

Les exceptions et les bonnes pratiques

par Anis Frikha ([Mon site](#))

Date de publication : 28/03/2007

Dernière mise à jour :

Après une présentation des exceptions, cet article tente d'illustrer quelques bonnes pratiques de programmation en rapport avec les exceptions.

- I - Introduction
- II - Qu'est-ce qu'une situation exceptionnelle ?
- III - La classe Error
- IV - La classe Exception
 - IV-A - Traitement des exceptions
 - IV-B - Créer son propre type d'exception
 - IV-C - La clause finally
- V - La classe RuntimeException
- VI - Quelques bonnes pratiques
 - VI-A - Ne jamais ignorer une exception
 - VI-B - Utiliser la clause throws de manière exhaustive
 - VI-C - Les exceptions ne sont pas faites pour le contrôle de flux
 - VI-D - Les exceptions et les entrées/sorties
 - VI-E - Attention au return dans un bloc finally !
 - VI-F - Utiliser les exceptions standards
 - VI-G - Une exception peut en cacher une autre !
 - VI-H - Bien déterminer la totalité du traitement qui sera interrompu lorsque une exception est levée
- VII - Conclusion
- VIII - Remerciements

I - Introduction

Bien souvent, un programme doit traiter des situations exceptionnelles qui n'ont pas un rapport direct avec sa tâche principale. Ceci oblige le programmeur à réaliser de nombreux tests avant d'écrire les instructions utiles du programme. Cette situation a deux inconvénients majeurs :

- Le programmeur peut omettre de tester une condition ;
- Le code devient vite illisible car la partie utile est masquée par les tests.

Java remédie à cela en introduisant un *Mécanisme de gestion des exceptions* qui est l'objet de cet article. Grâce à ce mécanisme, on peut améliorer grandement la lisibilité du code en découplant le code utile de celui qui traite des situations exceptionnelles, et on peut aussi déléguer au langage la tâche d'énumération des tests à effectuer.

II - Qu'est-ce qu'une situation exceptionnelle ?

Une situation exceptionnelle peut être assimilée à une erreur (dans le cadre de cet article), c'est à dire une situation qui est externe à la tâche principale d'un programme. En Java, on distingue trois types d'erreurs, qui sont de degrés de gravité différents, à savoir :

- Les erreurs graves qui causent généralement l'arrêt du programme et qui sont représentées par la classe **java.lang.Error** .
- Les erreurs qui doivent généralement être traitées et qui sont représentées par la classe **java.lang.Exception**.
- Les erreurs qui peuvent ne pas être traitées et qui sont des objets de la classe **java.lang.RuntimeException** qui hérite de **java.lang.Exception**.

Toutes ces classes héritent directement ou indirectement de la classe **java.lang.Throwable**. Voici un petit diagramme récapitulatif de tout cela :

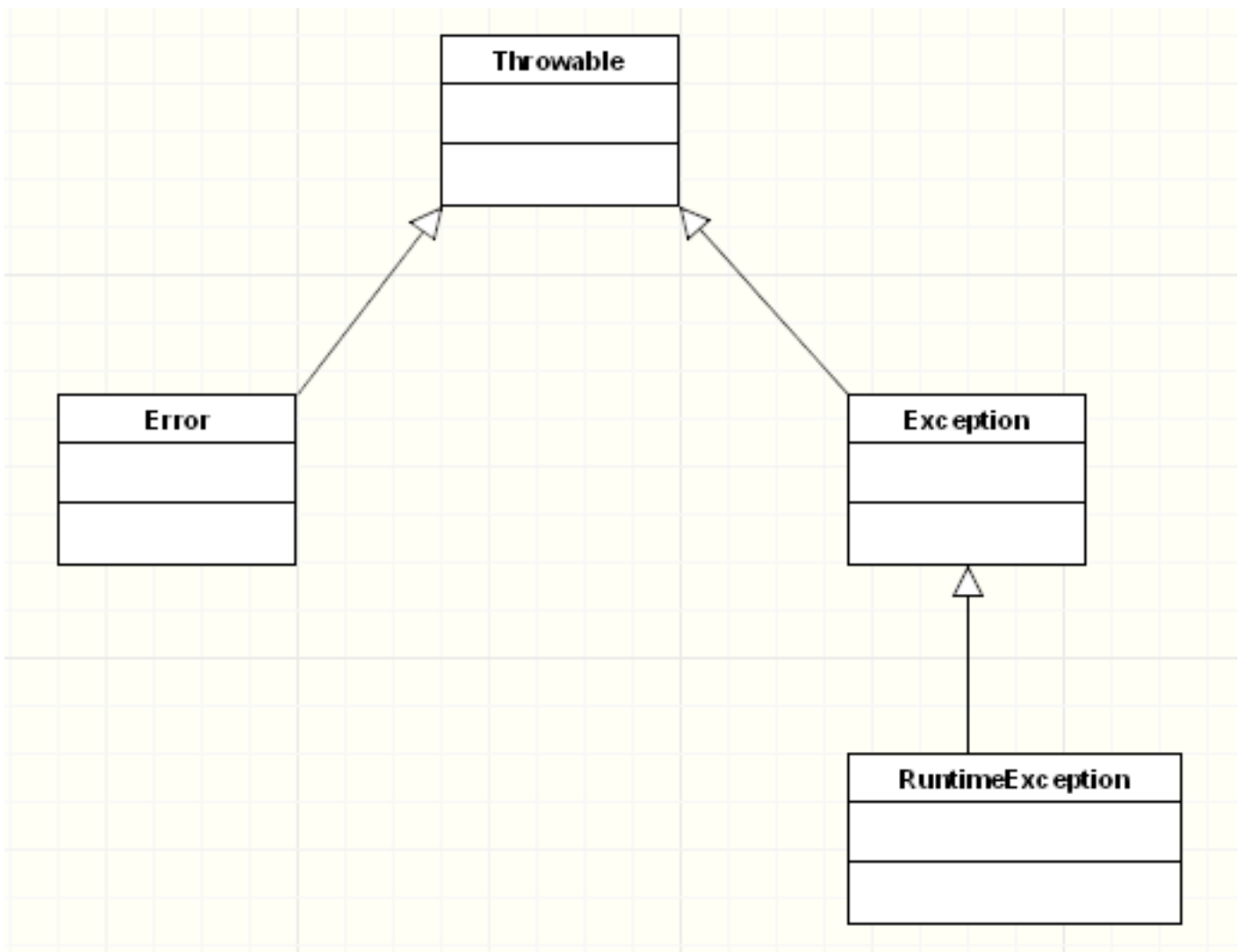


Diagramme de classe

III - La classe Error

Cette classe est instanciée lorsque une erreur grave survient, c'est à dire une erreur empêchant la JVM de faire correctement son travail. Les objets de type Error ne sont pas destinés à être traités et il est même déconseillé de le faire. Un exemple récurrent est **java.lang.OutOfMemoryError** qui signifie que la machine virtuelle java ne dispose plus d'assez de mémoire pour pouvoir allouer des objets. A titre d'exemple voici un code qui alloue de la mémoire pour un tableau de 1000000000 String et par ce fait déclenche une **OutOfMemoryError**

```
public class ErreurMemoire {  
  
    public static void main(String[] args) {  
        String[] tableau=new String[1000000000];  
    }  
  
}
```

Les erreurs, lorsqu'elles surviennent, ont la particularité d'arrêter le thread en cours, **sauf si** elles sont traitées par un catch. Ainsi n'importe quel type de **Throwable** peut être "catché". Le code précédent deviendrait alors :

```
try {  
    String[] tableau=new String[1000000000]; // OutOfMemoryError  
} catch (Error e) {  
    System.out.println("Oups ! Une erreur est survenue : " + e);  
}  
System.out.println("Fin du programme");
```

 *Mais comme mentionné plus haut, cette pratique est à éviter car les **Errors** correspondent à des problèmes graves qu'il n'est généralement pas possible de traiter dans le code.*

IV - La classe Exception

Les objets de type **Exception** ou bien de l'une de ses sous-classes sont instanciés lorsque une erreur au niveau applicatif survient. On dit, dans ce cas là, qu'une exception est levée. Lorsqu'une exception est levée, elle se propage dans le code en ignorant les instructions qui suivent et si aucun traitement survient, elle débouche sur la sortie standard. Voici un bout de code illustrant cela :

```
public class PropagationException {  
  
    public static void main(String[] args) {  
        String chemin="/Un/chemin/vers/une/classe/qui/n'existe/pas";  
        Class.forName(chemin);//levée d'une ClassNotFoundException  
        System.out.println("fin du programme");  
    }  
}
```

On voit bien sur cet exemple que l'instruction qui suit la levée de l'exception n'est pas exécutée : On n'obtient pas l'affichage *fin du programme*

IV-A - Traitement des exceptions

Les exceptions sont traitées via des blocs **try/catch** qui veulent littéralement dire essayer/attraper. On exécute les instructions susceptibles de lever une exception dans le bloc try et en cas d'erreur ce sont les instructions du bloc catch qui seront exécutées, pourvu qu'on attrape bien l'exception levée. Reprenons notre exemple de tout à l'heure et traitons l'exception. Ce qui donne le code suivant :

```
public class PropagationException {  
  
    public static void main(String[] args) {  
        try{  
            String chemin="/Un/chemin/vers/une/classe/qui/n'existe/pas";  
            Class.forName(chemin);//levée d'une ClassNotFoundException  
            System.out.println("fin du programme");  
        }catch(ClassNotFoundException ex){  
            System.out.println("Une exception est survenue");  
        }  
    }  
}
```

Comme prévu, on obtient bien l'affichage : "Une exception est survenue". Il faut tout de même faire attention au type d'exception qu'on met dans le catch : On aurait pu simplement déclarer une exception de type Exception. Cela aurait pour effet d'attraper toutes les exceptions levées dans le bloc try car l'ensemble des exceptions déclarées dans la JDK hérite de cette classe (Exception), il va sans dire que la réciproque n'est pas vraie. On peut également mettre plusieurs blocs **catch** qui se suivent afin de fournir un traitement spécifique pour chaque type d'exception. Cela doit être fait en respectant la hiérarchie des exceptions. Un code comme celui-ci ne compilera pas !

```
public class PropagationException {

    public static void main(String[] args) {
        try{
            String chemin="/Un/chemin/vers/une/classe/qui/n'existe/pas";
            Class.forName(chemin);//levée d'une ClassNotFoundException
            System.out.println("fin du programme");
        }catch(Exception e){
            //traitement
            //erreur de compilation car les autres blocs catch ne seront jamais executés
        }catch(ClassNotFoundException ex){
            System.out.println("Une exception est survenue");
        }
    }
}
```

Il faudrait plutôt écrire ceci :

```
public class PropagationException {

    public static void main(String[] args) {
        try{
            String chemin="/Un/chemin/vers/une/classe/qui/n'existe/pas";
            Class.forName(chemin);//levée d'une ClassNotFoundException
            System.out.println("fin du programme");
        }catch(ClassNotFoundException ex){
            System.out.println("Une exception est survenue");
        }catch(Exception e){
            //traitement
            //pas d'erreur de compilation
        }
    }
}
```

IV-B - Créer son propre type d'exception

Pour créer son propre type d'exception, il faut écrire une classe héritant de la classe **Exception**. Allons-y donc, créons une exception qu'on appellera **NombreNonValideException** qu'on lèvera si l'utilisateur de notre programme entre un nombre non compris entre 0 et 9 . Voici à quoi ressemble notre classe **NombreNonValideException** :

```
public class NombreNonValideException extends Exception{

    /** Créé une nouvelle instance de NombreNonValide */
    public NombreNonValide() {}

}
```

On pourrait se contenter du code précédent, cependant il est souvent préférable d'utiliser les mêmes constructeurs que la classe **Exception**, afin de simplifier leurs créations et l'encapsulation d'exception. Voici à quoi ça correspondrait :

```
public class NombreNonValideException extends Exception{
    /**
     * Crée une nouvelle instance de NombreNonValideException
     */
    public NombreNonValideException() {}
    /**
     * Crée une nouvelle instance de NombreNonValideException
     * @param message Le message détaillant exception
     */
    public NombreNonValideException(String message) {
        super(message);
    }
    /**
     * Crée une nouvelle instance de NombreNonValideException
     * @param cause L'exception à l'origine de cette exception
     */
    public NombreNonValideException(Throwable cause) {
        super(cause);
    }
    /**
     * Crée une nouvelle instance de NombreNonValideException
     * @param message Le message détaillant exception
     * @param cause L'exception à l'origine de cette exception
     */
    public NombreNonValideException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Et voici notre programme :

```
public class Nombre {

    public void parseAndPrint(String number) throws NombreNonValideException {
        try {
            int i = Integer.parseInt(number); // throw NumberFormatException
            if (i < 0 || i > 9) {
                throw new NombreNonValideException("bad value");
            }
            System.out.println(i);
        } catch (NumberFormatException e) {
            // encapsulation de l'exception
            throw new NombreNonValideException("parse error", e);
        }
    }
}
```



Les `NumberFormatException` sont encapsulés dans une `NombreNonValideException`.

Il y a ici deux choses à remarquer. Tout d'abord la présence de la clause **throws** dans la signature de la méthode, celle-ci est obligatoire pour toute méthode qui peut lever une exception. Ensuite on voit que pour lever une exception il faut user du mot clé **throw** suivi du type de l'exception qu'on instancie.

IV-C - La clause finally

Le mot clé **finally**, généralement associé à un **try**, permet l'exécution du code situé dans son bloc et ceci quelque soit la manière dont s'est déroulé l'exécution du bloc try. Voici sans plus attendre un exemple :

```
public class PropagationException {  
  
    public static void main(String[] args) {  
        try{  
            Object chaine="bonjour";  
            Integer i=(Integer)chaine;//levée d'une ClassCastException  
            System.out.println("fin du programme");  
        }finally{  
            System.out.println("on passe par le bloc finally");  
        }  
    }  
}
```

V - La classe RuntimeException

Les exceptions héritant de **java.lang.RuntimeException** représentent des erreurs qui peuvent survenir lors de l'exécution du programme. Le compilateur n'oblige pas le programmeur ni à les traiter ni à les déclarer dans une clause **throws**. Les classes **java.lang.ArithmeticException** (qui peut survenir lors d'une division par 0 par exemple) et la classe **java.lang.ArrayIndexOutOfBoundsException** (qui survient lors d'un dépassement d'indice dans un tableau) sont des exemples de **RuntimeException**. Autrement dit ce genre de code passe sans problème la compilation :

```
/**
 *
 * @author Anis Frikha
 */
public class CompilationRuntimeException {

    public static void main(String[] args) {
        String[] tableau={"A", "B", "C"};
        for(int i=0;i<=3;i++){
            System.out.println(tableau[i]);
        }
    }
}
```

Mais à l'exécution, on obtient bien une `ArrayIndexOutOfBoundsException` sans l'avoir préalablement déclarée dans une clause **throws**.

VI - Quelques bonnes pratiques

VI-A - Ne jamais ignorer une exception

Une erreur fréquente du débutant est de mettre un bloc catch vide sans aucune instruction afin de pouvoir compiler le programme. Ceci est très dangereux car cela risque de devenir une mauvaise habitude. En effet, si une exception survient, elle sera passée sous silence et le programme continuera de fonctionner ce qui peut déboucher sur des bugs incompréhensibles. Ayons donc le réflexe de bien traiter les exceptions dans les blocs catch ou au moins de mettre un `printStackTrace`, ça ne mange pas de pain ! .

```
public class NePasIgnorerUneException {
    public static void main(String[] args) {
        try{
            //traitement susceptible de lever une exception
        }catch(Exception ex){
            ex.printStackTrace();
        }
    }
}
```

VI-B - Utiliser la clause throws de manière exhaustive

Supposons que nous ayons une exception C qui hérite d'une exception B qui elle-même hérite d'une autre exception A, alors une méthode `f()` qui peut lancer ces trois exceptions doit le déclarer dans sa signature via la clause throws. Une mauvaise manière de déclarer `f()` est la suivante :

```
public void f() throws A{
    //corps de la méthode
}
```

Bien que cette façon de faire passe sans problème l'étape de compilation, elle est déconseillée car elle ne fournit pas toutes les exceptions qu'elle peut lever à l'utilisateur de cette méthode. La bonne manière de coder cela est la suivante :

```
public void f() throws A, B, C {
    //corps de la méthode
}
```

En effet, dans ce cas là, l'utilisateur de `f()` est en mesure de connaître toutes les exceptions susceptibles d'être levées par `f` et est à même de fournir une gestion fine de celles-ci.

VI-C - Les exceptions ne sont pas faites pour le contrôle de flux

C'est une très mauvaise idée que d'utiliser les exceptions pour contrôler le flux. Considérons à titre d'exemple le code suivant :

```
while(true){
//faire quelque chose
if(condition d'arrêt)
throw new FinDeBoucleException();
}
```

Ici on se sert de la levée d'une exception pour sortir de la boucle. Ce genre de code, bien qu'il fonctionne, a plusieurs inconvénients : il n'est pas efficace (création d'un objet supplémentaire à savoir l'exception), il est difficilement compréhensible et modifiable. Le langage Java fournit suffisamment d'instructions de contrôle pour éviter totalement ce genre de code, par exemple on peut utiliser l'instruction **break**.

VI-D - Les exceptions et les entrées/sorties

Concernant les entrées/sorties en Java, pensez à utiliser le pattern suivant :

```
try{
//déclaration de la ressource
try{
//utilisation de la ressource
}finally{
//fermeture de la ressource
}
}catch(ExceptionEntreeSortie ex){
//traitement de l'exception
}
```

Voici sans plus attendre une mise en oeuvre de ce pattern avec un programme qui affiche toutes les lignes d'un fichier texte en majuscule :

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

/**
 *
 * @author Anis Frikha
 */
public class FluxMajuscule {

    /** Créé une nouvelle instance de FluxMajuscule */
    public FluxMajuscule() {
    }

    public void readMaj(){
        try{
            BufferedReader br=new BufferedReader(new FileReader("monFichier.txt"));
```

```
try{
    String ligne;
    String ligneMajuscule;
    while((ligne=br.readLine())!=null){
        ligneMajuscule=ligne.toUpperCase();
        System.out.println(ligneMajuscule);
    }
}finally{
    br.close();
}
}catch(IOException ex){
    ex.printStackTrace();
}
}
```

VI-E - Attention au return dans un bloc finally !

On sort d'un bloc *try* lorsque l'une des situations suivantes se présente :

- Le bloc *try* se termine normalement.
- Une exception survient.
- Une instruction de rupture de séquence tel que *break*, *continue* ou *return* est utilisée.

Maintenant, examinons le code suivant :

```
public class ReturnFinally {

    public int methode1(){
        try{
            return 1;
        }catch(Exception e){
            return 2;
        }
    }

    public int methode2(){
        try{
            return 3;
        }finally{
            return 4;
        }
    }

    public static void main(String[] args) {
        ReturnFinally rf=new ReturnFinally();
        System.out.println("methode1 renvoie : "+rf.methode1());
        System.out.println("methode2 renvoie : "+rf.methode2());
    }
}
```

On obtient l'affichage suivant :


methode1 renvoie : 1

methode2 renvoie : 4

Bien que le premier résultat soit prévisible, le deuxième l'est beaucoup moins. En effet, on aurait tendance à penser qu'employer une instruction de rupture de séquence tel que *return* permet de quitter la méthode, ceci est vrai sauf si une clause **finally** existe.

Conclusion : évitez d'employer des instructions de rupture de séquence telle que *break*, *continue* ou *return* à l'intérieur d'un bloc *try*. Si c'est inévitable, assurez-vous qu'aucune clause **finally** ne modifie la valeur de retour de votre méthode.

VI-F - Utiliser les exceptions standards

Bien qu'il est aisé de créer son propre type d'exception, l'api Java en fournit suffisamment en standard pour vous éviter cette tâche. Vous en trouverez la liste assez exhaustive à cette adresse :  [javadoc](#)

VI-G - Une exception peut en cacher une autre !

L'exception qui apparaît sur la sortie standard n'est pas forcément celle qui est à l'origine de l'erreur. En effet, en exécutant ce code :

```
public class TestException {  
    public static void main(String[] args) throws Exception {  
        try{  
            throw new Exception("1");  
        }catch(Exception ex){  
            throw new Exception("2");  
        }  
    }  
}
```

On obtient la sortie suivante :

sur la sortie standard

```
Exception in thread "main" java.lang.Exception: 2
```

On voit bien que c'est la deuxième exception qui est renvoyée (celle qui se trouve dans le bloc **catch**) alors que c'est la première exception (celle qui se trouve dans le bloc **try**) qui est à l'origine de l'erreur .

Ceci nous amène à parler de l'**encapsulation** des exceptions qui consiste à regrouper plusieurs exceptions en une seule sans pour autant perdre l'information utile en cas d'erreur. L'encapsulation est réalisée généralement grâce au constructeur de la classe **Exception** qui prend en paramètre un **Throwable**.

Mais comme un bout de code vaut mieux qu'un long discours, voici tout de suite un exemple :

```
try {
    maMéthodeQuiRenvoiePlusieursTypesDException();
} catch (Exception e) {
    // En englobe toutes les exceptions dans une exception unique
    throw new Exception("Un problème est survenue", e);
}
```

Ce qui donne sur la sortie standard :

```
Exception in thread "main" java.lang.Exception: Un problème est survenue
at Main.main(Main.java:72)
Caused by: java.io.IOException: IO error
at Main.maMéthodeQuiRenvoiePlusieursTypesDException(Main.java:62)
at Main.main(Main.java:69)
```

Conclusion : En englobant des exceptions dans une autre, on peut simplifier la gestion des exceptions sans pour autant perdre l'information qui leurs est associée.

VI-H - Bien déterminer la totalité du traitement qui sera interrompu lorsque une exception est levée

Un exemple d'erreur typique, avec ce code qui utilise la reflection pour instancier un objet :

```
Class type = null;
Object object = null;

try {
    type = Class.forName("monpackage.MaClasse"); // throws ClassNotFoundException
} catch (ClassNotFoundException e) {
    e.printStackTrace();
    // + traitement particulier à ClassNotFoundException
}

try {
    object = type.newInstance(); // throws InstantiationException, IllegalAccessException
} catch (InstantiationException e) {
```

```
e.printStackTrace();
// + traitement particulier à InstantiationException
} catch (IllegalAccessException e) {
e.printStackTrace();
// + traitement particulier à IllegalAccessException
}

String string = object.toString();
```

Le code peut renvoyer 3 types d'exceptions selon les méthodes, qui sont bien englobé dans des **try/catch**, et le code marche correctement lorsque aucune exception n'est générée... Pourtant il n'est pas du tout sécurisé car lorsqu'une exception survient, elle n'interrompt qu'une partie du traitement : Ainsi par exemple, si la méthode `Class.forName()` remonte une exception, l'objet type restera toujours à null, mais on tentera quand même d'appeler la méthode `newInstance()` dessus, ce qui provoquera une `NullPointerException`...

La solution consiste à ce que les blocs `try/catch` doivent englober la totalité du traitement à interrompre en cas de problème. Dans ce cas il est donc préférable d'utiliser le code suivant :

```
try {

Class type = Class.forName("monpackage.MaClasse"); // throws ClassNotFoundException
Object object = type.newInstance(); // throws InstantiationException, IllegalAccessException
String string = object.toString();

} catch (ClassNotFoundException e) {
e.printStackTrace();
// + traitement particulier à ClassNotFoundException
} catch (InstantiationException e) {
e.printStackTrace();
// + traitement particulier à InstantiationException
} catch (IllegalAccessException e) {
e.printStackTrace();
// + traitement particulier à IllegalAccessException
}
```

De plus ce code a le mérite d'être bien plus lisible :

- 1 Tout le code utile est regroupé à l'intérieur du **try**.
- 2 Tous les **catch** sont au même niveau, ce qui pourrait permettre d'utiliser un traitement commun.

VII - Conclusion

Voilà, cet article touche à sa fin. J'espère qu'il vous aidera à mieux cerner le concept d'exception et qu'il vous permettra d'avoir un code à la fois plus élégant, plus propre et plus maintenable.

VIII - Remerciements

Je tiens à remercier toute l'équipe de la rédaction java, et en particulier **fabszn** pour ça relecture et ses encouragements, ainsi que **adiGuba**, **vbrabant**, **ZedroS**, **@om**, **Ricky81**, **millie** et enfin **valered** pour leurs remarques fort pertinentes.

