Programmation Orientée Objet

Amélie Lambert

USSE09 - Java

Les exceptions

La gestion des erreurs

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** (tout élément de l'ensemble de définition a une image).
- Rien n'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, etc ...

Exemple

- Soit un programme de saisie de données numériques comprises dans un certain intervalle $[a, b]$.
- 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

- Lecture d'un entier :

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

- Pour vérifier que la donnée lue est valide, une solution serait que la méthode renvoie un boolean :

```
boolean saisieEntiers(int a, int b) {
    Scanner in = new Scanner(System.in);
    boolean etat;
    int x = in.nextInt();
    if (a>= x || x >= b)
        etat = false; //erreur de saisie
    else
        etat = true;
  return etat;
}
```

Gestion traditionnelle des erreurs : inconvénients (1/2)

- Il faut ajouter un retour de fonction.
- 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`.

Les exceptions en Java

- Levée d'une exception
- Traitement d'une exception
- Propagation d'une exception

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**.

Exception : définition

- 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, par exemple :

  `IndexOutOfBoundsException`
  est levée lorsqu'un indice de tableau est hors limite.

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

  `ArithmeticException`
  est levée lorsqu'une opération arithmétique est mal utilisée.

- Le programmeur peut lui-même créer ses propres classes d'exception pour gérer les cas d'erreurs qu'il a identifiées.

Définir une exception

- **Définir une exception, c'est définir une nouvelle classe.**
- **Il existe 3 façons de définir une exception :**
  1. par extension de la classe `Error`
  2. par extension de la classe `Exception`
  3. 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 gérées par le programmeur.
- **Exemples :**
  - 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**.

- **Exemples :**
  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 programme**.

- Un grand nombre d'entre elles sont prédéfinies .

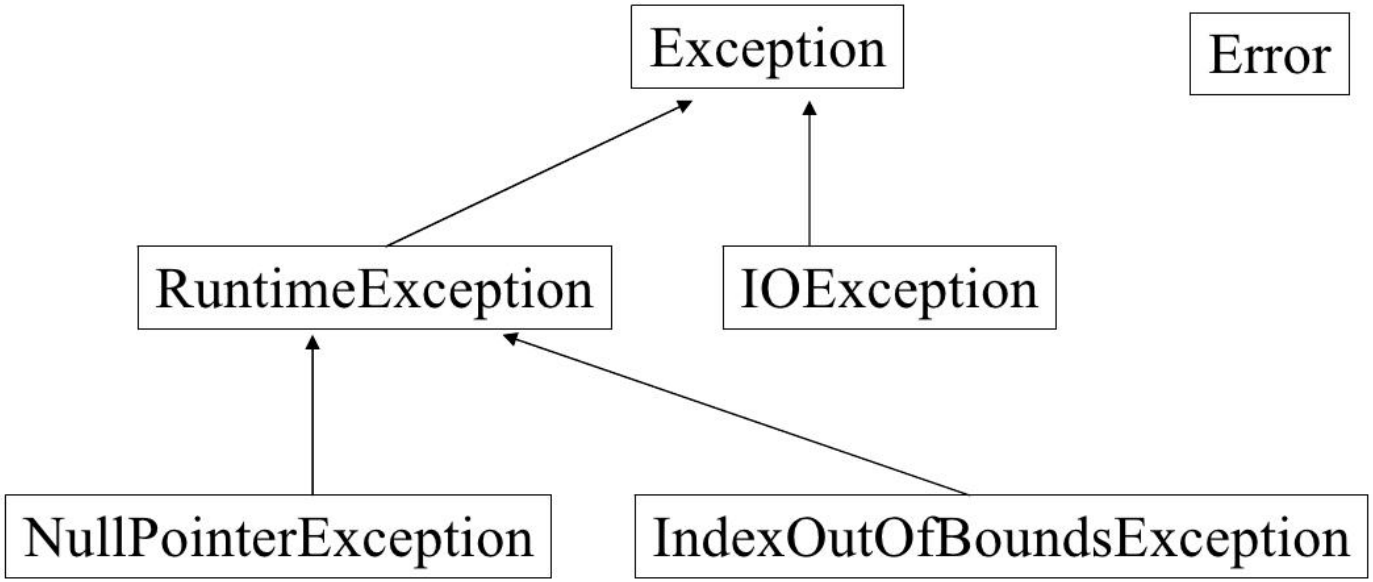
- **Exemples :**
  `NullPointerException` est levée si l'on tente d'accéder à un objet ou un tableau de valeur `null`.

  `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.

La hierarchie des exceptions



Mécanisme des exceptions(1/2)

- **Intérêt :**

  La séparation 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 crée 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.

La levée d'une exception

- Définition préalable d'une classe d'exception ou utilisation d'une classe prédéfinie :

  ```
  class Stop extends Exception
  ```

- Nécessite l'instanciation de cette classe :

  ```
  new Stop()
  ```

- Levée de l'exception :
  ```
  throw new Stop();
  ```

  Le programme s'arrête, les instructions qui suivent jusqu'à la fin du programme ne sont pas exécutées, à moins qu'un traitement ne soit prévu.

Exemple (1/2)

Définition de l'exception :

```
public class Stop extends Exception{
  public Stop(){
    super( "Erreur de saisie" );}

  public Stop( String msg ){
    super( msg );  }
}
```

Levée de l'exception :

```
import java.util.Scanner;
public class Arret{
  public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    int x = in.nextInt();
    System.out.println("instruction 1");
    // si la valeur saisie est positive, on lève l'exception
    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 8 du fichier.

Mécanisme des exceptions

Il existe 2 façons d'utiliser les exceptions :

- **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 remonte toute la pile d'exécution sans jamais trouver un bloc `catch` associé à l'exception propagée, 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
   }
}
```

L'exception est propagée vers la méthode appelante.

Exemple : Définition de l'exception (1/3)

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

  public DivisionParZero( String msg ){
    super( msg );
  }
}
```

Exemple : propagation de l'exception (2/3)

```
public class A{
  private double num;

  public A(){ }

  public double diviser(double den) throws DivisionParZero{
    if (den == 0) throw new DivisionParZero();
    return num/den; }
}

public class B{
  private A a;

  public B(A a){
   this.a=a;  }

  public double diviserPar(double den) throws DivisionParZero{
    return A.diviser(den); }
}
```

Exemple : traitement de l'exception (3/3)

```
public class Exemple{
  public static void main(String[] args){
    double num, den;
    Scanner in = new Scanner(System.in);
    System.out.println("Entrez le numérateur :");
    num = in.nextDouble();
    A a = new A(num);
    B b = new B(a);
    try{
      System.out.println("Entrez le dénominateur :");
      den = in.nextDouble();
      System.out.println("Le résultat est :", b.diviserPar(den));
    }
    catch (DivisionParZero e){
      System.out.println("Il est impossible de diviser par 0");
    }
  }
}
```

Les exceptions non contrôlées

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.

Exemple : Documentation de `nextInt()` (2/2)

```
nextInt

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
```

Exemple : Traitement d'un exception non contrôlée (2/2)

```
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.

```java
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();
   }
}

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

L'exception `IllegalStateException` est potentiellement levée par la méthode `next()` si le `Scanner in` est fermé. Elle est ensuite propagée vers la méthode appelante `lire` qui la propage à son tour vers la méthode `main`.

Les exceptions contrôlées

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.

Exemple : 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() );
  }
}
```

Exemple : 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);
}
```

L'exception est propagée, il faudra la traiter, par exemple dans le `main`.

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");
    }
  }

  public static void methode_2() throws Exception{
    try{ methode_3(); }
    catch(Exception exception){ throw new Exception(" -->methode_2");
    }
  }

  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
```

Les captures multiples

Les 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`.

```
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();
        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`

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`.

Exécution du bloc `finally`

3 cas :

1. Aucune exception dans le bloc `try` :
   ⇒ le bloc `finally` est exécuté
   ⇒ ainsi que les instructions qui suivent

2. Exception déclenchée dans le bloc `try` traitée dans un bloc `catch` :
   ⇒ les instructions qui suivent dans le bloc `try` sont sautées
   ⇒ le bloc `catch` est exécuté
   ⇒ le bloc `finally` est exécuté
   ⇒ si le bloc `catch` relance l'exception, le contrôle est passé vers l'appelant, sinon, les instructions qui suivent le bloc `finally` sont exécutées

3. Exception déclenchée dans le bloc `try` non traitée dans un bloc `catch` :
   ⇒ les instructions qui suivent dans le bloc `try` sont sautées
   ⇒ le bloc `finally` est exécuté
   ⇒ le contrôle est passé vers l'appelant (avec l'exception)

Exemple

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

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