

MOdélisation OBjet

2 - Héritage, classes abstraites et exceptions

Valentin Honoré

valentin.honore@ensiie.fr

FISA 1A

- ▶ Une structure de données (tuple ou tableau) s'appelle un objet
 - Un objet possède un **type appelé sa classe**
 - Si la classe de **l'objet o** est **C**, on dit que **o** est une **instance de C**
- ▶ La classe d'un objet
 - définit des **Champs & Méthodes d'instances**
 - Possède un **paramètre implicite** du type de la classe nommé **this**
 - **Constructeurs** : méthodes spéciales appelées après l'allocation
- ▶ Chacun de ces éléments a une visibilité
 - **public** (partout), **private** (local), pas de mot clé (package)

1 Héritage

- Pourquoi et comment ?
- Utilisation de l'héritage en Java

2 Exceptions

1 Héritage

- Pourquoi et comment ?
- Utilisation de l'héritage en Java

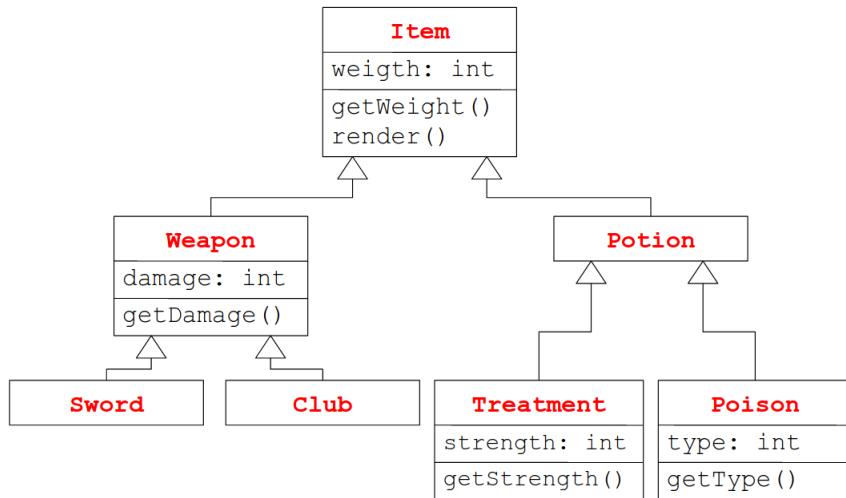
2 Exceptions

- ▶ Améliorer la réutilisabilité du code
 - ☐ En écrivant du code générique et spécialisable
 - ☐ En factorisant des fonctionnalités

- ▶ Exemples
 - ☐ Un objet générique dans un jeu, spécialisable en équipements de personnage
 - ☐ Un nœud générique dans un arbre binaire de recherche
 - ☐ Un flux générique spécialisable en flux réseau, flux vers un fichier ...
 - ☐ Une définition générique des matrices qui peuvent devenir creuse, carrée ...
 - ☐ ...

- ▶ Une classe dite *fil*le peut hériter d'une autre classe dite *mère*
 - La classe fille hérite (possède) les champs et méthodes de la mère
 - Et peut en ajouter de nouveaux pour spécialiser la classe mère
 - méthodes avec même nom et paramètres différents = surcharge de méthode (*overloading*)
 - La classe fille définit un nouveau type
 - Ce type est compatible avec le type de la mère
- ▶ L'héritage est, par définition, **transitif**
 - Si C hérite de B et B hérite de A, alors C hérite de A

Exemple d'héritage (1/2, spoiler du cours de modélisation)



Exemple d'héritage (2/2)

- ▶ Le sac du joueur est constitué d'objets de type Item
 - Affichés à l'écran avec la méthode `render()`
 - Le poids du sac est calculé en appelant `getWeight()` sur chaque Item

→ le code de gestion du sac est générique

- ▶ Le système de combat manipule des objets de type Weapon
 - Les dégâts sont connus avec la méthode `getDamage()`
 - Le fait de se battre à l'épée ou la massue ne change rien

→ le code de gestion des combats est générique

1 Héritage

- Pourquoi et comment ?
- Utilisation de l'héritage en Java


2 Exceptions

- ▶ Une classe hérite **au plus d'une unique classe**
 - Mot clé `extends` suivi du nom de la classe mère

```
class Item {  
    private int weight;  
    public void setWeight(int w) { weight = w; }  
    public Item() {}  
}
```

`Weapon` hérite de `Item` ⇒ possède les champs/méthodes de `Item`

```
class Weapon extends Item {  
    public Weapon(int w) { setWeight(w); }  
}
```

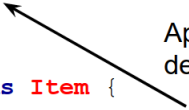


- ▶ La **première instruction** du constructeur d'une classe **filles** doit être une invocation du constructeur de la classe mère
 - Invocation constructeur mère avec le nom clé **super**

```
class Item {  
    private int weight;  
    public void setWeight(int w) { weight = w; }  
    public Item(int w) { weight = w; }  
}
```

```
class Weapon extends Item {  
    public Weapon(int w) { super(w); }  
}
```

Appel du constructeur
de **Item** avec le paramètre **w**



- ▶ La **première instruction** du constructeur d'une classe **filles** doit être une invocation du constructeur de la classe mère
 - Invocation constructeur mère avec le nom clé `super`
 - Invocation implicite si **constructeur mère sans paramètre**

```
class Item {  
    private int weight;  
    public void setWeight(int w) { weight = w; }  
    public Item() {}  
}
```

```
class Weapon extends Item {  
    public Weapon(int w) { super(); setWeight(w); }  
}
```

Si `super` omis
⇒ appel à `super` implicite (ajouté automatiquement à la compilation)

- ▶ Une fille possède aussi le type de sa mère → sur-typage valide

```
Item item = new Sword(); /* valide (upcast) */
```

- ▶ Un objet du type de la mère ne possède pas en général le type de la fille

- ☐ Possibilité de sous-typer (downcast) un objet explicitement
- ☐ Erreur de transtypage détectée à l'exécution

```
Item item = new Sword();  
Sword enduril = (Sword)item; /* ok execution */  
Club tharkun = (Club)item; /* erreur execution */
```

- ▶ Une fille possède aussi le type de sa mère → sur-typage valide

```
Item item = new Sword(); /* valide (upcast) */
```

- ▶ Un objet du type de la mère ne possède pas en général le type de la fille

- ☐ Possibilité de sous-typer (downcast) un objet explicitement
- ☐ Erreur de transtypage détectée à l'exécution

- ▶ `instanceof` permet de tester si un objet est une instance d'une classe donnée

```
if(item instanceof Club) {  
    Club asCl = (Club)item;  
    System.out.println("This is a club");  
}
```

Héritage et appel de méthode d'instance (1/3)

L'appel à une méthode déclenche un mécanisme de recherche dans le graphe d'héritage jusqu'à trouver une méthode ayant un nom et une signature correspondant à l'appel

- ▶ Surcharge = **résolu à la compilation**
- ▶ Redéfinition (la méthode a le même profil dans la classe fille)
 - ☐ résolu à **l'exécution** : la méthode réellement appelée dépend de la classe réelle de l'objet qui exécute cette méthode
 - ☐ **polymorphisme** d'héritage
 - ☐ Tag `@Override`
- ▶ Appel explicite à la méthode de la classe mère avec `super.methode()`

Héritage et appel de méthode d'instance (2/3)

- ▶ Si une fille redéfinit une méthode de la mère, l'appel va toujours vers celui de la fille (liaison tardive)

```
class Item {  
    public void render() {  
        /* prints ? */  
    }  
}  
  
class Sword extends Weapon {  
    @Override  
    public void render() {  
        /* prints a sword */  
    }  
}
```

```
class Engine {  
    public void test() {  
        Item i = new Sword();  
        i.render();  
    }  
}
```

- ▶ Appel à `Sword.render()` car `i` est un objet de type effectif `Sword` (même si son type statique dans le code source est `Item`)

Héritage et appel de méthode d'instance (3/3)

► Redéfinition d'une méthode d'une classe supérieure

- ☐ Même signature mais le traitement effectué est ré-écrit dans la sous-classe
- ☐ Accès à la méthode de la classe mère `super` + préfixe de la méthode
- ☐ **Utilisez le tag** `@Override` pour ces redéfinitions (*check* à la compilation)

```
class Sword extends Weapon {  
    @Override  
    public void render () {  
        super.render(); /* render() de Weapon */  
    }  
}
```

► Possible d'interdire la redéfinition d'un élément : on ne pourra lui donner une valeur qu'une seule fois dans le programme.

► Avec le mot-clé `final` pour une méthode / variable / classe

- ☐ Variables avec `final` : constant pour chaque instance (mais valeur différente possible entre 2 instances)
- ☐ Méthode avec `final` : redéfinition interdite dans classes filles
- ☐ Classe avec `final` : interdire **l'héritage**. `final` au début de la déclaration d'une classe (avant le mot-clé `class`).

Un exemple avec final [HeritageExemple.java]

```
class Item {
    public void render () {
        System.out.println("I am an item.");
    }
}

final class Sword extends Item {
    @Override
    public void render () {
        super.render();
        System.out.println("Even better, I am a sword!");
    }
}

public class HeritageExemple{
    public static void main(String[] args) {
        Sword enduril = new Sword();
        enduril.render();
    }
}
```

- ▶ Pour une variable locale : sa valeur ne peut être donnée qu'une seule fois
 - ☐ type primitif : la valeur ne peut changer
 - ☐ type référence vers objet : ne pourra pas référencer un autre objet (mais l'état de l'objet pourra être modifié)

```
final Perso gandalf = new Perso();  
gandalf.nom = "Gandalf"; // Autorisé  
gandal.pointsVie=1000000; // Autorisé  
gandalf = new Perso(); // INTERDIT
```

- ▶ Pour une variable de classe : constante dans tout le programme
 - peut ne pas être initialisée à sa déclaration mais doit avoir une valeur à la sortie de tous les constructeurs ou doit recevoir sa valeur dans un bloc d'initialisation `static`

```
static final double PI = 3.14;
```

► La visibilité `protected`

- `protected` permet de spécifier qu'une entité (champ ou méthode) est **visible des classes filles et du package**

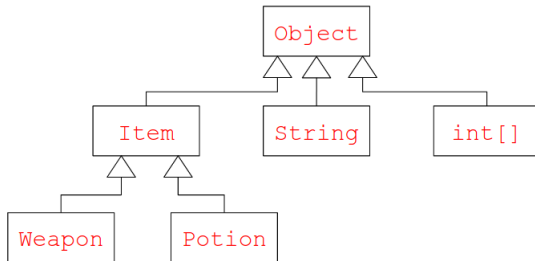
	<code>public</code>	<code>protected</code>	Défaut	<code>private</code>
Dans la classe	Oui	Oui	Oui	Oui
Dans une classe du package	Oui	Oui	Oui	Non
Dans une sous-classe d'un autre package	Oui	Oui	Non	Non
Dans une classe quelconque d'un autre package	Oui	Non	Non	Non

Table – Résumé des droits d'accès

Polymorphisme

- ▶ Polymorphisme : faculté attribuée à un objet d'être une instance de plusieurs classes
 - une seule classe "réelle" (la classe figurant après le `new`)
 - il peut aussi être déclaré avec une classe supérieure à sa classe "réelle"
- ▶ Toutes les classes héritent d'une classe nommée `Object`
 - Héritage implicite (`extends Object`) si pas de `extends`
 - Héritage par transitivité sinon
 - Vrai aussi pour les classes de tableaux

La relation d'héritage construit un arbre dont `Object` est la racine



La méthode String toString()

► Object fournit quelques méthodes génériques

- La plus importante à ce stade est **String** toString()
- Méthode appelée automatiquement pour convertir un objet en **String** quand conversion implicite requise (cf Cours 2)

```
class City {  
    private String name;  
    public String toString() {  
        return "City: " + name;  
    }  
}
```

```
City city = new City("Paris");  
System.out.println(city);
```

La méthode `String toString()`

La méthode `String toString()` sert principalement à déverminer les programmes

Utilisez la !

Règle générale : toute classe devrait redéfinir la méthode `String toString()`

Question : quel est l'affiche par défaut de cette méthode ?

La méthode `String toString()`

La méthode `String toString()` sert principalement à déverminer les programmes

Utilisez la !

Règle générale : toute classe devrait redéfinir la méthode `String toString()`

Question : quel est l'affiche par défaut de cette méthode ?

Par défaut, affiche la référence de l'objet

Les classes et méthodes abstraites (1/2)

► Propriétés du polymorphisme

- ☐ L'interpréteur Java trouve le traitement à effectuer lors de l'appel à méthode sur un objet
- ☐ Le traitement associé à une méthode donnée peut être différent (différentes classes réelles sous une même classe)

► Parfois, donner le code d'une méthode dans une classe mère n'a pas de sens

- ☐ La méthode n'est définie que pour être invoquée, c'est aux filles de la mettre en œuvre
- ☐ Exemple typique : une méthode `getDamage()` de `Weapon`

```
class Weapon extends Item {  
    public int getDamage() {  
        /* quelle valeur doit-on renvoyer ici ? */  
    }  
}
```

```
class Sword extends Weapon {  
    public int getDamage() {  
        return 17;  
    }  
}
```

Les classes et méthodes abstraites (2/2)

- ▶ Dans ce cas, on peut omettre le code d'une méthode
 - ☐ Marquer la méthode d'instance comme **abstract**
 - ☐ Marquer la classe comme **abstract** (cad, impossible à instancier)
- ▶ ATTENTION : les classes descendantes concrètes doivent mettre en œuvre les méthodes marquées **abstract** dans la classe mère
- ▶ Une classe abstraite peut donc contenir
 - ☐ des variables
 - ☐ des méthodes implémentées
 - ☐ des méthodes abstraites à implémenter
- ▶ Peut hériter d'une classe ou d'une classe abstraite. Dans ce cas, elle doit
 - ☐ implémenter les méthodes abstraites de sa super-classe en les dotant d'un corps
 - ☐ soit être elle-même abstraite si au moins une des méthodes abstraites de sa super-classe reste abstraite.

Utilisation des classes abstraites (1/3)

// classes abstraites versus classes concrètes

```
abstract class Item {  
    public void printWeigth() {  
        System.out.println("Weight: " + getWeigth());  
    }  
    public abstract int getWeight();  
}
```

Club et Sword mettent en œuvre
getWeight
(on suppose que Weapon hérite de
Item)

```
class Club extends Weapon {  
    public int getWeight() {  
        return 42;  
    }  
}
```

```
class Sword extends Weapon {  
    public int getWeight() {  
        return 17;  
    }  
}
```

Utilisation des classes abstraites (1/3)

Remarque : Weapon est une classe abstraite, pourquoi ?

```
abstract class Weapon extends Item { }
```

```
abstract class Item {  
    public void printWeigth() {  
        System.out.println("Weight: " + getWeigth());  
    }  
    public abstract int getWeigth();  
}
```

Club et Sword mettent en œuvre
getWeight
(on suppose que Weapon hérite de
Item)

```
class Club extends Weapon {  
    public int getWeight() {  
        return 42;  
    }  
}
```

```
class Sword extends Weapon {  
    public int getWeight() {  
        return 17;  
    }  
}
```

Utilisation des classes abstraites (2/3)

```
class Bag {  
    Item[] items;  
  
    // exemple d'initialisation de items  
    Bag() {  
        items = new Item[2];    // tableau de réfs vers Item  
        items[0] = new Club();  // ok car Club est un Item  
        items[1] = new Sword(); // ok car Sword est un Item  
    }  
    ...  
}
```

```
class Club extends Weapon {  
    public int getWeight() {  
        return 42;  
    }  
}
```

```
class Sword extends Weapon {  
    public int getWeight() {  
        return 17;  
    }  
}
```

Utilisation des classes abstraites (3/3)

```
class Bag {  
    ... // items vaut toujours { new Club(), new Sword() }  
  
    // exemple d'utilisation de getWeight  
    int bagWeight() {  
        int tot = 0;  
        for(int i=0; i<items.length; i++)  
            tot += items[i].getWeight();  
        return tot;  
    }  
}
```

⇒ tot vaut 59
à la fin de la boucle

getWeight()
pour items[0]

getWeight()
pour items[1]

```
class Club extends Weapon {  
    public int getWeight() {  
        return 42;  
    }  
}
```

```
class Sword extends Weapon {  
    public int getWeight() {  
        return 17;  
    }  
}
```

- ▶ Limitation de l'héritage Java
 - ☐ Une classe hérite au plus d'une unique classe
 - ☐ Que faire, par exemple, pour une classe Poison qui serait à la fois une arme (Weapon) et une potion (Potion) ?
- ▶ Solution : **l'interface** = généralisation des classes abstraites
 - ☐ Ne définit que des méthodes abstraites, pas de champs
 - ☐ Mot clé **interface** au lieu de **class**, plus besoin de marquer les méthodes **abstract**
 - ☐ Possibilité pour un objet d'exposer plusieurs interfaces
 - ☐ On dit alors que la classe met en œuvre la (les) interface(s)


```
interface Weapon {  
    public int getDamage();  
}
```

à peu près équivalent à :

```
abstract class Weapon {  
    abstract public int getDamage();  
}
```

L'interface - exemple (1/2)

```
interface Weapon {
    public int getDamage();
}
```

```
interface Potion {
    public void drink(Player p);
}
```

```
class Poison implements Weapon, Potion {
    public int getDamage() { ... }
    public void drink(Player player) { ... }
}
```

Utilisation :

```
Potion p = new Poison();  
p.drink(sauron);
```

- ▶ Si Y est une classe fille de X
 - ☐ Y possède les champs et méthodes de X, Y est aussi du type X
 - ☐ Déclaration avec `class Y extends X ...`
 - ☐ Appel du constructeur parent avec le mot clé `super`
 - ☐ Le receveur d'un appel de méthode est donné par le type effectif

- ▶ Méthode `abstract`
 - ☐ Méthode d'instance sans corps
 - ☐ La classe doit être marquée `abstract`, elle est non instanciable

- ▶ Interface = classe avec uniquement des méthodes `abstract`
 - ☐ Mise en œuvre d'une interface avec `implements`

1 Héritage

- Pourquoi et comment ?
- Utilisation de l'héritage en Java

2 Exceptions

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 `std::err` et `cerr` en C/C++

Problème 1 : signaler un cas d'exécution exceptionnel

- ▶ 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

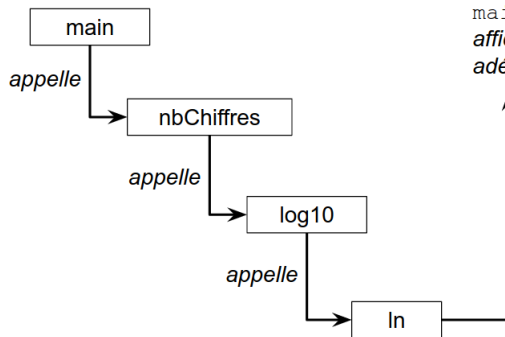
```
int nbChiffres(double x) { return log10(x); }  
double log10(double x) { return ln(x)/ln(10); }  
double ln(double x) { /* calcul log. népérien */ }  
  
... main() { System.out.println(nbChiffres(3024)); }
```

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

cette propagation rendrait le code vite illisible !

► Exception : **objet qu'on peut lever et attraper**

- **Lever** une exception : interrompt la chaîne d'appel jusqu'à un appelant capable d'**attraper** l'exception



main attrape l'exception pour afficher un message d'erreur adéquat

*ln lève une exception indiquant valeur négative
⇒ interrompt l'exécution de ln, de log10 et de nbChiffres, jusqu'au main qui attrape l'exception*

- ▶ 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`