Threads en Java

Élisabeth Henry

17 octobre 2012

Une petite recette

Recette de cookies

- Préparation : 20 minutes
- Cuisson : 10 minutes

- Préchauffer le four à 180 degrés
- Mélanger le beurre et le sucre en poudre
- Ajouter les œufs, 1 pincée de sel et mélanger encore
- Dans un autre récipient, mélanger la farine et la levure
- Incorporer cette farine au mélange œuf-beurre
- Ajouter les pépites de chocolat et mélanger
- Former des tas de pâte et mettre au four
- Laisser cuire 10 minutes

Programmation de la recette

Classe Recette

```
public class Recette {

    private static void preparerIngredients ()
    {
    (...)
    }

    public static void main (String[] args)
    {
        Four four = new Four ();
        four.prechauffer (180);
        preparerIngredients ();
        System.out.println ("On met au four");
    }
}
```

Programmation de la recette

Classe Four

```
public class Four {
    private int temperature;
    public Four ()
    {
        temperature = 17;
    }
    public void prechauffer (int temperatureDesiree)
    {
        /* Cela prend un certain temps, donc on affiche
             tous les 20 degres la temperature actuelle.*/
        /* (...) */
        System.out.println ("Le four est chaud");
    }
}
```

Résultats

```
20
40
60
80
100
120
140
160
Le four est chaud
On ajoute le beurre
On ajoute le sucre
On ajoute les oeufs
On melange la farine et la levure
On ajoute les pepites
On forme les cookies
On met au four
```

Résultats

```
20
40
60
80
100
120
140
160
Le four est chaud
On ajoute le beurre
On ajoute le sucre
On ajoute les oeufs
On melange la farine et la levure
On ajoute les pepites
On forme les cookies
On met au four
```

Pendant que le four préchauffe, on ne fait rien...

Importance du multithreading

- Il n'est pas toujours pertinent d'exécuter l'ensemble d'un programme séquentiellement
- $\rightarrow$ ressources bloquantes
  - four.prechauffer () ;
  - lecture/écriture sur des périphériques
  - demande d'entrée de l'utilisateur
  - attente d'une connexion réseau
  - ...
- $\rightarrow$ utilisation de plusieurs processeurs

Threads et processus

- Un processus est un programme en cours d'exécution
- Contient :
  - des instructions (les lignes de code) ;
  - de la mémoire (espace d'adressage) ;
  - des ressources (fichiers ouverts, sockets, ...).
- Il est possible d'avoir plusieurs threads dans un processus
  - la mémoire, les ressources, le code... sont partagés entre les threads d'un même processus ;
  - ce qui distingue deux threads c'est leur ≪position≫ dans le code (d'un point de vue technique, la pile et les registres)

Les threads en Java

En Java, il existe deux façons de gérer plusieurs Threads :

- l'interface Runnable ;
- la classe Thread, qui implémente cette interface.

Interface Runnable

L'interface Runnable définit principalement une méthode void run () qu'une classe doit implémenter pour lancer un nouveau thread. Il est ensuite possible de lancer cette méthode en parallèle du reste du programme.

Classe Thread

La classe Thread implémente l'interface Runnable. Là encore, il est nécessaire d'implémenter la méthode void run (). La classe Thread fournit par ailleurs un certain nombre de méthodes utiles.

Utilisation de la classe Thread

```
public class MaClass extends Thread
{
  void run ()
  {
    /* Du code */
  }

  void uneAutreMethode ()
  {
    /* Encore du code */
  }

  public static void main (String[] args)
  {
    MaClasse objet = new MaClasse ();
    objet.start ();
    objet.uneAutreMethode ();
    /* uneAutreMethode () et objet.run () s'executent en ↩
        parallele */
  }
}
```

Utilisation de l'interface Runnable

```
public class MaClass implements Runnable
{
  void run ()
  {
    /* Du code */
  }

  void uneAutreMethode ()
  {
    /* Encore du code */
  }

  public static void main (String[] args)
  {
    MaClasse objet = new MaClasse ();
    Thread thread = new Thread (objet);
    thread.start ();
    objet.uneAutreMethode ();
    /* uneAutreMethode () et thread.run () s'executent en ↩
        parallele */
  }
}
```

Retournons à nos cookies

Classe four

```
public class Four extends Thread {
    private int temperature;
    private int temperatureDesiree;
    public Four (int temperatureDesiree) {
        temperature = 17;
        this.temperatureDesiree = temperatureDesiree;
    }
    public void run () {
        prechauffer (temperatureDesiree);
    }
    /* La methode prechauffer, elle, ne change pas */
  /* (...) */
}
```

- run () se charge maintenant d'appeler prechauffer (temperature) ;
- comme run () ne prend pas de paramètres, et que prechauffer demande un entier, il faut le passer avant (ici, dans le constructeur).

Cookies multithreads

Classe Recette

```
public static void main (String[] args)
{
    Four four = new Four (180);
    four.start ();
    preparerIngredients ();
    System.out.println ("On met au four");
}
```

Différences

- Seul la méthode main change (un peu).
- Au lieu d'appeler directement four.prechauffer (temperature), il est nécessaire de passer la température au constructeur, puis d'appeler four.start () (et non pas four.run ()...)

Résultats

```
On ajoute le beurre
20
40
On ajoute le sucre
60
80
On ajoute les oeufs
100
120
On melange la farine et la levure
140
160
On ajoute les pepites
Le four est chaud
On forme les cookies
On met au four
```

Synchronisation entre threads

Dans cet exemple, on gagne du temps en préchauffant le four pendant qu'on commence à préparer

MAIS...

À aucun moment, on ne vérifie que le four est effectivement chaud avant de mettre le plat dedans.

Lorsqu'on lance plusieurs threads, on a souvent besoin de les synchroniser à un moment donné (en général lorsqu'ils ont fini leur traitement).
En effet, il faut vérifier qu'un traitement a bien été accompli avant de passer à la suite.

- Thread fournit la méthode join (), qui attend (éventuellement un temps limité) qu'un thread ait terminé son exécution.
- Lance une InterruptedException si le thread appelant a été interrompu.

Nouvelle méthode Recette.main

```
public static void main (String[] args) throws ↩
    InterruptedException
{
    Four four = new Four (180);
    four.start ();
    preparerIngredients ();
    four.join ();
    System.out.println ("On met au four");
}
```

- On s'assure ici que le thread ≪préchauffement du four≫ s'est terminé avant de continuer.

Sleep

La classe Thread fournit d'autres méthodes :

- static void sleep (int milliseconds) throws InterruptedException
- Cette méthode arrête le thread courant pendant le nombre de millisecondes passées en paramètre.

```
/* Dans la classe Recette */
public static void attendre (int duree) {
    try {
            Thread.sleep (1000 * duree);
    } catch (Exception e){}
}

private static void preparerIngredients () {
    System.out.println ("On ajoute le beurre");
    attendre (2);
    System.out.println ("On ajoute le sucre");
    attendre (2);
    /* Etc. */
}
```

Interruptions

Interrupt

- public void interrupt () throws SecurityException
- interrompt un thread existant : thread.interrupt ()
- le thread interrompu reçoit une InterruptedException s'il est dans une méthode (sleep, join) qui peut la lancer
- sinon, il peut savoir qu'il a été interrompu grâce à la méthode isInterrupted

isInterrupted

- public boolean isInterrupted ()
- permet de savoir si un thread a été interrompu
- si ça renvoie true, le thread doit se terminer proprement

Exemple sleep et interrupt

```
import java.io.*;
class Boucle extends Thread {
    public void run () {
        for (int i = 0; !this.isInterrupted (); i++)
        {
            try {
                Thread.sleep (1000);
                System.out.println (i);
            } catch (InterruptedException e) {
                return;
            }
        }
    }}
public class InterruptTest  {
    public static void main(String[] args) throws ↩
        IOException {
        Boucle b = new Boucle ();
        b.start ();
        BufferedReader reader = new BufferedReader (new ↩
            InputStreamReader (System.in));
        reader.readLine ();
        b.interrupt ();
}}
```

Autres méthodes de Thread

- boolean isAlive () : renvoie true si le thread est ≪vivant≫, c'est-à-dire s'il a commencé mais n'a pas été terminé ;
- static Thread currentThread () : renvoie une référence au thread en cours d'exécution ;
- final void setDeamon (boolean on) : permet de faire de ce thread un thread ≪daemon≫ (si on est true) ou ≪utilisateur≫ (si on est false) ;
- final boolean isDeamon () : renvoie true si le thread est ≪daemon≫ ;

Thread daemon

Par défaut, les thread créés sont ≪utilisateur≫, et le programme ne se termine que lorsque tous les threads sont terminés. À l'inverse, les threads ≪daemon≫ sont terminés dès qu'il n'y a plus de threads utilisateur.

Ressources partagées

- Dans les cas simples, des threads différents s'occupent de ressources différentes
- Lorsque deux threads sont amenés à manipuler les mêmes ressources (variables par exemple), il faut faire attention car rien n'indique qu'un autre thread n'a pas modifié une variable entre deux instructions

```
/* n est un entier */
system.out.printf ("n vaut %d\n", n);
system.out.printf ("Le carre de n vaut %d\n", n * n);
```

Dans cet exemple, si un autre thread peut modifier n, rien n'indique que d'une ligne à l'autre la valeur de n n'ait pas changé
$\rightarrow$ les données affichées ne sont pas cohérentes.

Mot-clé synchronized

Pour éviter que deux threads n'accèdent à une même variable en même temps, Java propose le mot-clé synchronized.

```
public class MaClasse {
    void synchronized uneFonction () {
        /* ... */
    }
    void synchronized uneAutreFonction () {
        /* ... */
    }
}
```

Quelque soit le nombre de threads créés, on s'assure ainsi que pour un objet monObjet de type MaClasse, les méthodes uneFonction () et uneAutreFonction () ne pourront pas être exécutées en même temps.
Autrement dit, pour accéder à une méthode synchronisée d'un objet, un thread met un verrou sur cet objet et s'assure ainsi qu'aucune autre méthode synchronisée ne sera exécutée en même temps.

Le retour du mot-clé synchronized

Il est aussi possible d'utiliser le mot clé synchronized d'une autre façon, en limitant le verrouillage à un seul bloc au lieu de toute une méthode.

```
public class MaClasse {
    public void uneFonction () {
        /* Pas de verouillage de cette portion du code */
        synchronized (this) {
            /* Verouillage de cette portion */
        }
        /* Pas de verrouillage de cette portion */
    }
    public void uneAutreFonction () {
        /* Pas de verrouillage */
        synchronized (this) {
            /* Verrouillage */
        }
        /* Pas de verrouillage */
    }
}
```

wait () et notifyAll ()

Parfois, un thread doit attendre qu'un autre thread ait effectué une tâche concernant un objet avant de continuer.
La classe Object propose des méthodes pour gérer cela :

- void wait () throws InterruptedException : le thread va attendre jusqu'à ce qu'un autre thread envoie une notification concernant cet objet (cela lance alors l'Exception). Avant d'utiliser monObjet.wait (), un thread doit avoir le verrou sur cet objet (via synchronized) ; cela a pour effet de libérer le verrou.
- void notifyAll () : réveille tous les threads qui attendent pour utiliser un objet verrouillé. Là encore, avant d'utiliser monObjet.notifyAll (), un thread doit avoir le verrou sur cet objet.
- void notify () : idem, mais ne réveille qu'un seul thread parmi ceux qui attendent. (Déconseillé, car le thread choisi dépend de l'environnement)

Deadlock

Imaginons...

- qu'un premier thread cherche à poser un verrou d'abord sur l'objet o1, puis sur l'objet o2 ;
- qu'un second thread cherche à poser un verrou d'abord sur l'objet o2, puis sur l'objet o1.

Il est alors possible que le premier thread pose un verrou sur l'objet o1, pendant que le second thread pose un verrou sur l'objet o2. Le premier thread attend alors de pouvoir accéder à l'objet o2, tandis que le second thread attend d'accéder à l'objet o1.
$\rightarrow$ On appelle cela deadlock (interblocage en français). Les deux threads se bloquent alors mutuellement.
$\rightarrow$ Très difficile à débugger, car deux threads peuvent se bloquer mutuellement ou pas en fonction de leur ordre d'activation. Il faut donc faire attention à l'ordre dans lequel on pose des verrous sur des objets.

Résumé

- Pour créer un thread, une classe doit hériter de la classe Thread (ou implémenter l'interface Runnable) et redéfinir la méthode void run () ;
- Pour lancer un thread :
  Thread t = new Thread () ;
  t.start () ;
- t.join () attend la fin du thread t ;
- Thread.sleep (duree) permet de mettre le thread appelant en veille ;
- t.interrupt () envoie une interruption au thread t ;
- this.isInterrupted () permet de savoir si le thread a été interrompu ;
- Pour le partage de ressources entre plusieurs threads : mot-clé synchronized.