Université Montpellier-II
UFR des Sciences - Département Informatique
Master Informatique - UE HMIN-102 -

Ingénierie Logicielle - Concepts et Outils de la modélisation et du développement de logiciel par et pour la réutilisation.

Exception Handling Patterns - Application Java

Notes de cours
Christophe Dony

1 Introduction

1.1 Définition : Exception

Le terme d'exception est employé usuellement pour distinguer, parmi un regroupement multi-critère d'éléments, ceux ne vérifiant pas un critère particulier.

Dans le contexte d'un programme en exécution, ce que l'on signalera comme une exception est une situation dans laquelle la poursuite standard du calcul (de l'exécution) est impossible.

Une exception ne dénote pas nécessairement une situation ne survenant que de manière exceptionnelle. Le terme prête donc à confusion.

Exemple : "plus de papier dans l'imprimante"

1.2 Classification historique - J.B Goodenough 1975

- Exception de domaine : qui correspond à un échec dans la satisfaction des assertions d'entrée d'une opération.

Exemple, une exception est signalée lorsque la procédure + reçoit un argument de type non numérique.

- Exception de portée : qui correspond à une situation dans laquelle une opération ne vérifie pas ses assertions de sortie ou ne sera pas en mesure de les vérifier.

Exemple : l'exception levée lorsque l'addition de deux nombres provoque un débordement arithmétique.

- Exception programmée : correspond à l'utilisation algorithmique des primitives de gestion des exceptions ; la notion d'exception n'y apparait plus obligatoirement comme l'impossibilité de réalisation d'une opération.

1.3 Définition : Système de gestion des exceptions

Un Système de gestion des exceptions permet de définir des programmes dits tolérants aux fautes ou résistants aux erreurs : programmes capables de réagir ou de laisser un système logiciel dans un état cohérent après l'occurrence d'une exception.

Un système de gestion d'exception offre aux programmeurs trois ensembles de primitives pour :

- Signaler des exceptions,

- Associer des "handlers" à des parties de programmes,

- Traiter les exceptions : provoquer la reprise de l'exécution standard après qu'une exception ait été signalée

1.4 Du bon usage des exceptions

http://wiki.c2.com/?ExceptionPatterns

2 Représentation des exceptions en Java

Le modèle de gestion des exceptions de Java est inspiré de ceux de *Smalltalk* et de *C++*, eux-même inspirés ...

2.1 Représentation

Chaque sorte d'exception est représenté par une classe, par exemple `ArithmeticException`.

Chaque exception effective survenant durant l'exécution d'un programme est représentée par une instance d'une classe d'exception.

2.2 Classification

Les sortes d'exception sont organisées en une hiérarchie de classes.

```
1  Throwable
2     Error
3        VirtualMachineError
4        InternalError
5        OutOfMemoryError
6        StackOverflowError
7        UnknownError
8        ...
9     Exception
10       RuntimeException
11          ArithmeticException
12          ClassCastException
13          ...
14       "toutes les exceptions applicatives ..."
```

2.3 Sémantique des exceptions prédéfinies

`Error` : Situation anormale détéctée par la machine virtuelle, jugée non traitable,

`RuntimeException` : Situation exceptionnelle détectée par la machine virtuelle, correspond à des erreurs de programmation, éventuellement traitable,

`Exception` : racine de la hiérarche des exceptions traitables.

3 Signalement d'une exception en Java

3.1 Quand et comment signaler une exception ?

Pattern - *report incontinuable situations* :

Signaler une exception quand la poursuite de l'exécution standard du programme est impossible. Par exemple, si l'on doit empiler quelque chose dans une pile déjà pleine.

```
1  class Stack {
2    int index = 0;
3    int taille = 10;
4    Object[] buffer = new Object[taille];

6    void push(Object o) throws Exception {
7      if (index == taille)
8        throw new StackException(''La pile est pleine'');
9      buffer[index++] = o;}
10 }
```

Patterns

— *throw the problem not the owner* : describe why the Exception is being thrown, not the place where it is thrown
(http://wiki.c2.com/?NameTheProblemNotTheThrower).

— *DontUseExceptionsForFlowControl*
(http://wiki.c2.com/?DontUseExceptionsForFlowControl)
Evitez :

```
1  try {
2      for (int i = 0; /*wot no test?*/ ; i++)
3          array[i]++;
4      } catch (ArrayIndexOutOfBoundsException e) {}
```

Mise en oeuvre du signalement

Le signalement commence par une instantiation de la classe d'exception choisie, avec utilisation de ses constructeurs.

```
1      throw new Exception(''La pile est pleine'');
```

L'instance créée, sera passée en argument à tout handler invoqué suite au signalement.

Mise en oeuvre du signalement - recherche de handler

le signalement proprement dit est réalisé, en Java, par la primitive `throw`, qui provoque, tant que aucun handler n'est trouvé et que le bloc de pile courant n'est pas le premier :

- une recherche d'un handler dans l'environnement lexical du bloc de pile courant et son invocation s'il y en a un,

- sinon

– si le modèle est à terminaison (Java), la desctruction du bloc de pile courant.

– la poursuite de la recherche dans l'environnement lexical associé au bloc de pile précédant (bloc précédent dans la pile d'exécution).

3.2 "Checked" et "Unchecked" exceptions

Checked : exception dont le signalement ou la propagation doit être déclarée dans la signature des méthodes concernées.

Les `Error` et les `RuntimeException` sont "unchecked".

Toutes les autres sont des "checked".

Exemple : conséquense de l'utilisation de la méthode `push` de la classe `Stack` (cf. section 3.1).

```
1  class DistributeurBonbons{
2    Stack conteneur = new Stack();

4    void ajouter(Bonbon b) throws Exception {
5      ...
6       conteneur push(b);
7    }
8  }
```

3.3 Quelle sorte d'exception signaler

Voir le pattern *DontThrowGenericExceptions*

http://wiki.c2.com/?DontThrowGenericExceptions

- exceptions génériques prédéfinies, voir l'exemple précédent.

- exception générique spécifique à un domaine

```
1  public final String readLine() throws IOException {
2    if (...) throw new IOException();
3  }
```

- ou exception spécifique à une classe ou à une fonction, voir l'exemple suivant.

3.4 Définition de nouvelles sortes d'exceptions

Création de sous-classes de *Exception*.

Possibilité de structurer les exceptions d'une application en une hiérarchie.

Un exemple avec les exceptions liées à l'application `Stack` :

```
1  abstract class StackException extends Exception {
2    Stack s;
3    StackException(Stack s2) {s = s2;}}

5  public class FullStackException extends StackException {
6      protected Object rejectedElement;
```

```
    public FullStackException(Stack s2, Object o) {
        super(s2);
        rejectedElement = o;}

    String toString() {
        return(“La pile,’’ + s + “est pleine; impossible d’y ajouter le’’ + rejectedElement.getClass() +
            ‘‘,’’ + rejectedElement);}
}
```

Une seconde sous-classe de `StackException`.

```
class EmptyStack extends StackException {
  EmptyStack(Stack s2) {super(s2);}

  String toString() {
    return(“La pile,’’ + s + “est vide.’’);
      }
}
```

Signalement d’une nouvelle sorte exception, une nouvelle version des méthodes `push` et `pop` de la classe `Stack` :

```
public void push(Object o) throws FullStack {
    if (index == taille) throw new FullStack(this, o);
    buffer[index++] = o;}}

public Object pop() throws EmptyStack {
    if (index == 0) throw new EmptyStack(this);
    else {return buffer[index−−];}}
```

checked ou *unchecked* ?

Problème :

```
class A{
  void m() { ... }
}

class B extends A{
  void m() throws Exception{ ... }
}
```

Listing (1) *– m de B n’est pas une redéfinition de m de A*

Essayer d’anciper tous les cas d’exceptions au niveau des classes abstraites. Sinon utiliser les *unchecked*.

4 Traitement d’une exception en Java

Le traitement d’une exception consiste, au sein d’un handler ayant rattrapé l’exception,

- à remettre le système dans un état d’exécution standard soit en modifiant l’état des variables locales soit en

forçant l'arrêt de la fonction en cours d'éxécution (return),

- ou à propager une nouvelle exception (pattern `ConvertExceptions`) ou la même (pattern `LetExceptionsPropagate`) , après restauration éventuelles de certaines entités.

4.1 définition de handlers

Java permet d'associer un handler à tout bloc, avec les instructions `try-catch` ou `try-catch-finally` ou `try-finally`.

Ces instructions permettent de rattraper une exception puis de définir un handler à exécuter suite au rattrapage.

Exemple :

```
class testException {
  public static void main(String[] args){
    Stack testStack = new Stack();
    try {testStack.use();}
    catch (FullStack e) { ... handler ... }
    catch (EmptyStack e) { ... handler ... }
    catch (Exception e) { ... handler ... }
    finally { ... restaurations inconditionnelles }
    }
}
```

La première clause `catch` qui correspond (selon la hiérarchie des types) à l'exception rattrapée est exécutée.

Une seule clause `catch` exécutée.

4.2 Exemple Concret - version 1 - retour à une exécution standard

```
class DistributeurBonbons{
  Stack conteneur = new Stack();

Bonbon void donner(Bonbon b, int prix) {
    try{
      return(conteneur pop());
    catch (EmptyStack e) {return null;}
    }
}
```

4.3 Exemple Concret - version2

```
class DistributeurBonbons{
  Stack conteneur = new Stack();

Bonbon void donner(Bonbon b, int prix) {
    try{
      return(conteneur pop());
```

```
    catch (EmptyStack e) {throw new DistributeurVide();}
    }
}
```

4.4 Utilisation du paramètre du handler

Tout handler est une fonction à un paramètre formel. Le type du paramètre détermine quelles sont exceptions qu'il "attrapera".

Définition, **Objet exception** : l'instance crée lors du signalement, automatiquement passée en argument à tout handler.

Toutes les méthodes publiques de la classe de l'exception signalée sont utilisables dans un handler via un envoi de message à l' "objet exception" reçu en argument.

```
    try {... something ...}
    catch (Exception e) {
      e.getMessage();
      e.printStackTrace();
      ...
    }
```

4.5 Ecriture de handlers

Un handler doit corriger la situation exceptionnelle rencontrée et remettre l'exécution du programme dans un état standard. S'ensuit une liste de diverses possibilités.

4.5.1 Modification de l'état des variables

Après exécution d'une clause `catch`, l'exécution reprends à l'instruction qui suit l'instruction `try-catch`.

```
float inverse(float x){
    float result;
    try {result = 1/x;}
    catch (ArithmeticException e){result = Float.infinity;}
    return(result);
}
```

Variante possible du précédent :

```
float inverse(float x){
    float result;
    try {return 1/x;}
    catch (ArithmeticException e){return Float.infinity;}
}
```

4.5.2 Ré-essai jusqu'au succès

Pour réessayer une clause `try` jusqu'au succès, il faut insérer l'instruction `try-catch` dans une boucle. (Exemple emprunté à M.Huchard).

Note : deux sortes d'exceptions sont potentiellement signalées dans la méthode suivante, les premières (`IOException`) sont propagées aux appelants, les secondes (`NumberFormatException`) sont rattrapées et traitées localement.

```
1  public int lireEntier()throws IOException {
2      BufferedReader clavier = ;
3      int ilu = 0;
4      boolean succes = false ;
5      while (! succes) {
6          try {
7              String s = clavier.readLine();
8              ilu = Integer.parseInt(s);
9              succes = true; }
10         catch (NumberFormatException e) {
11             System.out.println("Erreur : " + e.getMessage());
12             System.out.println("Veuillez recommencer "); }
13     } // end while
14     return ilu; }
```

4.5.3 Schéma de Réutilisation utlisant les exceptions

```
1  public class GrowingStack extends Stack{

3      public void push(Object o){
4          try {super.push(o);}
5          catch(FullStack e) {this.grow(); super.push(o);}
6      }

8      protected void grow() {...}
9  }
```

4.5.4 Structuration du contrôle avec les exceptions

Pour les exceptions de domaine, on a en général le choix de laisser où ne pas laisser une exception être signalée ? Critères de choix :

- pattern *DontUseExceptionsForFlowControl*,

- mais ... le test permettant de savoir si une exception va être signalée existe-t-il ?

- ce test est-il coûteux ? est-il répété ?

Un programmeur peut choisir de préférer une exception à des tests si ceux-ci sont trop couteux ou si le code résultant est trop complexe.

Exemple d'école de calcul du pgcd par modulos successifs sans test de zézo.

```
int pgcd (int a, int b){
    int aux;
    try{while (true){
         aux = a ;
         a = b ;
         b = modulo(aux, b); //en java, le
    catch (ZeroDivide e) {return aux;}};
    return aux;
}
```

4.5.5 Signalement d'une nouvelle exception dans un handler

Pattern *ConvertExceptions* (http://wiki.c2.com/?ConvertExceptions)

C'est un cas typique de bonne programmation, on rattrape une exception de bas niveau pour en signaler une correspondant sémantiquement à la classe en cours de réalisation (voir section 4.3).

```
class DistributeurBonbons{
  Stack conteneur = new Stack();
  void donner(Bonbon b) throws YaPlusDeBonbons {
    try {conteneur.pop();}
    catch (EmptyStack e) {throw new YaPlusDeBonbons();}}}
```

C'est un cas où l'utilisation du pattern *Nested Exception* (http://wiki.c2.com/?NestedException) n'est pas appropriée.

4.5.6 Propagation explicite d'une exception

Pattern *LetExceptionsPropagate*.

Un rattrapage suivi d'une propagation explicite de la même exception a un sens si des actions de restaurations sont à effectuer.

```
File f = new File(...);
f.open();
try {f.use();}
catch (IOException e) {f.close(); throw e;}
f.close;
```

4.5.7 Restaurations inconditionnelles

Le cas précédent s'écrit plus simplement avec l'instruction `try-finally`

```
File f = new File(...);
f.open();
try {f.use();}
finally {f.close();}
```

Les instructions de la clause finally sont exécutées après celle des instructions du `try` quoi qu'il arrive durant leur exécution.

4.5.8 Mauvaises pratiques typiques

— **Handlers vides**.

voir les pattern *Empty Catch Clause* et *ReplaceEmptyCatchWithTest*

```
public int m(){
    BufferedReader clavier = ...;
    try {String s = clavier.readLine();}
    catch (IOException e) {}
```

— Ne pas rattraper une exception si on souhaite uniquement la propager,

```
public int m() throws Exception{
    try {... something ...}
    catch (Exception e) {throw e;}
```

Sauf dans les cas de handlers à plusieurs entrées.

```
public int m() throws Exception{
    try {... something ...}
    catch (SpecificException e) {throw e;}
    catch (Exception e) { ... do something ...}
```