

- Interfaces
- Collections polymorphiques

## Les interfaces

## Développer avec des interfaces

- Développement d'un projet par des groupes disparates de programmeurs qui se mettent d'accord sur un contrat.
- Chaque groupe doit pouvoir écrire son code sans connaissance de comment le code des autres groupes est écrit.
- Les interfaces constituent le contrat.

## Les interfaces en Java

Une interface Java est une spécification d'un type (sous la forme d'un nom de type et d'un ensemble de méthodes) qui ne définit pas d'implémentation pour les méthodes.

```
public interface Animal {  
    // tous les animaux doivent implémenter les méthodes suivantes  
    public void manger(String nourriture);  
    public void seDeplacer();  
    public void respirer();  
}
```

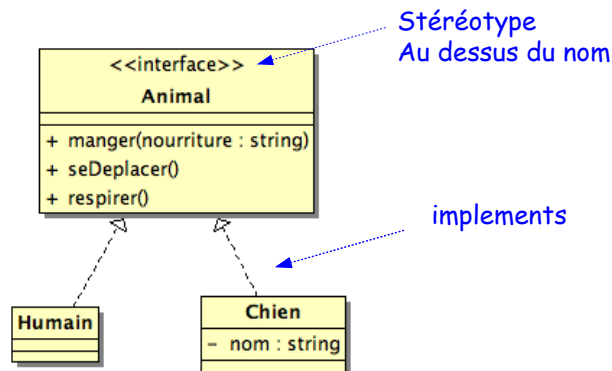
## Les interfaces

- Le rôle d'une interface est proche de celui d'une classe abstraite dans le cas d'un héritage de types.
- Une interface définit seulement un type.
- Toutes les méthodes sont abstraites.
- Une interface ne fournit aucune implémentation.
- Comme les classes, la définition est mise dans un fichier dont le nom est : *NomInterface.java*
- Comme les classes, les interfaces doivent être compilées.

## Que peut-on trouver dans une interface ?

- Toutes les méthodes sont publiques.
- Une interface ne contient pas de constructeurs.
- Une interface ne possède pas d'attribut d'instance.
- Les seuls attributs déclarés doivent être des constantes de classe :
  - `public static final double E = 2.718282.`

## Représentation UML



## Implémenter une interface

La classe Chien est liée par contrat à l'interface Animal

```
public class Chien implements Animal {  
    private String nom ;  
    //implémentations de l'interface Animal  
    public void manger(String patee){  
        System.out.println("Miam, " +patee+ " !");  
    }  
    public void seDeplacer(){  
        System.out.println("déplacement du chien");  
    }  
    public void respirer(){  
        System.out.println("respiration du chien");  
    }  
}
```

## utiliser plusieurs interfaces

```
public interface Animal {
    public void manger();
    public void seDéplacer() ;
    public void respirer ()
}
```

```
public interface Parler {
    public void parle(int phrase);
}
```

```
public interface Aboier {
    public void aboie();
}
```

## Implémenter plusieurs interfaces

La classe Chien doit fournir une implémentation à toutes les méthodes spécifiées par les 2 interfaces.

```
public class Chien implements Animal, Aboier {
    private String nom ;
    //implémentations de l'interface Animal
    . . .
    //implémentations de l'interface Aboier
    public void aboie(){
        System.out.println("Ouaf!");
    }
}
```

## L'API Java possède des interfaces

- interface **Comparable<T>**
  - **int compareTo(T o)** : retourne un entier négatif, nul ou positif si l'objet receveur est plus petit, égal ou plus grand que l'objet spécifié en paramètre.
- interface **Iterable**
  - **Iterator<T> iterator()** : retourne un itérateur sur un ensemble d'éléments de type T.

- interface **Cloneable**

Une classe implémente Cloneable pour indiquer que la méthode `Object.clone()` est légale pour faire une copie des instances de cette classe.

## Exemple de la classe String de l'API Java

```
public final class String
    extends Object
    implements Serializable,
        Comparable<String>,
        CharSequence
```

```
(Classe String)
public int compareTo(String anotherString)
```

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns 0 exactly when the `equals(Object)` method would return true

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

```
this.length()-anotherString.length()
```

#### Parameters:

`anotherString` - the `String` to be compared.

#### Returns:

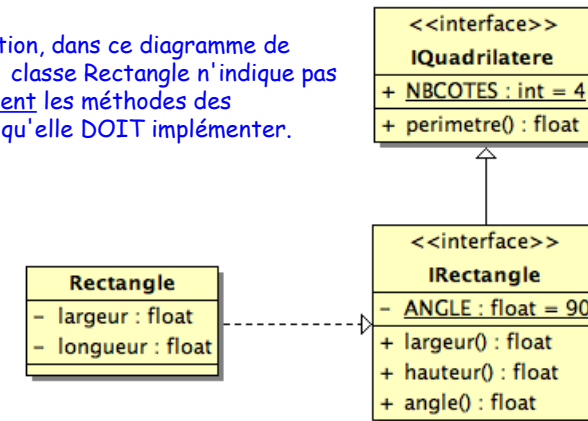
the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

IB - R2.01

13/28

## Héritage entre interfaces (notation UML)

Par convention, dans ce diagramme de classes, la classe `Rectangle` n'indique pas explicitement les méthodes des interfaces qu'elle DOIT implémenter.



IB - R2.01

14/28

## Implémentation et réalisation

```
public class Rectangle implements IRectangle {
    private float largeur, hauteur;

    public Rectangle() {...}

    public Rectangle(float l, float h){
        largeur=l;
        hauteur=h;
    }
    // implémentation de IQuadrilatere
    public float perimetre(){
        return 2*largeur()+2*hauteur();
    }
    // implémentation de IRectangle
    public float angle(){return IRectangle.ANGLE;}
    public float largeur(){return largeur;}
    public float hauteur(){return hauteur;}
}
```

IB - R2.01

15/28

## Interfaces vs classes abstraites

- Ce sont les deux mécanismes Java de définition d'un **type**.
- La différence la plus évidente : les classes abstraites peuvent contenir des implémentations pour certaines méthodes, pas les interfaces.
- Pour implémenter un type défini par une classe abstraite, au moins une classe doit être une sous-classe de la classe abstraite.
- Une classe peut implémenter une interface sans se soucier de la hiérarchie de classes.
- Il est plus facile de faire évoluer une classe abstraite qu'une interface. Pourquoi ?
- En résumé une interface est généralement la meilleure façon de définir un type sauf dans le cas où la facilité d'évolution est plus importante que la flexibilité.

IB - R2.01

16/28

## Evolution des interfaces

- Les possibilités des interfaces évoluent avec les nouvelles versions de Java.
- Par exemple, les interfaces supportent l'héritage multiple d'interfaces.
- Nous nous contenterons d'un usage standard :
  - Héritage simple entre les interfaces comme pour les classes
  - Implémentations multiples d'interfaces pour une classe.

## Types paramétrés multiples

```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}  
  
public class OrderedPair<K, V> implements Pair<K, V>{  
    private K key;  
    private V value;  
    . . .  
}
```

## Collections polymorphiques

## Polymorphisme

- Plusieurs classes implémentent une même méthode de façons différentes (par exemple toString()).
- Le polymorphisme généralise les abstractions pour qu'elles fonctionnent pour plusieurs types.
- En Java le polymorphisme est exprimé à travers la hiérarchie : un objet peut être déclaré avec un certain type (type apparent) et ensuite devenir un sous-type de ce type (type actuel)
  - BankAccount momsSaving = new SavingAccount(203,0.5) ;
  - BankAccount harrysCheking = new CheckingAccount(204) ;

## Collections polymorphiques

- Une collection est un objet qui regroupe de nombreux éléments dans une seule unité.
- Typiquement elles représentent des données qui forment des groupes naturels tels que :
  - une main de poker (collection des cartes)
  - un répertoire de mail (collections de lettres)
  - un répertoire de téléphone (des correspondances de noms et de numéros de téléphone)
- Les collections sont toutes typées et sont polymorphiques.

## Le framework des collections Java

- C'est une architecture unifiée pour représenter et manipuler des collections.
- Il contient
  - des interfaces : types abstraits de données
  - des implémentations concrètes des interfaces de collection
  - des algorithmes : calculs utiles tels que recherche, tri d'objets qui implémentent des interfaces de collection
- Nous utiliserons dans ce cours deux implémentations :
  - ArrayList : implémentation de l'interface List
  - HashMap : implémentation de l'interface Map
- Tous les éléments du framework sont dans le package *util*

## Les collections de type Map

- Une *map* est une collection polymorphique qui associe une clé à une valeur.
- La clé est unique, contrairement à la valeur qui peut être associée à plusieurs clés.
- La majorité des collections et, en particulier de type Map, ont deux constructeurs :
  - un constructeur sans paramètre créant une Map vide, et
  - un constructeur prenant en paramètre une Map qui crée une nouvelle Map en fonction de la Map passée en paramètre.

## Les HashMap

- Implémentation de base de l'interface Map.
- Les éléments d'une HashMap sont des paires d'objets :
  - une clé
  - une valeur
- Il n'y a pas de garantie sur l'ordre des éléments dans une HashMap.
- Comme pour un ArrayList on peut stocker un nombre flexible d'éléments.

## La classe java.util.HashMap

```
public class HashMap<K,V>
    extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
```

- C'est une classe paramétrée avec deux types :
  - K : le type pour les clés (keys)
  - V : le type pour les valeurs associées (values)
- Attention les clés et les valeurs sont des objets (pas de primitifs)

## Création d'un objet HashMap

- Constructeur :

```
public HashMap()
```

Création d'un HashMap :

```
HashMap<String, Integer> phoneBook =
    new HashMap<String, Integer>();
```

type de la valeur

type de la clé

## Les méthodes de base de HashMap

- `public V get(Object key)`  
retourne la valeur spécifiée par la clé, ou null s'il n'y a pas de mapping pour cette clé
- `public V put(K cle, V valeur)`  
associe la clé et la valeur dans la table. Si la clé était déjà présente, la valeur est remplacée. Retourne null si la clé n'était pas déjà présente dans la table, et l'ancienne valeur associée à la clé sinon.
- `public void clear()`  
retire tous les mappings
- `public int size()`  
retourne le nombre de mappings clé-valeur

## Suite des méthodes

- `public boolean containsKey(Object key)`  
retourne true si la table contient un mapping pour la clé et false sinon.
- `public boolean containsValue(Object value)`  
retourne true si la table a une ou plusieurs clés pour la valeur et false sinon.
- `public boolean isEmpty()`  
retourne vrai si le HashMap ne contient aucun mappings
- `public Set<K> keySet()` : retourne un ensemble des clés
- `public Collection<V> values()` : retourne une collection des valeurs
- `public V remove (Object key)`  
retire le mapping pour la clé s'il est présent et retourne la valeur associée à la clé ou null

## – Définition et usage des exceptions

- Traitement des exceptions
- Lancer des exceptions
- Définir des exceptions

## Erreurs d'exécution

- Pendant leur exécution les applications peuvent échouer sur plusieurs sortes d'erreurs de différents degrés de sévérité, provoquant la terminaison du programme.
- Bien qu'il soit nécessaire de prévoir un certain nombre de vérifications pour produire un logiciel fiable, parfois il peut être fastidieux de vérifier tous les cas possibles d'erreurs.
- Le langage java offre des mécanismes pour gérer et traiter les erreurs d'exécution.

## Introduction

- Tout programme comporte des erreurs ou est susceptible de générer des erreurs.
- La **fiabilité** d'un logiciel peut se composer de deux grandes propriétés :
  - La **robustesse** : capacité du logiciel à continuer de fonctionner en présence d'événements exceptionnels
  - La **correction** : capacité d'un logiciel à donner des résultats corrects lorsqu'il fonctionne normalement

## Un cas d'erreur d'exécution

```
public class TestException {  
    public static void main(java.lang.String[] args) {  
        int i = 3;  
        int j = 0;  
        System.out.println("résultat = " + (i / j));  
    }  
}
```

```
> java TestException  
Exception in thread "main" java.lang.ArithmeticException: / by  
zero  
at TestException.main(TestException.java:6)
```



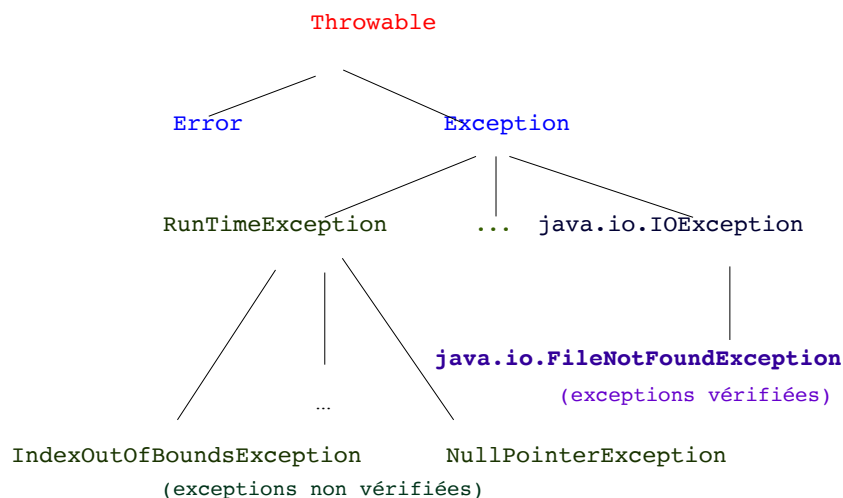
## Pourquoi utiliser des exceptions ?

- Le terme **exception** désigne tout événement arrivant durant l'exécution d'un programme interrompant son fonctionnement normal.
- Les exceptions fournissent un mécanisme de signalisation explicite des erreurs et de gestion des erreurs intégré au langage Java.
- Les exceptions peuvent améliorer la lisibilité d'un programme, faciliter sa lecture et sa maintenance.
- Mais si on utilise les exceptions incorrectement elles peuvent avoir les effets opposés.
- Il faut utiliser les exceptions pour des conditions exceptionnelles et elles ne remplacent en aucun cas les tests.

## Le principe des exceptions

- Les exceptions sont des objets représentant les erreurs. Une exception est lancée quand une condition d'erreur inattendue est rencontrée.
- Lors de la détection d'une erreur, un objet qui hérite de la classe `Exception` est créé (on dit qu'une exception est levée) et propagé à travers la pile d'exécution jusqu'à ce qu'il soit traité.
- Les exceptions se propagent en remontant les structures lexicales des méthodes, puis la pile d'appel des méthodes.

## Hierarchie des classes d'exception



## Les exceptions vérifiées

- Une **exception vérifiée** (ou contrôlée) décrit un problème qui a des chances d'arriver, peu importe si vous avez été prudent ou non.
  - La fin non attendue d'un fichier, panne réseau
- Une exception vérifiée hérite de la classe `Exception` et n'est pas du type `RuntimeException`.
- Elle est dite **vérifiée** car le compilateur vérifie que toutes les méthodes l'utilisent correctement
- Exemples :
  - `FileNotFoundException`
  - `ClassNotFoundException`
  - ...

## Les exceptions non vérifiées

- Une **exception non vérifiée est de votre faute**
  - Ex : vous êtes à blâmer pour un `NullPointerException` parce que votre code n'est pas bon et utilise une référence nulle
- Le compilateur ne vérifie pas si vous gérez un `NullPointerException` parce que vous devez tester vos références pour null avant de les utiliser plutôt que d'utiliser un gestionnaire pour cette exception.
- Java définit de nombreuses sous-classes `RuntimeException` :
  - `ArithmeticException`, `IndexOutOfBoundsException`, `IllegalArgumentException`...

IB – R2.01

9/29

## Différences exceptions vérifiées et non vérifiées

- Si une méthode doit lancer une exception vérifiée, Java demande que l'exception soit listée dans l'entête de la méthode.
  - Les exceptions non vérifiées n'ont pas besoin d'être listées dans l'entête.
- Si du code appelle une méthode qui doit lancer une exception vérifiée, Java demande qu'il gère l'exception.
  - Les exceptions non vérifiées n'ont pas besoin d'être gérées dans le code appelant.

IB – R2.01

10/29

## Lancer des exceptions

- Une méthode Java peut se terminer en lançant une exception.
- Il faut utiliser l'instruction

`throw <instance d'une classe Throwable>`

- Exemple pour une exception non vérifiée (juste pour le cours)

```
public int moyenneEntiere (int notes[], int nb){
    int moy = 0;
    if (nb < 0) throw new
        ArithmeticException("moyenneEntiere.division par 0");
    for (int i = 0; i < nb; i++) moy = moy + notes[i];
    moy = moy / nb;
    return moy;
}
```

IB – R2.01

11/29

## Lancer des exceptions (suite)

- Exemple pour une exception vérifiée
- On indique dans l'entête de la méthode que l'exception peut être lancée dans cette méthode.

```
public class DataSet {
    . . .
    public void read(String filename)throws
        FileNotFoundException {
        FileReader = new FileReader(filename);
        Scanner in = new Scanner(Reader)
        ""
    }
}
```

Le constructeur de `FileReader` peut lancer un `FileNotFoundException`, il faut le signaler au compilateur.

IB – R2.01

12/29

## Définir de nouveaux types d'exception

- La déclaration d'un nouveau type indique s'il est vérifié ou non par son super-type.

```
public class NewKindOfException extends Exception {  
    public NewKindOfException() {super();}  
    public NewKindOfException(String s) {super(s);}  
}
```

- La classe d'exception a uniquement besoin de constructeurs.

```
Exception e1 = new NewKindOfException ("this is the reason") ;  
  
String s = e1.toString() ;  
  
La chaîne s contient: "NewKindOfException: this is the reason"
```

## Gestion des exceptions

- Le mécanisme des exceptions en Java comprend trois mots-clés qui permettent de détecter et traiter les erreurs : **try**, **catch**, **finally**.
  - try** définit un bloc de code
  - catch** intercepte un type particulier d'exception
  - finally** contient un bloc généralement utilisé pour faire des nettoyages (fermer les fichiers, libérer des ressources, ...)

## Les blocs try/catch/finally

```
try  
{  
    //code a priori sans problème  
}  
catch (TypeException1 e1)  
{  
    //gère un objet exception e1 de type TypeException1  
}  
catch(TypeException2 e2)  
{  
    //gère un objet exception e2 de type TypeException2  
}  
finally  
{  
    // code exécuté lorsque l'on quitte la clause try  
}
```

## La clause try

- La clause try s'applique à un bloc d'instructions correspondant au fonctionnement normal mais pouvant générer des erreurs.
- Elle se contente de définir un bloc de code dont les sorties doivent être regardées de plus près.
- Si une clause **try** existe alors nécessairement il y a :
  - une (ou plusieurs) clauses **catch**
  - ou une clause **finally**
  - ou les deux, **catch** et **finally**
- Attention : un bloc ne compile pas si aucune de ses instructions n'est susceptible de lancer une exception.

## La clause `catch`

- Elle contient le code de gestion pour un type donné d'exception.  

```
catch (UneException e)
{
}
```
- L'argument `e` est du type **Throwable** ou une sous-classe.
- `UneException` doit être une sous-classe de `Throwable`
- Le code de la clause **catch** doit faire le nécessaire pour gérer la condition exceptionnelle.
- Par exemple si l'exception est « `FileNotFoundException` », dans la clause on peut demander à l'utilisateur de vérifier le nom du fichier.

## La clause `finally`

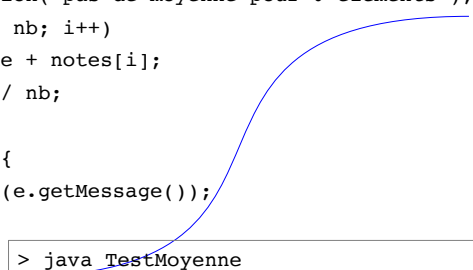
- La clause `finally` n'est pas obligatoire.
- Son intérêt est que son bloc de code est exécuté indépendamment de la façon dont on est sorti du bloc `try`.
- Trois cas sont envisageables pour l'enchaînement des clauses `try/catch/finally`:
  - cas normal : pas d'exception
  - cas exceptionnel 1 : un retour dû à `return/break/continue`
  - cas exceptionnel 2 : exception

## Règles sur les blocs `try-catch`

- Les blocs `try` et `catch` sont liés :
  - Tout bloc **try** doit être suivi par au moins un bloc **catch** ou un bloc **finally**.
  - Tout bloc **catch** doit être précédé par un autre bloc **catch** ou par un bloc **try**.

## Exécution de `finally` quand on sort de `try`

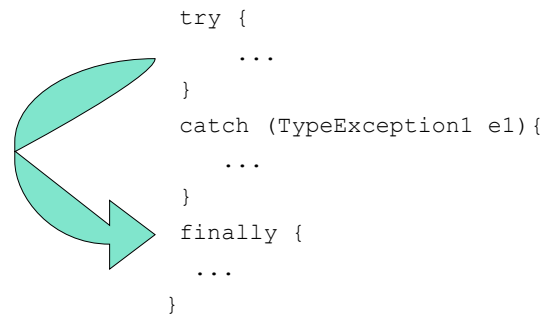
```
public int moyenneEntiere (int notes[], int nb) {
    int moyenne = 0;
    try {
        if (nb <=0)
            throw new Exception("pas de moyenne pour 0 éléments");
        for (int i = 0; i < nb; i++)
            moyenne = moyenne + notes[i];
        moyenne = moyenne / nb;
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
    finally{
        return moyenne;
    }
}
```



```
> java TestMoyenne
pas de moyenne pour 0 éléments
0
```

## Cas normal

- Pas d'exception générée dans le bloc `try`.
- Aucune instruction `return`, `break` ou `continue`.
- L'interpréteur atteint la fin du bloc `try` sans problème et poursuit avec l'exécution du code contenu dans le bloc `finally`.
- Aucune clause `catch` n'est exécutée.



IB - R2.01

21/29

## 1<sup>er</sup> cas exceptionnel

- L'interpréteur quitte le bloc `try` à cause d'une instruction `return`, `break` ou `continue`.
- L'interpréteur quitte le bloc `try` et poursuit avec l'exécution du code contenu dans le bloc `finally`.
- Aucune clause `catch` n'est exécutée.

IB - R2.01

22/29

## 2<sup>ème</sup> cas exceptionnel

- Une exception est déclenchée par une instruction du bloc `try`.
- L'interpréteur quitte immédiatement le bloc `try`.
- Si une clause `catch` est capable de gérer l'exception alors l'interpréteur exécute le code contenu dans cette clause.
- Puis l'interpréteur exécute le code défini dans la clause `finally`.
- Si aucune clause `catch` n'existe ou ne gère l'exception, le contrôle est passé au bloc `finally` puis, l'exception continue à se propager jusqu'à rencontrer la clause `catch` la plus proche dans la pile d'appel qui soit capable de gérer l'exception.
- Si aucune clause `catch` n'est rencontrée, l'interpréteur s'arrête et affiche un message d'erreur.

IB - R2.01

23/29

## Exemple

```
try {  
    String filename = ...;  
    Scanner in = new Scanner(new File(filename));  
    String input = in.next();  
    int value = Integer.parseInt(input);  
    ...  
}  
catch (IOException e) {  
    e.printStackTrace();  
}  
catch (NumberFormatException e) {  
    System.out.println("Input was not a number");  
}
```

Trois exceptions peuvent être lancées dans ce bloc `try` :

- Le constructeur de `Scanner` peut lancer un `FileNotFoundException`.
- `Scanner.next()` peut lancer un `NoSuchElementException`.
- `Integer.parseInt()` peut lancer un `NumberFormatException`

IB - R2.01

24/29

## Les méthodes utiles de la classe Throwable

- `String getMessage()` : retourne la chaîne qui a été donnée à l'objet exception par son constructeur
- `void printStackTrace()` : imprime ce throwable et toute sa trace
  - D'autre messages d'impression avec un flux de sortie en argument
- `String toString()` : retourne une description courte

## La méthode `getMessage()`

- Chaque exception possède une méthode `getMessage()` qui retrouve la chaîne qui a été donnée à l'objet exception par son constructeur.

```
try {  
    if (nb < 0) throw new Exception("pas de moyenne pour 0  
                                éléments");  
  
    ...  
}  
  
catch (Exception e) {  
    System.out.println(e.getMessage());  
}
```

- Le test `nb < 0` n'est pas bon et une exception sera levée. On obtient :  
\$ java TestMoyenne  
pas de moyenne pour 0 éléments

## Spécifier les exceptions lancées par une méthode

```
public void writeList() throws IOException{  
    PrintWriter out = new PrintWriter(  
                                new FileWriter("OutFile.txt"));  
    for (int i = 0; i < SIZE; i++) {  
        out.println("Value at: " + i + " = " + list.get(i));  
    }  
    out.close();  
}
```

Cet exemple peut lancer 2 exceptions

`IndexOutOfBoundsException` (non vérifiée donc on ne le signale pas)

`FileWriter` peut lancer `IOException`, on l'indique

## Définir de exceptions personnalisées

La forme générale est :

```
public class ExceptionName extends ExistingExceptionName{  
    public ExceptionName (String message) {  
        super(message) ;  
    }  
}
```

La plupart du temps on les définit comme sous-classe de :

`Exception`, `IOException` ou `FileNotFoundException`

Le paramètre permet de personnaliser le message.

## Pour vérifier les paramètres des méthodes

L'exception suivante pourra être utilisée si le test des paramètres d'une méthode révèlent une valeur non appropriée et que l'on souhaite arrêter l'exécution de l'application (typiquement dans les constructeurs) :

```
public class IllegalArgumentException
extends RuntimeException
Thrown to indicate that a method has been passed an illegal or
inappropriate argument.
```

Son constructeur :

```
public IllegalArgumentException(String s)
Constructs an IllegalArgumentException with the specified detail
message.
```