

# Programmation orientée objet : Les exceptions

E. Hyon, V. Bouquet<sup>1</sup>

<sup>1</sup>[valentin.bouquet@parisnanterre.fr](mailto:valentin.bouquet@parisnanterre.fr)

Licence MIASHS Miage - 2023/2024

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

- Les exceptions de type RuntimeException.

- Les exceptions contrôlées.

- Les exceptions contrôlées personnalisées.

Propagation des exceptions.

# Problématique.

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

- Les exceptions de type RuntimeException.

- Les exceptions contrôlées.

- Les exceptions contrôlées personnalisées.

Propagation des exceptions.

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).

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**.

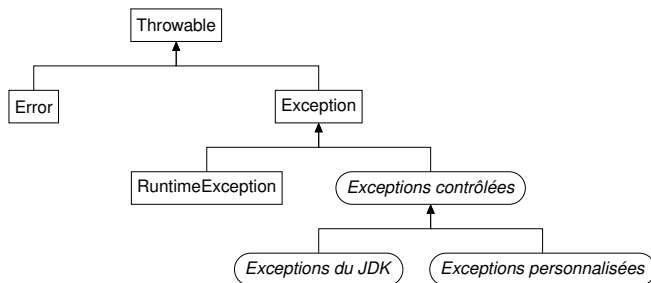
## Définition

*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)





# Les blocs : try-catch.

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

- Les exceptions de type RuntimeException.

- Les exceptions contrôlées.

- Les exceptions contrôlées personnalisées.

Propagation des exceptions.

## Définition

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

```
try {  
    ... suite d instructions ....  
}
```

## Attention

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

Exception is never thrown in body of corresponding try  
statement

## 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(TypeErreur1Exception e) {  
    ...  
}  
catch(TypeErreur2Exception e) {  
    ...  
}
```

# Les blocs try and catch : un exemple

Le code suivant provoque une exception :

```
String s = null ;  
int l = s.length();
```

On le place dans un bloc try and catch :

```
String s = null ;  
try {  
    int l = s.length();  
    System.out.println("longueur"+l);  
}  
catch (NullPointerException e){  
    System.out.println("je_viens_d_attraper");  
    System.out.println("une_exception");  
    e.printStackTrace();  
}
```

# Règles sur les blocs : try-catch.

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

- ▶ Tout bloc **try** doit être suivi 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 appelé bloc **try-catch**.*

## 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 blocs 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 est traité. Tous les autres blocs de traitement d'erreur sont ignorés.

# Les blocs try-catch : exemple

Le code suivant

```
String f = new String("chaîne");
try {
    int l = f.length();
    System.out.println("longueur_ "+l);
}
catch (NullPointerException e){
    System.out.println("une_ exception");
}
System.out.println("je_ continue");
System.out.println(f);
```

affiche :

```
longueur 6
je continue
chaîne
```



# Les blocs try-catch : exemple

Le code suivant

```
String f = null;  
try {  
    int l = f.length();  
    System.out.println("longueur_"+l);  
}  
catch (IndexOutOfBoundsException e){  
    System.out.println("une_exception_IOB");  
}  
catch (NullPointerException e){  
    System.out.println("une_exception_NP");  
}  
System.out.println("je_continue");
```

affiche :

une exception NP

je continue

## Les blocs try-catch : exemple II

Le code suivant

```
String f = "bon";  
try {  
    int l = f.length();  
    System.out.println("longueur_" + l);  
    System.out.println("case_" + f[l]);  
}  
catch (IndexOutOfBoundsException e){  
    System.out.println("une_exception_IOB");  
}  
catch (NullPointerException e){  
    System.out.println("une_exception_NP");  
}  
System.out.println("je_continue");
```

affiche :

longueur 3

une exception IOB

je continue

# Finally.

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

- Les exceptions de type RuntimeException.

- Les exceptions contrôlées.

- Les exceptions contrôlées personnalisées.

Propagation des exceptions.

# La clause finally : définition.

## Définition

*La clause **finally** définit un bloc d'instructions qui sera toujours exécuté, qu'une exception soit levée ou non. 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 lance pas d'exception : exécution après le `try` (**même** s'il contient un `return`).
- ▶ si la clause `try` lance une exception traitée par un `catch` : exécution après le `catch` (même s'il contient un `return`).
- ▶ si la clause `try` lance une exception non traitée 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

Un exemple générique avec plusieurs comportements possibles

```
public methodeLecture(String args) {  
    String[] line=new String[**];  
    line[0]="Coucou";  
    line[1]=**;  
    try {  
        System.out.println(line[0]);  
        if (line[1].compareTo(new String("ok"))==0)  
            System.out.println("break");  
    }  
    catch (NullPointerException e){  
        System.out.println("aie_aie_aie");  
    }  
    finally {  
        System.out.println("I_am_here");  
    }  
    System.out.println("Je_continue");  
}
```

# La clause finally : Exemple suite

## Execution 1

Si `** = 2` et `*** = "ok"` alors le code précédent est

```
public methodeLecture(String args) {  
    String[] line=new String[2];  
    line[0]="Coucou";  
    line[1]="OK";  
    try {  
        System.out.println(line[0]);  
        if (line[1].compareTo(new String("ok"))==0)  
            System.out.println("break");  
    }  
    catch(NullPointerException e){  
        System.out.println("aie_aie_aie");  
    }  
    finally{  
        System.out.println("I_am_here");  
    }  
    System.out.println("Je_continue");  
}
```

# La clause finally : Exemple suite

## Execution 1

A l'exécution il s'affiche

Coucou

break

I am Here

Je continue



# La clause finally : Exemple suite

## Execution 2

Si `** = 2` et `*** = null` alors le code précédent est

```
public methodeLecture(String args) {  
    String[] line=new String[2];  
    line[0]="Coucou";  
    line[1]=null;  
    try {  
        System.out.println(line[0]);  
        if (line[1].compareTo(new String("ok"))==0)  
            System.out.println("break");  
    }  
    catch(NullPointerException e){  
        System.out.println("aie_aie_aie");  
    }  
    finally{  
        System.out.println("I_am_here");  
    }  
    System.out.println("Je_continue");  
}
```

# La clause finally : Exemple suite

## Execution 2

A l'exécution il s'affiche

Coucou

aie aie aie

I am here

Je continue

# La clause finally : Exemple suite

## Execution 3

Si `** = 1` et la ligne `*** = null` est supprimée alors le code précédent est

```
public methodeLecture(String args) {  
    String[] line=new String[1];  
    line[0]="Coucou";  
    try {  
        System.out.println(line[0]);  
        if (line[1].compareTo(new String("ok"))==0)  
            System.out.println("break");  
    }  
    catch(NullPointerException e){  
        System.out.println("aie_aie_aie");  
    }  
    finally{  
        System.out.println("I_am_here");  
    }  
    System.out.println("Je_continue");  
}
```

# La clause finally : Exemple suite

## Execution 3

A l'exécution il s'affiche

Coucou

I am here

**N.B.** Dans ce dernier cas l'exception n'est pas levée.

# La clause finally : Exemple de gestion de ressource

```
public methodeLecture(String args) {  
    BufferedReader reader;  
    String line;  
    try {  
        reader = new BufferedReader(new FileReader(args));  
        try{  
            while((line=reader.nextLine())!=null){  
                System.out.println(line);  
                if (line.compareTo(new String("ok"))==0)  
                    break;  
            }  
            finally{ reader.close();}  
        }  
    } catch (FileNotFoundException e){  
        e.printStackTrace();  
    }  
}
```

# Les types d'exceptions.

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

- Les exceptions de type RuntimeException.

- Les exceptions contrôlées.

- Les exceptions contrôlées personnalisées.

Propagation des exceptions.

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

- Les exceptions de type RuntimeException.

- Les exceptions contrôlées.

- Les exceptions contrôlées personnalisées.

Propagation des exceptions.

# Les exceptions de type RuntimeException.

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

- Les exceptions de type RuntimeException.

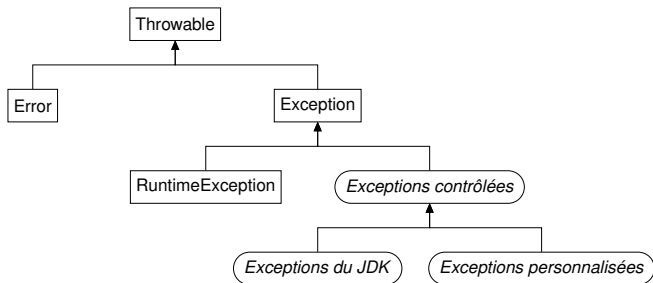
- Les exceptions contrôlées.

- Les exceptions contrôlées personnalisées.

Propagation des exceptions.



# 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. Le compilateur n'oblige pas le programmeur à les traiter.*

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 / 0");
}
```

division par 0 : 1 / 0

Problématique.

Les blocs : try-catch.

Finally.

**Les types d'exceptions.**

Les exceptions de type RuntimeException.

**Les exceptions contrôlées.**

Les exceptions contrôlées personnalisées.

Propagation des exceptions.

# Les exceptions contrôlées.

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

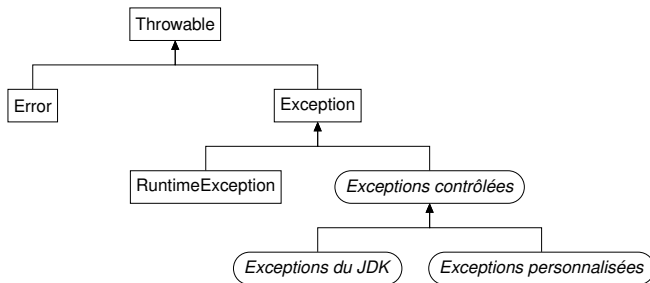
Les exceptions de type RuntimeException.

**Les exceptions contrôlées.**

Les exceptions contrôlées personnalisées.

Propagation des exceptions.

# 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 savent la gérer 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]);
        Scanner sc = new Scanner(f);
        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.

```
public class ReadFile {  
    public static void main(String[] args) {  
        try{  
            FileReader f = new FileReader(args[1]);  
            Scanner sc = new Scanner(f);  
            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 trouve");  
        }  
    }  
}
```

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

- Les exceptions de type RuntimeException.

- Les exceptions contrôlées.

- Les exceptions contrôlées personnalisées.

Propagation des exceptions.

# Les exceptions contrôlées personnalisées.

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

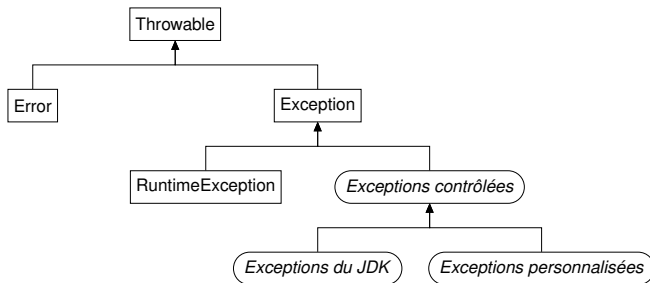
- Les exceptions de type RuntimeException.

- Les exceptions contrôlées.

- Les exceptions contrôlées personnalisées.

Propagation des exceptions.

# 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;  
    }  
}
```

## 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

# Propagation des exceptions.

Problématique.

Les blocs : try-catch.

Finally.

Les types d'exceptions.

- Les exceptions de type RuntimeException.

- Les exceptions contrôlées.

- Les exceptions contrôlées personnalisées.

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 laisser 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.

- ▶ `throws` (avec un `s`).  
Dans l'en tête d'une méthode.  
Indique que la méthode est susceptible de lancer des exceptions. Attention pour les exceptions non contrôlées cela n'est pas la peine de l'indiquer.
- ▶ `throw` (sans `s`).  
Dans un bloc d'instructions.  
Attention le mot clef `throw` provoque une rupture de séquence (et un branchement vers une gestionnaire d'exceptions).

## Traitement délégué : exemple

```
public void f() throws MonException {  
    throw new MonException(3);  
}  
  
public void g() throws MonException {  
    f();  
}  
  
public void h() {  
    try {  
        g();  
    }  
    catch (MonException e) {  
        System.err.println("Traitement_délegue_a_h");  
    }  
}
```

Traitement délégué à h

## Traitement local partiel : exemple

```
public void f() throws MonException {  
    try {  
        throw new MonException(1);  
    }  
    catch (MonException e) {  
        System.err.println("Traitement_local_par_f");  
        throw new MonException(2);  
    }  
}  
  
public void g() {  
    try { f(); }  
    catch (MonException e) {  
        System.err.println("Traitement_delegue_a_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.