

# Introduction à la Programmation Orientée Objet

## 6 - Les exceptions

**Valentin Honoré**

valentin.honore@ensiie.fr

FISA 1A

# À quoi servent les exceptions ?

- ▶ Les exceptions servent à gérer les cas d'exécution **exceptionnels**
  - c'est-à-dire les cas d'exécution rares, par exemple les erreurs
  - intercepter et propager des erreurs, des valeurs indésirables etc
  
- ▶ **Évite de traiter les cas exceptionnels dans le flot d'exécution normal**
  - code plus clair et plus facilement réutilisable
  
- ▶ Idée de `stderr` et `cerr` en C/C++

- ▶ Exemple : méthode calculant le logarithme népérien
  - `double ln(double x)`
  
- ▶ Que faire si  $x$  est inférieur ou égal à 0 (non défini par la fonction) ?
  - Renvoyer une valeur absurde... Mais `ln` est surjective dans  $\mathbb{R}$  !
  - Renvoyer un objet avec un champ indiquant l'erreur et un champ indiquant le résultat...  
Mais coûteux en calcul/mémoire !

pas de solution satisfaisante

## Problème 2 : propager un cas d'exécution exceptionnel

- ▶ Bien souvent, on est obligé de propager d'appelé vers appelant les situations exceptionnelles

- ▶ Exemple : calcul du nombre de chiffres d'un nombre

```
int nbChiffres(double x) { return log10(x); }  
double log10(double x) { return ln(x)/ln(10); }  
double ln(double x) { /* calcul log. népérien */ }  
  
... main() { System.out.println(nbChiffres(3024)); }
```

- ▶ `log10` et `nbChiffres` doivent propager l'erreur si erreur dans `ln`

cette propagation rendrait le code vite illisible !



- ▶ Facilité de programmation et de lisibilité
  - Regrouper la gestion d'erreurs à un même niveau
  - Evite des redondances dans l'écriture de traitements d'erreurs
  - Encombre peu le reste du code avec ces traitements.
  
- ▶ Une gestion des erreurs propre et explicite
  - Valeur de retour parfois utilisée pour décrire une erreur
  - Dissocier valeur de retour et exception pour être plus précis (quelle ligne de code est concernée etc)

- ▶ Une exception est un objet qui hérite de la classe `Exception`
  - ❑ `class NegativeValue extends Exception`
  - ❑ Toute instance de `NegativeValue` est une exception
  
- ▶ **Une exception est un objet comme les autres**, mais il peut en plus être levé et attrapé
  
- ▶ Deux principaux constructeurs offerts par `Exception`
  - ❑ Pas d'argument
  - ❑ Un message de type `String` accessible via `getMessage()`

# Lever une exception

- ▶ `throw e` où `e` est un objet de type exception

- Exemple dans le code de la méthode `ln` :

```
if(x <= 0) throw new NegativeValue();
```

- ▶ Les exceptions levées par une méthode (directement ou indirectement) doivent être indiquées avec `throws`

```
int nbChiffres(double x) throws NegativeValue {  
    return log10(x); }  
double log10(double x) throws NegativeValue {  
    return ln(x)/ln(10); }  
double ln(double x) throws NegativeValue { ... }
```

Ne confondez pas `throw` (sans s) pour lever une exception

avec

`throws` (avec s) pour indiquer quelles exceptions sont levées

# Attraper une exception

## ► Trois éléments + 1 optionnel

- Délimiteur de zone d'attrapage avec `try`
- Suivi de filtres d'exceptions attrapées avec `catch`
- Suivi du code à exécuter si exception attrapée
- Optionnellement suivi de `finally` exécuté dans tous les cas

```
try { nbChiffres(); }  
catch(NegativeValue|ZeroValue e) { ... }  
catch(AutreException e) { ... }  
catch(Exception e) { ... }  
finally { ... }
```

Si exception levée dans ce bloc  
⇒ applique les filtres

Attrape tout objet qui hérite de `Exception`  
⇒ toutes les autres Exceptions

## Exemple complet [TestNegativeValue.java]

```
class NegativeValue extends Exception {}

class TestNegativeValue {
    static double log(double x) throws NegativeValue {
        if(x <= 0) { throw new NegativeValue(); }
        else { return Math.log(x); } }
    public static void main(String[] args) {
        try {
            double val = Double.parseDouble(args[0]);
            System.out.println(log(val));
        } catch(NegativeValue v) {
            System.out.println("Chiffre positif attendu");
        }
    }
}
```

- ▶ N'utilisez les exceptions que pour les cas exceptionnels
  - ❑ Les exceptions rendent le code difficile à lire dans le cas normal (plus facile à lire uniquement dans les cas exceptionnels)
  - ❑ Allouer une exception avec `new` et lever une exception avec `throw` sont deux opérations particulièrement lentes
  - ❑ `try/catch` est en revanche gratuit
  
- ▶ N'attrapez les exceptions que si vous avez quelque chose d'intéressant à faire
  - ❑ Exemple à ne jamais faire :

```
try { f() } catch(Exception e) { throw e; }
```

- ▶ Si `e` est de type `Exception`
  - ❑ `e.getMessage()` → renvoie le message associé à l'exception
  - ❑ `e.printStackTrace()` → affiche la pile d'exécution (la suite d'appels ayant mené jusqu'au `throw`)
  - ❑ La machine virtuelle Java affiche message/pile d'exécution si le `main` lève une exception
  
- ▶ Très utile pour déverminer vos programmes

## 4 exceptions prédéfinies à connaître

- ▶ `ClassCastException` : mauvais transtypage
- ▶ `IllegalArgumentException` : mauvais argument (vous pouvez réutiliser cette exception, elle est là pour ça)
- ▶ `IndexOutOfBoundsException` : dépassement de capacité d'une structure (tableau, chaîne de caractères, vecteurs)
- ▶ `NullPointerException` : accès à la référence `null`

- ▶ Erreurs de compilation
  - Compilateur renvoie des erreurs
  - Solution : réviser le code
  
- ▶ Erreurs d'exécution
  - Arrêt/blocage du programme à l'exécution
  - Solution : résoudre la configuration de la JVM
  
- ▶ Exception non vérifiée
  - Trace de la pile des exceptions, sans gestion
  - Solution : modifier le programme
  
- ▶ Exception vérifiée
  - Trace de la pile montre une exception attrapée
  - Solution : changer les paramètres du programme

- ▶ Le mécanisme d'exception sert à gérer les cas exceptionnels
  
- ▶ Exception = objet qui hérite de la classe `Exception`
  - Peut être levée avec `throw`
  - Peut être attrapée avec `try/catch`