Université Montpellier-II
UFR des Sciences - Département Informatique
Master Informatique - UE FMIN 107 -

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

Gérer les exceptions : une introduction avec support 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

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.

```
Throwable
    Error
        VirtualMachineError
        InternalError
        OutOfMemoryError
        StackOverflowError
        UnknownError
        ...
    Exception
        RuntimeException
            ArithmeticException
            ClassCastException
            ...
        ‘‘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 ?

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.

```
class Stack {
```

```
  int index = 0;
  int taille = 10;
  Object[] buffer = new Object[taille];

  void push(Object o) throws Exception {
    if (index == taille)
      throw new Exception(“La pile est pleine’’);
    buffer[index++] = o;}
}
```

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

```
    throw new Exception(“La pile est pleine’’);
```

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

le signalement proprement dit est réalisé, en Java, par la primitive `throw`, qui provoque :

- une recherche locale d'un handler et son invocation s'il y en a un,

- sinon, l'arrêt de l'exécution de la méthode signalante, puis la propagation du signalement à l'appelant (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 **??**).

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

  void ajouter(Bonbon b) throws Exception {
    ...
     conteneur push(b);
  }
}
```

3.3 Quelle sorte d'exception signaler ?

- exceptions génériques prédéfinies, voir l'exemple précédent.

- exception spécifique prédéfinies, par exemple :

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

- ou une nouvelle sorte d'exception, définie par une nouvelle classe.

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  public class abstract class StackException extends Exception { toto n toto
2    Stack s;
3    StackException(Stack s2) {s = s2;}}

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

8      public FullStackException(Stack s2, Object o) {
9          super(s2);
10         rejectedElement = o;}

12     String toString() {
13         return(“La pile,’’ + s + “est pleine; impossible d’y ajouter le’’ + rejectedElement.getClass() +
               ‘‘,’’ + rejectedElement);}
14 }
```

Une seconde sous-classe de `StackException`.

```
1  public class class EmptyStack extends StackException { toto n toto
2    EmptyStack(Stack s2) {super(s2);}

4    String toString() {
5      return(“La pile,’’ + s + “est vide.’’);
6        }
7  }
```

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

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

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

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 cohérent soit en modifiant l'état des variables soit en forçant le retour d'une valeur ad.hoc. pour la fonction en cours d'éxécution,

- à propager une nouvelle exception,

- ou à propager la même après restauration de certaines entités.

4.1 Rattrapage d'exception, 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

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

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

4.3 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();
      ...
    }
```

4.4 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.

4.4.1 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;}
}
```

4.4.2 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.

```
public int lireEntier()throws IOException {
    BufferedReader clavier = ;
    int ilu = 0;
    boolean succes = false ;
    while (! succes) {
        try {
            String s = clavier.readLine();
            ilu = Integer.parseInt(s);
            succes = true; }
       catch (NumberFormatException e) {
            System.out.println("Erreur : " + e.getMessage());
            System.out.println("Veuillez recommencer "); }
    } // end while
    return ilu; }
```

4.4.3 Réutilisation utlisant les exceptions

```
public class public class GrowingStack extends Stack{ toto n toto

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

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

4.4.4 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 :

- 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 (ArithmeticException e) {return aux;}};
    return aux;
}
```

4.4.5 Signalement d'une nouvelle exception dans un handler

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.

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

4.4.6 Propagation explicite d'une exception

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;
```

4.4.7 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.

4.4.8 Mauvaises pratiques typiques

- Ne pas rattraper l'exception si on souhaite uniquement la propager.

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

- Sauf cas exceptionnel, ne pas définir de handler vide. Cela rend la mise au point des programmes très difficile.
  Le code suivant est très fréquent dans les programmes Java, il est néanmoins à proscrire même s'il est tentant.

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