Cours programmation par objets et Java

2ème partie: Concepts avancés du langage Java

Jacques Ferber

dernière mise à jour : 17 octobre 1999

1. Types et polymorphisme en Java

Parler du choix des méthodes en fonction des types des arguments. Parler aussi du problème des constructeurs. {{à finir}}

1.1 Types et affectations

Principe de gestion des types {{à finir}}

1.2 Polymorphisme des méthodes

Montrer comment cela fonctionne pour les méthodes, mais aussi pour les constructeurs {{à finir}}

2. Erreurs et exceptions

2.1 Principe

Technique du try, catch, finally, throw.

Syntaxe:

```
try {
        . . .
    }
catch (<classe d'exception> <variable>){
        . . .
    }
catch ( . . . ) {
        . . .
    } . . .
```

Le cas classique: erreur sur une division par 0:

```
int i = 0;

try {
   i = i/i;
}

catch(ArithmeticException 0) {
   System.out.println("on a une division par zéro, mais on continue");
}
```

Autre exemple: on veut imprimer sur un fichier mais cela conduit à une erreur.

```
int a = 7; //par exemple

try {

   System.out.println("on démarre le try statement");
   pStr = new PrintStream(
               new BufferedOutputStream(
                     new FileOutputStream("fichierSortie.txt")));
   pStr.println("valeur de a:" + a);
}
catch (IOException e) {
       System.err.println("Erreur entrée-sortie: " + e.getMessage());
}
finally {
   if (pStr != null) {
      System.out.println("on ferme le PrintStream");
      pStr.close();
```

```
    } else
        System.out.println("le PrintStream est fermé");
}
...
```

Note:

1. les catch blocs sont essayés les uns après les autres jusqu'à ce que l'un d'entre eux corresponde (teste de type isa?).
2. On peut renvoyer l'exception avec le code **`throw(e)`** où e est l'exception.
3. Si l'on veut récupérer plusieurs exceptions avec un seul gestionnaire (handler), il faut utiliser la plus petite classe commune entre toutes les exceptions que l'on veut utiliser ainsi. Mais évidemment, on récupère alors des tas d'autres exceptions qui dérivent de cette classe.
4. Il y a n+2 sorties pour un bloc try, où n est le nombre de gestionnaire (catch) attaché à un bloc. Soit tout se passe normalement et le contrôle est passé au bloc d'instruction qui suit le try et les catch; soit il y a une "erreur" et l'un des gestionnaires traite l'exception; soit aucun gestionnaire ne peut la traiter et c'est récupéré par une exception de haut niveau (**`Exception`** par exemple).

Le bloc **`finally`** permet de décrire du code qui sera exécuté quelle que soit la sortie du bloc **`try`**. Exemple, pour l'impression dans un fichier:

2.2 La hiérarchie des principales exceptions et erreurs

Seules provenant des packages java.lang et io sont listées. Il y en a vraiment beaucoup !!

class java.lang.**Erreur! Signet non défini.** (implements java.io.**Erreur! Signet non défini.**)

- class java.lang.**Erreur! Signet non défini.**
  - class java.awt.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**
      - class java.lang.**Erreur! Signet non défini.**
      - class java.lang.**Erreur! Signet non défini.**
      - class java.lang.**Erreur! Signet non défini.**
      - class java.lang.**Erreur! Signet non défini.**
      - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**

- class java.lang.**Erreur! Signet non défini.**
- class java.lang.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**

- class java.lang.**Erreur! Signet non défini.**
  - class java.awt.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
  - class java.io.**Erreur! Signet non défini.**
    - class java.io.**Erreur! Signet non défini.**
    - class java.io.**Erreur! Signet non défini.**
    - class java.io.**Erreur! Signet non défini.**
    - class java.io.**Erreur! Signet non défini.**
    - class java.io.**Erreur! Signet non défini.**
      - class java.io.**Erreur! Signet non défini.**
      - class java.io.**Erreur! Signet non défini.**
      - class java.io.**Erreur! Signet non défini.**
      - class java.io.**Erreur! Signet non défini.**
      - class java.io.**Erreur! Signet non défini.**
      - class java.io.**Erreur! Signet non défini.**
      - class java.io.**Erreur! Signet non défini.**
    - class java.io.**Erreur! Signet non défini.**
    - class java.io.**Erreur! Signet non défini.**
    - class java.io.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
  - class java.lang.reflect.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**
      - class java.lang.**Erreur! Signet non défini.**
      - class java.lang.**Erreur! Signet non défini.**
    - class java.lang.**Erreur! Signet non défini.**

- class java.lang.**Erreur! Signet non défini.**
- class java.lang.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
  - class java.lang.**Erreur! Signet non défini.**
- class java.lang.**Erreur! Signet non défini.**
- class java.lang.**Erreur! Signet non défini.**
- class java.lang.**Erreur! Signet non défini.**

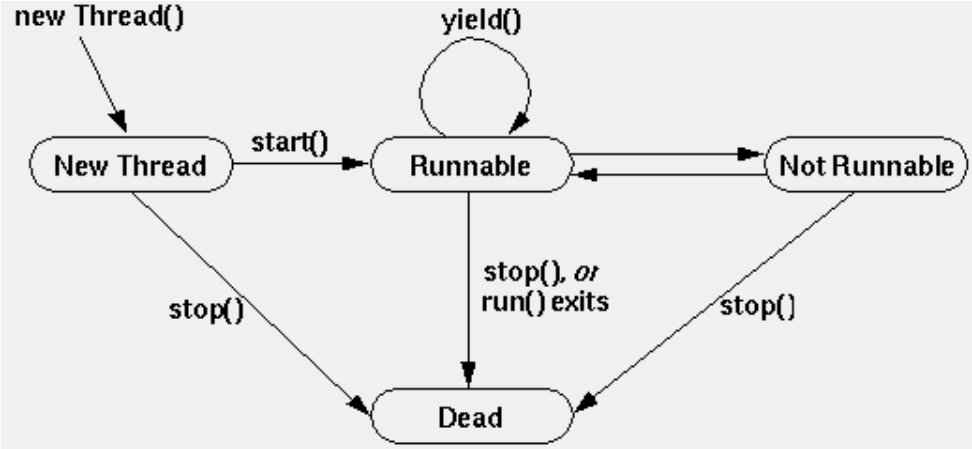
3. Fichiers. Flots d'entrées-sorties. Reader et writer

{{à finir}}

4. Threads et programmation concurrente en Java

Les threads servent à construire des unités de programmes qui travaillent en "parallèles" (ce que l'on appelle la concurrence).

Un thread est dans l'un des états suivants:



Les méthodes qu'un thread reconnaît sont les suivantes:

- **`start()`**: démarre l'exécution d'un thread
- **`stop()`**: termine son exécution (il passe dans l'état "mort") (a n'utiliser qu'avec ciconspection).
- **`suspend()`**: suspend l'exécution (doit être relancé par un "resume" ensuite. Fait passer le thread dans un état d'attente (not runnable).
- **`resume()`**: relance et "résume" l'exécution.
- **`yield()`**: redonne explicitement le contrôle au scheduler.
- **`sleep(long)`**: fait passer le thread en état d'attente pendant une certaine durée (comptée en millisecondes).
- **`run()`**: la méthode qui est exécutée lorsque le thread s'exécute. C'est souvent une boucle.

4.1.1 Exemple1

```
class SimpleThread extends thread {
   essaiThread(String name) {
      super(name)
   }

   public void run() {
      for(int=0; i < 10; i++)
         System.out.println(getName()+i);
      }
}

class EssaiThread {
   public static void main(String argv[]) {
      SimpleThread essaiA = new SimpleThread("c
      SimpleThread essaiB = new SimpleThread("Thread B:");
      SimpleThread essaiC = new SimpleThread("Thread C:");

      essaiA.start();
      essaiB.start();
      essaiC.start();
   }
}
```

Résultats:

```
Thread A:1
Thread B:1
Thread C:1
Thread A:2
Thread B:2
Thread C:2
Thread A:3
Thread B:3
Thread C:3
```

Autre possibilité pour construire un Thread. Faire une implémentation de "runnable".

```
import java.awt.*;
import java.util.Date;

public class Clock extends Applet implements Runnable {

    Thread clockThread = null;

    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
    public void run() {
      // boucle se termine lorsque la clock est mise à null dans
      // la méthode stop()
      while (Thread.currentThread() == clockThread) {
            repaint();
            try {
                clockThread.sleep(1000); // il s'endort une seconde
            } catch (InterruptedException e){}
      }
    }
    public void paint(Graphics g) {
        Date now = new Date(); // a chaque fois on crée une date (lourd)
        g.drawString(now.getHours() + ":" + now.getMinutes() + ":" +
                now.getSeconds(), 5, 10);
    }
```

```
    public void stop() { //si on stoppe l'applet(pas le thread)
        clockThread = null;
    }
}
```

Note: à la place de la boucle **while**, on aurait pu écrire:

```
while(clockThread != null && clockThread.isAlive()) {
      ...
}
```

Pour faire un objet qui devient "threadable", il suffit d'implémenter l'interface **Runnable** et de décrire une méthode **run**. Puis on construit un thread en lui passant l'objet en argument de sa création:

```
new Thread(unObjetRunnable,"une chaîne")
```

4.1.2 Problèmes de "scheduling"

Tout ne se passe pas comme c'est écrit. Les threads sont activés par un "scheduler" qui dépend de la machine. Il arrive parfois (et en particulier sur les Sun d'après ce que j'ai entendu) que le système d'exploitation ne donne pas bien la main aux threads.

On peut aussi changer la priorité du thread avec la méthode **setPriority(int)**. Le thread avec la plus grande valeur (le plus prioritaire) est activé préférentiellement.

Si le temps découpé (time slicing) n'est pas bien implémenté, il faut implémenter à la main des techniques du genre:

```
yield()
```

Par exemple, si le **EssaiThread** ne fonctionne pas parfaitement, une technique consiste à introduire la commande **yield()** directement dans le code. On peut aussi utiliser des **sleep(n)** de manière à donner aux autres threads l'occasion d'avoir la main.

```
class SimpleThread extends thread {
   essaiThread(String name) {
      super(name)
   }

   public void run() {
      for(int=0; i < 10; i++)
         System.out.println(getName()+i);
         yield();    // donne la main aux autres threads
      }
}
```

Attention: l'usage des threads introduit les techniques et concepts de la programmation concurrente. Les problèmes d'accès partagés, de deadlocks, etc. sont le lot normal de ces programmes.

4.2 Interfaces et "runnable"

On peut créer une classe qui implémente l'interface Runnable. Il suffit de décrire la méthode run, et de construire un processus en passant cet objet en paramètre. Exemple:

```
public class Truc extends Chose implements Runnable{

   int j = 0;

   Truc() {
      new Thread(this);
   }

   run() {
      while (true) {
         System.out.println("je suis vivant: " + j);
   }

}
```

4.3 Synchronisation

On ajoute le mot clé "synchronized" dans l'en-tête des méthodes. Ces méthodes servent à construire ce que l'on appelle des "moniteurs" c'est-à-dire des structures de données qui sont protégées de telle manière que seules des procédures qui travaillent en exclusion mutuelles puissent accéder aux objets.

Exemple: deux processus veulent accéder à un compte en banque:

```
public class Compte
   int solde = 0;
   public void deposer(int s) {
      int so = solde + s;
      solde = so;
   }
   public void retirer(int s) {
      int so = solde - s;
      solde = so;
   }
}


public class MyThread extends Thread {

   Compte c;

   MyThread(Compte aCompte){
      c = aCompte;
   }

   public void run()
      c.deposer(100);
}


....
Compte c1 = new Compte();
MyThread t1, t2;
t1 = new MyThread(c1);
t2 = new MyThread(c2);

t1.start();
t2.start();
```

Solution: utiliser la commande "synchronized" qui fait en sorte que les méthodes soient exécutées en exclusion mutuelle. Dans l'exemple précédent, voici la nouvelle classe **Compte**:

```
public class Compte
   int solde = 0;
   public synchronized void deposer(int s) {
      int so = solde + s;
      solde = so;
   }
   public synchronized void retirer(int s) {
      int so = solde - s;
      solde = so;
   }

}
```

5. Collections

{{à finir}}

5.1 Vector

5.2 Dictionary