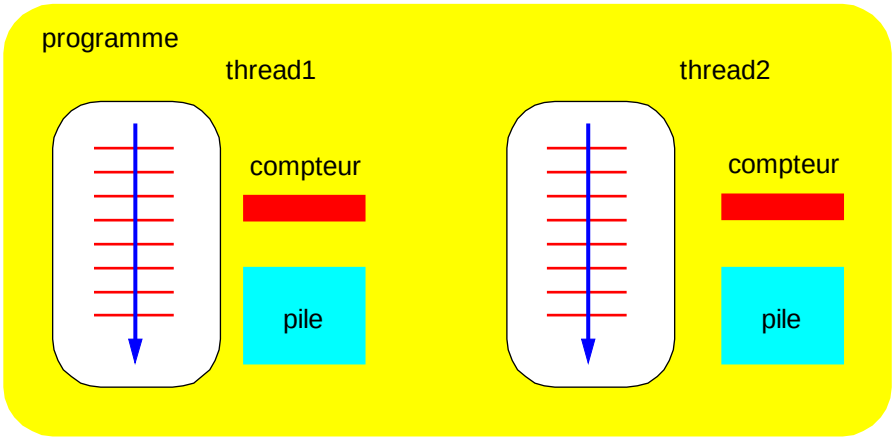
Chapitre 4

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.



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*

Sous-classement de la classe *Thread*

Un objet de la classe *Thread* est un objet de contrôle permettant de lancer et de suivre l'exécution d'un *thread*. Lorsqu'un nouvel objet contrôleur est créé, le *thread* démarre par un appel à la méthode *start()* sur ce contrôleur. La méthode *start()* appelle la méthode *run()* sur l'objet.

Exemple:

```
public class TwoThreadsTest {
   public static void main (String[] args) {
      SimpleThread un= new SimpleThread("UN");          // instanciation d'un thread
      SimpleThread deux = new SimpleThread("DEUX");
      un.start();                                       // activation d'un thread
      deux.start();
   }
}
```

```
class SimpleThread extends Thread {                    // thread - classe interne
   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("DONE! " + getName());
   }
}
```

La solution avec une classe qui implémente l'interface *Runnable* nécessite la création d'un objet (d'objets) *runnable* et ensuite la création du *thread* qui va exécuter l'objet *runnable*. La méthode *start()* lance cette exécution.

```
class SimpleRunnable implements Runnable {  // classe interne qui peut etre utilisee (runnable) par un thread
   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.println(i);
         try {
            Thread.sleep((long)(Math.random() * 1000));
         } catch (InterruptedException e) {}
      }
      System.out.println("Fini! ");
   }
}
public class TwoRunnableTest {
   public static void main (String[] args) throws Exception{
      Thread thread_un = new Thread(new SimpleRunnable(),"un");   // creation d'un thread de l'objet runnable
      Thread thread_deux = new Thread(new SimpleRunnable(),"deux");
      thread_un.start();              // activation du thread
      thread_deux.start();
   }
}
```

Cycle de vie d'un *thread*

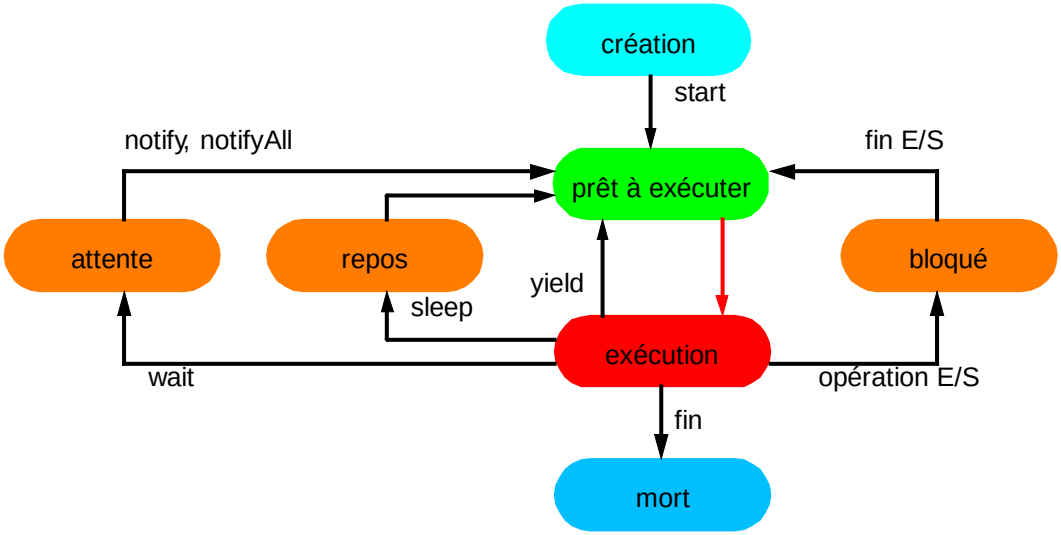
Un *thread* se termine normalement lorsque sa méthode *run()* se termine. Le *thread* est alors effacé de la machine virtuelle; mais son objet contrôleur est toujours accessible. Il ne permet pas de reprendre l'exécution du *thread*, mais il permet de connaître son état grâce à la méthode *isAlive()*.

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 "mort" lorsque sa méthode *run()* s'achève ou se termine par un événement.

Généralement, un *thread* entre dans l'état "bloqué" lorsqu'il effectue une requête d'entrée-sortie. Un *thread* "bloqué" devient "prêt à exécuter" quand l'opération entrée/sortie s'achève.

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

Lorsqu'un *thread* à l'état "exécution" 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.



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.

Selon le type de système d'exploitation supportant les *threads*, la gestion des *threads* peut ou non être effectuée en mode découpage de temps *time-sharing*). Dans le mode *time-sharing*, chaque *thread* reçoit un bref intervalle de temps du processeur appelé **quota**, durant lequel le *thread* a accès au processeur ("exécution"). Au moment de l'expiration du **quota** le processeur est affecté à un autre *thread* de priorité identique.

L'ordonnanceur (*scheduler*) de Java garantit que plusieurs *threads* de priorités identiques s'exécutent de façon équitable dans le temps. Selon leurs priorités, les *threads* sont organisés en plusieurs files d'attente. Tous les *threads* d'une file d'attente (plus) prioritaire sont exécutés avant les *threads* appartenant à une file d'attente d'un niveau inférieur.

Attention: Les *threads* ayant la priorité la plus faible peuvent attendre indéfiniment leur tour d'exécution (phénomène de famine).

La priorité d'un *thread* est modifiable grâce à la méthode *setPriority()*; son argument doit être compris entre 1 et 10. La méthode *getPriority()* donne la priorité actuelle d'un *thread*.

Un *thread* peut arrêter son exécution de lui même (passage à 'état "prêt à exécuter") par l'appel à la méthode *yield()*. Cette méthode donne la chance à un autre *thread* de priorité égale de prendre le contrôle du processeur.

Exemple:

```
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");
  }
}
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");
}
}
```

Le résultat d'une exécution: (l'interprétation?)

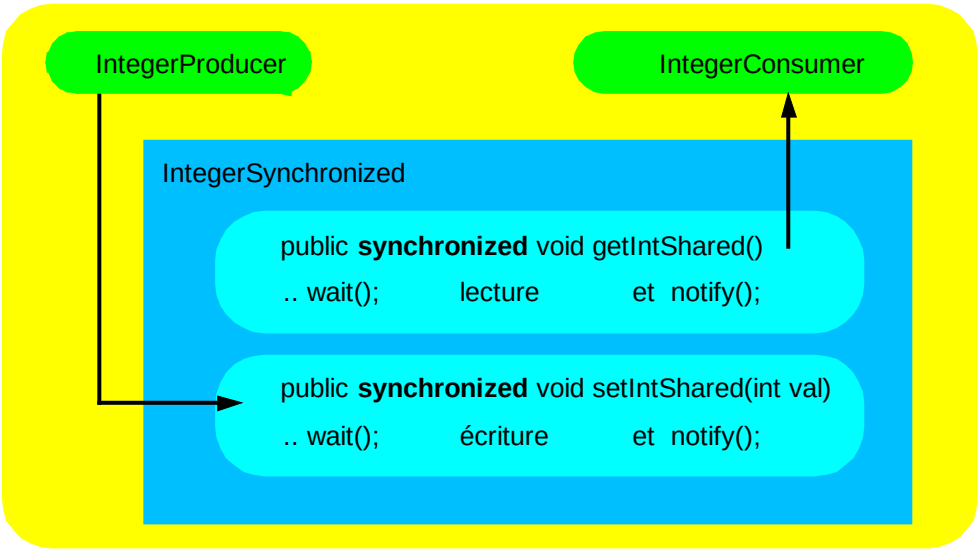
```
Nom: thread1  temps de repos: 4564
Nom: thread2  temps de repos: 322
Nom: thread3  temps de repos: 1346
Nom: thread4  temps de repos: 4013
Threads crees
Threads en etat pret
thread3  Mise en repos
thread4  Mise en repos
thread1  Mise en repos
thread2  Mise en repos
thread2  repos fini
thread3  repos fini
thread4  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 le *thread* "prêt à exécuter" d'invoquer l'exécution d'une méthode *synchronized*.

Un *thread* qui sollicite une méthode *synchronized* peut libérer le processeur volontairement par l'appel *wait()* pour attendre la fin d'exécution d'un autre *thread* en possession de la méthode à ce moment là. Le *thread* qui termine l'exécution une méthode *synchronized* peut le 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". En principe, les *threads* qui ont explicitement invoqué le *wait()* ne peuvent continuer que lorsqu'ils reçoivent une notification *notify()* ou *notifyAll()* envoyé par un autre *thread*.

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.



Le programme de l'exemple:

```
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
  }
}
```

```
public class IntegerProducer extends Thread {          // thread producteur
private IntegerSynchronized ikeep;
public IntegerProducer(IntegerSynchronized ipar) {
super("IntegerProducer");
        ikeep = ipar;
}
public void run() {
        for(int counter=1; counter<=10;counter++) {          // boucle d'ecriture
          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"); }
}
```

```
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);}
}

public class IntegerSynchronized {
        private int intShared = -1;
        private boolean inscriptible = true;                    // initialisation pour l'ecriture
public synchronized void setIntShared(int val) {       // methode synchronisee d'acces a un entier : ecriture
while(!inscriptible) {                                  // attente de validation en ecriture
  try {       wait(); }                                         // attente sur wait pour etre reveillee par 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();                                               // notification de la fin d'execution du thread producteur
}

public synchronized int getIntShared() {               // methode synchronisee d'acces a un entier : lecture
while(inscriptible) {                                   // attente de validation en lecture
try {       wait(); }                                           // attente sur wait pour etre reveillee par un notify()
        catch(InterruptedException e) { e.printStackTrace(); } }
inscriptible = true;
notify();
System.out.println(Thread.currentThread().getName() + " get value from intShared " + intShared);
return intShared; }                                     // lecture da la variable partagee
}
```

Le résultat d'une exécution:

```
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
IntegerConsumer  end of consumption - the sum is 55
```

Résumé

Dans ce chapitre nous avons étudié le mécanisme *Thread* permettant de lancer en exécution plusieurs activités parallèles. Les *threads* évoluent dans le temps et peuvent se trouver dans différents états (*prêt, actif, suspendu,..*); à un instant donné un seul *thread* peut être dans l'état "exécution". La gestion des *threads* est basée sur la notion de priorité. Différentes méthodes de gestion des *threads* permettent de modifier leur priorité, arrêter ou suspendre un *thread*. Afin d'organiser une synchronisation et une communication entre les *threads*, Java introduit le mécanisme de moniteur. Un moniteur permet de synchroniser l'accès aux ressources partagées par plusieurs *threads*.

Le mécanisme des *threads* est également utilisé pour l'implémentation des applets qui seront étudiées dans le chapitre suivant.