Claude Delannoy

Programmer en Java

3e édition

ISBN : 2-212-11501-6

EYROLLES

10

La gestion des exceptions

Même lorsqu'un programme est au point, certaines circonstances exceptionnelles peuvent compromettre la poursuite de son exécution ; il peut s'agir par exemple de données incorrectes ou de la rencontre d'une fin de fichier prématurée (alors qu'on a besoin d'informations supplémentaires pour continuer le traitement).

Bien entendu, on peut toujours essayer d'examiner toutes les situations possibles au sein du programme et prendre les décisions qui s'imposent. Mais outre le fait que le concepteur du programme risque d'omettre certaines situations, la démarche peut devenir très vite fastidieuse et les codes quelque peu complexes. Le programme peut être rendu quasiment illisible si sa tâche principale est masquée par de nombreuses instructions de traitement de circonstances exceptionnelles.

Par ailleurs, dans des programmes relativement importants, il est fréquent que le traitement d'une anomalie ne puisse pas être fait par la méthode l'ayant détectée, mais seulement par une méthode ayant provoqué son appel. Cette dissociation entre la détection d'une anomalie et son traitement peut obliger le concepteur à utiliser des valeurs de retour de méthode servant de "compte rendu". Là encore, le programme peut très vite devenir complexe ; de plus, la démarche ne peut pas s'appliquer à des méthodes sans valeur de retour donc, en particulier, aux constructeurs.

La situation peut encore empirer lorsque l'on développe des classes réutilisables destinées à être exploitées par de nombreux programmes.

Java dispose d'un mécanisme très souple nommé *gestion d'exception*, qui permet à la fois :

- de dissocier la détection d'une anomalie de son traitement,

- de séparer la gestion des anomalies du reste du code, donc de contribuer à la lisibilité des programmes.

D'une manière générale, une exception est une rupture de séquence déclenchée par une instruction *throw* comportant une expression de type classe. Il y a alors branchement à un ensemble d'instructions nommé "gestionnaire d'exception". Le choix du bon gestionnaire est fait en fonction du type de l'objet mentionné à *throw* (de façon comparable au choix d'une fonction surdéfinie).

1 Premier exemple d'exception

1.1 Comment déclencher une exception avec throw

Considérons une classe *Point*, munie d'un constructeur à deux arguments et d'une méthode *affiche*. Supposons que l'on ne souhaite manipuler que des points ayant des coordonnées non négatives. Nous pouvons, au sein du constructeur, vérifier la validité des paramètres fournis. Lorsque l'un d'entre eux est incorrect, nous "déclenchons[1]" une exception à l'aide de l'instruction *throw*. A celle-ci, nous devons fournir un objet dont le type servira ultérieurement à identifier l'exception concernée. Nous créons donc (un peu artificiellement) une classe que nous nommerons *ErrCoord*. Java impose que cette classe dérive de la classe standard *Exception*. Pour l'instant, nous n'y plaçons aucun membre (mais nous le ferons dans d'autres exemples) :

```
class ErrConst extends Exception
{ }
```

Pour lancer une exception de ce type au sein de notre constructeur, nous fournirons à l'instruction *throw* un objet de type *Errconst*, par exemple de cette façon :

```
throw new ErrConst() ;
```

En définitive, le constructeur de notre classe *Point* peut se présenter ainsi :

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0)) throw new ErrConst() ;  // lance une exception de
    this.x = x ; this.y = y ;                    //  type ErrConst
  }
```

Notez la présence de *throws ErrConst*, dans l'en-tête du constructeur, qui précise que la méthode est susceptible de déclencher une exception de type *ErrConst*. Cette indication est obligatoire en Java, à partir du moment où l'exception en question n'est pas traitée par la méthode elle-même.

En résumé, voici la définition complète de nos classes *Point* et *ErrCoord* :

1. On emploie aussi les verbes "lancer" ou "lever".

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0)) throw new ErrConst() ;
    this.x = x ; this.y = y ;
  }
  public void affiche()
  { System.out.println ("coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
class ErrConst extends Exception
{ }
```

Exemple d'une classe Point *dont le constructeur déclenche une exception* ErrConst

1.2 Utilisation d'un gestionnaire d'exception

Disposant de notre classe *Point*, voyons maintenant comment procéder pour gérer convenablement les éventuelles exceptions de type *ErrConst* que son emploi peut déclencher. Pour ce faire, il faut :

- inclure dans un bloc particulier dit "bloc *try*" les instructions dans lesquelles on risque de voir déclenchée une telle exception ; un tel bloc se présente ainsi :

```
try
{
   // instructions
}
```

- faire suivre ce bloc de la définition des différents gestionnaires d'exception (ici, un seul suffit). Chaque définition de gestionnaire est précédée d'un en-tête introduit par le mot clé *catch* (comme si *catch* était le nom d'une méthode gestionnaire). Voici ce que pourrait être notre unique gestionnaire :

```
catch (ErrConst e)
{ System.out.println ("Erreur construction ") ;
  System.exit (-1) ;
}
```

Ici, il se contente d'afficher un message et d'interrompre l'exécution du programme en appelant la méthode standard *System.exit* (la valeur de l'argument est transmis à l'environnement qui peut éventuellement l'utiliser comme "compte rendu").

1.3 Le programme complet

A titre indicatif, voici la liste complète de la définition de nos classes, accompagnée d'un petit programme de test dans lequel nous provoquons volontairement une exception :

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0)) throw new ErrConst() ;
    this.x = x ; this.y = y ;
  }
  public void affiche()
  { System.out.println ("coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
class ErrConst extends Exception
{ }
public class Except1
{ public static void main (String args[])
  { try
    { Point a = new Point (1, 4) ;
      a.affiche() ;
      a = new Point (-3, 5) ;
      a.affiche() ;
    }
    catch (ErrConst e)
    { System.out.println ("Erreur construction ") ;
      System.exit (-1) ;
    }
  }
}
```

```
coordonnees : 1 4
Erreur construction
```

Premier exemple de gestion d'exception

Remarque

Avec certains environnements, la fenêtre console disparaît dès la fin du programme. Pour éviter ce désagrément, vous avez probablement pris l'habitude d'ajouter une instruction telle que *Clavier.lireInt()* à la fin de votre programme. Cette fois ici, cette précaution ne suffit plus ; il faut intervenir dans le gestionnaire d'exception en ajoutant une telle instruction d'attente, avant l'appel de *exit*.

1.4 Premières propriétés de la gestion d'exception

Ce premier exemple était très restrictif pour différentes raisons :

- on n'y déclenchait et on n'y traitait qu'un seul type exception ; nous verrons bientôt comment en gérer plusieurs ;

- le gestionnaire d'exception ne recevait aucune information ; plus exactement, il recevait un objet sans valeur qu'il ne cherchait pas à utiliser ; nous verrons comment utiliser cet objet pour communiquer une information au gestionnaire ;
- nous n'exploitions pas les fonctionnalités de la classe *Exception* dont dérive notre classe *ErrCoord* ;
- le gestionnaire d'exception se contentait d'interrompre le programme ; nous verrons qu'il est possible de poursuivre l'exécution.

On peut dores et déjà noter que le gestionnaire d'exception est défini indépendamment des méthodes susceptibles de la déclencher. Ainsi, à partir du moment où la définition d'une classe est séparée de son utilisation (ce qui est souvent le cas en pratique), il est tout à fait possible de prévoir un gestionnaire différent d'une utilisation à une autre de la même classe. Dans l'exemple précédent, tel utilisateur peut vouloir afficher un message avant de s'interrompre, tel autre préférera ne rien afficher ou encore tenter de trouver une solution par défaut...

D'autre part, le bloc *try* et les gestionnaires associés doivent être contigus. Cette construction est erronée :

```
try
{ .....
}
  .....
catch (ErrConst)    // erreur : catch doit être contigu au bloc try
{ ..... }
```

Enfin, dans notre exemple, le bloc *try* couvre toute la méthode *main*. Ce n'est nullement une obligation et il est même théoriquement possible de placer plusieurs blocs *try* dans une même méthode[1] :

```
void truc()
{ .....
  try { .....          // ici, les exceptions ErrConst sont traitées
      }
  catch (ErrConst)
  { ..... }
  .....                // ici, elles ne le sont plus
  try { .....
      }
  catch (ErrConst)
  { ..... }            // ici, elles le sont de nouveau
  .....                // ici, elles ne le sont plus
}
```

1. Et même de les imbriquer !

En C++

C++ dispose d'un mécanisme de gestion des exceptions proche de celui de Java. On emploie aussi l'instruction *throw* dans un bloc *try* pour déclencher une exception qui sera traitée par un gestionnaire introduit par *catch*. Mais on peut lui fournir une expression d'un type quelconque (pas nécessairement classe). Par ailleurs, il existe une clause *throw* jouant un rôle comparable à *throws*.

2 Gestion de plusieurs exceptions

Voyons maintenant un exemple plus complet dans lequel on peut déclencher et traiter deux types d'exceptions. Pour ce faire, nous considérons une classe *Point* munie :

- du constructeur précédent, déclenchant toujours une exception *ErrConst*,
- d'une méthode *deplace* qui s'assure que le déplacement ne conduit pas à une coordonnée négative ; si tel est le cas, elle déclenche une exception *ErrDepl* (on crée donc, ici encore, une classe *ErrDepl*) :

```
public void deplace (int dx, int dy) throws ErrDepl
{ if ( ((x+dx)<0) || ((y+dy)<0)) throw new ErrDepl() ;
  x += dx ; y += dy ;
}
```

Lors de l'utilisation de notre classe *Point*, nous pouvons détecter les deux exceptions potentielles *ErrConst* et *ErrDepl* en procédant ainsi (ici, nous nous contentons comme précédemment d'afficher un message et d'interrompre l'exécution) :

```
try
{   // bloc dans lequel on souhaite détecter les exceptions ErrConst et ErrDepl
}
catch (ErrConst e)      // gestionnaire de l'exception ErrConst
{ System.out.println ("Erreur construction ") ;
  System.exit (-1) ;
}
catch (ErrDepl e)       // gestionnaire de l'exception ErrDepl
{ System.out.println ("Erreur deplacement ") ;
  System.exit (-1) ;
}
```

Voici un exemple complet de programme dans lequel nous provoquons volontairement une exception *ErrDepl* ainsi qu'une exception *ErrConst* :

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0)) throw new ErrConst() ;
    this.x = x ; this.y = y ;
  }
```

```
 public void deplace (int dx, int dy) throws ErrDepl
  { if ( ((x+dx)<0) || ((y+dy)<0)) throw new ErrDepl() ;
    x += dx ; y += dy ;
  }
  public void affiche()
  { System.out.println ("coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
class ErrConst extends Exception
{ }
class ErrDepl extends Exception
{ }
public class Except2
{ public static void main (String args[])
  { try
    { Point a = new Point (1, 4) ;
      a.affiche() ;
      a.deplace (-3, 5) ;
      a = new Point (-3, 5) ;
      a.affiche() ;
    }
    catch (ErrConst e)
    { System.out.println ("Erreur construction ") ;
      System.exit (-1) ;
    }
    catch (ErrDepl e)
    { System.out.println ("Erreur deplacement ") ;
      System.exit (-1) ;
    }
  }
}
```

```
coordonnees : 1 4
Erreur deplacement
```

Exemple de gestion de deux exceptions

Bien entendu, comme la première exception (*ErrDepl*) provoque la sortie du bloc *try* (et, de surcroît, l'arrêt de l'exécution), nous n'avons aucune chance de mettre en évidence celle qu'aurait provoquée la tentative de construction d'un point par l'appel *new Point(-3, 5)*.

Remarque

La construction suivante serait rejetée par le compilateur :

```
public void static main (...)
{ try
  { Point a = new Point(2, 5) ; a.deplace (2, 5) ;
  }
```

```
  catch (ErrConst)
  { .....
  }
}
```

En effet, dès qu'une méthode est susceptible de déclencher une exception, celle-ci doit obligatoirement être traitée dans la méthode ou déclarée dans son en-tête (clause *throws*). Ainsi, même si l'appel de *deplace* ne pose pas de problème ici, nous devons soit prévoir un gestionnaire pour *ErrDepl*, soit indiquer *throws ErrDepl* dans l'en-tête de la méthode *main*. Nous reviendrons plus en détail sur ce point.

3 Transmission d'information au gestionnaire d'exception

On peut transmettre une information au gestionnaire d'exception :

- par le biais de l'objet fourni dans l'instruction *throw*,
- par l'intermédiaire du constructeur de l'objet exception.

3.1 Par l'objet fourni à l'instruction throw

Comme nous l'avons vu, l'objet fourni à l'instruction *throw* sert à choisir le bon gestionnaire d'exception. Mais il est aussi récupéré par le gestionnaire d'exception sous la forme d'un argument, de sorte qu'on peut l'utiliser pour transmettre une information. Il suffit pour cela de prévoir les champs appropriés dans la classe correspondante (on peut aussi y trouver des méthodes).

Voici une adaptation de l'exemple de programme du paragraphe 1, dans lequel nous dotons la classe *ErrConst* de champs *abs* et *ord* permettant de transmettre les coordonnées reçues par le constructeur de *Point*. Leurs valeurs sont fixées lors de la construction d'un objet de type *ErrConst* et elles sont récupérées directement par le gestionnaire (car les champs ont été prévus publics ici, pour simplifier les choses).

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0)) throw new ErrConst(x, y) ;
    this.x = x ; this.y = y ;
  }
  public void affiche()
  { System.out.println ("coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
```

```
class ErrConst extends Exception
{ ErrConst (int abs, int ord)
  { this.abs = abs ; this.ord = ord ;
  }
  public int abs, ord ;
}
public class Exinfo1
{ public static void main (String args[])
  { try
    { Point a = new Point (1, 4) ;
      a.affiche() ;
      a = new Point (-3, 5) ;
      a.affiche() ;
    }
    catch (ErrConst e)
    { System.out.println ("Erreur construction Point") ;
      System.out.println ("  coordonnees souhaitees : " + e.abs + " " + e.ord) ;
      System.exit (-1) ;
    }
  }
}
```

```
coordonnees : 1 4
Erreur construction Point
  coordonnees souhaitees : -3 5
```

Exemple de transmission d'information à un gestionnaire d'exception (1)

3.2 Par le constructeur de la classe exception

Dans certains cas, on peut se contenter de transmettre un "message" au gestionnaire, sous forme d'une information de type chaîne. La méthode précédente reste bien sûr utilisable, mais on peut aussi exploiter une particularité de la classe *Exception* (dont dérive obligatoirement votre classe). En effet, celle-ci dispose d'un constructeur à un argument de type *String* dont on peut récupérer la valeur à l'aide de la méthode *getMessage* (dont héritera votre classe).

Pour bénéficier de cette facilité, il suffit de prévoir dans la classe exception, un constructeur à un argument de type *String*, qu'on retransmettra au constructeur de la super-classe *Exception*. Voici une adaptation dans ce sens du programme précédent

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0))
     throw new ErrConst("Erreur construction avec coordonnees " + x + " " + y) ;
    this.x = x ; this.y = y ;
  }
```

```
  public void affiche()
  { System.out.println ("coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
class ErrConst extends Exception
{ ErrConst (String mes)
  { super(mes) ;
  }
}
public class Exinfo2
{ public static void main (String args[])
  { try
    { Point a = new Point (1, 4) ;
      a.affiche() ;
      a = new Point (-3, 5) ;
      a.affiche() ;
    }
    catch (ErrConst e)
    { System.out.println (e.getMessage()) ;
      System.exit (-1) ;
    }
  }
}
```

```
coordonnees : 1 4
Erreur construction avec coordonnees -3 5
```

Exemple de transmission d'information au gestionnaire d'exception (2)

Remarque

En pratique, on utilise surtout cette seconde méthode de transmission d'information pour identifier une exception par un bref message explicatif. Les éventuelles valeurs complémentaires seront plutôt fournies par l'objet lui-même, suivant la première méthode proposée.

4 Le mécanisme de gestion des exceptions

Plusieurs exemples vous ont permis de vous familiariser avec cette nouvelle notion d'exception, dans des situations relativement simples. Nous apportons ici un certain nombre de précisions concernant :

- la poursuite de l'exécution après le traitement d'une exception par le gestionnaire,
- le choix du gestionnaire,

- le cheminement des exceptions, c'est-à-dire la manière dont elles peuvent remonter d'une méthode à une méthode appelante,
- les règles d'écriture de la clause *throws*,
- les possibilités de redéclencher une exception,
- l'existence d'un bloc particulier dit *finally.*

4.1 Poursuite de l'exécution

Dans tous les exemples précédents, le gestionnaire d'exception mettait fin à l'exécution du programme en appelant la méthode *System.exit*. Cela n'est pas une obligation ; en fait, après l'exécution des instructions du gestionnaire, l'exécution se poursuit simplement avec les instructions suivant le bloc *try* (plus précisément, le dernier bloc *catch* associé à ce bloc *try*).

Observez cet exemple qui utilise les mêmes classes *ErrConst* et *ErrDepl* que l'exemple du paragraphe 2 :

```
// définition des classes Point, ErrConst et ErrDepl comme dans paragraphe 2
public class Suitex
{ public static void main (String args[])
  { System.out.println ("avant bloc try") ;
    try
    { Point a = new Point (1, 4) ;
      a.affiche() ;
      a.deplace (-3, 5) ;
      a.affiche() ;
    }
    catch (ErrConst e)
    { System.out.println ("Erreur construction ") ;
    }
    catch (ErrDepl e)
    { System.out.println ("Erreur deplacement ") ;
    }
    System.out.println ("apres bloc try") ;
  }
}
```

```
avant bloc try
coordonnees : 1 4
Erreur deplacement
apres bloc try
```

Lorsque l'exécution se poursuit après le gestionnaire d'exception

Ici, le bloc *try* ne couvre pas toute la méthode *main*. Mais souvent un bloc *try* couvrira toute une méthode, de sorte que, après traitement d'une exception par un gestionnaire ne provoquant pas d'arrêt, il y aura retour de ladite méthode.

Un gestionnaire d'exception peut aussi comporter une instruction *return*. Celle-ci provoque la sortie de la méthode concernée (et pas seulement la sortie du gestionnaire !). Voyez ce schéma (on suppose que *E1* est une classe dérivée de *Exception*) :

```
int f()
{ try
  { .....
  }
  catch (Exception E1 e)
  { .....
    return 0 ;              // OK ; en cas d'exception, on sort de f
  }
  .....                     // sans exécuter ces instructions
}
```

Informations complémentaires

En ce qui concerne la portée des identificateurs, les blocs *catch* ne sont pas traités comme des méthodes mais bel et bien comme de simples blocs (on l'a déjà constaté ci-dessus avec le rôle de *return*). En théorie, il est possible dans un bloc *catch* d'accéder à certaines variables appartenant à un bloc englobant :

```
int f()
{ int n = 12 ;
  try
  { float x ;
    .....
  }
  catch (Exception E1 e)
  { // ici, on n'a pas accès à x (bloc disjoint de celui-ci)
    // mais on a accès à n (bloc englobant)
  }
```

4.2 Choix du gestionnaire d'exception

Lorsqu'une exception est déclenchée dans un bloc *try*, on recherche parmi les différents gestionnaires associés celui qui correspond à l'objet mentionné à *throw*. L'examen a lieu dans l'ordre où les gestionnaires apparaissent. On sélectionne le premier qui est soit du type exact de l'objet, soit d'un type de base (polymorphisme). Cette possibilité peut être exploitée pour regrouper plusieurs exceptions qu'on souhaite traiter plus ou moins finement. Supposons par exemple que les exceptions *ErrConst* et *ErrDepl* sont dérivées d'une même classe *ErrPoint* :

```
class ErrPoint extends Exception { ..... }
class ErrConst extends ErrPoint  { ..... }
class ErrDepl  extends ErrPoint  { ..... }
```

Considérons une méthode *f* quelconque (peu importe à quelle classe elle appartient) déclenchant des exceptions de type *ErrPoint* et *ErrDepl* :

```
void f ()
{ .....
  throw ErrConst ;
  .....
  throw ErrDepl ;
}
```

Dans un programme utilisant la méthode *f*, on peut gérer les exceptions qu'elle est susceptible de déclencher de cette façon :

```
try
{ .....   // on suppose qu'on utilise f
}
catch (ErrPoint e)
{ // on traite ici à la fois les exceptions de type ErrConst
  //  et celles de type ErrDepl
}
```

Mais on peut aussi les gérer ainsi :

```
try
{ .....   // on suppose qu'on utilise f
}
catch (Errconst e)
{ // on traite ici uniquement les exceptions de type ErrConst
}
catch (ErrDepl e)
{ // et ici, uniquement celles de type ErrDepl
}
```

ou encore ainsi :

```
try
{ .....   // on suppose qu'on utilise f
}
catch (Errconst e)
{ // on traite ici uniquement les exceptions de type ErrConst
}
catch (ErrPoint e)
{ // et ici, toutes celles de type ErrPoint ou dérivé (autre que Errconst)
}
```

Remarque

Si l'on procédait ainsi :

```
try { ..... }
catch (ErrPoint e)  { ..... }
catch (ErrConst e)  { ..... }
```

on obtiendrait une erreur de compilation due à ce que le gestionnaire *ErrConst* n'a plus désormais aucune chance d'être atteint.

4.3 Cheminement des exceptions

Lorsqu'une méthode déclenche une exception, on cherche tout d'abord un gestionnaire dans l'éventuel bloc *try* contenant l'instruction *throw* correspondante. Si l'on n'en trouve pas ou si aucun bloc *try* n'est prévu à ce niveau, on poursuit la recherche dans un éventuel bloc *try* associé à l'instruction d'appel dans une méthode appelante, et ainsi de suite.

Le gestionnaire est rarement trouvé dans la méthode qui a déclenché l'exception puisque l'un des objectifs fondamentaux du traitement d'exception est précisément de séparer déclenchement et traitement !

Bien qu'intuitifs, les exemples précédents correspondaient bien à une recherche dans un bloc *try* de l'appelant. Par exemple, l'exception *ErrConst* déclenchée par un constructeur de *Point* était traitée non dans un bloc *try* de ce constructeur, mais dans un bloc *try* de la méthode *main* qui avait appelé le constructeur.

4.4 La clause throws

Nous avons déjà noté sa présence dans l'en-tête de certaines méthodes (constructeur de *Point*, méthode *deplace*). D'une manière générale, Java impose la règle suivante :

> Toute méthode susceptible de déclencher une exception qu'elle ne traite pas localement doit mentionner son type dans une clause *throws* figurant dans son en-tête.

Bien entendu, cette règle concerne les exceptions que la méthode peut déclencher directement par l'instruction *throw*, mais aussi toutes celles que peuvent déclencher (sans les traiter) toutes les méthodes qu'elle appelle. Autrement dit, la clause *throws* d'une méthode doit mentionner au moins la réunion de toutes les exceptions mentionnées dans les clauses *throws* des méthodes appelées.

Grâce à cette contrainte, le compilateur est en mesure de détecter tout écart à la règle. Ainsi, au vu de l'en-tête d'une méthode, on sait exactement à quelles exceptions on est susceptible d'être confronté.

On notera que si aucune clause *throws* ne figure dans l'en-tête de la méthode *main*, on est certain que toutes les exceptions sont prises en compte. En revanche, si une clause *throws* y figure, les exceptions mentionnées ne seront pas prises en compte. Comme il n'y a aucun bloc *try* englobant, on aboutit alors à une erreur d'exécution précisant l'exception concernée.

Remarque

Nous verrons un peu plus loin qu'il existe des exceptions dites implicites, qui ne respectent pas la règle que nous venons d'exposer. Elles pourront provoquer une erreur d'exécution sans avoir été déclarées dans une clause *throws*.

4.5 Redéclenchement d'une exception

Dans un gestionnaire d'exception, il est possible de demander que, malgré son traitement, l'exception soit retransmise à un niveau englobant, comme si elle n'avait pas été traitée. Il suffit pour cela de la relancer en appelant à nouveau l'instruction *throw* :

```
try
{  .....
}
catch (Excep e)   // gestionnaire des exceptions de type Excep
{ .....
  throw e ;       // on relance l'exception e de type Excep
}
```

Voici un exemple de programme complet illustrant cet aspect :

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<=0) || (y<=0)) throw new ErrConst() ;
    this.x = x ; this.y = y ;
  }
  public void f() throws ErrConst
  { try
    { Point p = new Point (-3, 2) ;
    }
    catch (ErrConst e)
    { System.out.println ("dans catch (ErrConst) de f") ;
      throw e ;        // on repasse l'exception à un niveau superieur
    }
  }
  private int x, y ;
}

class ErrConst extends Exception
{ }

public class Redecl
{ public static void main (String args[])
  { try
    { Point a = new Point (1, 4) ;
      a.f() ;
    }
    catch (ErrConst e)
    { System.out.println ("dans catch (ErrConst) de main") ;
    }
  System.out.println ("apres bloc try main") ;
  }
}
```

```
dans catch (ErrConst) de f
dans catch (ErrConst) de main
apres bloc try main
```

Exemple de redéclenchement d'une exception

Cette possibilité de redéclenchement d'une exception s'avère très précieuse lorsque l'on ne peut résoudre localement qu'une partie du problème posé.

Informations complémentaires

Une exception peut en déclencher une autre. Cette situation est tout à fait légale, même si elle est rarement utilisée.

```
try {  ..... }
catch (Ex1 e)         // gestionnaire des exceptions de type Ex1
{ .....
  throw new Ex2() ;   // on lance une exception de type Ex2
}
```

Voici un exemple d'école :

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<=0) || (y<=0)) throw new ErrConst() ;
    this.x = x ; this.y = y ;
  }
  public void f() throws ErrConst, ErrBidon
  { try
    { Point p = new Point (-3, 2) ;
    }
    catch (ErrConst e)
    { System.out.println ("dans catch (ErrConst) de f") ;
      throw new ErrBidon() ;        // on lance une nouvelle exception
    }
   }
  private int x, y ;
}
class ErrConst extends Exception
{ }
class ErrBidon extends Exception
{}

public class Redecl1
{ public static void main (String args[])
  { try
    { Point a = new Point (1, 4) ;
      a.f() ;
    }
```

```
    catch (ErrConst e)
    { System.out.println ("dans catch (ErrConst) de main") ;
    }
    catch (ErrBidon e)
    { System.out.println ("dans catch (ErrBidon) de main") ;
    }
    System.out.println ("apres bloc try main") ;
  }
}
```

```
dans catch (ErrConst) de f
dans catch (ErrBidon) de main
apres bloc try main
```

Exemple de gestionnaire déclenchant une nouvelle exception

En C++

On peut relancer une exception par l'instruction *throw* (sans paramètres). Mais il s'agit obligatoirement d'une exception de même type quelle celle traitée par le gestionnaire.

4.6 Le bloc finally

Nous avons vu que le déclenchement d'une exception provoque un branchement inconditionnel au gestionnaire, à quelque niveau qu'il se trouve. L'exécution se poursuit avec les instructions suivant ce gestionnaire.

Cependant, Java permet d'introduire, à la suite d'un bloc *try*, un bloc particulier d'instructions qui seront toujours exécutées :

- soit après la fin "naturelle" du bloc *try*, si aucune exception n'a été déclenchée (il peut s'agir d'une instruction de branchement inconditionnel telle que *break* ou *continue*),
- soit après le gestionnaire d'exception (à condition, bien sûr, que ce dernier n'ait pas provoqué d'arrêt de l'exécution).

Ce bloc est introduit par le mot clé *finally* et doit obligatoirement être placé après le dernier gestionnaire.

Bien entendu, cette possibilité n'a aucun intérêt lorsque les exceptions sont traitées localement. Considérez par exemple :

```
try { ..... }
catch (Ex e) {  ..... }
finally
{ // instructions A
}
```

Ici, le même résultat pourrait être obtenu en supprimant tout simplement le mot clé *finally* et en conservant les *instructions A* (avec ou sans bloc) à la suite du gestionnaire.

En revanche, il n'en va plus de même dans :

```
void f (...) throws Ex
{ .....
  try
  { ..... }
  finally
  { // instructions A }
  .....
}
```

Ici, si une exception *Ex* se produit dans *f*, on exécutera les *instructions A* du bloc *finally* avant de se brancher au gestionnaire approprié. Sans la présence de *finally*, ces mêmes instructions ne seraient exécutées qu'en l'absence d'exception dans le bloc *try*.

D'une manière générale, le bloc *finally* peut s'avérer précieux dans le cadre de ce que l'on nomme souvent l'*acquisition de ressources*. On range sous ce terme toute action qui nécessite une action contraire pour la bonne poursuite des opérations : la création d'un objet, l'ouverture d'un fichier, le verrouillage d'un fichier partagé... Toute ressource acquise dans un programme doit pouvoir être convenablement libérée, même en cas d'exception. Le bloc *finally* permet de traiter le problème puisqu'il suffit d'y placer les instructions de libération de toute ressource allouée dans le bloc *try*.

Remarque

En toute rigueur, il est possible d'associer un bloc *finally* à un bloc *try* ne comportant aucun gestionnaire d'exception. Dans ce cas, ce bloc est exécuté après la sortie du bloc *try*, quelle que soit la façon dont elle a eu lieu :

```
try
{ .....
  if (...) break ;    // si ce break est exécuté, on exécutera d'abord
  .....
}
finally               // ce bloc finally
{ ..... }
.....                 // avant de passer à cette instruction
```

Informations complémentaires

En théorie, il est possible de rencontrer des instructions *return* à la fois dans un bloc *try* et dans un bloc *finally*, comme dans :

```
try
{ .....
  return 0 ;      // provoque d'abord l'exécution
}
finally           // de ce bloc finally
{ .....
  return -1 ;      // et un retour avec la valeur -1
}
```

Ici, la méthode semble devoir exécuter une instruction *return 0*, mais il lui faut quand même exécuter le bloc *finally*, qui contient à son tour une instruction *return -1*. Dans ce cas, c'est la valeur prévue en dernier qui sera renvoyée (ici -1).

5 Les exceptions standard

Java fournit de nombreuses classes prédéfinies dérivées de la classe *Exception*, qui sont utilisées par certaines méthodes standard ; par exemple, la classe *IOException* et ses dérivées sont utilisées par les méthodes d'entrées-sorties. Certaines classes exception sont même utilisées par la machine virtuelle à la rencontre de situations anormales telles qu'un indice de tableau hors limites, une taille de tableau négative...

Ces exceptions standard se classent en deux catégories.

- Les *exceptions explicites* (on dit aussi *sous contrôle*) correspondent à ce que nous venons d'étudier. Elles doivent être traitées par une méthode, ou bien être mentionnées dans la clause *throws*.
- Les *exceptions implicites* (ou *hors contrôle*) n'ont pas à être mentionnées dans une clause *throw* et on n'est pas obligé de les traiter (mais on peut quand même le faire).

En fait, cette classification sépare les exceptions susceptibles de se produire n'importe où dans le code de celles dont on peut distinguer les sources potentielles. Par exemple, un débordement d'indice ou une division entière par zéro peuvent se produire presque partout dans un programme ; ce n'est pas le cas d'une erreur de lecture, qui ne peut survenir que si l'on fait appel à des méthodes bien précises.

Vous trouverez la liste des exceptions standard en Annexe D[1]. Voici un exemple de programme qui détecte les exceptions standard *NegativeArraySizeException* et *ArrayIndexOutOfBoundsException* et qui utilise la méthode *getMessage* pour afficher le message correspondant[2]. Il est accompagné de deux exemples d'exécution.

```
public class ExcStd1
{ public static void main (String args[])
  { try
    { int t[] ;
      System.out.print ("taille voulue : ") ;
      int n = Clavier.lireInt() ;
      t = new int[n] ;
      System.out.print ("indice : ") ;
      int i = Clavier.lireInt() ; t[i] = 12 ;
      System.out.println ("*** fin normale") ;
     }
```

1. Vous y verrez qu'il existe de nombreuses exceptions implicites et qu'il serait donc fastidieux de devoir toutes les traiter ou les déclarer.

2. Notez qu'il n'est guère explicite !

```
      catch (NegativeArraySizeException e)
      { System.out.println ("Exception taille tableau negative : "
                            + e.getMessage() ) ;
      }
      catch (ArrayIndexOutOfBoundsException e)
      { System.out.println ("Exception indice tableau : " + e.getMessage() ) ;
      }
   }
}
```

```
taille voulue : -2
Exception taille tableau negative :
```

```
taille voulue : 10
indice : 15
Exception indice tableau : 15
```

Exemple de traitement d'exceptions standard

Remarque

Nous pourrions écrire le programme précédent sans détection d'exceptions (une telle version figure sur le CD-Rom sous le nom *ExcStd2.java*) Comme les exceptions concernées sont implicites, il n'est pas nécessaire de les déclarer dans *throws*. Dans ce cas, les deux exemples d'exécution conduiraient à un message d'erreur et à l'abandon du programme.

Informations complémentaires

La classe *Exception* dérive en fait de *Throwable*. Il existe une classe *Error*, dérivée de *Throwable* (et sans lien avec *Exception*), qui correspond à des exceptions particulières (on parle parfois d'erreurs plutôt que d'exceptions) que vous n'aurez généralement pas à traiter[1].

1. Pour vous en convaincre, voici des exemples de telles exceptions : *VirtualMachineError*, *LinkageError*, *InstantiationError*, *InternalError*.