# Chapitre 12

# Exceptions en Java

Les exceptions constituent le mécanisme avec lequel Java gère les erreurs à l'exécution. Une méthode en Java, plutôt que de renvoyer un code d'erreur pour signaler que quelque chose d'anormal s'est produit, *lèvera une exception*. La philosophie est que le code appelant la méthode pourra se *saisir* de l'exception, pour remédier à la situation lorsque cela est possible.

## 1 Les exceptions sont des objets java

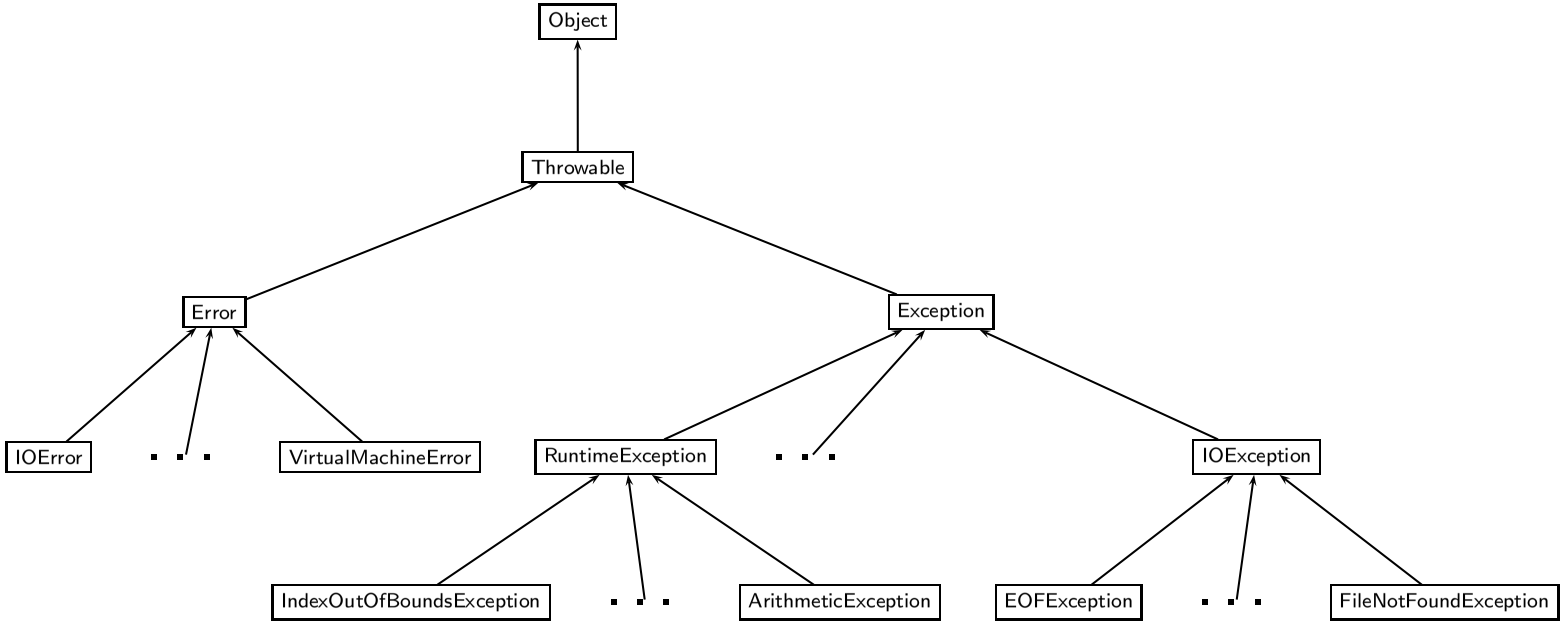
Une exception est un *objet* en Java, destiné à signaler une exécution anormale dans une suite d'instructions.

### 1.1 Hiérarchie des exceptions

Le type de cet objet est java.lang.Throwable ou l'une de ses sous-classes. Comme le montre la hiérarchie des classes représentée ci-dessous, Throwable possède deux sous-classes standard : Error et Exception.

Error est la super-classe pour les exceptions indiquant des problèmes en général irrécupérables, telles que :
- la machine virtuelle n'a plus de mémoire,
- un fichier de classe est corrompu,
- ...

La super-classe Exception indique plutôt des erreurs auxquelles on peut réagir, telles que :
- EOFException qui indique qu'on a rencontré une fin de fichier,
- ArrayOutOfBoundsException qui indique qu'on a adressé un tableau en dehors de ses limites,
- ...


Object
Throwable
Error
Exception
IOError
VirtualMachineError
RuntimeException
IOException
IndexOutOfBoundsException
ArithmeticException
EOFException
FileNotFoundException

## 1.2 Personnalisation d'une exception

En tant qu'objet, une exception possède des attributs et des méthodes. La classe Throwable et toutes ses sous-classes possèdent :

- un attribut de type String destiné à contenir un message d'erreur décrivant l'anomalie,
- une méthode getMessage() permettant de récupérer ce message,

Le message est personnalisable au moment de l'instanciation de l'exception. Par exemple, si la variable fileName (de type String) contient un nom de fichier, on pourra créer une exception indiquant que ce fichier n'a pas été trouvé au moyen de :

```
new FileNotFoundException("Le fichier "+fileName+" n'existe pas");
```

Il est également possible de créer sa propre classes d'exception en héritant de l'une des classes existantes.

## 1.3 Levée et propagation d'une exception

### 1.3.1 Levée d'une exception

En cas d'anomalie, on dit qu'on *lève* (ou encore qu'on *lance)* une exception. Cela consiste à instancier l'exception, puis à la lever à proprement parler au moyen de l'instruction throw. Comme cela ne peut se produire qu'au sein d'une méthode, on dit de la méthode qu'elle lève l'exception.

Par exemple, considérons une méthode (statique) factorielle(n) chargée de calculer la factorielle d'un entier $n$ fourni en paramètre. Si $n$ est négatif, l'opération n'est pas définie et on va le signaler en levant une exception.

```
public static long factorielle(int n) {
    if (n < 0)
        throw new IllegalArgumentException("Factorielle d'un nombre négatif indéfinie");

    long fact = 1;
    for (int i=n; i > 1; i--)
        fact *= i;

    return fact;
}
```

Ici, on choisit d'instancier une exception dont le type est IllegalArgumentException. Le message d'erreur est personnalisé au moment de l'instanciation.

### 1.3.2 Mécanisme de propagation d'une exception

Aussitôt qu'une exception est levée avec throw, l'interpréteur java *interrompt* l'exécution normale du programme et l'exception est propagée par le mécanisme suivant : elle se propage de bloc de code appelant en bloc de code appelant, attendant de rencontrer un bloc (nommé catch, *cf.* partie 2.3) capable de la traiter.

Si c'est le cas, l'exception est traitée et le programme reprend ensuite normalement son exécution, juste après le bloc catch qui a réussi à traiter l'exception.

Si aucun bloc catch capable d'arrêter l'exception n'est rencontré, celle-ci va se propager jusqu'à la méthode main(). Si elle n'y est toujours pas traitée, alors l'interpréteur java :

1. affiche le message d'erreur de l'exception ;
2. affiche une trace de la pile des appels permettant de localiser où l'exception s'est produite ;
3. arrête le programme.

**N.B.** L'affichage de la pile des appels peut être obtenu au moyen de la méthode printStackTrace() définie dans Throwable.

# 2 Traitement des exceptions : *catch me if you can*

## 2.1 Exceptions contrôlées et non contrôlées

Les exceptions se divisent entre celles qui sont *contrôlées* : le programmeur est *obligé* de les traiter ; et celles qui sont *non contrôlées* : leur traitement n'est pas imposé.

**Exceptions non-contrôlées.** Les exceptions non-contrôlées sont celles qui sont *a priori* imprévisibles. Pour cette raison, on ne peut pas imposer au programmeur de prévoir prendre le contrôle lorsqu'elles se produisent. Elles peuvent entraîner un plantage du programme.

Les exceptions non-contrôlées sont celles du type Error, avec en plus celles du type RuntimeException.

**Exceptions contrôlées.** Elles correspondent à des cas d'erreur prévisibles, entraînant l'obligation pour le programmeur de prévoir comment réagir. Les exceptions contrôlées sont celles du type Exception, sauf les RuntimeException.

Une méthode qui lève une exception contrôlée doit obligatoirement le déclarer à la fin de sa signature au moyen du mot clé throws suivi du type de l'exception levée.

Par exemple, soit une méthode void readFile(String fileName) chargée de lire un fichier dont le nom est fileName. Si elle ne trouve pas le fichier, elle va lever une exception du type FileNotFoundException. Sa signature devient alors :

```
void readFile(String fileName) throws FileNotFoundException
```

**Remarque.** Dans l'exemple précédent de la méthode factorielle(), l'exception levée était du type IllegalArgumentException, qui est une sous-classe de RuntimeException, et donc non-contrôlée. factorielle() n'avait donc pas l'obligation de déclarer throws IllegalArgumentException, même si elle aurait pu le faire.

## 2.2 Se préoccuper des exceptions

Lorsque vous invoquez une méthode qui lève une exception contrôlée, vous avez l'obligation de vous en préoccuper en la traitant.

**Question.** Comment puis-je savoir si une méthode que je souhaite invoquer est susceptible de lever une exception ?

**Réponse.** Je peux le voir dans l'API, en consultant sa signature qui fait apparaître une clause throws ; ou sinon, java me le signalera par une erreur au moment de la compilation si je ne me suis pas préoccupé de traiter l'exception.

Traiter l'exception levée par une méthode invoquée peut se faire de deux façons : soit en se saisissant de l'exception, soit en déléguant cette tâche au programme (méthode) appelant.

Pour le deuxième cas, il suffit de rajouter le mot clé throws dans la signature de votre propre méthode (celle que vous écrivez et qui invoque une méthode qui lève une exception).

Dans le premier cas, il faut englober l'appel de la méthode qui lève une exception dans une structure try/catch/finally.

### 2.3 L'instruction try/catch/finally pour se saisir d'une exception

L'instruction try/catch/finally permet de définir des blocs d'instruction pour se saisir des exceptions :

```
try {
      // code levant une exception
}
catch (TypeDException e) {
      // code réagissant à l'exception
}
finally {
      // code « de nettoyage »
}
```

Le bloc try englobe du code susceptible de lever une exception.

Un bloc catch est conçu pour réagir à un type d'exception spécifique. Le *type* (au sens de classe java) de l'exception traitable est précisé entre parenthèses. Ce bloc attrape toutes les exceptions de ce type ou d'un type dérivé levées dans le bloc try associé.

Le bloc finally contient du code qui sera toujours exécuté, que l'exception se soit produite ou non. Son but est de contenir du « code de nettoyage », par exemple qui referme des fichiers ouverts.

Un bloc try doit être accompagné soit d'un bloc catch, soit d'un bloc finally, soit des deux. De plus, plusieurs blocs catch peuvent être définis pour un même bloc try.

## 3 Un exemple

Soit une méthode (statique) subArrayString(int[] tab, int from, int to) qui retourne une chaîne de caractères représentant le sous-tableau considéré entre les indices from et to. Si les indices sont absurdes, on veut le signaler par une exception.

Commençons par créer une classe d'exceptions qui indique des mauvaises valeurs d'indice :

```
public class BadIndexesException extends Exception {
        BadIndexesException() {
            super();
        }

        BadIndexesException(String message) {
            super(message);
        }
}
```

On peut écrire la méthode subArrayString(...) :

```
public static String subArrayString(int[] tab, int from, int to)
                                        throws BadIndexesException {
    if (from < 0 || from >= tab.length || to < 0 || to >= tab.length)
        throw new BadIndexesException("Les indices "+from+" et "+to+
           " doivent être dans l'intervalle [0, "+(tab.length-1)+"]");

    if (from > to)
        throw new BadIndexesException("L'indice de début "+from+
                    " doit être plus petit que l'indice de fin "+to);

    // Si ce point est atteint, alors on n'a pas levé d'exception.
    // Les indices sont corrects, on peut commencer le traitement
    // sans risque.
    String sas = "[";
    for (int i=from; i <= to; i++) {
        sas += tab[i];
        if (i < to)
            sas += ", ";
    }
    return sas+"]";
}
```

On veut maintenant écrire une méthode printSubArrayString(int[] tab, int from, int to) chargée tout simplement d'afficher la chaîne de caractère calculée par subArrayString(...). Comme subArrayString(...) lève une exception, printSubArrayString(...) doit s'en préoccuper : soit en la renvoyant au code appelant (cas 1), soit en la traitant par un try/catch (cas 2).

**Cas 1 - Déléguer au code appelant.** Il suffit de déclarer que notre méthode printSubArrayString(...) « throws » l'exception :

```
public static void printSubArrayString1(int[] tab, int from, int to)
                                            throws BadIndexesException {
    // version 1 : l'exception, si elle se produit, ne fait que transiter
    // sans être traitée
   System.out.println("de "+from+" à "+to+" : "+subArrayString(tab, from, to));
}
```

**Cas 2 - Se saisir de l'exception.** Il faut englober l'appel de subArrayString(...) dans un bloc try, et récupérer l'erreur éventuelle dans un bloc catch. Ici, le bloc finally (facultatif) n'a pas lieu d'être.

```
public static void printSubArrayString2(int[] tab, int from, int to) {
    // version 2 : l'exception, si elle se produit, est saisie (ou arrêtée)
    try {
     System.out.println("de "+from+" à "+to+" : "+subArrayString(tab, from, to));
    }
    catch (BadIndexesException e) {
       System.out.println(e.getMessage());
    }
}
```

La différence est que dans le cas 1, l'exception peut continuer à se propager, au risque de faire planter le programme si personne ne l'arrête. Dans le cas 2, l'exception est attrapée, ce qui permet au programme de continuer sans planter.

Ceci est illustré par exemple par le programme main() suivant.

```
public static void main(String[] args) throws BadIndexesException {
    // Selon les exécutions (au hasard), ce programme peut s'exécuter
    // avec ou sans erreur (ou exception). Si une exception se produit
    // le programme peut soit planter (cas 1, exception non rattrapée),
    // soit arriver quand même à son terme (cas 2, exception rattrapée)

    int[] tab = new int[20];

    for (int i=0; i < tab.length; i++)
        tab[i] = i; // on remplit par les valeurs de 0 à 19

    // On génère les indices au hasard
    int from = Hasard.entier(20)-3;
    int to = Hasard.entier(20)+3;

    // Cas 1 ou 2 tiré au hasard :
    if (Hasard.entier(2) == 0) {
        System.out.println("Cas 1 : appel non protégé par un try");
        printSubArrayString1(tab, from, to);
    }
    else {
        System.out.println("Cas 2 : appel protégé par un try");
        printSubArrayString2(tab, from, to);
    }

    // En cas d'exception, ce point ne peut être atteint que si celle-ci
    // a été rattrapée (cas 2). Sinon (cas 1), le programme a planté avant
    // d'atteindre ce point
    System.out.println("FIN NORMALE DU PROGRAMME");
}
```