Développons en Java v 2.00

8. Le multitâche

Niveau : Supérieur

Un thread est une unité d'exécution faisant partie d'un programme. Cette unité fonctionne de façon autonome et parallèlement à d'autres threads. En fait, sur une machine mono processeur, chaque unité se voit attribuer des intervalles de temps au cours desquels elles ont le droit d'utiliser le processeur pour accomplir leurs traitements.

La gestion de ces unités de temps par le système d'exploitation est appelée scheduling. Il existe deux grands types de scheduler:

- le découpage de temps utilisé par Windows et Macintosh OS jusqu'à la version 9. Ce système attribue un intervalle de temps prédéfini quelque soit le thread et la priorité qu'il peut avoir
- la préemption utilisée par les systèmes de type Unix. Ce système attribut les intervalles de temps en tenant compte de la priorité d'exécution de chaque thread. Les threads possédant une priorité plus élevée s'exécutent avant ceux possédant une priorité plus faible.

Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leur exécution "simultanée".

La classe java.lang.Thread et l'interface java.lang.Runnable sont les bases pour le développement des threads en java. Par exemple, pour exécuter des applets dans un thread, il faut que celles-ci implémentent l'interface Runnable.

Le cycle de vie d'un thread est toujours le même qu'il hérite de la classe Thread ou qu'il implémente l'interface Runnable. L'objet correspondant au thread doit être créé, puis la méthode start() est appelée qui à son tour invoque la méthode run(). La méthode stop() permet d'interrompre le thread.

Avant que le thread ne s'exécute, il doit être démarré par un appel à la méthode start(). On peut créer l'objet qui encapsule le thread dans la méthode start() d'une applet, dans sa méthode init() ou dans le constructeur d'une classe.

Ce chapitre contient plusieurs sections :

8.1. L'interface Runnable

Cette interface doit être implémentée par toute classe qui contiendra des traitements à exécuter dans un thread.

Cette interface ne définit qu'une seule méthode : void run().

Dans les classes qui implémentent cette interface, la méthode run() doit être redéfinie pour contenir le code des traitements qui seront exécutés dans le thread.

Exemple :

```
01. package com.jmdoudoux.test;
```

```
02.
03. public class MonThread3 implements Runnable {
04.
05.   public void run() {
06.     int i = 0;
07.     for (i = 0; i > 10; i++) {
08.       System.out.println("" + i);
09.     }
10.   }
11. }
```

Lors du démarrage du thread, la méthode run() est appelée.

8.2. La classe Thread

La classe Thread est définie dans le package java.lang. Elle implémente l'interface Runnable.

Elle possède plusieurs constructeurs : un constructeur par défaut et plusieurs autres qui peuvent avoir un ou plusieurs des paramètres suivants :

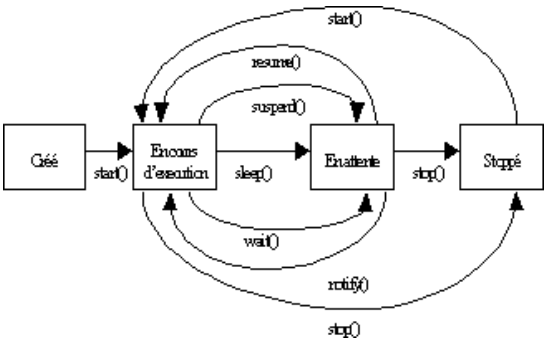
Paramètre	Rôle
un nom	le nom du thread
un objet qui implémente l'interface Runnable	l'objet qui contient les traitements du thread
un groupe	le groupe auquel sera rattaché le thread

Un thread possède une priorité et un nom. Si aucun nom particulier n'est donné dans le constructeur du thread, un nom par défaut composé du suffixe "Thread-" suivi d'un numéro séquentiel incrémenté automatiquement lui est attribué.

La classe Thread possède plusieurs méthodes pour gérer le cycle de vie du thread.

Méthode	Rôle
void destroy()	met fin brutalement au thread : à n'utiliser qu'en dernier recours.
int getPriority()	renvoie la priorité du thread
ThreadGroup getThreadGroup()	renvoie un objet qui encapsule le groupe auquel appartient le thread
boolean isAlive()	renvoie un booléen qui indique si le thread est actif ou non
boolean isInterrupted()	renvoie un booléen qui indique si le thread a été interrompu
void join()	attend la fin de l'exécution du thread
void join(long)	attend, au plus le nombre de millisecondes fourni en paramètre, la fin de l'exécution du thread
void resume()	reprend l'exécution du thread( ) préalablement suspendu par suspend( ). Cette méthode est dépréciée
void run()	méthode déclarée par l'interface Runnable : elle doit contenir le code qui sera exécuté par le thread
void sleep(long)	mettre le thread en attente durant le temps exprimé en millisecondes fourni en paramètre. Cette méthode peut lever une exception de type InterruptedException si le thread est réactivé avant la fin du temps.
void start()	démarrer le thread et exécuter la méthode run()
void stop()	arrêter le thread. Cette méthode est dépréciée
void suspend()	suspend le thread jusqu'au moment où il sera relancé par la méthode resume( ). Cette méthode est dépréciée
void yield()	indique à l'interpréteur que le thread peut être suspendu pour permettre à d'autres threads de s'exécuter.

Le cycle de vie avec le JDK 1.0 est le suivant :



Le comportement de la méthode start() de la classe Thread dépend de la façon dont l'objet est instancié. Si l'objet qui reçoit le message start() est instancié avec un constructeur qui prend en paramètre un objet Runnable, c'est la méthode run() de cet objet qui est appelée. Si l'objet qui reçoit le message start() est instancié avec un constructeur qui ne prend pas en paramètre une référence sur un objet Runnable, c'est la méthode run() de l'objet qui reçoit le message start() qui est appelée.

A partir du J.D.K. 1.2, les méthodes stop(), suspend() et resume() sont dépréciées. Le plus simple et le plus efficace est de définir un attribut booléen dans la classe du thread initialisé à true. Il faut définir une méthode qui permet de basculer cet attribut à false. Enfin, dans la méthode run() du thread, il suffit de continuer les traitements tant que l'attribut est à true et que les autres conditions fonctionnelles d'arrêt du thread sont négatives.

Exemple : exécution du thread jusqu'à l'appui sur la touche Entrée

```
01. public class MonThread6 extends Thread {
02.   private boolean actif = true;
03.
04.   public static void main(String[] args) {
05.     try {
06.       MonThread6 t = new MonThread6();
07.       t.start();
08.       System.in.read();
09.       t.arreter();
10.     } catch (Exception e) {
11.       e.printStackTrace();
12.     }
13.
14.   }
15.
16.   public void run() {
17.     int i = 0;
18.     while (actif) {
19.       System.out.println("i = " + i);
20.       i++;
21.     }
22.   }
23.
24.   public void arreter() {
25.     actif = false;
26.   }
27. }
```

Si la méthode start() est appelée alors que le thread est déjà en cours d'exécution, une exception de type IllegalThreadStateException est levée.

Exemple :

```
01. package com.jmdoudoux.test;
02.
03. public class MonThread5 {
04.
05.   public static void main(String[] args) {
06.     Thread t = new Thread(new MonThread3());
07.     t.start();
08.     t.start();
09.   }
10. }
```

Résultat :

```
1. java.lang.IllegalThreadStateException
2.       at java.lang.Thread.start(Native Method)
3.       at com.jmdoudoux.test.MonThread5.main(MonThread5.java:14)
4. Exception in thread "main"
```

La méthode sleep() permet d'endormir le thread durant le temps en millisecondes fourni en paramètres de la méthode.

La méthode statique currentThread() renvoie le thread en cours d'exécution.

La méthode isAlive() renvoie un booléen qui indique si le thread est en cours d'exécution.

8.3. La création et l'exécution d'un thread

Pour que les traitements d'une classe soient exécutés dans un thread, il faut obligatoirement que cette classe implémente l'interface Runnable puis que celle-ci soit associée directement ou indirectement à un objet de type Thread

Il y a ainsi deux façons de définir une telle classe

- la classe hérite de la classe Thread
- la classe implémente l'interface Runnable

8.3.1. La dérivation de la classe Thread

Le plus simple pour définir un thread est de créer une classe qui hérite de la classe java.lang.Thread.

Il suffit alors simplement de redéfinir la méthode run() pour y inclure les traitements à exécuter par le thread.

Exemple :

```
01. package com.jmdoudoux.test;
02.
03. public class MonThread2 extends Thread {
04.
05.   public void run() {
06.     int i = 0;
07.     for (i = 0; i > 10; i++) {
08.       System.out.println("" + i);
09.     }
10.   }
11.
12. }
```

Pour créer et exécuter un tel thread, il faut instancier un objet et appeler sa méthode start(). Il est obligatoire d'appeler la méthode start() qui va créer le thread et elle-même appeler la méthode run().

Exemple :

```
01. package com.jmdoudoux.test;
02.
03. public class MonThread2 extends Thread {
04.
05.   public static void main(String[] args) {
06.     Thread t = new MonThread2();
07.     t.start();
08.   }
09.
10.   public void run() {
11.     int i = 0;
12.     for (i = 0; i > 10; i++) {
13.       System.out.println("" + i);
14.     }
15.   }
16.
17. }
```

8.3.2. L'implémentation de l'interface Runnable

Si on utilise l'interface Runnable, il faut uniquement redéfinir sa seule et unique méthode run() pour y inclure les traitements à exécuter dans le thread.

Exemple :

```
01. package com.jmdoudoux.test;
02.
03. public class MonThread3 implements Runnable {
04.
05.   public void run() {
06.     int i = 0;
07.     for (i = 0; i > 10; i++) {
08.       System.out.println("" + i);
09.     }
```

```
  }
}
```

Pour pouvoir utiliser cette classe dans un thread, il faut l'associer à un objet de la classe Thread. Ceci se fait en utilisant un des constructeurs de la classe Thread qui accepte un objet implémentant l'interface Runnable en paramètre.

Exemple :

```
package com.jmdoudoux.test;

public class LancerDeMonThread3 {

  public static void main(String[] args) {
    Thread t = new Thread(new MonThread3());
    t.start();
  }
}
```

Il ne reste plus alors qu'à appeler la méthode start() du nouvel objet.

8.3.3. La modification de la priorité d'un thread

Lors de la création d'un thread, la priorité du nouveau thread est égale à celle du thread dans lequel il est créé. Si le thread n'est pas créé dans un autre thread, la priorité moyenne est attribuée au thread. Il est cependant possible d'attribuer une autre priorité plus ou moins élevée.

En java, la gestion des threads est intimement liée au système d'exploitation dans lequel s'exécute la machine virtuelle. Sur des machines de type Mac ou Unix, le thread qui a la plus grande priorité a systématiquement accès au processeur s'il ne se trouve pas en mode « en attente ». Sous Windows 95, le système ne gère pas correctement les priorités et il choisit lui-même le thread à exécuter : l'attribution d'une priorité supérieure permet simplement d'augmenter ses chances d'exécution.

La priorité d'un thread varie de 1 à 10, la valeur 5 étant la valeur par défaut. La classe Thread définit trois constantes :

MIN_PRIORITY :	priorité inférieure
NORM_PRIORITY :	priorité standard
MAX_PRIORITY :	priorité supérieure

Exemple :

```
package com.jmdoudoux.test;

public class TestThread10 {

  public static void main(String[] args) {
    System.out.println("Thread.MIN_PRIORITY  = " + Thread.MIN_PRIORITY);
    System.out.println("Thread.NORM_PRIORITY = " + Thread.NORM_PRIORITY);
    System.out.println("Thread.MAX_PRIORITY  = " + Thread.MAX_PRIORITY);
  }
}
```

Résultat :

```
Thread.MIN_PRIORITY  = 1
Thread.NORM_PRIORITY = 5
Thread.MAX_PRIORITY  = 10
```

Pour déterminer ou modifier la priorité d'un thread, la classe Thread contient les méthodes suivantes :

Méthode	Rôle
int getPriority()	retourne la priorité d'un thread
void setPriority(int)	modifie la priorité d'un thread

La méthode setPriority() peut lever l'exception IllegalArgumentException si la priorité fournie en paramètre n'est pas comprise en 1 et 10.

Exemple :

```
package com.jmdoudoux.test;
```

```
02.
03. public class TestThread9 {
04.
05.   public static void main(String[] args) {
06.     Thread t = new Thread();
07.
08.     t.setPriority(20);
09.   }
10. }
```

Résultat :

```
1. java.lang.IllegalArgumentException
2.     at java.lang.Thread.setPriority(Unknown Source)
3.     at com.jmdoudoux.test.MonThread9.main(TestThread9.java:8)
4. Exception in thread "main"
```

8.4. La classe ThreadGroup

La classe ThreadGroup représente un ensemble de threads. Il est ainsi possible de regrouper des threads selon différents critères. Il suffit de créer un objet de la classe ThreadGroup et de lui affecter les différents threads. Un objet ThreadGroup peut contenir des threads mais aussi d'autres objets de type ThreadGroup.

La notion de groupe permet de limiter l'accès aux autres threads. Chaque thread ne peut manipuler que les threads de son groupe d'appartenance ou des groupes subordonnés.

La classe ThreadGroup possède deux constructeurs :

Constructeur	Rôle
ThreadGroup(String nom)	création d'un groupe avec attribution d'un nom
ThreadGroup(ThreadGoup groupe_parent, String nom)	création d'un groupe à l'intérieur du groupe spécifié avec l'attribution d'un nom

Pour ajouter un thread à un groupe, il suffit de préciser le groupe en paramètre du constructeur du thread.

Exemple :

```
01. package com.jmdoudoux.test;
02.
03. public class LanceurDeThreads {
04.
05.   public static void main(String[] args) {
06.     ThreadGroup tg = new ThreadGroup("groupe");
07.     Thread t1 = new Thread(tg,new MonThread3(), "numero 1");
08.     Thread t2 = new Thread(tg,new MonThread3(), "numero 2");
09.   }
10. }
```

L'un des avantages de la classe ThreadGroup est de permettre d'effectuer une action sur tous les threads d'un même groupe. On peut, par exemple avec Java 1.0, arrêter tous les threads du groupe en lui appliquant la méthode stop().

8.5. Un thread en tâche de fond (démon)

Il existe une catégorie de threads qualifiés de démons : leur exécution peut se poursuivre même après l'arrêt de l'application qui les a lancés.

Une application dans laquelle les seuls threads actifs sont des démons est automatiquement fermée.

Le thread doit d'abord être créé comme thread standard puis transformé en démon par un appel à la méthode setDaemon() avec le paramètre true. Cet appel se fait avant le lancement du thread, sinon une exception de type IllegalThreadStateException est levée.

8.6. L'exclusion mutuelle

Chaque fois que plusieurs threads s'exécutent en même temps, il faut prendre des précautions concernant leur bonne exécution. Par exemple, si deux threads veulent modifier la même variable, il ne faut pas qu'ils le fassent en même temps.

Java offre un système simple et efficace pour réaliser cette tâche. Si une méthode déclarée avec le mot clé synchronized est déjà en cours d'exécution, alors les threads qui en auraient également besoin doivent attendre leur tour.

Le mécanisme d'exclusion mutuelle en Java est basé sur le moniteur. Pour définir une méthode protégée, afin de s'assurer de la cohérence des données, il faut utiliser le mot clé synchronized. Cela crée à l'exécution, un moniteur associé à l'objet qui empêche les méthodes déclarées synchronized d'être utilisées par d'autres objets dès lors qu'un objet utilise déjà une des méthodes synchronisées de cet objet. Dès l'appel d'une méthode synchronisée, le moniteur verrouille tous les autres appels de méthodes synchronisées de l'objet. L'accès est de nouveau automatiquement possible dès la fin de l'exécution de la méthode.

Ce procédé peut bien évidemment dégrader les performances lors de l'exécution mais il garantit, dès lors qu'il est correctement utilisé, la cohérence des données.

8.6.1. La sécurisation d'une méthode

Lorsque l'on crée une instance d'une classe, on crée également un moniteur qui lui est associé. Le modificateur synchronized place la méthode (le bloc de code) dans ce moniteur, ce qui assure l'exclusion mutuelle.

La méthode ainsi déclarée ne peut être exécutée par plusieurs processus simultanément. Si le moniteur est occupé, les autres processus seront mis en attente. L'ordre de réveil des processus pour accéder à la méthode n'est pas prévisible.

Si un objet dispose de plusieurs méthodes synchronized, ces dernières ne peuvent être appelées que par le thread possédant le verrou sur l'objet.

8.6.2. La sécurisation d'un bloc

L'utilisation de méthodes synchronisées trop longues à exécuter peut entrainer une baisse d'efficacité lors de l'exécution. Avec java, il est possible de placer n'importe quel bloc de code dans un moniteur pour permettre de réduire la longueur des sections de code sensibles.

```
synchronized  void methode1() {
   // bloc de code sensible
   ...
}

    void methode2(Object obj) {
   ...
   synchronized (obj) {
      // bloc de code sensible
      ...
   }
}
```

L'objet dont le moniteur est à utiliser doit être passé en paramètre de l'instruction synchronized .

8.6.3. La sécurisation de variables de classes

Pour sécuriser une variable de classe, il faut un moniteur commun à toutes les instances de la classe. La méthode getClass() retourne la classe de l'instance dans laquelle on l'appelle. Il suffit d'utiliser un moniteur qui utilise le résultat de getClass() comme verrou.

8.6.4. La synchronisation : les méthodes wait() et notify()

La suite de ce chapitre sera développée dans une version future de ce document