Programmation Java

threads et synchronisation

P. Bakowski

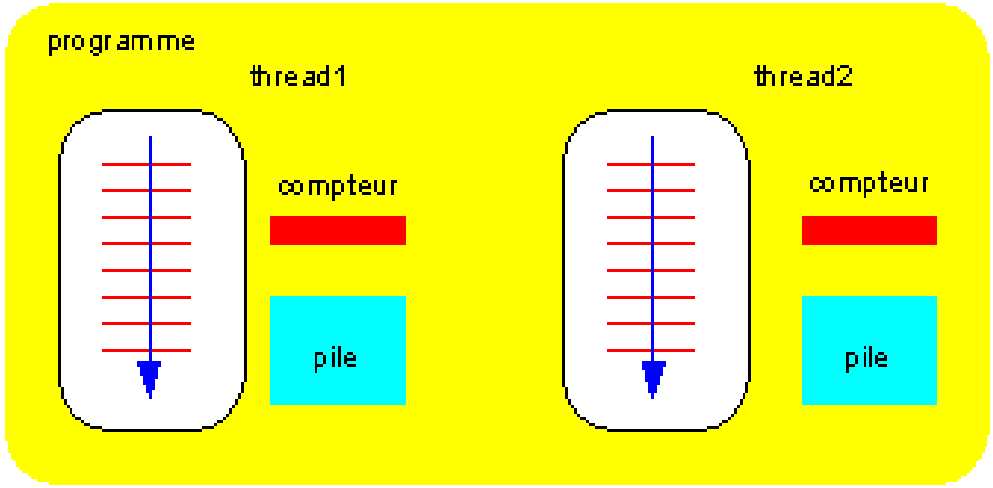
bako@ieee.org

Les *threads*

Les applications en Java peuvent s'exécuter comme un programme simple ou comme un programme à plusieurs fils d'exécutions appelés threads.
Par ailleurs, un programme simple est exécuté comme un seul thread.

Un thread a un début, une séquence d'instructions et une fin.
Un thread ne peut pas s'exécuter sans l'enveloppe du programme; il doit être lancé dans un programme.

Il peut exploiter les ressources d'un programme mais il utilise son propre compteur du programme et sa propre pile



Les *threads*

Pour exécuter un thread il faut lancer la méthode `run()`. Cette méthode contient le programme à exécuter par le *thread*.

La classe **`Thread`** offre le cadre pour la création d'un thread.

La méthode `run()` de cette classe est vide; elle doit être remplacée par notre méthode `run()`.

Il y a deux manières permettant d'implémenter une méthode `run()`:

- par la création d'une sous-classe de **`Thread`** avec notre méthode `run()`
- par l'implémentation de notre méthode `run()` dans l'interface **`Runnable`**

SimpleThread extends Thread

```
public class TwoThreadsTest {
  public static void main (String[] args) {
  SimpleThread un = new SimpleThread("UN");
  SimpleThread deux = new SimpleThread("DEUX");
  un.start();
  deux.start(); }
}
class SimpleThread extends Thread {
  public SimpleThread(String str) { super(str); }
  public void run() {
  for(int i = 0; i < 10; i++) {
    System.out.println(i + " " + getName());
    try { sleep((long)(Math.random() * 1000));
    } catch (InterruptedException e) {}
  }
  System.out.println("FIN! " + getName());
  }
}
```

SimpleRunnable implements Runnable

```
public class TwoRunnableTest {
  public static void main (String[] args) {
  Thread t_un = new Thread(new SimpleRunnable(),«un»);
  Thread t_deux = new Thread(new SimpleRunnable(),«deux»);
  t_un.start();
  t_deux.start(); }
}
class SimpleThread extends Runnable {
  public void run() {
  for(int i = 0; i < 10; i++) {
    System.out.println(i + " " + getName());
    try { Thread.sleep((long)(Math.random() * 1000));
    } catch (InterruptedException e) {}
  }
  System.out.println("FIN! " + getName());
  }
}
```

Cycle de vie d'un *thread*

Un thread se termine normalement lorsque sa méthode run() se termine. On ne peut pas reprendre l'exécution du thread, mais on peut connaître son état grâce à la méthode `isAlive()`.

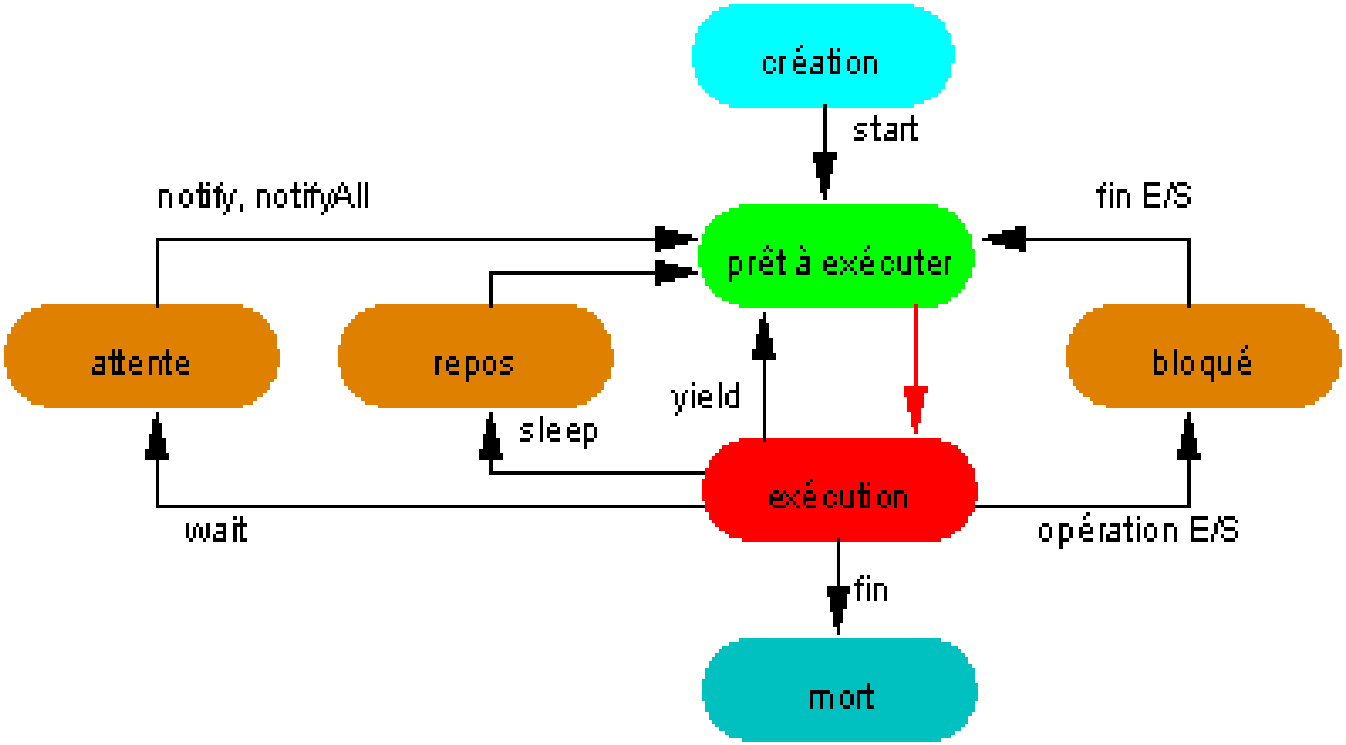
Le thread est effectivement lancé par sa méthode `start()`; à ce moment il passe à l'état "prêt à exécuter". Le thread "prêt à exécuter" de la plus haute priorité entre dans l'état "exécution" lorsque le système assigne un processeur au thread.

Un thread entre dans l'état "bloqué" lorsqu'il effectue une requête d'entrée-sortie. A la sortie de l'opération E/S un thread "bloqué" devient "prêt à exécuter" .

L'appel de la méthode `sleep()` par un thread en état "exécution" provoque son passage à l'état de "repos".

Lorsqu'un thread effectue l'appel `wait()`, il entre en état "attente" pour l'objet donné sur lequel l'appel `wait()` a été effectué. Un thread en état "attente" est débloqué par un appel `notify()` généré par un autre thread associé à cet objet. Chaque thread à l'état "attente" passe à l'état "prêt à exécuter" après la notification par `notifyAll()` perpétrée par un autre thread associé à cet objet.

Cycle de vie d'un *thread*



Chaque thread a une priorité comprise entre `Thread.MIN_PRIORITY` (valeur 1) et `Thread.MAX_PRIORITY` (valeur 10). Au moment de l'activation, un thread reçoit la priorité `Thread.NOR_PRIORITY` égale à 5.

Cycle de vie d'un *thread*

```
public class ThreadStateTest {
public static void main (String[] args) {
ThreadState thread1 = new ThreadState("thread1");
ThreadState thread2 = new ThreadState("thread2");
ThreadState thread3 = new ThreadState("thread3");
ThreadState thread4 = new ThreadState("thread4");
thread3.setPriority(10);thread4.setPriority(10);
System.out.println("Threads crees");
thread1.start();thread2.start();
thread3.start();thread4.start();
System.out.println("Threads en etat pret"); }
}
```

Cycle de vie d'un *thread*

```
class ThreadState extends Thread { // classe interne
private int timeSleep;
  public ThreadState(String name) { super(name);
  timeSleep = (int) (Math.random()*5000);
  System.out.println("Nom:"+getName()+"temps de repos:"+
                      timeSleep);
  }
  public void run() {
  try {
    System.out.println(getName() + " Mise en repos");
    Thread.sleep(timeSleep);
    } catch (InterruptedException e) {
    System.out.println(e.toString()); }
    System.out.println(getName() + " repos fini");
  }
}
```

Cycle de vie d'un *thread*

Résulat d'exécution :

```
Nom:thread1 temps de repos:3551
Nom:thread2 temps de repos:2356
Nom:thread3 temps de repos:1566
Nom:thread4 temps de repos:1466
Threads crees
thread1 Mise en repos
thread2 Mise en repos
thread3 Mise en repos
Threads en etat pret
thread4 Mise en repos
thread4 repos fini
thread3 repos fini
thread2 repos fini
thread1 repos fini
```

Synchronisation

Le partage et la protection de ressources utilisées par plusieurs threads en exécution nécessite un mécanisme de synchronisation.
Java utilise le mécanisme des moniteurs pour assurer la synchronisation.

Chaque objet qui dispose de méthodes `synchronized` est un moniteur.

Le moniteur permet de verrouiller l'exécution d'une méthode.

S'il y a plusieurs méthodes `synchronized`, une seule de ces méthodes est active à la fois sur un objet; tous les autres objets qui tentent d'invoquer des méthodes `synchronized` doivent attendre.

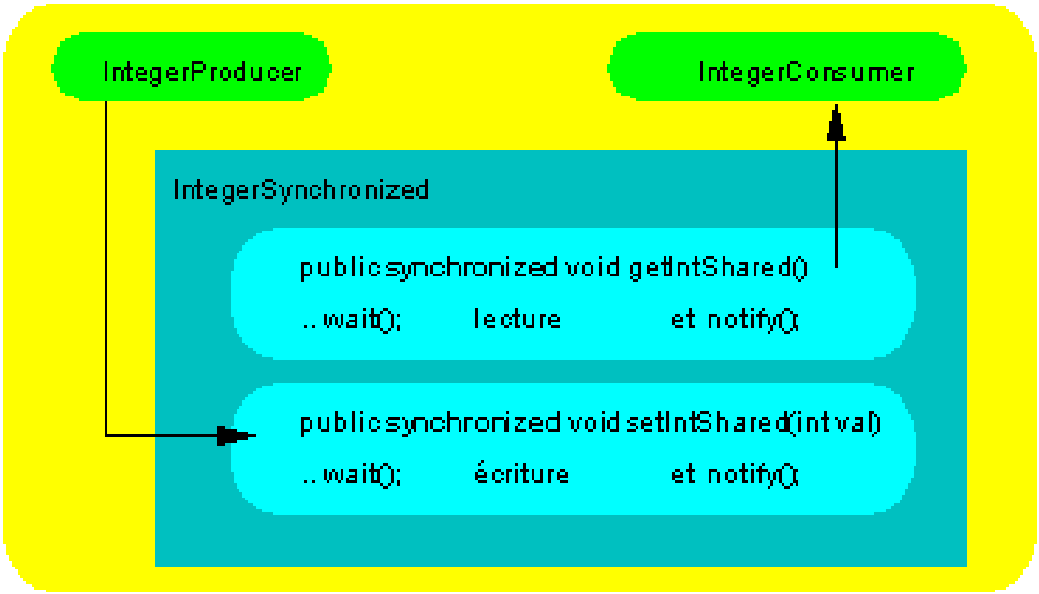
Quand une méthode synchronisée termine son exécution, le verrou sur l'objet est libéré et le moniteur laisse signaler au processus en attente par la méthode `notify()`.
Si un thread appelle `notifyAll()`, alors tous les threads qui attendent que l'objet devienne éligible entrent dans l'état "prêt à exécuter".

Synchronisation

Exemple:
Dans l'exemple suivant, nous utilisons les méthodes de synchronisation dans une application qui lance deux threads:

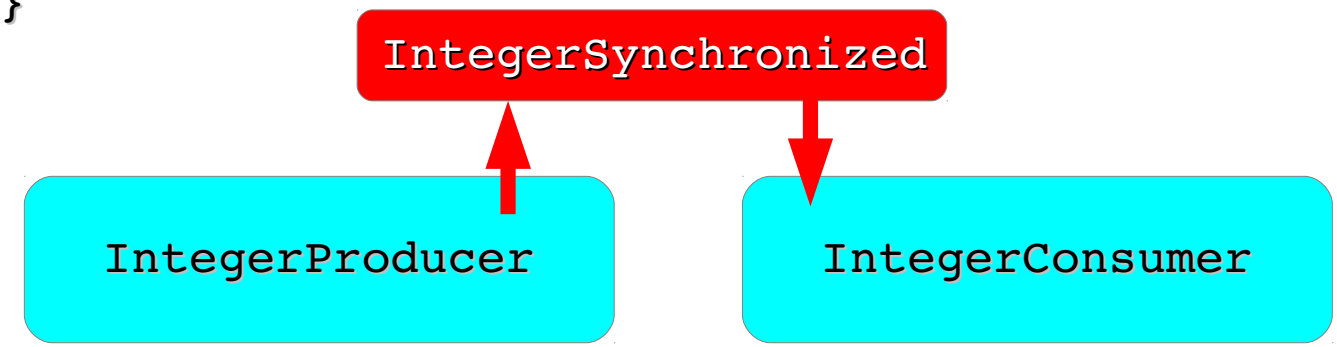
le thread producteur et le thread consommateur.

Les valeurs (entiers) produites par le producteur sont déposées dans une case synchronisée par la méthode `setIntShared(int val)` et récupérées par une autre méthode synchronisée `getIntShared()` lancée par le thread consommateur.



Producer - Cunsumer

```
public class ProducerConsumer {
public static void main (String[] args) {
IntegerSynchronized is = new IntegerSynchronized();
IntegerProducer ip = new IntegerProducer(is);
IntegerConsumer ic = new IntegerConsumer(is);
ip.start(); // activation du thread producteur
ic.start(); // activation du thread consommateur
}
}
```



Producer

```
public class IntegerProducer extends Thread {
private IntegerSynchronized ikeep;
public IntegerProducer(IntegerSynchronized ipar) {
super("IntegerProducer");
ikeep = ipar;
}
public void run() {
for(int counter=1; counter<=10;counter++) {
try { Thread.sleep((int) (Math.random()*3000));
} catch (InterruptedException e)
{ System.out.println(e.toString()); }
ikeep.setIntShared(counter); }
// ecriture dans la variable partagee
System.out.println(getName() + " end of production"); }
}
```

Consumer

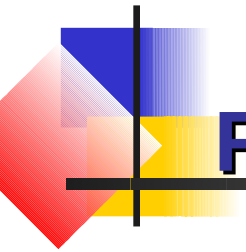
```
public class IntegerConsumer extends Thread {
private IntegerSynchronized ikeep;
public IntegerConsumer(IntegerSynchronized ipar) {
super("IntegerConsumer");
ikeep = ipar;
}
public void run() {
int val, sum = 0;
  do { // boucle de lecture
  try { Thread.sleep((int) (Math.random()*3000));
      } catch (InterruptedException e)
  { System.out.println(e.toString()); }
  val = ikeep.getIntShared();
  // lecture dans la variable partagee
  sum += val; }
  while (val != 10);
System.out.println(getName()+"end of consumption-the sum
is "+sum);}
}
```

Variable partagée (synchronized)

```
public class IntegerSynchronized {
private int intShared = -1;
private boolean inscriptible = true; // droit a ecrire
public synchronized void setIntShared(int val) {
while(!inscriptible) {
try { wait(); }  // attente d'un notify()
catch(InterruptedException e) { e.printStackTrace(); } }
System.out.println(Thread.currentThread().getName() + " put
intShared in " + val);
intShared = val; // ecriture dans la variable partagee
inscriptible = false;
notify();
}
public synchronized int getIntShared() {
while(inscriptible) { // en ecriture ?
try { wait(); }
catch(InterruptedException e) { e.printStackTrace(); } }
inscriptible = true;
notify();
System.out.println(Thread.currentThread().getName() + " get value
from intShared " + intShared);
return intShared; }  // lecture dans la variable partagee
}
```

Variable partagée (synchronized)

```
IntegerProducer put intShared in 1
IntegerConsumer get value from intShared 1
IntegerProducer put intShared in 2
IntegerConsumer get value from intShared 2
IntegerProducer put intShared in 3
IntegerConsumer get value from intShared 3
IntegerProducer put intShared in 4
IntegerConsumer get value from intShared 4
IntegerProducer put intShared in 5
IntegerConsumer get value from intShared 5
IntegerProducer put intShared in 6
IntegerConsumer get value from intShared 6
IntegerProducer put intShared in 7
IntegerConsumer get value from intShared 7
IntegerProducer put intShared in 8
IntegerConsumer get value from intShared 8
IntegerProducer put intShared in 9
IntegerConsumer get value from intShared 9
IntegerProducer put intShared in 10
IntegerProducer end of production
IntegerConsumer get value from intShared 10
IntegerConsumerend of consumption-the sum is 55
```



Résumé

Le mécanisme de Thread

Les états d'un thread (prêt, actif, suspendu,..);

la notion de priorité

Le mécanisme de moniteur (objet synchronisé)