Chapitre 7

Erreurs et exceptions

La gestion des erreurs est profondément intégrée au langage Java. Nous allons voir comment examiner, contrôler des exceptions, un mécanisme qui sépare le code normal du code de gestion des erreurs. Nous pourrons même créer nos propres exceptions.

Lors de l'exécution d'un programme, il se produit parfois des erreurs. Certaines erreurs sont normalement fatales à l'exécution du programme. Java dispose de moyens permettant de traiter ces erreurs proprement en prévoyant des procédures de traitement alternatives. C'est le mécanisme des exceptions. Un grand nombre d'exceptions sont prévues par le langage, mais l'utilisateur peut en définir lui-même.

7.1 Exemple d'erreur

Prenons la classe suivante, qui tente un début de modélisation des nombres rationnels.

```
public class Fraction {


    private int numerateur;
    private int denominateur;

    public Fraction(int num,int denom) {
        numerateur = num;
        denominateur = denom;
    }

    public double ValeurDecimale(){
        return (double)numerateur/denominateur;
    }

    public long ValeurEntiere(){
        return numerateur/denominateur;
    }

    public static void main(String[] args){
        Fraction fraction=new Fraction(1,2);
        System.out.println("Valeur␣décimale␣"+fraction.ValeurDecimale());
        System.out.println("Valeur␣entière␣"+fraction.ValeurEntiere());
    }
```

```
}
```

L'exécution de la méthode `main` produit normalement l'affichage suivant :

```
Valeur décimale 0.5
Valeur entière 0
```

Si on remplace 2 par 0 lors de la création de l'objet dans la méthode `main()`, voici ce qui apparaît :

```
Valeur décimale Infinity
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exceptions.Fraction.ValeurEntiere(Fraction.java:34)
        at exceptions.Fraction.main(Fraction.java:40)
Java Result: 1
```

La méthode `valeurDecimale` produit une valeur particulière nommée `Infinity`, pas très utile pour la suite des calculs, mais la méthode `valeurEntiere`, en tentant une division par 0, provoque l'arrêt du programme et l'affichage d'un message, suivi de la liste des appels de méthodes en cours au moment de l'incident (l'état de la pile).

On peut tenter de rattraper l'erreur. Il faut à ce moment déterminer à l'avance à quel type d'erreur on veut faire face. Dans la plupart des cas, il s'agira d'attraper une *exception*, une classe fréquemment employée pour gérer le mécanisme de l'erreur.

```
public static void main(String[] args){
    Fraction fraction=new Fraction(1,0);
     try{
         Fraction fraction=new Fraction(1,0);
         System.out.println("Valeur décimale "+fraction.ValeurDecimale());
         System.out.println("Valeur entière "+fraction.ValeurEntiere());
     }
     catch(java.lang.ArithmeticException ex)
     {
         System.err.println("Une fraction à diviseur nul a été manipulée");
         System.err.println(ex.getMessage());
         ex.printStackTrace();
     }
}
```

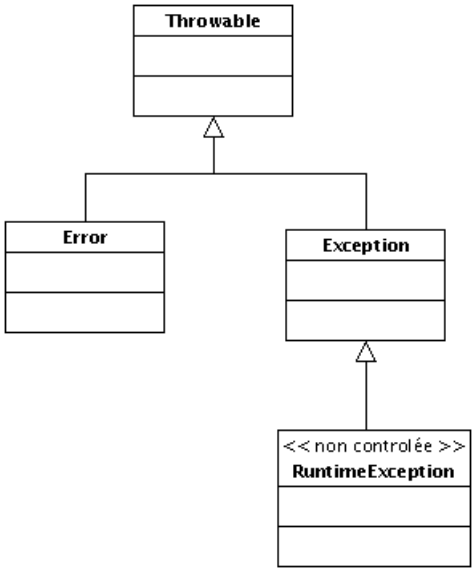
Nous plaçons les instructions susceptibles de provoquer l'erreur dans un bloc contrôlé par `try`. Un bloc `catch` est ensuite prévu, pour rattraper les erreurs potentielles (ici uniquement une exception arithmétique). Nous pouvons utiliser les méthodes `getMessage()` et `printStackTrace()` de l'exception pour afficher proprement la cause de l'erreur.

Cette fois, nous pouvons intervenir et contrôler l'affichage des messages d'erreurs.

```
Valeur décimale Infinity
Une fraction à diviseur nul a été manipulée
/ by zero
java.lang.ArithmeticException: / by zero
        at exceptions.Fraction.ValeurEntiere(Fraction.java:34)
        at exceptions.Fraction.main(Fraction.java:44)
```

7.2 Types d'erreurs

Toutes les erreurs dérivent d'une classe unique qui dérive directement d'`Object` : la classe `Throwable`, qui est la seule dont les instances (directes ou dérivées) peuvent être lancées par l'instruction `throw` ou relancées par la directive `throws`.



La classe `Error` permet de gérer des erreurs exceptionnelles et généralement fatales : problèmes dans la machine virtuelle, incidents majeurs provoquant généralement la mort de l'application. Elles ne sont généralement pas attrapées, parce que cela n'est pas utile. On peut en fait les ignorer lors de la programmation, parce qu'elles ne doivent normalement pas se produire. Leur seul intérêt est de provoquer des affichages sur le canal d'erreur, utiles pour comprendre la source du crash.

De nombreuses classes dérivent d'une deuxième fille de la classe de base : `Exception`. Elles sont trop nombreuses pour être énumérées. Viennent s'y ajouter des exceptions prévues par des extensions à l'API de base de Java. De nombreuses exceptions dérivent de la classe `RuntimeException`. À la différence des autres *throwables*, il n'est pas obligatoire de prévoir du code pour les contrôler (elles sont dites non-contrôlées ou `unchecked`). Les autres exceptions déclenchent automatiquement une erreur de compilation si on ne prévoit pas du code pour les contrôler. Ce point va être développé dans la section suivante.

7.3 Lancement, relancement et traitement des exceptions

Les descendants de `Throwable` sont des classes dont les instances possèdent un caractéristique originale : elles peuvent être « levées » par le biais d'une instruction originale, nommée `throw`. Le fait de lever une exception a une conséquence immédiate : le code en cours d'exécution est immédiatement interrompu, son contexte est perdu, toutes les méthodes en cours d'exécution sont oubliées jusqu'à celle qui contient la dernière instruction `try` rencontrée. Si aucun `try` n'a été rencontré, le thread en cours meurt. S'il y a une interface graphique, celle-ci peut continuer à s'exécuter. S'il y a eu un `try`, toutes les instructions qui suivent dans le bloc sont ignorées, le contexte du bloc est perdu et l'exécution reprend par l'examen du ou des blocs commençant par `catch`. Le code contenu dans le premier bloc mentionnant une exception du même type que l'exception levée, ou l'un de ses ancêtres, est ensuite exécuté. Si

les blocs de traitement des exceptions sont suivi d'un bloc `finally`, son code sera également exécuté. Il faut noter que le code de `finally` est toujours exécuté, sauf lancement d'une nouvelle exception, quel que soit le moyen par lequel on quitte le bloc `try`. Ce bloc est en principe destiné à libérer des ressources bloquées par les instructions qui précèdent (ouverture de fichier, de canaux réseau...).

En principe, une exception doit être traitée à l'endroit où elle est susceptible de survenir. Par exemple, la méthode suivante peut provoquer le lancement de plusieurs exceptions.

```
// méthode susceptible de provoquer une exception.
private int convertInt(JTextField t){
   return new Scanner(t.getText()).nextInt();
}
```

Elle est censée convertir le contenu d'une zone de texte en un entier, en faisant appel à une instance de `Scanner` et à sa méthode `nextInt()`. Comme le texte peut contenir des tas de caractères inattendus, cette méthode peut provoquer des exceptions. Pour éviter l'erreur de compilation, on peut choisir de relancer l'exception vers la méthode qui a fait appel à `convertInt()` ou traiter l'erreur localement.

```
// traitement local de l'exception
private int convertInt(JTextField t){
   try{
      return new Scanner(t.getText()).nextInt();
   }
   catch(Exception ex) {
      // en cas de problème on renvoie 0
      return 0;
}
```

Cette manière de procéder masque totalement le fait qu'il y a eu un problème. La méthode qui a fait appel à `convertInt()` ne pourra pas distinguer un 0 tapé par l'utilisateur dans une zone de texte d'un 0 obtenu suite à une erreur.

```
// renvoi de l'exception
private int convertInt(JTextField t) throws RuntimeException{
      return new Scanner(t.getText()).nextInt();
}
```

Avec cette nouvelle version, il appartient à la méthode ayant fait appel à `convertInt()` de traiter les éventuelles exceptions. Ce traitement n'est pas obligatoire, parce que nous avons relancé les exceptions de type *runtime*. En choisissant de relancer une exception générique, nous aurions forcé la méthode appelante à gérer l'exception.

Que choisir : exception contrôlée ou non ? traitement local ou lointain ? Il n'y a évidemment pas de réponse universelle. Une exception non contrôlée risque de rendre le programme instable faute d'un traitement approprié de l'erreur. Un trop grand nombre d'exceptions contrôlées vont obliger le programmeur à multiplier les blocs `try/catch` à tout bout de champ. Une exception traitée localement oblige à une réaction immédiate (le renvoi de 0 dans notre exemple), parfois sans tenir compte d'un contexte plus général. Un traitement différé peut au contraire manquer d'informations précises sur la nature exacte de l'incident survenu.

7.4 Philosophie de l'erreur et retour sur l'exemple initial

Le mécanisme des exceptions présente de nombreux avantages :

- il permet de distinguer clairement le traitement normal du traitement de l'erreur en plaçant le code de gestion de cette dernière à un endroit séparé
- en forçant une réflexion sur les erreurs possibles, il oblige le programmeur à les anticiper et à réfléchir sur leur cause réelle.

Si nous revenons à notre exemple de nombre rationnel, nous prendrons conscience qu'au lieu de traiter la division par zéro, il convient de se demander comment elle a pu se produite. L'erreur consiste, non à appeler une méthode renvoyant un valeur entière approchant le nombre rationnel, mais à créer un nombre rationnel dont le dénominateur vaut 0. C'est donc dans le créateur qu'on va gérer le problème. Au lieu de subir l'erreur et de tenter de réagir quand elle se produit, nous allons lever une exception lorsqu'on tente de créer un nombre rationnel aberrant, ce qui permettra de gérer l'erreur à sa source.

```
public class FractionAvecException {

    private int numerateur;
    private int denominateur;

    /** Creates a new instance of Fraction */
    public FractionAvecException(int num,int denom) throws
       InvalidParameterException{
        if(denom==0)
            throw new InvalidParameterException(
             "Le dénominateur d'une fraction ne peut être nul");
        numerateur = num;
        denominateur = denom;
    }

    public double ValeurDecimale(){
        return (double)numerateur/denominateur;
    }

     public long ValeurEntiere(){
        return numerateur/denominateur;
    }
    public static void main(String[] args){
        try{
            FractionAvecException fraction
               =new FractionAvecException(1,0);
            System.out.println("Valeur décimale "
               +fraction.ValeurDecimale());
            System.out.println("Valeur entière"
               +fraction.ValeurEntiere());
        }
        catch(InvalidParameterException ex)
        {
            System.err.println(ex.getMessage());
        }
    }
```

```
}
```

7.5 Définition d'une classe d'exception

La définition d'une exception n'est pas toujours nécessaire car il existe de nombreuses exceptions préalablement définies. Pour en obtenir la liste, on pourra consulter la JavaDoc des classes `Exception` et `RuntimeException`. Chaque classe d'exception possède notamment un constructeur permettant d'initialiser un message que le traitement de l'exception permettra de tester ou d'afficher. Voici un exemple d'une méthode dont on veut vérifier les deux paramètres :

```
public int Operation(int para1,int para2){
if(para1<0)
  throw new InvalidParameterException(
    "Le premier paramètre ne peut être négatif");
if(para2<1 && para2>10)
  throw new InvalidParameterException(
    "Le second paramètre doit être compris entre 1 et 10");
/* suite du traitement */
...
}
```

On peut aussi dériver sa propre classe d'une exception judicieusement choisie[1]. On veillera à lui donner au moins deux constructeurs (l'un avec message et l'autre sans). On peut ajouter autant de méthodes que nécessaire, par exemple pour récolter des informations sur le contexte de l'erreur.

Voici deux exceptions créées comme classes internes d'une classe de test. L'une est contrôlée, l'autre pas. On verra comment les méthodes `boumControle()` et `boumNonControle()` intègrent ces exceptions. On constate que l'appel de `boumNonControle` provoque un arrêt du programme.

```
package exceptions;
public class TestException{

    public TestException() {
    }

     public void boumControle() throws ExceptionControlee{
        throw new ExceptionControlee("Tout va mal");

    }

    public void boumNonControle(){
        throw new ExceptionNonControlee("Rien ne va plus");
    }

    public static void main(String[] arg){
        TestException te = new TestException();
```

[1] Au moins `Exception` ou `RuntimeException`, mais un choix plus précis facilitera le groupement de son traitement avec celui d'autres exceptions apparentées dans un `catch`.

```
        // le bloc try est obligatoire
        try{
        te.boumControle();
        System.out.println("Tout s'est bien passé");
        }
        catch(ExceptionControlee ex)
        {
            System.err.println(ex.getMessage());
        }
        // pas nécessaire de prévoir un traitement de l'erreur
        te.boumNonControle();

    }


    // Définition d'une classe d'exception non controlée
    private class ExceptionNonControlee extends RuntimeException{

        public ExceptionNonControlee(String message){
            super(message);
        }

    }

    // Définition d'une classe d'exception contrôlée
    private class ExceptionControlee extends Exception{
        public ExceptionControlee(String message){
            super(message);
        }
    }
}
```

7.6 Remarque importante

Une exception n'est pas assimilable à une transaction. Lors de la levée de l'exception, nous avons vu que le contexte de la méthode en cours d'exécution et de toutes celles qui la séparent du dernier bloc `try` vont disparaître. Les objets en cours de création avortent, les variables locales s'évanouissent. Il faut néanmoins se rappeler que tous les objets créés ou modifiés qui possèdent des références extérieures au contexte dans lequel l'erreur se produit subsisteront ou maintiendront leurs modifications, à moins que le bloc prévu pour traiter l'exception ne corrige la situation en supprimant les objets créés ou en rétablissant les valeurs initiales.