Chapitre 6

Les exceptions en Java

Lorsqu'on conçoit un programme, on essaie évidemment qu'il soit *correct*, c'est-à-dire qu'il fonctionne parfaitement dans les conditions prévues de son utilisation. L'utilisateur peut cependant ne pas bien dominer ces conditions et essayer des valeurs non prévues, ce qui conduit à des **erreurs d'exécution**. Les plus fréquentes sont la division par zéro, l'entrée d'un entier négatif alors que l'on attend un entier naturel, le dépassement des limites d'un tableau par un indice, l'épuisement de la mémoire...

Il n'est pas possible de prévoir toutes les erreurs d'exécution. L'expérience permet cependant de dégager un certain nombre d'erreurs potentielles. On peut traiter ces erreurs, par exemple en vérifiant systématiquement que b est différent de zéro avant d'effectuer la division de a par b. Si b est égal à zéro et que l'on ne connaît pas l'instruction à effectuer dans ce cas, on peut tout simplement l'indiquer à l'utilisateur. On parle d'**exception** pour les erreurs potentielles traitées au moment de la conception du programme.

Prendre en compte les exceptions alourdit évidemment beaucoup le programme, tant du point de vue de sa taille que de sa compréhension. Certains langages de programmation, tel que Java, possèdent une prise en compte des exceptions qui en facilite la compréhension.

6.1 Méthodes de traitement des exceptions

Il existe fondamentalement deux méthodes de traitement des exceptions : la méthode directe et la méthode par gestionnaire.

Méthode directe.- La méthode directe est celle que l'on utilise spontanément. Les erreurs (à lesquelles on pense) sont gérées aux endroits où les erreurs peuvent se produire.

L'avantage de cette approche réside dans la possibilité qu'elle offre au programmeur qui relit du code de voir le code de gestion d'erreurs dans la proximité immédiate des lignes de programme génératrices d'erreurs éventuelles, et de déterminer ainsi si la vérification d'erreur adéquate est bien mise en place.

Le problème de cette stratégie est que le code est en quelque sorte pollué par le traitement des erreurs. Le programmeur intéressé à relire l'application elle-même éprouve quelques difficultés à lire le programme et à déterminer s'il fonctionne correctement. Ceci complique donc la compréhension et la maintenance de l'application.

Méthode par gestionnaire des erreurs.- Une autre méthode consiste à séparer le code de gestion de l'erreur du programme proprement dit. Lorsqu'une erreur est rencontrée, on **lève une exception**. Le traitement de l'exception , c'est-à-dire le code qui s'exécute lorsqu'une exception a été détectée, est l'objet du **gestionnaire d'exception** associé.

Lorsqu'on lève une exception, il se peut qu'il n'y ait pas de gestionnaire de l'exception associé. Si un tel gestionnaire existe, l'exception est **capturée** et **traitée**.

6.2 Traitement des exceptions par gestionnaire

Il n'y a pas grand chose à dire sur le traitement des exceptions par la méthode directe. Intéressons-nous donc au traitement des exceptions par gestionnaire en Java.

6.2.1 Principe

6.2.1.1 Les classes d'exception

Notion.- Comme Java est un langage orienté objet, une exception sera liée à un objet. Toutes les classes de types d'exceptions dériveront de la classe :

```
Exception
```

Définition d'une classe d'exception.- Une classe d'exception possède normalement deux constructeurs : le constructeur par défaut et un constructeur dont le paramètre est une chaîne de caractères. L'idée est que dans le premier cas, on ait un message d'exception prédéfini et dans le second cas un message passé à travers la chaîne de caractères.

6.2.1.2 Lancement d'une exception

Notion.- Pour indiquer qu'une exception s'est produite, on **lance** ou on **lève une exception**. En Java, il s'agit tout simplement d'une instruction qui consiste à instancier une classe d'exception précédé du mot clé `throw` :

`throw new` *TypeException*`() ;`

Il appartient évidemment au programmeur de décider dans quelles conditions quand il veut lever une exception. Les concepteurs du langage Java ont créé un certain nombre de classes d'exception, celles-ci étant levées d'une façon transparente au programmeur.

Lien entre une méthode et un type d'exception.- Comme d'habitude en Java, le lien doit être explicite entre une méthode dans laquelle on lance une exception et la classe d'exception coreespondante. Cela se fait grâce à la clause `throws` devant suivre l'en-tête de la définition de la méthode :

```
type nom(param) throws a,b,c
     {
     - - -
     }
```

où `a`, `b` et `c` sont des classes d'exception.

Nous l'avons déjà vu mis en œuvre à propos de la classe `IOException`. Comme d'habitude également, il est inutile de faire ce lien pour les classes d'exception par défaut, c'est-à-dire du paquetage `java.lang`.

6.2.1.3 Capture des exceptions

Syntaxe.- Le programmeur enferme dans un bloc précédé du mot clé `try` (*essayer* en anglais) le code qui peut générer une exception :

```
try
   {
   - - -
   }
```

Le bloc `try` est immédiatement suivi de zéro, un ou plusieurs **blocs de capture** de la forme suivante :

```
catch({\it TypeException} ref)
     {
     - - -
     }
```

où *TypeException* est une classe d'exception. La variable `ref` pourra servir éventuellement dans le corps du bloc.

Après le dernier bloc de capture on peut placer un bloc optionnel, précédé du mot clé `finally`. Il doit y avoir au moins un bloc, soit de capture, soit `finally`.

Sémantique.- Lors du lancement d'une exception, le contrôle du programme quitte le bloc `try` et une recherche débute dans l'ordre parmi les blocs de capture pour trouver un gestionnaire approprié. Si le type de l'exception levée correspond au type de paramètre de l'un des blocs de capture, le code de ce dernier est exécuté. Si aucune exception n'est lancée dans le bloc `try`, les gestionnaires d'exception de ce bloc sont éludés et le programme poursuit son exécution après le dernier bloc de capture. Si le bloc `finally` se présente après le dernier bloc de capture, il est exécuté sans regard de la levée d'une exception ou non.

6.2.2 Un exemple

Écrivons une application Java consistant à saisir deux entiers naturels, à calculer le quotient dans la division euclidienne du premier par le deuxième et à afficher ce quotient.

6.2.3 Programme sans traitement d'exception

L'application suivante, que nous savons écrire depuis longtemps, convient :

```
import java.io.*;

class Quotient
   {
   public static void main(String args[]) throws IOException
      {
      InputStreamReader fich;
      BufferedReader ligne;
      String mot;
      int a, b, q;
      fich = new InputStreamReader(System.in);

      ligne = new BufferedReader(fich);
      System.out.print("a = ");
      mot = ligne.readLine();
      a = Integer.parseInt(mot);

      ligne = new BufferedReader(fich);
      System.out.print("b = ");
      mot = ligne.readLine();
      b = Integer.parseInt(mot);

      q = a/b;
      System.out.println("a/b = " + q);
      }
   }
```

Il n'y a aucun traitement d'exception dans cette application bien qu'on lance `IOException`, dont nous ne savons pas quoi faire pour le moment.

L'exécution se fait de façon attendue :

```
linux:/windows/C/applis/info/java/program/ch6 # java Quotient
a = 12
b = 4
a/b = 3
```

6.2.3.1 Programme avec traitement d'exceptions

Introduction.- Plusieurs types d'exceptions sont prévisibles pour un tel programme. Le premier est évidemment la division par zéro. Remarquons que Java traite cette exception :

```
linux:/windows/C/applis/info/java/program/ch6 # java Quotient
a = 12
b = 0
java.lang.ArithmeticException: / by zero
        at Quotient.main(Quotient.java:23)
linux:/windows/C/applis/info/java/program/ch6 #
```

Le second type est la saisie d'autre chose qu'un entier. Java traite également ce type d'exceptions :

```
linux:/windows/C/applis/info/java/program/ch6 # java Quotient
a = 43
b = bon
java.lang.NumberFormatException: bon
        at java.lang.Integer.parseInt(Integer.java)
        at java.lang.Integer.parseInt(Integer.java)
        at Quotient.main(Quotient.java:21)
linux:/windows/C/applis/info/java/program/ch6 #
```

Nous allons considérer ces deux types d'exception. Pour le deuxième type nous utiliserons la classe de Java dont nous venons de voir le nom, à savoir `NumberFormatException`. Pour le second type, nous allons créé une classe nous-même.

Un programme.- Ceci nous conduit au programme suivant :

```
import java.io.*;

public class Quotient2
   {
   public static void main(String args[]) throws IOException
      {
      InputStreamReader fich;
      BufferedReader ligne;
      String mot;
      int a, b, q;
      fich = new InputStreamReader(System.in);

      try
         {
         ligne = new BufferedReader(fich);
         System.out.print("a = ");
         mot = ligne.readLine();
         a = Integer.parseInt(mot);

         ligne = new BufferedReader(fich);
         System.out.print("b = ");
```

```
         mot = ligne.readLine();
         b = Integer.parseInt(mot);

         q = quotient(a,b);
         System.out.println("a/b = " + q);
         }
      catch(ExceptionDivisionParZero e)
         {
         System.out.print("Tentative de division ");
         System.out.println("par zero");
         }
      catch(NumberFormatException e)
         {
         System.out.print("Format d'entier non ");
         System.out.println("valable");
         }
      }

// methode avec lancement d'une exception

   public static int quotient(int a, int b) throws ExceptionDivisionParZero
      {
      if (b == 0) throw new ExceptionDivisionParZero();
      return a/b;
      }
   }

// Definition d'un type d'exception

class ExceptionDivisionParZero extends Exception
    {
    public ExceptionDivisionParZero()
       {
       super("Essai de division par zero");
       }
    public ExceptionDivisionParZero(String message)
       {
       super(message);
       }
    }
```

6.3 Les exceptions prédéfinies

Java implémente un certain nombre de types d'exceptions, dont beaucoup se comprennent d'elles-mêmes. En voici la hiérarchie :

```
Exception
  IOException
    EOFException
    FileNotFoundException
    MalformedURLException
    UnknownHostException
  ClassNotFoundException
  CloneNotSupportedException
  RuntimeException
    ArithmeticException
    ClassCastException
    IllegalArgumentException
       NumberFormatException
    IllegalStateException
    IndexOutOfBoundsException
       ArrayIndexOutOfBoundsException
    NoSuchElementException
    NullPointerException
```

Exercice 1.- *Écrire une application Java demandant le nom d'un fichier (texte) et affichant le contenu de ce fichier à l'écran ou « fichier non présent » si on ne trouve pas ce fichier.*

Exercice2.- *Écrire une application Java permettant d'entrer un tableau d'entiers, de le trier (grâce à un tri à bulle) et d'afficher le résultat. On affichera « Problème d'index dans le tableau » lorsqu'il y a un débordement.*