

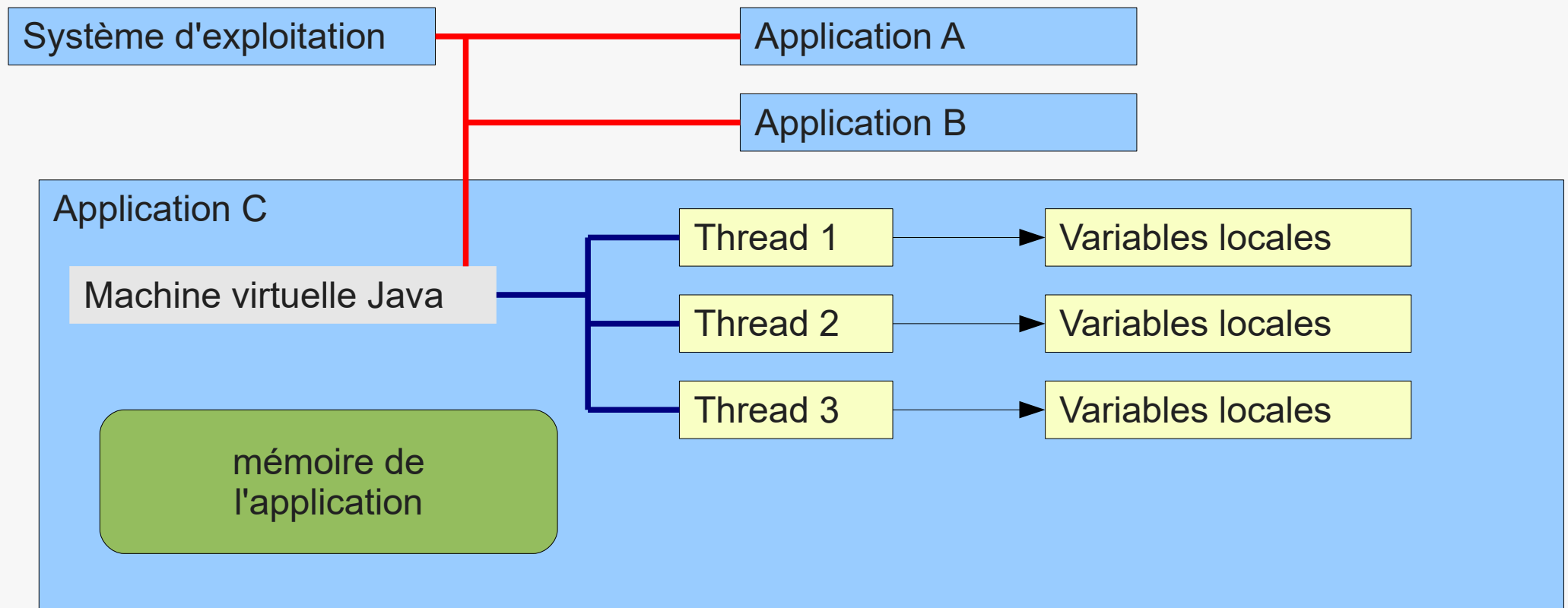
Programmation concurrente

Les threads

- Java possède le concept de thread
 - la JVM s'appuie sur les threads de l'OS sous-jacent
 - ou les simule si OS monotâche
- Permet de créer des applications dont certaines parties s'exécutent en parallèles
- La JVM possède des threads
 - le thread principal => méthode main
 - des threads secondaires pour la gestion de la JVM

Multitâche et threads

- Un thread est un flux de contrôle à l'intérieur d'une application



Les threads

- Chaque thread exécute son code indépendamment des autres threads
 - les threads peuvent coopérer entre eux
- Les threads donnent l'illusion d'une exécution simultanée de plusieurs tâches
- Accès aux données par un thread
 - les variables locales des méthodes d'un thread sont différentes pour chaque thread
 - si deux threads exécutent la même méthode, chacun obtient un espace mémoire séparé pour les variables locales de la méthode
 - les objets et variables d'instance peuvent être partagés entre les threads d'un même programme Java.
 - les variables statiques sont automatiquement partagées

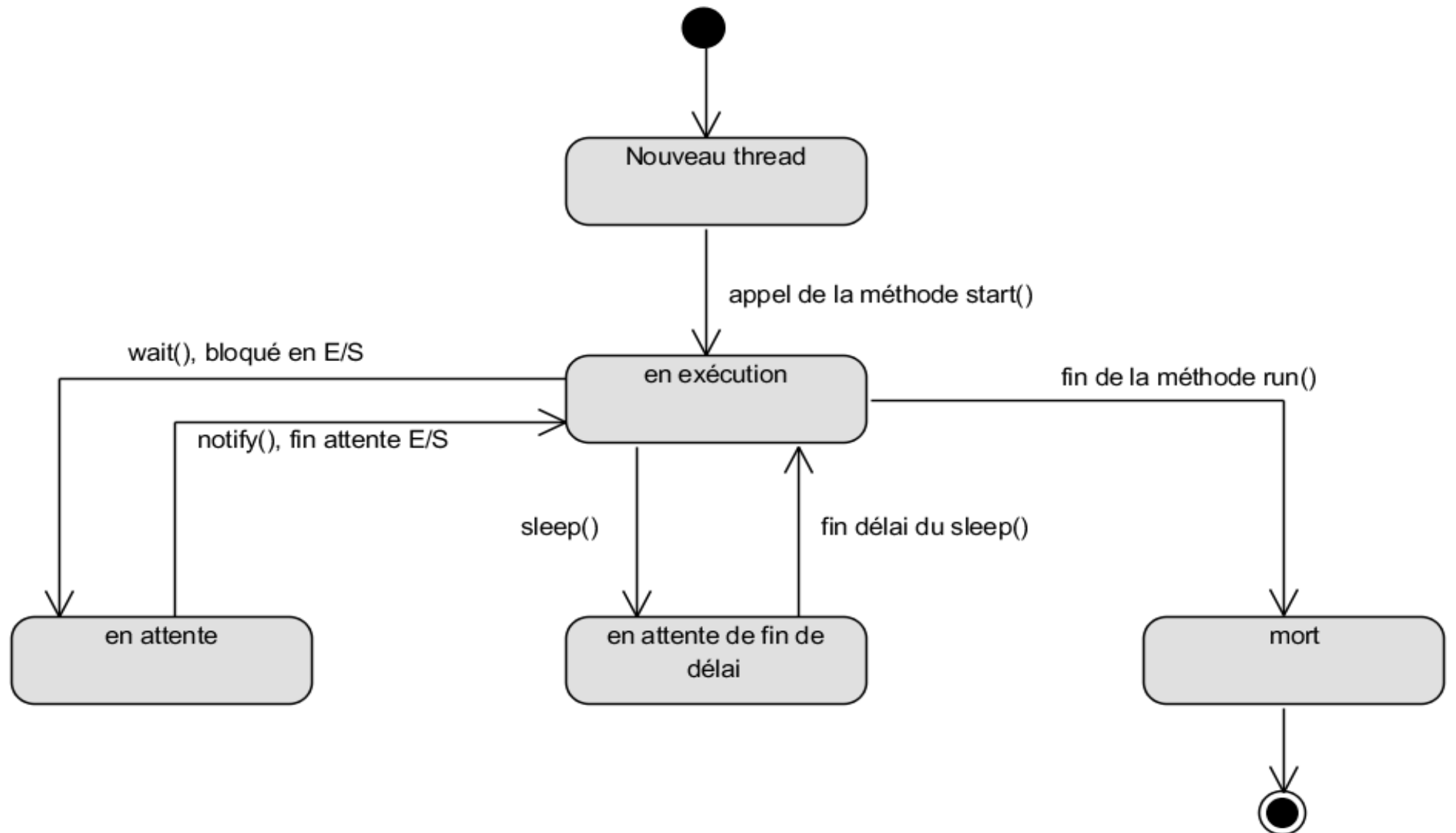
Les threads

- Un thread possède
 - un nom
 - un identifiant
 - une priorité
 - `Thread.MIN_PRIORITY` => 1
 - `Thread.NORM_PRIORITY` => 5
 - `Thread.MAX_PRIORITY` => 10
 - le status daemon ou non
 - un groupe
 - un code à exécuter
 - méthode `run()`
 - un état

Les threads

- La JVM arrête son exécution lorsque :
 - la méthode `System.exit()` est invoquée
- ou
- tous les threads sont morts
 - si le thread n'est pas un daemon
 - les daemons n'empêchent pas la JVM de s'arrêter

Thread – cycle de vie



Les thread

- Création d'un thread
 - par spécialisation de la classe `Thread`
 - redéfinition de la méthode `run()`
 - lancement par la méthode `start()` sur l'instance
 - par implémentation de l'interface `Runnable`
 - codage de la méthode `run()`
 - passer l'instance de la classe à une instance de `Thread`
 - méthode `start()` sur l'instance de `Thread`

Spécialisation de la classe Thread

```
public class Task extends Thread {  
    private int delai;  
  
    public Task(int delai, String nom) {  
        this.delai = delai;  
        setName(nom);  
        this.setName(nom);  
    }  
  
    @Override  
    public void run() {  
        System.out.format(">>> DEBUT TACHE %s de %d s.\n", getName(), delai);  
        try {  
            this.sleep(delai * 1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.format(">>> FIN TACHE %s de %d s.\n", getName(), delai);  
    }  
}
```

Implémentation de Runnable

```
class MyRunnable implements Runnable{
    @Override
    public void run() {
        try {
            System.out.println("DEBUT THREAD");
            Thread.sleep(1_000);
            System.out.println("FIN THREAD");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class MainRunnable {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

Arrêt d'un thread

- Une partie des méthodes de la classe `Thread` sont dépréciées
 - `stop()`, `resume()`, `suspend()`, `destroy()`
- Le thread s'arrête à la fin de sa méthode `run()`
- La méthode `interrupt()` positionne un flag demandant l'interruption
 - si le thread n'est pas en attente
 - utiliser alors les méthodes
 - `interrupted()` : renvoie le flag et le remet à `false`
 - `isInterrupted()` : renvoie le flag sans le remettre à `false`

Arrêt d'un thread

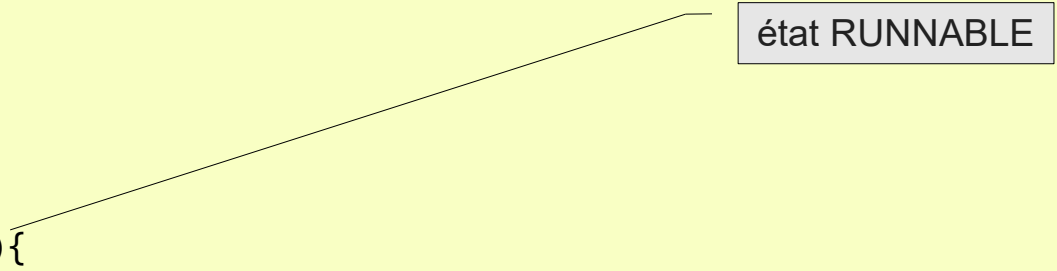
- Si l'état du thread est `RUNNABLE`
 - `interrupted()` évalue le flag d'interruption et on peut stopper la méthode `run()`
- Si l'état du thread est en attente
 - suite à `sleep()`, `join()` ou `wait()`
 - il reçoit une `InterruptedException`
 - suite à une attente sur une E/S
 - il reçoit un `ClosedByInterruptedException`
 - on peut alors stopper le thread dans le traitement de l'exception

Arrêt d'un thread

```
public class ThreadInterrupt implements Runnable {
    private int id;

    public ThreadInterrupt(int id){
        this.id = id;
    }

    @Override
    public void run() {
        int i = 0;
        while(!Thread.interrupted()){
            System.out.printf("thread id : %d - valeur de la boucle %d\n",id,i++ );
            //Traitement long
            for(long k=1 ; k<1000000 ; k++)
                for(long r=1; r<1000 ;r++)
                    ;
        }
        System.out.printf(">>> FIN EXECUTION thread %d - status : %b\n",id,Thread.currentThread().isInterrupted());
    }
}
```



état RUNNABLE

Arrêt d'un thread

```
public class ThreadInterrupt implements Runnable {
    private int id;

    public ThreadInterrupt(int id){
        this.id = id;
    }

    @Override
    public void run() {
        int i = 0;
        while(!Thread.interrupted()){
            System.out.printf("thread id : %d - valeur de la boucle %d\n",id,i++ );
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.printf(">>> CATCH thread %d - status : %b\n",id,Thread.currentThread().isInterrupted());
                Thread.currentThread().interrupt();
            }
        }
        System.out.printf(">>> FIN EXECUTION thread %d - status : %b\n",id,Thread.currentThread().isInterrupted());
    }
}
```

état TIMED_WAITING

Arrêt d'un thread

```
public class ThreadInterruptMain {  
  
    public static void main(String[] args) {  
        Thread t1 = new Thread(new ThreadInterrupt(1));  
        Thread t2 = new Thread(new ThreadInterrupt(2));  
  
        t1.start();  
        t2.start();  
  
        Scanner in = new Scanner(System.in);  
        System.out.println("Entrez le numéro du thread à arrêter : ");  
        int num = in.nextInt();  
        switch(num){  
            case 1:  
                System.out.printf("Etat du thread t1 : %s\n",t1.getState());  
                t1.interrupt();  
                break;  
            case 2:  
                System.out.printf("Etat du thread t2 : %s\n",t2.getState());  
                t2.interrupt();  
                break;  
        }  
    }  
}
```

Arrêt d'un thread

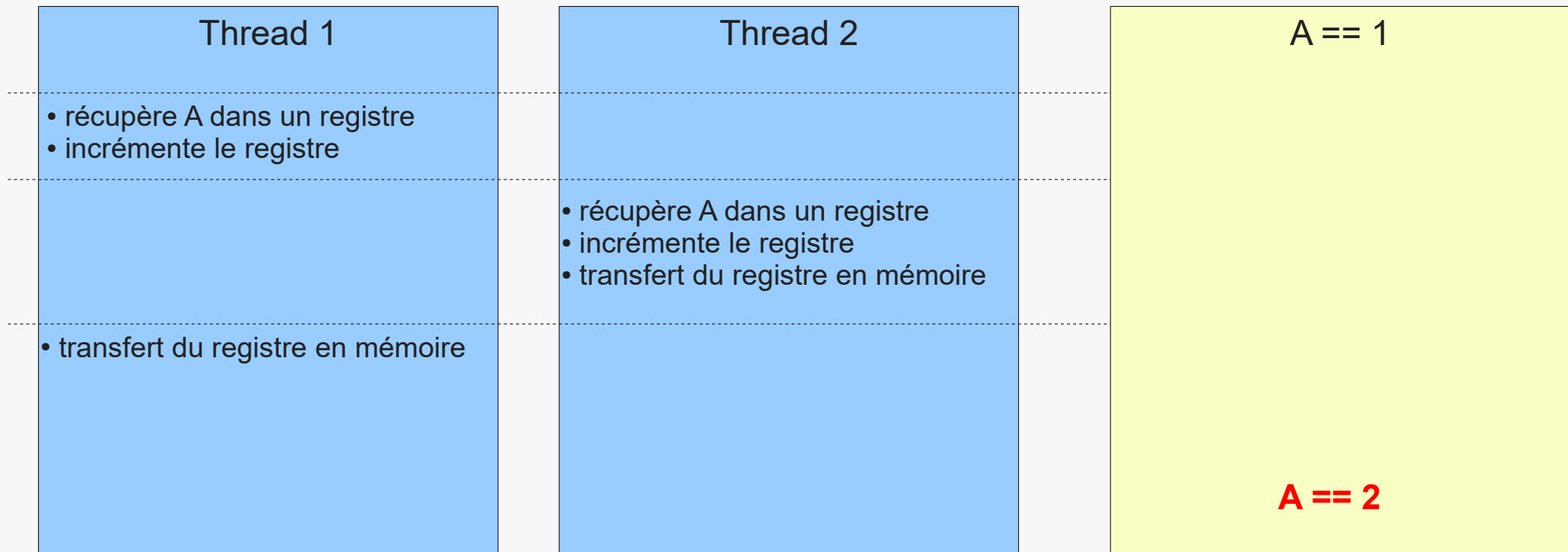
- Attente de l'arrêt
 - méthode `join()`
 - attend la terminaison d'un thread
 - appel bloquant tant que le thread est en vie
 - retourne immédiatement si le thread n'est pas en vie
 - pas démarrer, arrêter
- Redémarrage d'un thread
 - un thread mort ne peut pas être redémarré
 - une instance de `Thread` ne peut-être utilisée qu'une seule fois
 - une exception `IllegalThreadStateException` est levée

Concurrence

- Dans un environnement monothread
 - si il y a écriture d'une valeur dans une variable
 - la lecture de la variable renverra la valeur écrite
 - si celle-ci n'a pas été modifiée entre temps
- Dans un environnement multithread
 - les lectures et écritures peuvent avoir été exécutées dans des threads différents il n'y a pas de garantie qu'un thread lise la valeur écrite par un autre thread

Concurrence

- Deux threads incrémentent un variable
 - au départ la variable vaut 1
 - la valeur attendue après les deux incrémentation est 3



Concurrence

- L'accès et l'affectation est garantie en Java sur tous les types sauf long et double
- Il n'y a pas de garantie qu'un thread ne prenne pas le contrôle entre 2 opérations atomiques
- Pour que le code soit atomique, il faut que l'objet soit utilisé comme moniteur
 - technique pour synchroniser plusieurs tâches qui utilisent des ressources partagées

Concurrence

```
class Foo{
    private int a;
    private int b = 2;

    public Foo(int val){
        a=val;
        b=2*a;
    }

    public void changer(){
        a++;
        b=2*a;
        System.out.printf("%s - a == %d, b ==%d\n",
                           Thread.currentThread().getName(),a,b);
    }
}
```

code non atomique

```
class ThreadFoo extends Thread{
    private Foo foo;

    public ThreadFoo(Foo foo){
        this.foo = foo;
    }

    public void run(){
        for(int i=0 ; i<10 ; i++)
            foo.changer();
    }
}
```

Concurrency

```
public class SynchronisationMain {  
    public static void main(String[] args) {  
        Foo foo = new Foo(10);  
  
        ThreadFoo t0 = new ThreadFoo(foo);  
        ThreadFoo t1 = new ThreadFoo(foo);  
  
        t0.start();  
        t1.start();  
    }  
}
```

```
Thread-1 - a == 12, b == 24  
Thread-0 - a == 12, b == 24  
Thread-1 - a == 13, b == 26  
Thread-1 - a == 15, b == 30
```

Concurrence

- Verrou d'exclusion mutuelle
 - verrou mutex
 - un seul thread peut acquérir un mutex
 - si deux threads essaient d'acquérir un verrou, un seul y parvient
 - l'autre thread doit attendre que le premier thread restitue le verrou pour continuer

Concurrence

- Le mot clé `synchronized` permet d'éviter que plusieurs threads utilisent le même code en même temps
- Tout objet peut jouer le rôle de moniteur
 - chaque objet possède un verrou
 - si un thread prend le verrou aucun autre ne peut utiliser le code `synchronized` marqué par ce verrou
 - différentes approches d'utilisation de `synchronized`

Concurrence

- Réentrance
 - si un thread demande un verrou déjà verrouillé par un autre thread, le thread demandeur est bloqué
 - si un thread tente de prendre un verrou qu'il détient déjà la requête réussit
 - les verrous sont acquis par le thread et non par un appel
 - au verrou est associé un thread propriétaire et un compteur d'acquisition
 - lorsque le compteur passe à zéro le verrou est libéré

Concurrence

- Synchronisation implicite sur `this`
 - `synchronized` est placé en tant que modificateur de méthode
 - efficace car toute la méthode est verrouillée
 - peut poser un problème de performance si les traitements sont longs

```
...  
public synchronized void changer(){  
    a++;  
    b=2*a;  
    System.out.printf("%s - a == %d, b == %d\n",  
        Thread.currentThread().getName(),a,b);  
}  
...
```

```
Thread-0 - a == 11, b == 22  
Thread-0 - a == 12, b == 24  
Thread-1 - a == 13, b == 26  
Thread-1 - a == 14, b == 28  
Thread-1 - a == 15, b == 30
```

Concurrency

- Synchronisation sur bloc de code
 - on ne synchronise qu'une partie du code en le protégeant par `synchronized`

```
...  
public void changer(){  
    synchronized (this) {  
        a++;  
        b=2*a;  
    }  
    System.out.printf("%s - a == %d, b == %d\n"  
        ,Thread.currentThread().getName(),a,b);  
}  
...
```

Concurrence

- Synchronisation sur un objet
 - le moniteur est alors l'objet

```
...  
private Object monitor = new Object();  
...  
public void changer(){  
    synchronized (monitor) {  
        System.out.printf("DEBUT %s - a == %d, b == %d\n",  
            Thread.currentThread().getName(),a,b);  
  
        a++;  
        b=2*a;  
        System.out.printf("FIN %s - a == %d, b == %d\n",  
            Thread.currentThread().getName(),a,b);  
    }  
}  
...
```

le moniteur est une instance d'Object

Concurrence

- Synchronisation de la classe
 - permet de protéger les variables statiques d'une classe

```
class Foo1 {  
    private static int a = 0;  
  
    static void setA(int a) {  
        Foo1.a = a;  
    }  
  
    static int getA() {  
        return a;  
    }  
}  
  
class Foo2 {  
    void methodFoo2(int a) throws ClassNotFoundException {  
        synchronized (Class.forName("org.antislashn.formation.Foo1")) {  
            Foo1.setA(a);  
            System.out.printf("%s - valeur de a : %d\n", Thread.currentThread().getName(), Foo1.getA());  
        }  
    }  
}
```

Synchronisation entre threads

- Pour éviter l'interblocage de threads on utilise les méthodes `wait()`, `notify()`, `notifyAll()`
 - si `wait()` est invoqué à l'intérieur d'une méthode synchronisée
 - le thread actuel est bloqué et mis en liste d'attente
 - le moniteur est déverrouillé
 - le thread suspendu sera réveillé par une notification sur le moniteur
 - `notify()` pour libérer un thread de la liste
 - `notifyAll()` pour libérer tous les threads de la liste

Concurrence

- `Thread` : méthodes d'attente et de notification
 - `wait()`
 - attend l'arrivée d'une condition
 - le thread en cours est mis en attente
 - le verrou est libéré, d'autres threads peuvent alors exécuter des méthodes synchronisées avec le même verrou
 - doit être invoquée dans un bloc `synchronized`
 - `notify()`
 - notifie un thread en attente d'une condition
 - doit être invoquée dans un bloc `synchronized`

Concurrence

- Thread : méthodes d'attente et de notification
 - `notifyAll()`
 - notifie tous les threads en attente d'une condition
 - doit être invoquée dans un bloc `synchronized`
 - `wait(int delay)`
 - attend l'arrivée d'une condition avec un timeout
 - le verrou est libéré
 - `sleep(int delay)`
 - provoque une attente
 - le verrou n'est pas libéré
 - cf. aussi `TimeUnit.MILLISECONDS.sleep(int ms)`

Concurrence

- Double verrou

```
public class DoubleLock {  
    private final Object lock1 = new Object();  
    private final Object lock2 = new Object();  
  
    public void foo() throws InterruptedException{  
        synchronized (lock1) {  
            synchronized (lock2) {  
                while(...){  
                    lock2.wait();  
                }  
            }  
        }  
    }  
  
    public void notifier(){  
        lock2.notify();  
    }  
}
```

foo() verrouille lock1 tant que lock2.wait()
n'est pas terminé

Concurrence et collections

- Parcours de collection par itérateur ou boucle type for-each
 - les itérateurs renvoyés les collections synchronisées ne sont pas conçus pour gérer les modifications concurrentes
 - exception `ConcurrentModificationException`
 - solution : verrouiller la collection pendant le parcours
- certains itérateurs sont cachés
 - exemple : appel `toString()` sur une collection
 - itération cachée pour appel de `toString()` sur chaque élément