Les exceptions

Au niveau d'un programme, ou d'une application Java, certaines circonstances exceptionnelles peuvent inhiber la poursuite de son exécution. Pour assurer le suivi et le contrôle nécessaire de ces erreurs, Java utilise le concept des exceptions.

C'est une approche, différente de celle traditionnelle, qui peut se résumer par le fait de séparer le code du traitement souhaité par rapport au code gérant l'erreur survenue. Il faut préciser que Java distingue entre les erreurs (problèmes graves ou erreurs fatales survenues : crash de disque, non disponibilité de mémoire, etc.) et les exceptions (problèmes plutôt proches de l'application : débordement d'un tableau, division par zéro, une tentative de conversion d'une mauvaise donnée, etc.). Ce chapitre présente en premier lieu une description du concept d'exception et son utilité. Ensuite on décrira la gestion des exceptions en précisant la structure syntaxique à utiliser, la prise en charge d'une exception particulière, la génération (provocation) d'une exception et la possibilité de sa propagation . On verra par la suite l'hiérarchie des classes d'exception et on terminera avec les exceptions personnalisées qui sont simplement des classes créées par le développeur.

1 Présentation

Il doit être toujours envisageable, même si le programme développé a été minutieusement conçu et implémenté, que certains événements se manifestent pour compromettre la fin souhaitée de son exécution. Ces événements peuvent être de différents types : on peut avoir un cas de données incorrectes ou un cas de rencontre d'une fin prématurée de fichier (alors qu'il y a un besoin de données supplémentaires), etc.

Evidemment, dans l'approche classique, il est possible de tenter de prévoir toutes les situations possibles au sein d'un programme et préparer les bonnes réactions. Le problème est que, mis à part le fait que le développeur du programme risque d'ignorer ou omettre certaines situations, cette approche peut devenir fastidieuse et le code source un peu complexe.

Les exceptions sont justement des événements qui peuvent se déclencher lors de l'exécution d'un programme et qui arrêtent le flux normal des instructions.

Les exceptions sont principalement utilisées pour représenter des problèmes d'exécution de différentes catégories : des erreurs matérielles (crash du disque, ...), des erreurs de programmation (indice d'un tableau hors limites, ...), des erreurs liées à l'environnement d'exécution (mémoire insuffisante, ...), des erreurs spécifiques à une application (numéro d'article non-défini), etc.

En effet, pour faire face à ces situations Java offre une technique très souple appelée *gestion d'exception*, permettant d'assurer deux aspects:

- distinguer la détection d'une anomalie de son traitement ;
- isoler la gestion des anomalies du reste du code source, donc aider à assurer la lisibilité des applications.

2 Utilité des exceptions

L'utilisation des exceptions, comme déjà indiqué, augmente la lisibilité du code en séparant les instructions, qui traitent le cas normal des actions nécessaires au traitement, par rapport aux erreurs et événements exceptionnels. En effet, elles permettent (voire même imposent) la déclaration explicite des exceptions qu'une méthode peut lever.

Les avantages d'utilisation des exceptions peuvent se résumer par :

- forcer le programmeur à prendre en compte (traiter ou propager) les cas exceptionnels (erreurs ou autres événements) qui sont déclarés dans chaque méthode qu'il invoque ;
- permettre de créer une hiérarchie d'événements et de les traiter avec une granularité différente selon les situations ;
- offrir un mécanisme de propagation automatique ce qui permet au programmeur de choisir à quel niveau il souhaite traiter l'exception.

3 Gestion des exceptions

3.1 Prendre en charge une exception

En Java, les exceptions sont représentées par des objets de type *Throwable*. Il existe plusieurs familles d'exceptions représentées par différentes sous-classes de *Throwable* (par exemple : *Exception*).

La mise en œuvre de la gestion des exceptions peut être exprimée par l'instruction *throw* qui sert à générer une exception (on dit également lever une exception, lancer une exception, …) et aussi par l'instruction *try / catch / finally* qui constitue le cadre dans lequel les exceptions peuvent être détectées (capturées) et traitées.

D'autre part le mot-clé *throws* (à ne pas confondre avec *throw*) est utilisé dans la déclaration de la méthode (signature) pour annoncer la liste des exceptions qu'elle peut générer. L'utilisateur de la méthode est ainsi informé des exceptions qui peuvent survenir lors de son invocation et peut prendre les mesures nécessaires pour gérer ces événements exceptionnels (c'est-à-dire les traiter ou les propager). Pour le cas de l'instruction *throw* elle permet, plutôt, de provoquer l'exception indiquée.

3.2 Générer une exception

Les exceptions peuvent être générées :

- soit par le **système**, lors de l'exécution de certaines instructions ;
- soit par l'exécution de l'instruction **throw**(qui est aussi utilisée dans les classes prédéfinies de la plateforme Java ou dans le code que l'on écrit soi-même).

Parmi les instructions qui génèrent des exceptions, on peut citer :

- la division entière par zéro qui génère :
  ArithmeticException
- l'indexation d'un tableau hors de ses limites qui génère :
  ArrayIndexOutOfBoundsException
- l'utilisation d'un tableau ou objet dont la référence vaut **null** qui génère :
  NullPointerException
- la création d'un tableau avec une taille négative qui génère :
  NegativeArraySizeException
- la conversion d'un objet dans un type non compatible qui génère :
  ClassCastException

Pour générer explicitement une exception on utilise le mot-clé *throw* : *throw <Exception_Object> ;*
L'expression qui suit l'instruction *throw* doit être un objet qui représente une exception (un objet de type *Throwable*). Lors de la création de l'objet exception, il est généralement possible de lui associer un message (*String*) qui décrit l'événement.

Exemple :

```
public static long factorielle(int x) throws Exception {
   long result = 1;
   if (x<0)
      throw new Exception("x doit être >= 0");
   while (x>=0) {
      result *= x ;
      x-- ;
   }
   return result ;
}
```

3.3 Traiter une exception

Les instructions susceptibles de lever des exceptions peuvent être insérées dans un bloc **try** / **catch / finally** qui se présente de la manière suivante :

```
try {
          // Bloc contenant des instructions pouvant
          // générer des exceptions.
}
catch (ExceptionType1  e1) {
          // Bloc contenant les instructions qui
          // traitent les exceptions
          // du type ExceptionType1(ou d'une de ses
          // sous-classes)
          // On peut référencer l'objet exception à
          // l'aide de la variable e1.
}
catch (ExceptionType2  e2) {
          // Bloc contenant les instructions qui
          // traitent les exceptions
          // du type ExceptionType2(ou d'une de ses
          // sous-classes). L'objet est référencé par e2.
}
catch (…) {
      …  //  Et ainsi de suite…
}
[ finally{
      . . .  }  ]
```

Si une des instructions contenues dans le bloc *try*, lève une exception, le contrôle est passé au premier bloc *catch* dont le type d'exception correspond à l'exception qui a été levée (même classe ou classe parente de l'exception levée).

Après l'exécution de la dernière instruction du bloc *catch* considéré, le contrôle est passé à l'instruction qui suit le dernier bloc *catch* (ou à la clause *finally* si elle existe).

Si aucun bloc *catch* ne correspond au type d'exception qui a été levée, l'exception est propagée au niveau supérieur c'est-à-dire que le contrôle est transféré au traitement d'exception de la méthode invoquante (appelante) ou du bloc englobant. Si aucun traitement n'existe au niveau supérieur pour ce type d'exception, la propagation se poursuit jusqu'à trouver un bloc catch traitant cette exception. Si ce n'est pas le cas, le programme se termine avec un message d'erreur sur la console de sortie.

Les divers types d'exceptions détectées peuvent appartenir à différentes familles et peuvent dériver d'une même famille. En effet, une exception spécialisée par exemple : *FileNotFoundException* fait partie d'une famille plus vaste : *IOException* qui, elle-même, fait partie d'une famille plus générale : *Exception* qui dérive de *Throwable*.

Si plusieurs clauses *catch* sont compatibles avec le type d'exception qui a été détectée (c-a-d font partie de la même famille), c'est la première qui capturera l'exception et se chargera du traitement.

Exemple :

```
try {
    ...
}
catch ( FileNotFoundException   notFound) {
    ...
}
catch ( IOException   ioErr) {
    ...
}
catch ( Exception   genErr) {
    ...
}
```

Lorsqu'un problème est détecté (une exception est générée) et que l'on est en mesure de le traiter, il y a différentes mesures que l'on peut envisager, selon le cas, pour régler la situation. En effet, dans un traitement d'exception (dans le bloc *catch*) on peut par exemple :

- régler le problème et recommencer le traitement ;
- faire quelque chose d'autre à la place (algorithme de substitution) ;
- sortir de l'application (*System.exit()*) après affichage d'un message ou écriture dans un fichier *log*, etc.
- regénérer l'exception (pour procéder, éventuellement, à sa propagation à un niveau supérieur) ;

- générer une nouvelle exception (après avoir éventuellement effectué certaines opérations) ;
- retourner une valeur spéciale ou valeur par défaut (pour une fonction) ;
- terminer la méthode (si elle n'a pas de valeur de retour) ;
- ne rien faire (ou afficher un message) et continuer (c'est rarement une bonne solution).

3.4 Propager une exception

Comme il a été présenté dans un paragraphe précédent l'exception est propagée dans le cas où aucun bloc *catch* ne correspond au type d'exception levée. Le contrôle sera donc transféré au bloc invoquant et la propagation peut se poursuivre jusqu'à trouver un bloc *catch* traitant cette exception sinon l'interruption d'exécution avec un message d'erreur sur la console de sortie devient inévitable.

Essayons d'exécuter manuellement l'exemple suivant :

```
public class PropEx {

   public static void main(String[] args) {
      System.out.println("main commence");
      b(5);
      b(0);
      System.out.println("main termine");
   }
//-----------------------------------------------
//                       b
//-----------------------------------------------
    public static void b(int i) {
      System.out.println("b commence");
      try {
         System.out.println("b étape 1");
         c(i);
         System.out.println("b étape 2");
      }
      catch (Exception e) {
         System.out.println("b intercepte " + e);
      }
      System.out.println("b termine ");
   }
//-----------------------------------------------
//                       c
//-----------------------------------------------
   public static void c(int i) throws Exception {
      System.out.println("c commence");
      d(i);
      System.out.println("c termine");
   }
//-----------------------------------------------
//                       d
//-----------------------------------------------
   public static void d(int i) throws Exception {
     System.out.println("d commence");
     int a=10/i; // Cette instruction peut générer une
                 // exception
     System.out.println("d termine");
   }
}
```

Dans le programme *PropEx*, lors de la deuxième invocation de la méthode *b()*, une exception sera levée dans la méthode *d()* (division par zéro).

Le déroulement de l'application sera alors altéré et l'exception sera propagée jusqu'à la méthode *b()* qui dispose d'un bloc *catch* pour traiter cette exception.

b(5) 1ère invocation	b(0) 2ème invocation
main commence b commence b étape 1 c commence d commence d termine c termine b étape 2 b termine	b commence b étape 1 c commence d commence b intercepte ArithmeticException : / by zero b termine main termine

3.5 La clause finally

Une clause *finally* peut être ajoutée à une instruction *try / catch***.** Cette clause définit un bloc d'instructions qui sera exécuté à la fin de l'instruction *try/ catch* indépendamment du fait que des exceptions ont été levées ou non. Le bloc *finally* sera donc exécuté dans tous les cas.

En fait on a deux situations possibles :

- Si aucune exception n'est levée dans le bloc *try*, le bloc *finally* sera exécuté après la dernière instruction du bloc *try* ;
- Si une exception est levée dans le bloc *try* et est capturée par un bloc *catch*, le bloc *finally* sera exécuté après la dernière instruction du bloc *catch*. Si cette exception n'est pas capturée par un bloc *catch*, le bloc *finally* sera exécuté avant la propagation de l'exception au niveau supérieur.

Un bloc *finally* est généralement utilisé pour effectuer des opérations de conclusion (fermeture de fichiers, de connexion réseau, de base de données, etc.) qui devraient être effectuées dans tous les cas de figure. Le bloc *finally*évite donc de devoir placer ces instructions de conclusion dans le bloc *try* et dans tous les blocs *catch*.

Remarques

1) L'utilisation des instructions *break*, *continue*, *return* ou *throw* (dans les blocs *try* ou *catch*) n'empêche pas l'exécution préalable du bloc *finally* ;

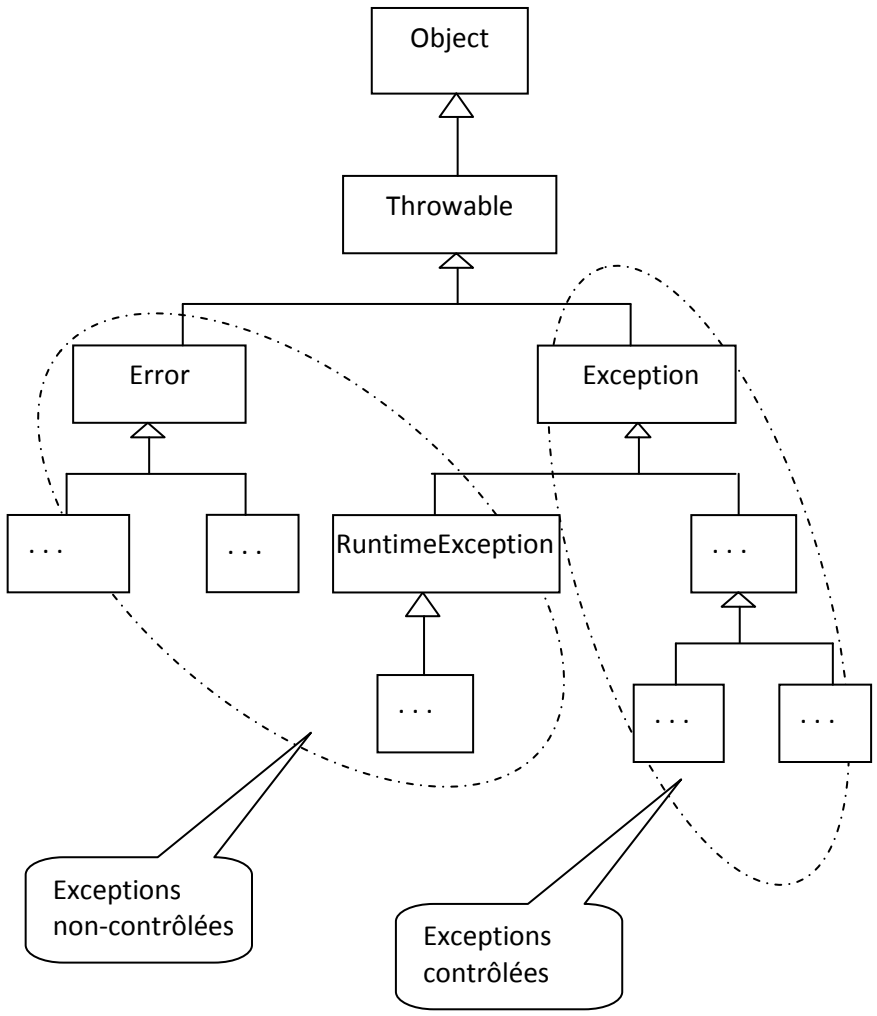
2) Le bloc *finally* est optionnel mais un bloc *try* doit être accompagné, obligatoirement, d'au moins un bloc *catch* ou d'un bloc *finally* (ou naturellement des deux).

Exemple d'utilisation de la clause finally :

```
try {
    ouvertureFichier(f);
    contenu = lectureFichier(f);
    impression(contenu);
}

catch (InvalidData  invdat) {
    System.out.println("Les données du fichier sont
            erronées");
}

catch (PrintError  prerr) {
    System.out.println("Erreur durant l'impression");
}

finally {
    fermetureFichier(f); //- Effectué dans tous les cas
}
```

4 Hiérarchie des classes

Dans le langage Java, toutes les exceptions sont représentées par des objets qui spécialisent la classe de base *Throwable*. On peut, par exemple, voir les trois sous-classes *Error*, *Exception*et *RuntimeException* qui dérivent de *Throwable* et qui sont décrits selon l'arborescence suivante :



Les exceptions peuvent être donc classées en deux catégories :

- les exceptions contrôlées sont celles qui dérivent de la classe *Exception* (en dehors des exceptions dérivant de *RuntimeException*). Elles doivent être déclarées dans la signature de méthode ;
- les exceptions non contrôlées sont celles dérivant des classes *Error* et *RuntimeException*. Elles n'ont pas besoin d'être déclarées dans la signature de la méthode.

5 Exceptions personnalisées

Pour créer ses propres types d'exceptions, il suffit de dériver (spécialiser) une classe de type *Throwable*. En général on choisit un nom de classe personnalisé mais qui doit hériter de la classe *Exception* (qui dérive elle même de *Throwable*).

Généralement, dans cette sous-classe, on crée uniquement deux constructeurs : un constructeur sans paramètre et un constructeur qui prend un message (String) en paramètre. On crée rarement de nouveaux attributs ou de nouvelles méthodes.

Exemple :

```
public class ClockException extends Exception {

    public ClockException() {
       super();
    }

    public ClockException(String message) {
       super(message);
    }
}
```

L'utilisation des types d'exceptions que l'on a créés est identique à l'utilisation des types d'exceptions prédéfinis.

Exemple :

```
. . .
if (heure < 0 || heure > 23) {
    throw new ClockException("Heure incorrect ! ");
}
. . .


. . .
try {
    . . .
}
catch (ClockException horlogeErr) {
    System.out.println(horlogeErr);
    . . .
}
catch (Exception autresErr) {
    . . .
}
```

Exercices d'application

Exercice 1

Soit le programme suivant :

```
class Erreur extends Exception
{public int num; }
class Erreur_d extends Erreur
{public int code; }

class A {
   public A(int n) throws Erreur_d {
      if (n==1){
         Erreur_d e = new Erreur_d();
         e.num=999; e.code=12;
         throw e;
      }
   }
}
public class Chemin{
   public static void main(String [] args){
      try {
         A a = new A(1);
         System.out.println("Après création a(1)");
      }
      catch (Erreur e){
         System.out.println(" *** exception Erreur " +
                     e.num);
      }
      System.out.println("Suite de main");
      try {
          A b = new A(1);
           System.out.println("Après création b(1)");
      }
      catch (Erreur_d e) {
         System.out.println(" *** exception Erreur_d " +
                e.num+ "  "+e.code);
      }
      catch (Erreur e){
         System.out.println(" *** exception Erreur " +
                e.num);
       }
  }
}
```

Questions :

Avant de compiler et d'exécuter ce programme essayer de prévoir :

1) Que fournit le programme suivant ?

2) Que se passe-t-il si l'on inverse l'ordre des deux gestionnaires d'exception (les *catch*) dans le second bloc *try* ?

Exercice 2

Soit le programme suivant :

```
class Except extends Exception {  }
public class FinReThrow {
```

```
   public static void f (int n ) throws Except{
      try{
         if (n!=1) throw new Except();
      }
      catch (Except e){
         System.out.println ("catch dans f  avec n = "
                           +n);
         throw e;
       }
      finally {
         System.out.println("dans finally de f avec n = "
                           +n);
       }
  }
  public static void main (String [] args){
     int n = 0;
     try {
       for (n=1; n<5; n++)
          f(n);
     }
     catch (Except e){
       System.out.println("catch dans main avec n = "
                           +n);
     }
     finally {
         System.out.println("dans finally de main avec n
                           = " +n);
     }
  }
}
```

Question :
Avant de compiler et d'exécuter ce programme, quel est le résultat prévu par ce programme ?

Exercice 3

Soit le programme Temps suivant :

```
public class Temps {
      private int heures;
      private int minutes;
      private int secondes;
      public Temps (int h, int m, int s){
            heures = h;
            minutes = m;
            secondes = s;
      }
      public static void main(String args[]){
            int h,m,s;
            // saisie des valeurs de l'heure, les
            // minutes et les secondes représentées
            // par les entiers h, m, s
            h = . . . . ;
            m = . . . . ;
            s = . . . . ;
            Temps t = new Temps (h, m, s);
      }
}
```

Questions :

1) Modifier le constructeur de la classe *Temps* de manière à ce qu'il lance une exception de type *TempsException* si les heures, les minutes ou les secondes ne correspondent pas à un temps valide. La classe *TempsException* est à définir.

2) Modifier le code de la méthode *main()* de manière à ce que l'exception *TempsException* soit traitée en affichant le message suivant : "Temps Invalide".

Exercice 4

Créer une classe permettant de manipuler des entiers naturels (positifs ou nuls) et disposant :

- d'un constructeur à un argument de type *int*, il générera une exception *ErrConst* si la valeur de son argument est négative ;
- des méthodes statiques de somme, de différence et de produit de deux naturels. Elles généreront respectivement des exceptions *ErrSom, ErrDiff* et *ErrProd* lorsque le résultat ne sera pas représentable; la limite des valeurs des naturels sera fixée à la plus grande valeur du type *int ;*
- une méthode d'accès *getN* fournissant sous forme d'un int la valeur de l'entier naturel.

On s'arrangera pour que toutes les classes d'exception dérivent d'une classe *ErrNat* et pour qu'elles permettent à un éventuel gestionnaire de récupérer les valeurs ayant provoquées l'exception.

Ecrire ensuite deux exemples d'utilisation de la classe :

- l'un se contentant d'intercepter sans discernement les exceptions de type dérivé de *ErrNat ;*
- l'autre qui explicite la nature de l'exception en affichant les informations disponibles.

Les deux exemples pourront figurer dans deux blocs try d'un même programme.

Exercice 5

Nous disposons d'un parking permettant de garer des voitures. Les voitures seront stationnées sur des places numérotées en partant de 0. Le nombre de places est fixé, une fois pour toutes, à la construction du parking. On supposera qu'il existe une classe Voiture déjà programmée, stockée dans un package *véhicule*, et englobant la méthode *toString* qui a été redéfinie.

1) Ecrire la classe *Parking* et ses constructeurs.
2) Ecrire une méthode *garer* qui prend en paramètres une voiture, un numéro de place, et qui gare la voiture à la place donnée si elle est libre, et lève une exception de type *PlaceNonLibreException*.
3) Ecrire une méthode *sortir* qui prend en paramètre un numéro de place et qui retire du garage la voiture à cette place. La méthode retourne l'objet Voiture récupéré. Elle lève une exception de type *PlaceLibreException* si la place est vide.
4) Ajouter une exception *HorsParkingException* declenchée lorsqu'un numéro de place n'est pas valide.
5) Ecrire une méthode *toString* affichant l'état du garage : pour chaque place, le numéro, ainsi que les caractéristiques de la voiture qui l'occupe si elle existe.