
    }
    public synchronized void P(){
        while (valeur == 0)
            try{
                wait();
            }catch(InterruptedException e){}
        valeur = 0;
    }

    public synchronized void V(){
        valeur = 1;
        notify();
    }
}
```

La synchronisation

- *wait*

- ♦ ne relâche que le verrou courant
- ♦ attention interférence *join* (basé sur *wait/notify*)

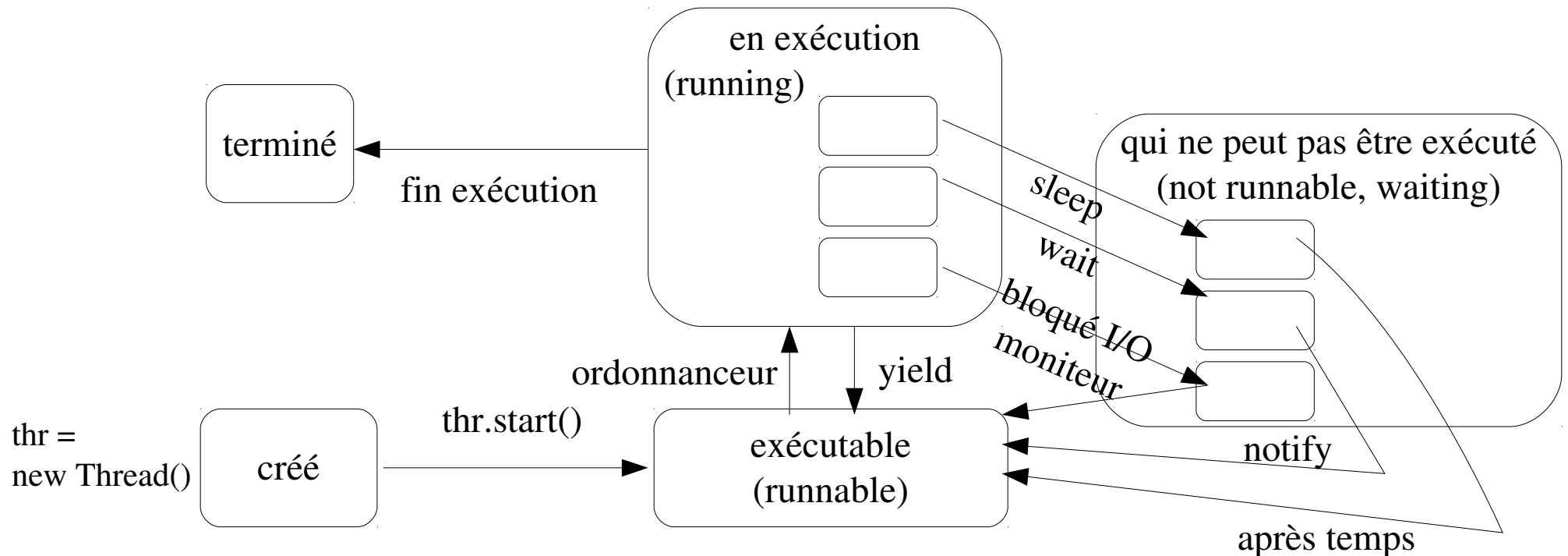
- risques

- ♦ deadlock : interblocage
- ♦ famine (starvation) : coalition qui empêche l'accès
- ♦ livelock : chacun cède à l'autre indéfiniment

- pour aller plus loin

- ♦ paquetage *java.util.concurrent*

Etat des threads



Thread

Thread.State getState()

static class Thread.State (enum)

NEW
 RUNNABLE
 BLOCKED
 WAITING
 TIMED_WAITING
 TERMINATED

Lecteurs/rédacteurs en Java

- sans sémaphores
- outils de base
- gestion coopérative avec
 - ▶ booléens : redaction / lecture
 - ▶ compteur : nbLecteurs
 - ▶ wait / notify
- pas d'ordre FIFO (les réveils – un lecteur ou un rédacteur)
- priorité ?

Classe *Texte* – partie lecteur

```
public synchronized void debutLecture() {
    while (redaction == true) {
        try{    wait();
        } catch (InterruptedException e) {}
    }
    nbLecteurs++;
    if (nbLecteurs == 1) lecture = true;
}
public synchronized void finLecture() {
    nbLecteurs--;
    if (nbLecteurs == 0) {
        lecture = false;
        // réveille les rédacteurs en attente
        notifyAll();
    }
}
```

Classe *Texte* – partie rédacteur

```
public synchronized void debutRedaction(){
    while (lecture == true || redaction == true){
        try{ wait();
        } catch(InterruptedException e) {}
    }
    redaction = true;
}
public synchronized void finRedaction(){
    redaction = false;
    // réveille tous les lecteurs ou rédacteurs
    notifyAll();
}
```