Chapitre 10 : exceptions

Gestion traditionnelle des erreurs
Exception : définition
Levée d'une exception
Traitement d'une exception
Propagation d'une exception

Exceptions

Introduction

- Une exception traduit un comportement exceptionnel, une erreur, survenant pendant l'exécution d'un programme
- Traditionnellement, en cas d'erreur, le programme s'interrompt définitivement avec l'affichage d'un message à l'écran décrivant l'erreur
- L'idée est alors de munir les langages de programmation d'un mécanisme permettant de gérer les erreurs sans que le programme s'arrête définitivement
- Il devient possible d'exécuter un morceau de code spécifique au traitement de l'erreur

Qu'est-ce ?

- Une exception est une **erreur** ou bien une **action inattendue** qui se produit à **l'exécution**
- Une exception se traduit par un **signal** qui **interrompt** le déroulement normal du programme
- En Java, ce **signal** est accompagné d'un **objet** qui identifie l'exception

Exceptions prédéfinies

- Elles sont représentées par des classes
- Elles reflètent les cas d'erreur les plus courants
  - `IndexOutOfBoundsException`
  - `NullPointerException`
  - `ArithmeticException`
- Le programmeur peut lui-même créer ses propres classes d'exception pour gérer les cas d'erreurs qu'il a identifiées

Gestion des erreurs : point de vue théorique

- Un **programme** correspond au calcul d'une **fonction** mathématique
- Tout programme établit une correspondance entre un **ensemble de définition** (les données du programme) et un ensemble **image ou co-domaine** (les résultats)
- Cette assimilation ne vaut que pour les **fonctions totales**
- Rien interdit à un programme de s'exécuter sur des valeurs pour lesquelles la fonction n'est pas définie (ex:division par zéro)

Exceptions et Langages de Programmation

- Pour construire des programmes **robustes**, il faut que les langages de programmation soient capables de prendre en compte le cas des données exceptionnelles pour lesquelles la fonction n'est pas définie.
- Le mécanisme des exceptions est présent dans les langages modernes : C++, Ada, Java, Eiffel, etc ...

Exemple

- Soit un programme de saisie de données numériques comprises dans un certain intervalle.
- Pour être robuste, le programme doit déterminer si les valeurs tapées par l'utilisateur sont bien comprises dans cet intervalle
- Dans le cas contraire, il doit proposer un traitement alternatif. Ici, il pourra demander une nouvelle saisie des données

Gestion traditionnelle des erreurs (1)

- Lecture d'un entier

```
Scanner in = new Scanner(System.in);
int x = in.nextInt();
```

- Pour vérifier que la donnée lue est valide, une solution est de modifier la méthode `nextInt()` en lui ajoutant un paramètre

```
Boolean etat = new Boolean();
int x = in.nextInt(etat);
if ( !etat ) // erreur de saisie
```

Gestion traditionnelle des erreurs inconvénients

- Il faut ajouter un paramètre supplémentaire à `nextInt()`
- Déclarer une variable supplémentaire dans le code : `etat`
- ajouter un test de cette variable alors que dans la grande majorité des cas elle aura pour valeur `true`

Gestion traditionnelle des erreurs inconvénients

```
int saisirEntier( ...,Boolean etat )
{
   int x = in.nextInt( etat );
  if ( !etat ) return 0;
  // traitement normal
}

// l'état est propagé avec adjonction
// d'un nouveau paramètre à l'appelant
```

Définir une exception

- Définir une exception, c'est définir une nouvelle classe
- 3 manières :
  - par extension de la classe `Error`
  - par extension de la classe `Exception`
  - par extension de la classe `RuntimeException`
- Exemples

```
class Critique extends Error
class ValeurNegative extends Exception
class HorsLimite extends RuntimeException
```

La classe `java.lang.Error`

- Les exceptions définies par extension de la classe `Error` représentent des erreurs critiques
- Elles ne sont pas normalement gérés par le programmeur
- Exemple :
  - l'exception `OutOfMemoryError` est déclenchée lorsqu'il n'y a plus de mémoire disponible
  - l'exception `StackOverflowError` est levée en cas de dépassement de capacité de pile à la suite d'un trop grand nombre d'appels récursifs

La classe `java.lang.Exception`

- Elle représente les erreurs qui typiquement doivent être gérées par le programme

- Exemple :
  la class `IOException`, définie par extension de la classe `Exception` traduit les erreurs lors d'une entrée/sortie

- Quelques éléments de son interface:
  constructeur avec paramètre
  ```
  public Exception(String message)
  ```
  méthodes
  ```
  public String getMessage()
  public void printStackTrace()
  ```

Exemple de classe d'exception

```
public class DivisionParZero extends Exception
{
  public DivisionParZero()
  {
    super( "division par zéro" );
  }
  public DivisionParZero( String msg )
  {
    super( msg );
  }
}
```

La classe `java.lang.RuntimeException`

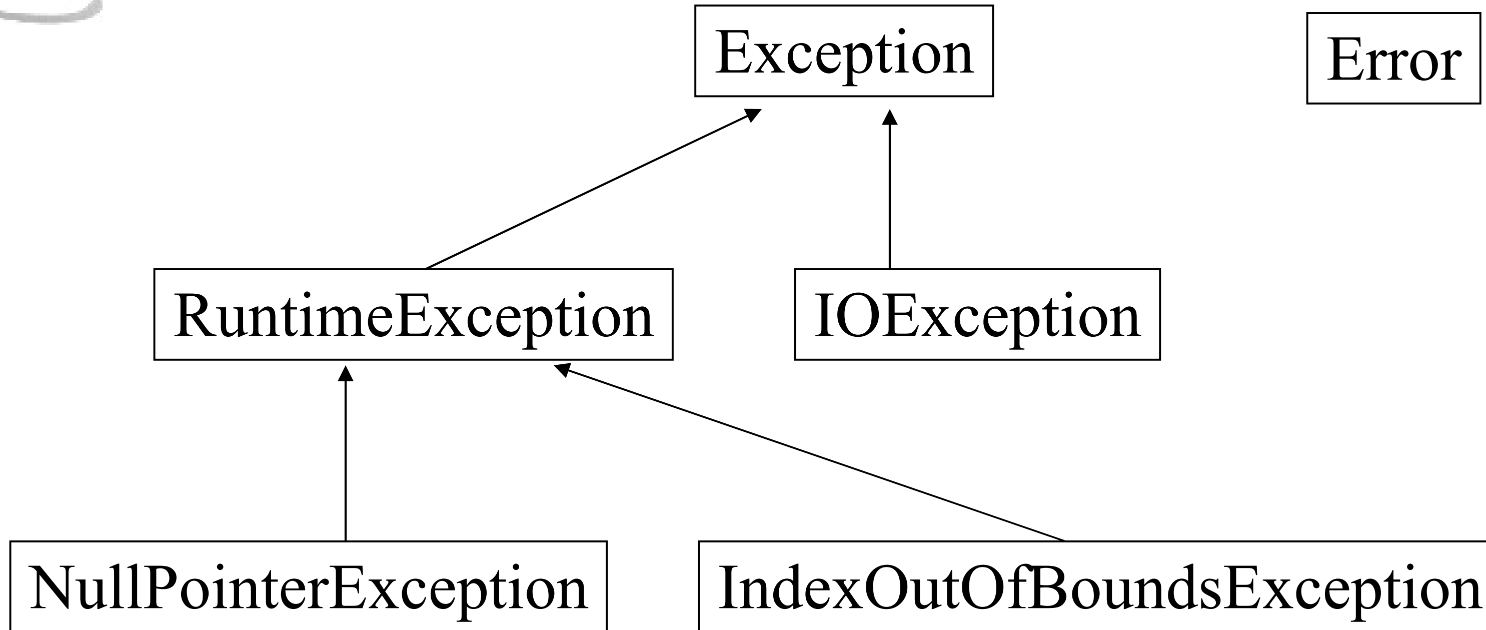
- Les exceptions définies par extension de la classe `RuntimeException` représentent des erreurs pouvant éventuellement être gérées par le programmeur
- Un grand nombre d'entre elles sont prédéfinies
- Exemples :

  `NullPointerException` est levée si l'on tente d'accéder à un objet ou un tableau de valeur `null`

  `IndexoutOfBoundsException` est levée lorsqu'un indice de tableau est hors limite

  `NumberFormatException` indique qu'une chaîne de caractères ne peut être traduite en une valeur numérique

  `NoSuchMethodException` est levée lorsqu'une méthode ne peut être trouvée


Exception
Error
RuntimeException
IOException
NullPointerException
IndexOutOfBoundsException

Mécanisme des exceptions(1/2)

- Intérêt :

  sépare dans le code les cas normaux des cas exceptionnels

- Opération :

  la levée (`throw`) est la seule opération possible.

  la levée d'une exception produit l'envoi d'un signal d'interruption accompagné d'un objet spécifique

La clause `throw`

- Permet de *lever* une exception déclarée par le programmeur
- Lorsque la clause **`throw`** est exécutée, le programme est interrompu et un objet du type de l'exception est créé et propagé
- Si l'objet exception est récupéré par un bloc **`catch`**, le code de ce bloc est exécuté, le contrôle est ensuite passé aux instructions qui suivent ce bloc

Levée d'une exception

- Définition préalable d'une classe d'exception ou utilisation d'une classe prédéfinie
  - `class HorsLimite extends Exception`
- Nécessite l'instanciation de cette classe
  - `new HorsLimite()`
- Levée de l'exception
  - `throw new HorsLimite();`

```
// le programme s'arrête, les instructions qui
// suivent jusqu'à la fin du programme ne sont
// pas exécutées à moins qu'un traitement soit
// prévue
```

Exemple (1/2)

```
class Stop extends RuntimeException{}
import java.util.Scanner;
public class Arret{
  public static void main(String[] args)
                                       throws Stop{
    Scanner in = new Scanner(System.in);
    int x = in.nextInt();
    System.out.println("instruction 1");
    if(x>0) throw new Stop();
    System.out.println("instruction 2");
    System.out.println("instruction 3");
  }
}
```

Exemple (2/2)

Le programme `Arret` s'exécute

L'opérateur saisit la valeur `7` pour la variable `x`

L'exception `Stop` est levée

L'affichage suivant apparaît sur l'écran :

```
instruction 1
Exception in thread "main" Stop
        at Arret.main(Arret.java:6)
```

On constate que les instructions 2 et 3 ne sont pas exécutées et que le programme se termine à la ligne 6 du fichier

Mécanisme des exceptions(2/2)

- Chaque méthode susceptible de lever une exception dans son corps assume une responsabilité vis à vis de celle-ci

- De 2 manières possibles :
  - traiter l'exception : un traitement peut être associé à une exception (**`catch`**). Dans ce cas, le programme ne s'interrompt pas. Il continue son exécution après le bloc **`catch`**.
  - propager l'exception : vers la méthode appelante si aucun traitement n'est prévu. La méthode doit alors déclarer cette propagation dans l'en-tête de la méthode (clause **`throws`**). Si la propagation continue jusqu'à l'exécutif java, le programme s'interrompt avec l'affichage d'un message d'erreur banalisé

Exemple : levée avec traitement

```
public void errone(){
   try{
// morceau de code dans lequel l'exception ErreurException
// pourrait être levée
   }
   catch( ErreurException objet ){
   // code de traitement de l'exception
}
```

Dès qu'une exception est levée dans le bloc **`try`**, le contrôle est immédiatement passé au bloc **`catch`** correspondant.

Exemple : levée avec propagation

```
public void errone()throws ErreurException{
   try{
// morceau de code dans lequel l'exception
// ErreurException pourrait être levée
   }
}
```

Exceptions contrôlées

- Elles dérivent de la classe `Exception`
  - si elles ne sont pas traitées dans la méthode dans laquelle elle apparaissent, elles doivent être déclarées par la clause **`throws`** dans l'en-tête de la méthode
  - elles sont analysées par le compilateur. Lorsqu'une méthode comporte un **`throw`**, il vérifie que l'exception levée *est traitée* ou bien *déclarée*. Sinon, il génère une erreur.

Exceptions non contrôlées

- Elles n'ont pas besoin d'être déclarées dans un clause **`throws`** car elles sont trop nombreuses
- Elles dérivent de la classe `RuntimeException`
- Elles peuvent apparaître n'importe où
- Exemples :

```
NullPointerException,
IndexoutOfBoundsException,
IllegalStateException,
InputMismatchException, ...
```

Levée implicite d'une exception non contrôlée

L'exception peut être déclenchée par une méthode importée. La documentation de cette méthode nous indique le type de ou des exception(s) susceptible(s) d'être levée(s)

Dans ce cas, l'instruction `throw` appartient à cette méthode

Il nous appartient ensuite de la traiter ou de la laisser se propager

`public int nextInt()`

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `int` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

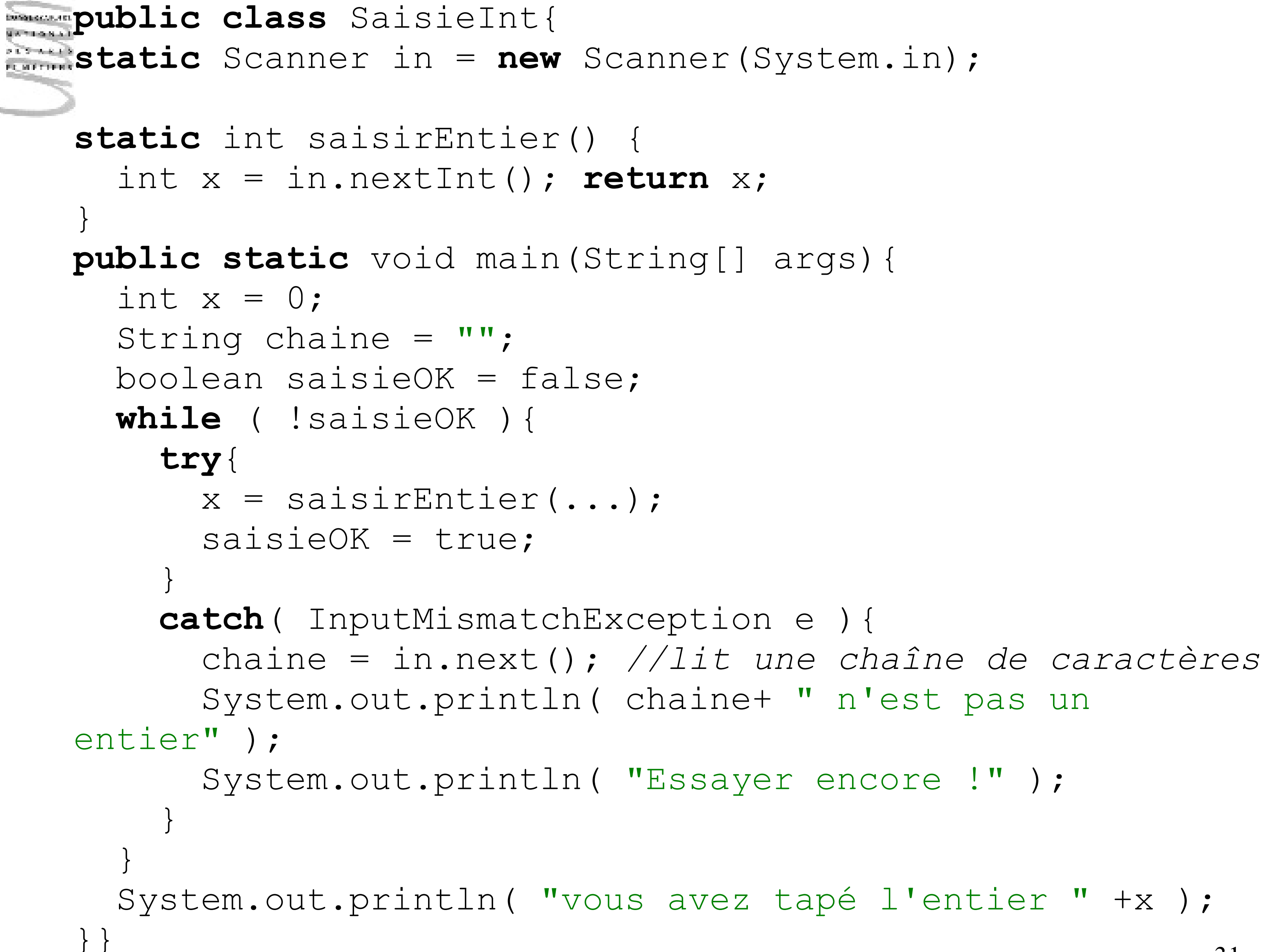
`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

Traitement d'un exception non contrôlée

```
int x;
try{
  x = in.nextInt();
}
catch( InputMismatchException e ){
  // code de traitement de l'exception
}
```

La méthode `nextInt()` est susceptible de lever (**`throw`**) une exception de type `InputMisMatchException` si la chaîne lue n'est pas un entier valide.
Dans ce cas, aucun caractère du flux d'entrée n'est consommé.
La variable `x` n'a subi aucun changement



```
public class SaisieInt{
static Scanner in = new Scanner(System.in);

   static int saisirEntier() {
      int x = in.nextInt(); return x;
   }
   public static void main(String[] args){
      int x = 0;
      String chaine = "";
      boolean saisieOK = false;
      while ( !saisieOK ){
         try{
            x = saisirEntier(...);
            saisieOK = true;
         }
         catch( InputMismatchException e ){
            chaine = in.next(); //lit une chaîne de caractères
            System.out.println( chaine+ " n'est pas un 
entier" );
            System.out.println( "Essayer encore !" );
         }
      }
      System.out.println( "vous avez tapé l'entier " +x );
}}
```

Propagation d'une exception

- Si une exception levée pendant l'exécution d'une méthode n'est pas traitée, elle est propagée vers la méthode appelante
- La méthode appelante qui reçoit le signal est elle-même interrompue
- Il appartient à la méthode appelante de traiter ou bien de propager de nouveau cette exception .

Propagation d'une exception non contrôlée

```
class Exemple{
  static Scanner in = new Scanner( System.in );

  public static String lire() {
    in.close();
    return in.next();
    // l'exception IllegalStateException est potentiellement
    // levée par la méthode next() si le scanner est fermé, et
    // propagée vers la méthode appelante lire qui la propage
    // vers la méthode main puis enfin vers l'exécutif java
  }
}

public class Client2{
    public static void main( String[] args ){
      System.out.println("j'ai lu : "+ Exemple.lire());
    }
}
```

Levée d'exception contrôlée

- L'exception est représentée par une sous-classe de `Exception` ou bien par la classe `Exception` elle-même.
- L'instruction `throw` est accompagnée d'une instance de la classe d'exception qui correspond à l'erreur détectée.

Levée et traitement d'une exception contrôlée

```
public void diviser(){
  try{
    System.out.println( "numérateur" );
    double num = in.nextDouble();
    System.out.println( "dénominateur" );
    double denom = in.nextDouble();
    if ( denom == 0 )
      throw new DivisionparZero();
    double quotient = num/denom;
    System;out.println(num+ "/" +denom+ "=" +quotient);
  }
  catch( DivisionParZero e ){
   System.out.println( e.getMesssage() );
  }
}
```

Levée et propagation d'une exception contrôlée

```
public void diviser()throws DivisionParZero{
  System.out.println( "numérateur" );
  double num = in.nextDouble();
  System.out.println( "dénominateur" );
  double denom = in.nextDouble();
  if ( denom == 0 )
    throw new DivisionparZero();
  double quotient = num/denom;
  System;out.println(num+ "/" +denom+ "=" +quotient);
}
```

Transmission d'une cause d'exception

```
Exception(String message, Throwable cause)
```

Constructs a new exception with the specified detail message and cause.

The Throwable class is the superclass of all errors and exceptions in the Java language.

Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.

Similarly, only this class or one of its subclasses can be the argument type in a catch clause.

Propagations en chaîne

```
public class ExceptionsEnChaine{
  public static void main( String[] args )
  {
     try{ methode_1(); }
     catch( Exception exception ){ exception.printStackTrace(); }
  }
  public static void methode_1() throws Exception
  {
     try{ methode_2(); }
     catch( Exception exception )
     { throw new Exception( " --> methode_1" ,exception ); }
  }
  public static void methode_2() throws Exception
  {
     try{ methode_3(); }
     catch( Exception exception )
     { throw new Exception( " --> methode_2" ,exception ); }
  }
  public static void methode_3() throws Exception
  {
         throw new Exception( " --> methode_3" );
  }
}
```

Résultats

```
java.lang.Exception:  --> methode_1
	at ExceptionsEnChaine.methode_1(ExceptionsEnChaine.java:21)
	at ExceptionsEnChaine.main(ExceptionsEnChaine.java:12)
	at __SHELL7.run(__SHELL7.java:6)
	at bluej.runtime.ExecServer.vmSuspend(ExecServer.java:178)
	at bluej.runtime.ExecServer.main(ExecServer.java:143)
Caused by: java.lang.Exception:  --> methode_2
	at ExceptionsEnChaine.methode_2(ExceptionsEnChaine.java:29)
	at ExceptionsEnChaine.methode_1(ExceptionsEnChaine.java:18)
	... 4 more
Caused by: java.lang.Exception:  --> methode_3
	at ExceptionsEnChaine.methode_3(ExceptionsEnChaine.java:34)
	at ExceptionsEnChaine.methode_2(ExceptionsEnChaine.java:26)
	... 5 more
```

Captures multiples

- Un bloc `try` peut être suivi de plusieurs blocs `catch`
- Si une exception est levée dans le bloc `try`, elle est traitée dans le premier bloc `catch` avec lequel le type de l'exception s'unifie.
- Le type de l'exception peut être celui de la super-classe de l'exception levée
- Lorsque l'exécution du bloc `catch` est terminée, le contrôle est passé derrière le dernier bloc `catch`

```java
import java.util.*;
public class CatchMultiples{
  static class NegatifException extends Exception{}
  public static void main( String[] args ){
    Scanner in = new Scanner( System.in );
    int x = 0;boolean ok = false;String chaine = "";
    System.out.println( "Tapez un entier !" );
    while ( !ok ){
      try{
        x = in.nextInt();
        if ( x<0 ) throw new NegatifException();
        else System.out.println( "entier saisi : " + x );
       ok = true;
      }
      catch ( InputMismatchException ime ){
        chaine = in.next();
        System.out.print( chaine + "n'est pas un entier :" );
        System.out.println( " nouvel essai !" );
      }
      catch ( NegatifException ne ){
        System.out.println( "on continue avec la valeur 0 !" );
        x = 0;ok = true;
      }} }
}
```

La clause **`finally`**

- Cette clause suit une construction **`try-catch`**
- Elle spécifie une portion de code qui sera exécutée qu'une exception soit levée ou pas

```
try
{  ...}
catch(...)
{  ...}
catch(...)
{  ...}
finally
{// portion de code obligatoirement exécutée}
```

La clause `finally`

- la clause `finally` contient du code pour rendre des ressources acquises dans le bloc `try` correspondant (fichiers, connexion BD ou réseau).
- Java garantit que la clause `finally` s'exécutera y compris si le bloc `try` termine par un `return`, un `break` ou un `continue`

Exemple

```
public class Exemple{
  public static void main( String[] args ){
     try{
          lanceException();
     }
     catch( Exception exception ){
        System.err.println
                        ( "main → exception" );
     }
  }
```

Exemple

```
public static void lanceException() throws Exception{
   try{
     System.out.println( "méthode lanceException" );
     throw new Exception();
   }
    catch( RuntimeException re ){
     System.err.println
                ( "lanceException → RuntimeException" );
     }
     finally{
     System.err.println( "finally" );
     }
 }
 }
```

Résultat

```
méthode lanceException

finally
main → exception
```

Exercice 1

Supposons que `message2` déclenche une exception dans le bloc `try-catch` suivant :

```
try{
   message1;
   message2;
   message3;
}
catch( Exception1 e1 )
   {...}
catch( Exception2 e2 )
   {...}
message4;
```

Questions :

- message3 sera-t-il exécuté ?
- si l'exception n'est pas traitée, message4 sera-t-il exécuté?
- si l'exception est traitée, message4 sera-t-il exécuté?
- si l'exception est propagée, message4 sera-t-il exécuté?

Exercice 2

Supposons que `message2` déclenche une exception dans le bloc suivant :

```
try{
        message1;
        message2;
        message3;
}
catch( Exception1 e1 ){...}
catch( Exception2 e2 ){...}
catch( Exception3 e3 ){ throw e3}
finally{message4;}
message 5;
```

Questions :
- message5 sera-t-il exécuté si l'exception n'est pas traitée?
- si l'exception est de type 3, message4 sera-t-il exécuté, message5 sera-t-il exécuté

Exercice 1 : réponses

Supposons que `message2` déclenche une exception dans le bloc `try-catch`

Questions :

- message3 sera-t-il exécuté ? NON
- si l'exception n'est pas traitée, message4 sera-t-il exécuté? NON
- si l'exception est traitée, message4 sera-t-il exécuté? OUI
- si l'exception est propagée, message4 sera-t-il exécuté? NON

Exercice 2 : réponses

Supposons que `message2` déclenche une exception dans le bloc `try-catch`

Questions :

-message5 sera-t-il exécuté si l'exception n'est pas traitée? NON

- si l'exception est de type 3, message4 sera-t-il exécuté, OUI
message5 sera-t-il exécuté NON