Programmation Orientée Objets : Les exceptions

Souheib Baarir[1]

[1]Université Paris Nanterre.
Souheib.baarir@parisnanterre.fr

L2 MIASHS/DL-Info-Gestion/CMI
2024/2025

Grandes lignes du cours

Problématique.

Les blocs : try-catch.

Les types d'exceptions.

- Les exceptions de type RuntimeException.
- Les exceptions contrôlées.
- Les exceptions contrôlées personnalisées.

Finally.

Propagation des exceptions.

Les erreurs d'exécution

Tout programme comporte des erreurs (**bugs**) ou est susceptible de générer des erreurs (*e.g* suite à une action de l'utilisateur, de l'environnement, etc ...).

Problème

Ces erreurs peuvent provoquer l'interruption brutale du programme !

Comment prévenir les erreurs ?

Une solution consiste dans un premier temps à « prévoir les erreurs possibles » et mettre en place :

- soit des séquences de tests pour les empêcher,
- soit un système de codes d'erreurs, c'est-à-dire un système permettant de retourner des valeurs spécifiques pour signaler un fonctionnement anormal de l'application.

Toutefois cette solution est loin d'être satisfaisante car elle rend difficile la maintenance du programme (à cause d'une trop grande imbrication de tests conditionnels (if .. else) par exemple).

La gestion d'erreurs en Java.

Le langage Java propose un mécanisme particulier pour gérer les erreurs : **les exceptions**. Ce mécanisme repose sur deux principes :

1. Les différents types d'erreurs sont modélisées par des classes.

2. Les instructions susceptibles de générer des erreurs sont séparées du traitement de ces erreurs : concept de **bloc d'essai** et de **bloc de traitement d'erreur**.

Les exceptions : définition.

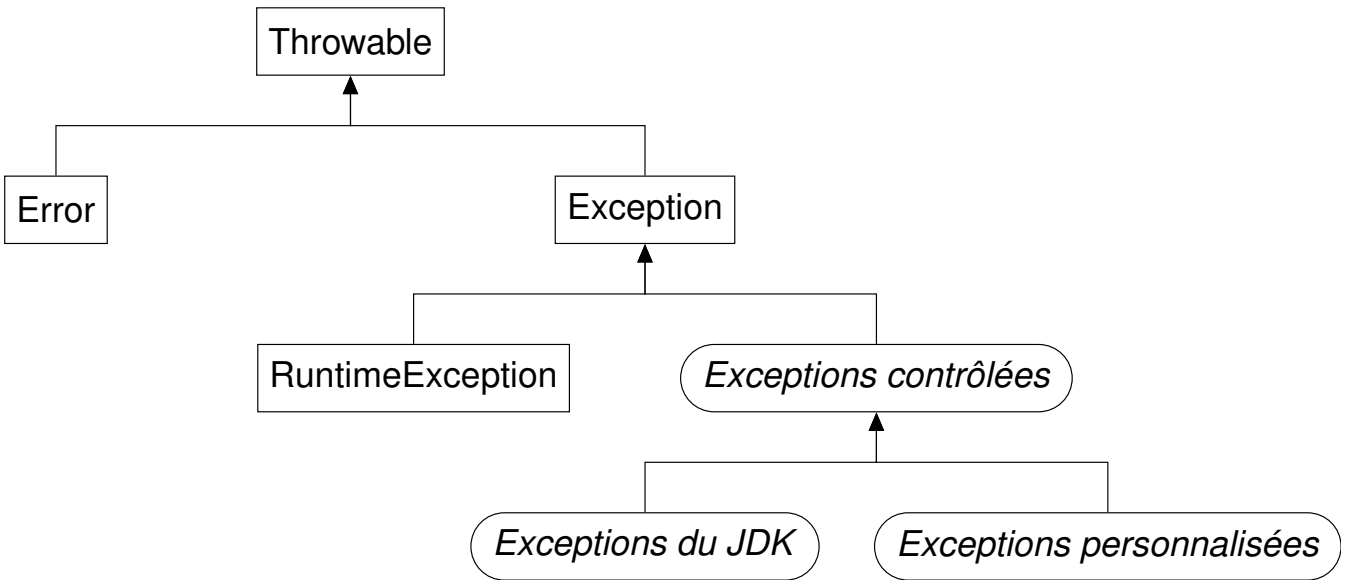
Définition

*Le terme **exception** désigne tout événement arrivant durant l'exécution d'un programme interrompant son fonctionnement normal.*

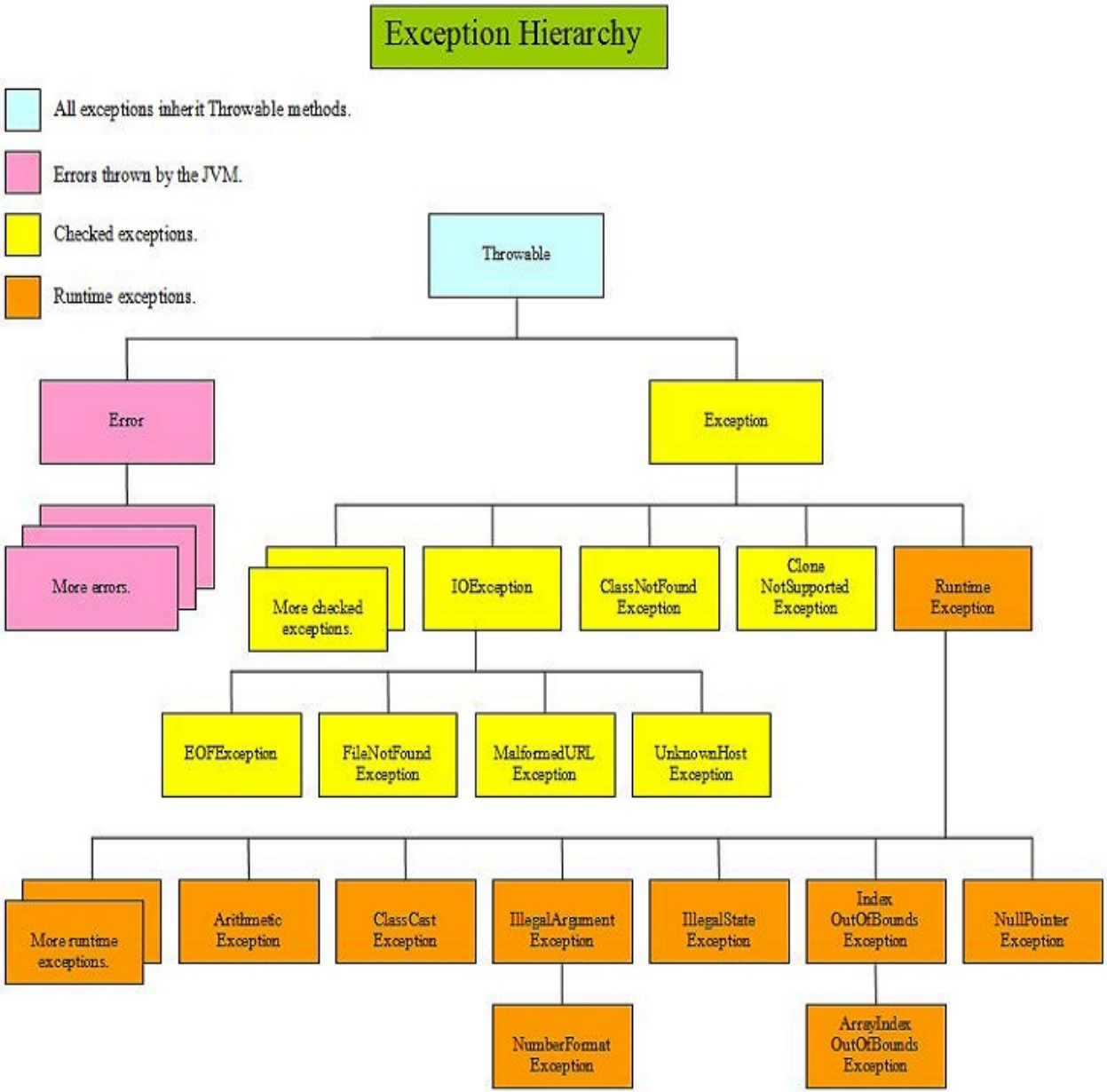
En java, les exceptions sont matérialisées par des instances de classes héritant de la classe **java.lang.Throwable**.

A chaque évènement correspond une sous-classe précise, ce qui peut permettre d'y associer un traitement approprié.

Arbre d'héritage des exceptions (1/2)



Arbre d'héritage des exceptions (2/2)



Les blocs : try-catch.

Les blocs : try.

Définition

*La clause **try** s'applique à un bloc d'instructions correspondant au fonctionnement normal mais pouvant générer des erreurs.*

```
try {
    ...
    ...
}
```

Attention

Un bloc **try** ne compile pas si aucune de ses instructions ne lance d'exception.

Les blocs : catch.

Définition

*La clause **catch** s'applique à un bloc d'instructions définissant le traitement d'un type d'erreur. Ce traitement sera lancé sur une instance de la classe d'exception passée en paramètre.*

```
try{
    ...
}
catch(TypeErreur1 e) {
    ...
}
catch(TypeErreur2 e) {
    ...
}
```

Règles sur les blocs : try-catch.

Si les blocs **try** et **catch** corresponde à deux types de traitement (resp. normal et erreur), ils n'en sont pas moins liés. Ainsi :

- Tout bloc **try** doit être suivit par au moins un bloc **catch** ou par un bloc **finally** (étudié plus loin).
- Tout bloc **catch** doit être précédé par un autre bloc **catch** ou par un bloc **try**.

Définition

*Un ensemble composé d'un bloc **try** et d'au moins un bloc **catch** est communément appellé bloc **try-catch**.*

Lancer et lever des exceptions : définitions.

Définition

Lorsqu'une instructions du bloc d'essai génère une erreur et y associe une exception, on dit qu'elle ***lance*** *cette exception.*

Définition

Lorsqu'un bloc de traitement d'erreur est déclenché par une exception, on dit qu'il ***lève*** *cette exception.*

Les bloc try-catch : fonctionnement.

Le fonctionnement d'un bloc **try-catch** est le suivant :

1. si aucune des instructions du bloc d'essai ne lance d'exception, il est entièrement exécuté et les blocs de traitement d'erreur sont ignorés.

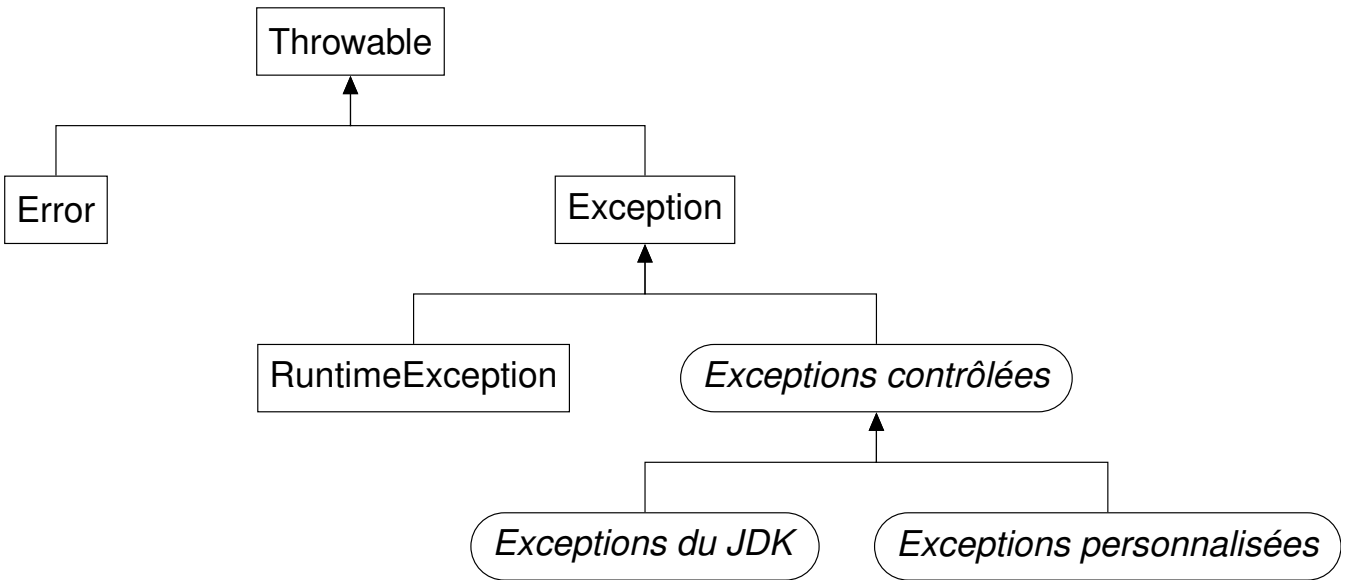
2. si une des instructions du bloc d'essai lance une exception, alors toutes les instructions du bloc d'essai après elle sont ignorées et le premier bloc de traitement d'erreur correspondant au type d'exception lancée. Tous les autres blocs de traitement d'erreur sont ignorés.

Les types d'exceptions.

Outline

Les exceptions de type RuntimeException.

Les exceptions de type RuntimeException.



Les exceptions de type RuntimeException.

Définition

Les exceptions de type ***RuntimeException*** *correspondent à des erreurs qui peuvent survenir dans toutes les portions du codes.*

Java définit de nombreuses sous-classe de `RuntimeException` :

- `ArithmeticException` : division par zéro (entiers), etc ....
- `IndexOutOfBoundsException` : dépassement d'indice dans un tableau.
- `NullPointerException` : référence **null** alors qu'on attendait une référence vers une instance.
- etc...

Remarque(s)

Attention `1.0d/0.0d` renvoie `Infinity` !

RuntimeException : exemple.

Exemple

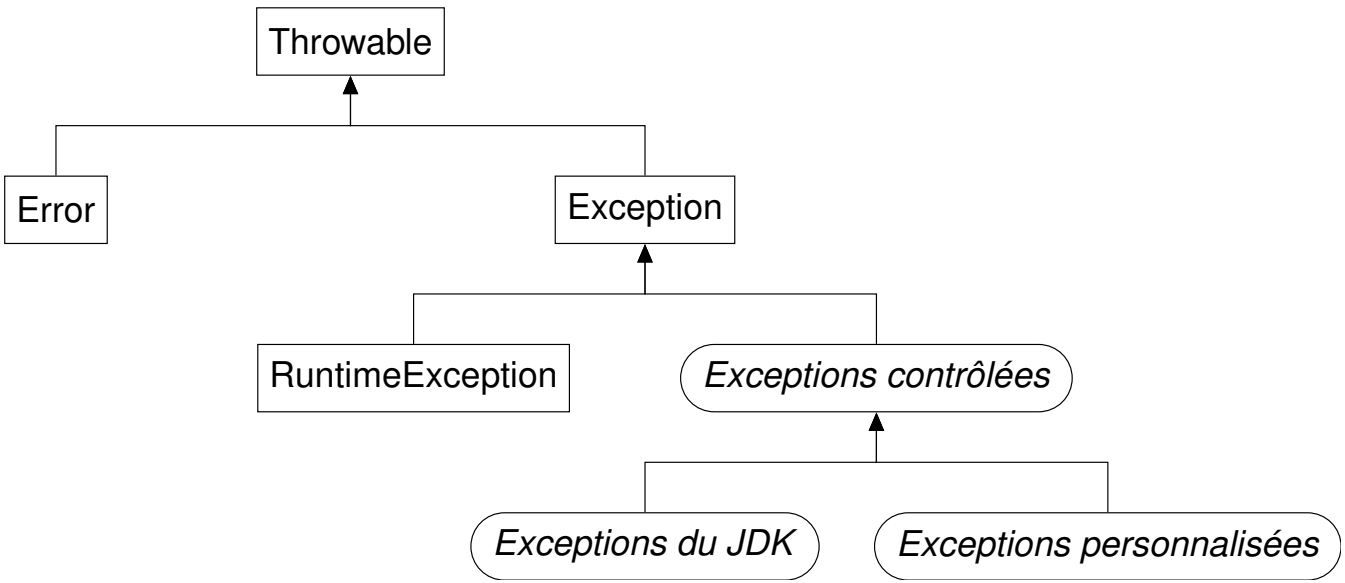
```
int a = 0;
try {
  int x = 1 /a;
  System.out.println("␣x␣=␣" + x);
}
catch (ArithmeticException e) {
  System.out.println("division␣par␣0␣:␣1␣/␣" + a);
}
```

```
division par 0 : 1 / 0
```

Outline

Les exceptions contrôlées.

Les exceptions contrôlées.



Les exceptions contrôlées.

Définition

On appelle ***exception contrôlée****, toute exception qui hérite de la classe* `Exception` *et qui n'est pas une* `RuntimeException`*. Elle est dite contrôlée car le compilateur vérifie que toutes les méthodes l'utilisent correctement.*

Le JDK définit de nombreuses exceptions :

- `EOFException` : fin de fichier.
- `FileNotFoundException` : erreur dans l'ouverture d'un fichier.
- `ClassNotFoundException` : erreur dans le chargement d'une classe.
- etc...

Le contrôle des exceptions : throws.

Toute exception contrôlée, du JDK ou personnalisée, pouvant être émise dans une méthode doit être :

- soit levée dans cette méthode. Elle est alors lancée dans un bloc try auquel est associé un catch lui correspondant.
- soit être indiquées dans le prototype de la méthode à l'aide du mot clé **throws**.

Le contrôle des exceptions : exemple.

```
import java.io.*;
import java.util.Scanner;
public class ReadFile {
  public static void main(String[] args) {
   FileReader f = new FileReader(args[1]);
   BufferedReader reader = new BufferedReader(f);
   Scanner sc = new Scanner(reader);
   while((sc.hasNext())){
            String line=sc.nextLine();
       System.out.println(line);
       if (line.compareTo(new String("ok"))==0)
         break;
   } } }
```

Compilation : Unhandled exception type FileNotFoundException.

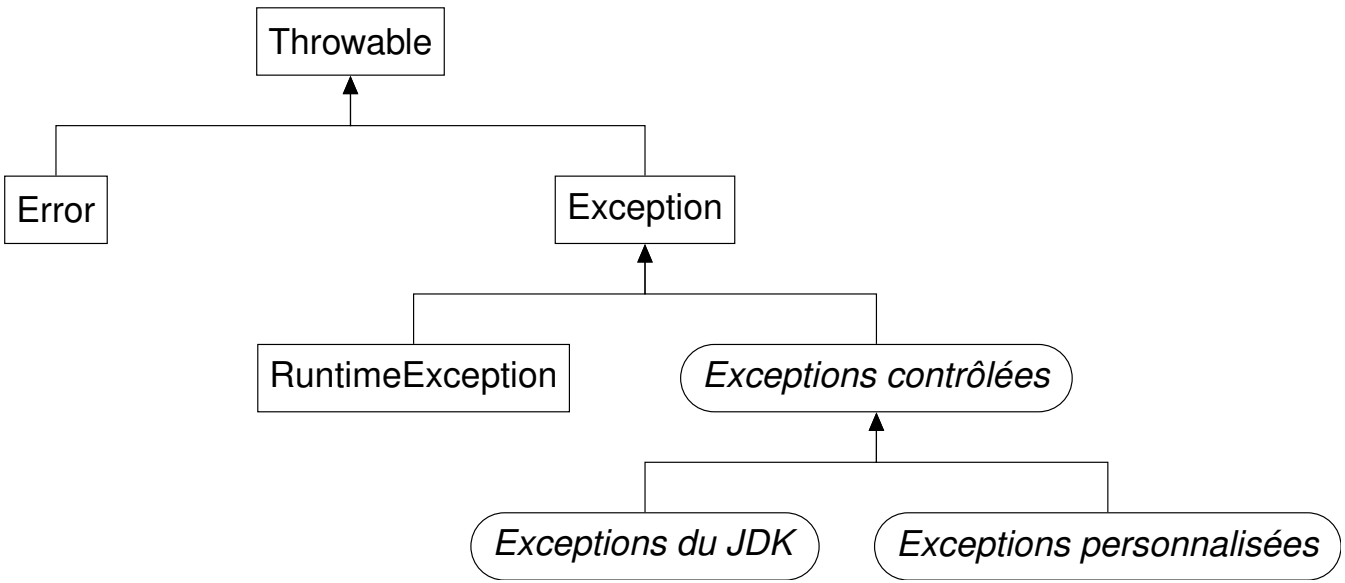
Le contrôle des exceptions : correction exemple.

```
import java.io.*;
import java.util.Scanner;
public class ReadFile {
  public static void main(String[] args) {
   try{
        FileReader f = new FileReader(args[1]);
        BufferedReader reader = new BufferedReader(f);
        Scanner sc = new Scanner(reader);
        while((sc.hasNext())) {
              String line=sc.nextLine();
          System.out.println(line);
          if (line.compareTo(new String("ok"))==0)
            break;
        }
    } catch (FileNotFoundException e) {
       System.out.println("le fichier n'est pas trouvé");
 }
}
```

Outline

Les exceptions contrôlées personnalisées.

Les exceptions contrôlées personnalisées.



Les exceptions contrôlées personnalisées.

On peut définir ses propres exceptions en définissant une sous-classe de la classe **Exception**.

Exemple

```
public class MonException extends Exception {
  private int x;
  public MonException(int x) {
    this.x = x;
  }
  public String toString() {
    return "valeur incorrecte : " + x;
  }
}
```

Lancer une exception

Définition

*Pour lancer une exception, on peut utiliser la clause **throw**.*

Exemple

```
try {
   int x = -1;
   if (x < 0) {throw new MonException(x);}
   x = x + 3;
   System.out.println(x);
}
catch (MonException e) {
   System.err.println(e);
}
```

```
valeur incorrecte : -1
```

Finally.

La clause finally : définition.

Définition

*La clause **finally** définit un bloc d'instruction qui sera exécuté même si une exception est lancée dans le bloc d'essai. Elle permet de forcer la bonne terminaison d'un traitement en présence d'erreur, par exemple : la fermeture des fichiers ouverts.*

```
try {
    ...
}
catch (...) {
    ...
}
finally {
    ...
}
```

La clause finally : quand ?

Le code de la clause **finally** sera toujours exécuté :

- si la clause *try* ne lève pas d'exception : exécution après le *try* (même s'il contient un *return*).
- si la clause *try* lève une exception traité par un *catch* : exécution après le *catch* (même s'il contient un *return*).
- si la clause *try* lève une exception non traité par un *catch* : exécution après le lancement de l'exception.

Remarque(s)

Attention : Un appel à la méthode *System.exit()* dans le bloc *try* ou dans un bloc *catch* arrête l'application sans passer par la clause *finally*.

La clause finally : exemple.

Exemple

```
try {
   int x = -1;
   if (x < 0) {throw new MonException(x);}
   System.out.println(x);
}
catch (MonException e) {
   System.err.println(e);
}
finally {
   System.out.println("Tout est bien qui ...");
}
```

```
valeur incorrecte : -1
Tout est bien qui ...
```

Propagation des exceptions.

Pourquoi propager les exceptions ?

Plus une méthode est éloignée de la méthode **main** dans la pile d'exécution, moins elle a de vision globale de l'application. Une méthode peut donc avoir du mal à savoir traiter une exception.

Il est souvent difficile pour une méthode effectuant un petit traitement de décider si une exception est critique :

> *"Il vaut mieux laissé remonter une exception que de décider localement d'interrompre l'ensemble du programme."*

Comment faire remonter une exceptions ?

Pour traiter une exception, on distingue finalement trois méthodes :

1. **Traitement local :** l'exception est levée dans la méthode .
2. **Traitement délégué :** l'exception, qui n'est pas levée dans la méthode, est transmise à la méthode appelante.
3. **Traitement local partiel :** l'exception est levée dans la méthode, puis re-lancée pour être transmise à la méthode appelante.

Traitement délégué : exemple

```
public void f() throws MonException {
    throw new MonException();
}

public void g() throws MonException {
  f();
}

public void h() {
  try {
   g();
  }
  catch (MonException e) {
    System.err.println("Traitement délégué à h");
  }
}
```

```
Traitement délégué à h
```

Traitement local partiel : exemple

```
public void f() throws MonException {
  try {
    throw new MonException();
  }
  catch (MonException e) {
    System.err.println("Traitement local par f");
    throw new MonException();
  }
}
public void g() {
  try { f(); }
  catch (MonException e) {
    System.err.println("Traitement délégué à g");
  }
}
```

```
Traitement local par f
Traitement délégué à g
```

Et si aucune méthode ne lève l'exception ?

Si une exception remonte jusqu'à la méthode **main** sans être traitée par cette méthode :

- l'exécution du programme est stoppée,
- le message associé à l'exception est affiché, avec une description de la pile des méthodes traversées par l'exception

Remarque(s)

Seul le thread qui a généré l'exception non traitée meurt ; les autres threads continuent leur exécution.