




COURS JAVA AVANCÉ

Mme Nouria Sana

SOMMAIRE

1. **LES PACKAGES ET L'ENCAPSULATION**
 2. **LES INTERFACES – GESTION DES EXCEPTIONS**
 3. **INTERFACES FONCTIONNELLES ET EXPRESSIONS LAMBDA**
 4. **LES CLASSES DE BASE ET COLLECTIONS**
 5. **LES STREAMS**
 6. **JAVAFX ET PROGRAMMATION ÉVÉNEMENTIELLE**
 7. **IMPLÉMENTATION DU MVC EN JAVA**
 8. **PROGRAMMATION CONCURRENTE ET THREADS**
 8. **E/S : ACCÈS AUX BASES DE DONNÉES AVEC JAVA – LES FICHIERS (À FAIRE PAR L'ETUDIANT)**
- 

LES PACKAGES ET L'ENCAPSULATION

○ Les packages

- Un package peut être considéré comme une bibliothèque de classes
- Il permet de regrouper un certain nombre de classes qui sont proches dans une seule famille et ainsi d'apporter un niveau de hiérarchisation supplémentaire au langage.
- Les packages sont eux-mêmes organisés hiérarchiquement. : on peut définir des sous-packages, des sous-sous-packages, ...



LES PRINCIPAUX PACKAGES DU JDK

- **java.applet** Classes de base pour les Applets
- **java.awt** Interface Utilisateur
- **java.io** Entrées/Sorties, fichiers, ...
- **java.lang** Classes faisant partie du langage
- **java.math** Utilitaires mathématiques
- **java.net** Accès au réseau
- **java.security** Gestion de la sécurité
- **java.sql** Accès aux bases de données
- **java.util** Conteneurs, dates, ...
- **java.util.zip** Pour compresser/décompresser



Il y a deux manières d'utiliser une classe stockée dans un package :

- En utilisant le nom du package suivi du nom de la classe

```
java.util.Date now = new java.util.Date() ;  
System.out.println(now) ;
```

En utilisant le mot clé **import** pour importer (inclure) le package auquel appartient la classe

```
import java.util.Date ;           // Doit être en tête du fichier !!  
// Ne permet d'importer que la classe Date de java.util
```

```
...  
Date now = new Date() ;  
System.out.println(now) ;
```

```
import java.util.* ;              // Permet d'importer toutes les  
classes de java.util
```

```
...  
Date now = new Date() ;  
System.out.println(now) ;
```

On peut généralement utiliser l'une ou l'autre de ces deux méthodes, mais il existe un cas où l'on doit obligatoirement utiliser la première : si deux classes portant le même nom sont définies dans deux packages différents.

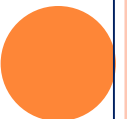
LA STRUCTURE DE STOCKAGE DES CLASSES ET DES PACKAGES

// Fichier *Classe1.java* dans le répertoire *test/util/*

```
package test.util ;  
public class Classe1 {  
    public void test() { ... }  
}
```

// la déclaration de package est inutile, puisque cette classe n'est pas destinée à être utilisée par d'autres

```
import test.util.* ;  
public class Appli1 {  
    public static void main(String args[]) {  
        Classe1 c = new Classe1() ;  
        c.test() ;  
    }  
}
```



LES RÈGLES DE VISIBILITÉ DES ATTRIBUTS ET MÉTHODES : PUBLIC, PROTECTED, PRIVATE, FRIENDLY

- *La visibilité friendly est celle prise par défaut.*

Accessible aux :	méthodes de la même classe	classes dérivées dans le même package	classes du même package	classes dérivées dans un autre package	classes des autres packages
public	X	X	X	X	X
protected	X	X	X	X	
friendly	X	X	X		
private	X				

LES CLASSES IMBRIQUÉES ET LES CLASSES ANONYMES

- Il est possible de définir des classes dans des classes. Il existe deux types de classes ainsi définies :
 - **Les classes imbriquées** : Elles possèdent un nom et sont définies au même niveau qu'une méthode ou un attribut.
 - **Les classes anonymes** : Elles ne possèdent pas de nom et sont définies là où elles sont utilisées, c'est à dire dans le code



EXEMPLE DE CLASSE IMBRIQUÉE :

```
public class Voiture {  
    class Roue {  
        private String modele ;  
        Roue(String modele) { this.modele = modele ; }  
    }  
    private Roue[] roues = new Roue[4] ;  
    private int puissance =10 ;  
  
    public Voiture(String modele_roue, int puissance) {  
        this.puissance = puissance ;  
        for ( int i=0 ; i<roues.length ; i++ ) roues[i] = new  
        Roue(modele_roue) ;  
    }  
}
```



EXEMPLE DE CLASSE ANONYME

```
class Commande {  
    public void go() { System.out.println("Pas de commande") ; }  
}  
  
class Shell {  
    private Vector commandes = new Vector() ; // Objet permettant de contenir N objets  
    public void addCommande( Commande commande) {  
        commandes.addElement(commande) ;  
    }  
    public void executer() {  
        for ( /* Parcours de tous les éléments */ )  
            ((Commande)e.nextElement()).go() ; } }  
  
class Principale {  
    Shell shell = new Shell() ;  
    public static void main(String[] args) {  
        shell.addCommande(  
            new Commande() {  
                public void go() { System.out.println("commande") ;  
                ... shell.executer() ; } }  
        }) ;  
    }  
}
```



LES INTERFACES

- Une interface est une déclaration permettant de décrire un ensemble de méthodes abstraites et de constantes. On peut considérer une interface comme étant une classe abstraite ne contenant que des méthodes abstraites et que des attributs final et static.
- en utilisant le mot clé **implements**



- Une interface est définie de manière similaire à une classe, mais est caractérisée par le mot clé **interface**.

```
interface Printable {  
    void print() ;  
}
```

```
class Point extends Object implements Printable {  
    private double x,y ;  
    ...  
    void print() {  
        System.out.println( "(" + x + "," + y)" );  
    }  
}
```



INTERFACES ET HÉRITAGE

- Une interface peut aussi hériter d'une ou plusieurs interfaces : elle hérite alors de l'ensemble des méthodes abstraites et constantes de ses ancêtres


```
interface Printable {          /* exprime le fait de pouvoir être imprimé */
    void print() ;
}

interface InputStream {        /* exprime le fait de pouvoir être une source de
    caractères*/
    public int read() ;
}

interface OutputStream {       /* exprime le fait de pouvoir accepter des
    caractères */
    public void write(int) ;    }

interface DataStream extends InputStream{
    public double readDouble() ;
    public void writeDouble(double) ;
}

class MonStream implements DataStream,
Printable {
    void print() { // ...}
    public int read() { // ...}
    public double readDouble(){// }
    public void writeDouble(double) { ...}
}
```



- Parmi les nouveautés apportées par Java 8, on en trouve deux qui concernent les interfaces : **les méthodes statiques et les méthodes par défaut**.
 - Les méthodes statiques définies sur les interfaces fonctionnent exactement de la même façon que celles portées par les classes, il n'y a donc pas grand-chose à en dire.
 - En revanche, les méthodes par défaut risquent de modifier assez profondément notre façon de concevoir nos API.
- En Java 7 et antérieur, une méthode déclarée dans une interface ne fournit pas d'implémentation. Ce n'est qu'une signature, un contrat auquel chaque classe dérivée doit se conformer en fournissant une implémentation propre.
- Mais il arrive que plusieurs classes similaires souhaitent partager une même implémentation de l'interface. Dans ce cas, deux stratégies sont possibles
 - Factoriser le code commun dans une classe abstraite, mais il n'est pas toujours possible de modifier la hiérarchie des classes
 - Extraire le code commun dans une classe utilitaire, sous forme de méthode statique (ex: `Collections.sort()`).
- On conviendra qu'aucune des deux n'est réellement satisfaisante. Heureusement, Java 8 nous offre maintenant une troisième possibilité.



AVEC JAVA QUELQUES NOUVEAUTÉS

- Java 8 propose en effet une solution plus propre :
 - permettre aux méthodes déclarées dans les interfaces d'avoir une implémentation !
 - Là, tout le monde se frappe le front en disant, bon sang mais c'est bien sûr, pourquoi n'y a-t-on pas pensé avant ? Tout simplement parce que les concepteurs du langage voulaient absolument éviter les problèmes d'héritage en diamant, bien connu des développeurs C++. On verra (plus loin) que ce n'est finalement pas un problème en Java.



○ Syntaxe

La syntaxe est simple et sans surprises : il suffit de fournir un corps à la méthode, et de la qualifier avec le mot-clé default

```
public interface Foo {  
    public default void foo() {  
        System.out.println("Default implementation of foo()");  
    }  
}
```



- Les classes filles sont alors libérées de l'obligation de fournir elles-mêmes une implémentation à cette méthode - en cas d'absence d'implémentation spécifique, c'est celle par défaut qui est utilisée.

```
public interface Itf {  
  
    /** Pas d'implémentation - comme en Java 7 et antérieur */  
    public void foo();  
  
    /** Implémentation par défaut, qu'on surchargera dans la classe fille */  
    public default void bar() {  
        System.out.println("Itf -> bar() [default]");  
    }  
  
    /** Implémentation par défaut, non surchargé dans la classe fille */  
    public default void baz() {  
        System.out.println("Itf -> baz() [default]");  
    }  
  
}
```

```
public class Cls implements Itf {

    @Override
    public void foo() {
        System.out.println("Cls -> foo()");
    }

    @Override
    public void bar() {
        System.out.println("Cls -> bar()");
    }
}
```

Et le test :

```
public class Test {
    public static void main(String[] args) {
        Cls cls = new Cls();
        cls.foo();
        cls.bar();
        cls.baz();
    }
}
```

Résultat :

```
Cls -> foo()
Cls -> bar()
Itf -> baz() [default]
```

Comme prévu, l'implémentation de la classe est préférée à celle de l'interface (méthode bar()), et si la classe ne fournit pas d'implémentation, c'est celle de l'interface qui est utilisée (méthode baz()).

INTERFACE FONCTIONNELLE-

- **Une interface fonctionnelle** est une interface comprenant exactement une seule méthode abstraite.
- L'annotation **@FunctionalInterface** peut précéder la déclaration des interfaces fonctionnelles . Elle n'est pas obligatoire
- Interface fonctionnelle qui permet de définir une interface disposant d'une unique méthode abstraite, c'est-à-dire une seule méthode ne possédant pas d'implémentation par défaut.

- **Exemple : interface Printable**

@FunctionalInterface

public interface Printable { void print(); }



INTERFACE DE MARQUAGE

- **Une interface de marquage** est une interface entièrement vide. Une classe peut implémenter cette interface en la nommant dans la clause implements sans avoir à implémenter de méthodes. Toutes les instances de la classe deviennent des instances valables de l'interface. Il est donc possible de tester si un objet implémente l'interface de marquage à l'aide de l'opérateur **instanceof**:

if (obj instanceof Cloneable) {...}

- Les interfaces **java.lang.Cloneable** et **java.io.Serializable** constituent deux exemples d'interfaces de marquage de la plate-forme *Java*.
- Les interfaces peuvent avoir des sous-interfaces. On utilise le mot clé: extends. Contrairement aux classes, une interface peut posséder plusieurs interfaces parentes.
 - **Exemple : public interface Zoomable extends Modifiable, Translatable, Rotatable {...}**
- Une classe qui implémente une sous-interface doit implémenter les méthodes abstraites définies directement par l'interface ainsi que les méthodes abstraites héritées de toutes les interfaces parentes de la sous-interface



LES EXPRESSIONS LAMBDA

- Les deux importants apports du Java 8 résident dans les **expressions Lambda et les streams**.
 - Ils sont utilisés conjointement pour offrir les possibilités de la « **programmation fonctionnelle** ».
- L'idée est de traiter des structures de données parce qu'on souhaite obtenir, sans avoir besoin d'explicitement comment y parvenir.
- Comme premier exemple d'expression lambda est de l'employer pour remplacer avantageusement **le recours à une classe anonyme**:
 - Ce mécanisme ne fonctionne que si l'interface comporte une seule méthode abstraite qui est le cas des interfaces fonctionnelles.
 - La méthode évoquée est appelée méthode fonctionnelle.
 - Le compilateur est a priori en mesure de s'assurer que l'interface représentée par une expression Lambda est bien fonctionnelle.
 - Il est bien possible d'ajouter une annotation de la forme (@FunctionalInterface).
 - Parmi les interfaces fonctionnelles standards, on peut citer :
 - Iterable, Closeable, Comparable, Callable, Readable, Flushable, Formattable, FileFilter...



QUELQUES RÈGLES POUR LES EXPRESSIONS LAMBDA :

- Une expression Lambda peut renvoyer ou non une valeur.
- Cette valeur peut être un bloc avec des return classiques.
- Elle peut comporter plusieurs arguments et préciser leurs types.
 - Exemples :
 - `(x,y) -> x*x+x*y ;`
 - `(Point p, Float x) -> {System.out.println(x) ; p.afficher() ;} ;`
 - `()-> System.out.println(.....) ;`
- L'expression Lambda doit avoir autant d'arguments que la méthode fonctionnelle correspondante
- et les expressions renvoyées doivent être de type compatible avec celui attendu par l'interface fonctionnelle.
 - --- voir le projet « exemplesLambda » sous NetBeans IDE



EXEMPLESLAMBDA »

```
public interface Calculable {  
    public int calculer(int n);  
}
```

// interface crée pour tester la composition des expressions Lambda

```
public interface FabriqueCalcul {  
    public Calculable fabriquer() ;  
}
```

```
public class ExemplesLambda {
```

```
    public static void traiter (int n, Calculable cal)  
    { int res = cal.calculer(n);  
      System.out.println(res);}
```

// expression Lambda dans une instruction return

```
    public static Calculable fabriquer(){  
        double x = Math.random();  
        if(x < 0.5) return y -> y*y;  
        else return y -> 2*y;}
```

```
public static void main(String[] args) {
```

// test de l'expression Lambda dans une instruction return

```
    for (int i=0;i<3;i++)  
        traiter(4,fabriquer());
```

// test de la composition des expression Lambda

```
    FabriqueCalcul fabCarre = () -> x -> x*x ;  
    FabriqueCalcul fabDouble = () -> x -> 2*x ;
```

```
    traiter(4,fabCarre.fabriquer());  
    traiter(4,fabDouble.fabriquer());
```

// tableau d'expression Lambda

```
    System.out.println("tableau des expressions Lambda");
```

```
    Calculable[] tab = {x -> x*x, x -> 2*x,  
                        x -> (int) Math.sqrt(x)} ;
```

```
    for (Calculable cal : tab) // balayage d'un tableau par  
        foreach  
        {  
            traiter(15,cal) ;  
        }  
}
```



RÉFÉRENCES DE MÉTHODES

- Comme les expressions Lambda permettent d'exprimer une méthode fonctionnelle d'une interface fonctionnelle en introduisant le code correspondant à l'emplacement où l'on a besoin, les références de méthodes vont offrir une autre sorte de raccourci qui peut s'appliquer à des méthodes statiques, des méthodes de classe, des méthodes associées à un objet particulier ou à des constructeurs.

- Exemple :

- **Rq :**

- // Référence à un constructeur
 - // constructeur de la classe A : A ::new
 - // constructeur d'un tableau d'éléments de type A: A[] ::new

EXEMPLE RÉFÉRENCE

```
public class TestReferences {  
    // méthodes statiques  
  
    public static void traiter (int n, Calculable cal)  
    {   int res = cal.calculer(n);  
        System.out.println(res);}  
  
    public static int carre(int n){return n*n;}  
  
    public static void main(String[] args) {  
  
        // référence à une méthode statique  
        traiter(5,TestReferences::carre);  
  
        // référence à une méthode de classe  
        Point p1 = new Point(1,3);  
        Point p2 = new Point(2,5);  
        Distanciable db = Point::distance;  
        System.out.println(db.distance(p1,p2));  
  
        // référence à une méthode d'objet  
        Point Origine = new Point(0,0);  
        Distanciable1 db1 = Origine::distance;  
        System.out.println(db1.distance(p1));  
    }  
}
```

```
public interface Calculable {  
    public int calculer(int n);  
}
```

```
public interface Distanciable {  
    public double distance(Point p1, Point p2);  
}
```

```
public interface Distanciable1 {  
    public double distance(Point p);  
}
```

```
public class Point {  
    double x, y;  
    Point (double x, double y)  
    {this.x=x; this.y=y;}  
    public double distance( Point p)  
    {return(0);}  
}
```



EXEMPLE INTERFACE FONCTIONNELLE //NETBEANS STREAM1

- **Exercice :**

- Soit le code Java implémentant les interfaces fonctionnelles suivantes :
- `Function <Integer,Double> fun = new Function <Integer,Double>(){`
- `public Double apply (Integer i){return i*10.0;}};`
- `Consumer <String> cons = new Consumer <String>(){`
- `public void accept (String s){System.out.println(s+" is consumed");} };`
- `Supplier <String> supp = new Supplier <String>(){`
- `public String get(){return "Success";} };`
- `Predicate <Double> pre = new Predicate <Double>(){`
- `public boolean test (Double n){return n<20;} };`
- a) transformez ce code en utilisant les expressions Lambda
- b) Donnez un exemple d'application avec les « List » et « Map » vu précédemment dans lequel on a utilisé implicitement l'interface Consumer.

a) `Function <Integer,Double> fun1 = i->i*10.0;`
`Consumer <String> cons1 = s->System.out.println(s+" is consumed");`
`Supplier <String> supp1 = ()-> "Suceess";`
`Predicate <Double> pre1 = n-> n<20;`

b) **// création de d'une liste avant Java 9**

`List <String> liste = new ArrayList();`
`liste.add("Camion");liste.add("voiture"); liste.add("train");`
`liste.forEach(System.out::println);`
// en une seule instruction
`Arrays.asList("Camion","voiture","train").forEach(System.out::println);`
// autres collections

`Map <Integer,String> map = new HashMap();`
`map.put(1,"Camion"); map.put(2,"voiture"); map.put(3,"train");`
`map.entrySet().forEach(System.out::println);`



INTERFACE FONCTIONNELLE UNARYOPERATOR

```
UnaryOperator<Integer> operator = t -> t * 2;  
System.out.println(operator.apply(5));  
System.out.println(operator.apply(10));  
System.out.println(operator.apply(15));  
UnaryOperator<Integer> operator1 = t -> t + 10;  
UnaryOperator<Integer> operator2 = t -> t * 10;  
int a = operator1.andThen(operator2).apply(5);  
System.out.println(a);  
int b = operator1.compose(operator2).apply(5);  
System.out.println(b);
```

- // autres

```
DoubleBinaryOperator operator3 = (p, q) -> (p + q);  
System.out.println(operator3.applyAsDouble(5, 6));
```

- // Il existe des interfaces fonctionnelles spécifiques au Java 8:
- // LongUnaryOperator, LongToIntFunction, IntSupplier, DoubleConsumer,
- // IntToDoubleFunction, LongPredicate



LES EXCEPTIONS

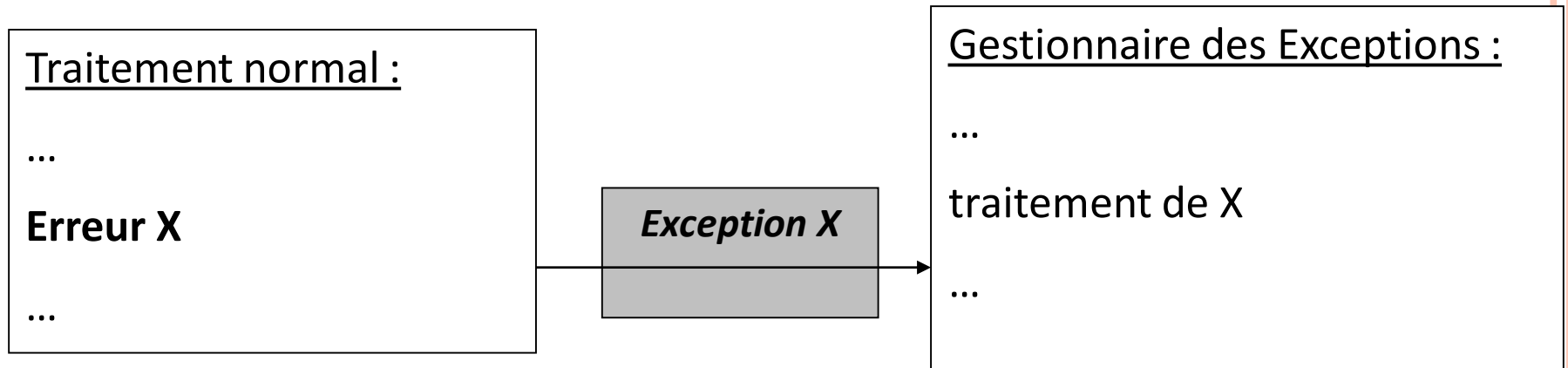
- En java les exceptions représentent le mécanisme de gestion des erreurs
- Pourquoi des exceptions ?
 - Le code est peu lisible, on ne distingue pas le traitement normal des traitements des cas exceptionnels (qui ont souvent une logique très différente)
 - Des traitements d'erreurs sont souvent oubliés par le programmeur.
 - Il est souvent difficile de traiter de manière cohérente l'ensemble des erreurs : comment distinguer un résultat valide d'un cas d'erreur
 - Par exemple si le code retour d'une fonction est utilisé pour transmettre le résultat, où doit-on passer le code d'erreur ?



LA NOTION D'EXCEPTION

- Un mécanisme facilitant le traitement de tous les cas exceptionnels et permettant en particulier de séparer ces traitements des traitements habituels du programme.
- Un cas exceptionnel est représenté **par un objet** (une exception), qui contient la description du cas exceptionnel, et qui peut être transmis de l'endroit où il a été déclenché jusqu'à celui où l'on sait le traiter (un gestionnaire d'exception).

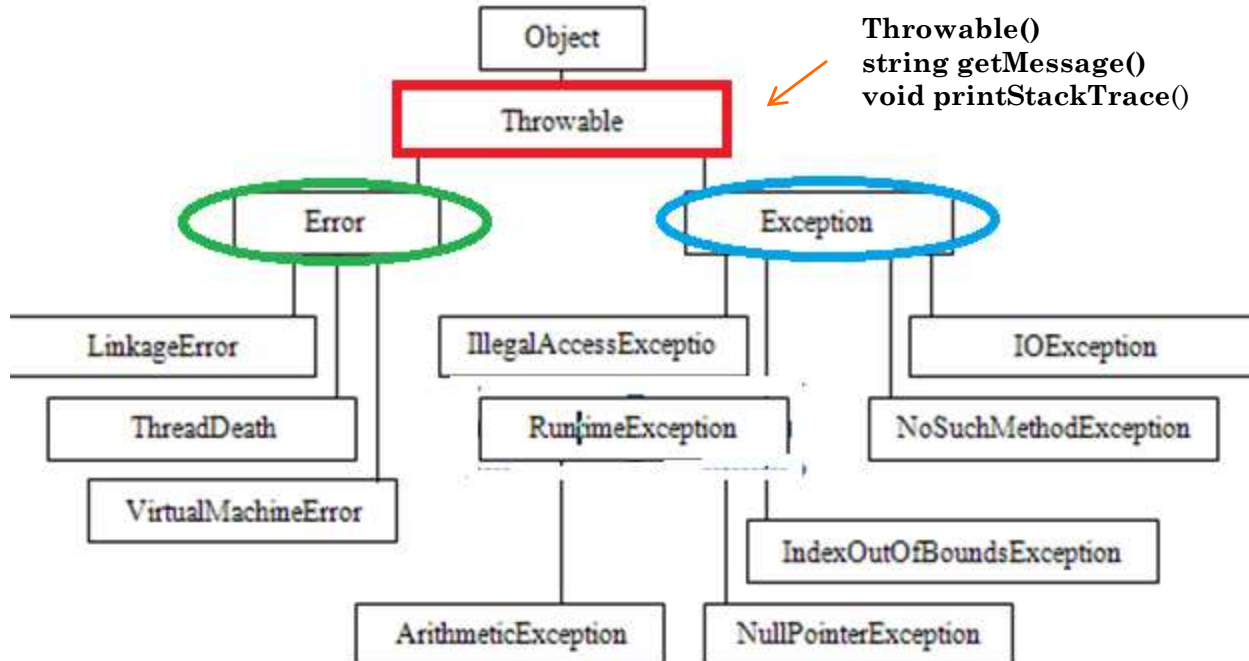
Dans Java, contrairement à C++, le traitement des exceptions est vérifié par le compilateur.



REPRÉSENTATION ET DÉCLENCHEMENT DES EXCEPTIONS

○ Classe Exception et Error

- C'est deux classes dérivent de Throwable
- La classe Error représente une erreur Grave intervenue dans la machine virtuelle Java ou dans un sous système Java
 - L'application java s'arrête instantanément des l'apparition d'une exception de la classe Error
- La classe Exception représente des erreurs moins grave
 - Une exception dans Java est un objet appartenant à une classe dérivée de la classe **java.lang.Exception**



LANCER UNE EXCEPTION

- Le mot clé opérateur « **throw** »
- Une exception est déclenchée avec le mot clé **throw**, après avoir créé le nouvel objet **Exception** correspondant

```
class MonException extends Exception {  
    // constructeur  
}  
class Test{  
    public static void main(String []args)  
    {  
        ....  
        if ( erreurDétectée ) throw new MonException();  
        ...  
    }  
}
```



RATTRAPER UNE EXCEPTION

- Lorsqu'une exception est déclenchée dans une méthode
 - soit directement avec **throw**,
 - soit parce que une méthode appelée dans cette méthode la déclenche et la propage)
- il y a deux possibilités
 - 1 - On ne traite pas l'exception localement **et on la propage**.
 - 2 - On traite l'exception localement (try – catch).



1 - PROPAGATION DES EXCEPTIONS

○ le mot clé **throws**

- Toute exception pouvant être propagée par une méthode doit être signalée dans la déclaration de celle-ci.
- Pour cela on utilise le mot clé **throws** suivi de la liste des exceptions.



```
class PasDeSolution extends Exception { ...//Constructeur..... }
```

```
class Equation { /* Equation du second degré  $ax^2+bx+c$  */
```

```
    private double a, b, c ;
```

```
    public Equation(double a, double b, double c) { this.a = a ; this.b = b ; this.c = c ; }
```

```
    public double resultat(){
```

```
    public double resultat() throws PasDeSolution { // Cette méthode propage une exception
```

```
        double discriminant =  $b*b-4*a*c$  ;
```

```
        if (discriminant < 0) throw new PasDeSolution() ;
```

```
        return ( b + Math.sqrt(discriminant) ) / ( 2 * a ) ;
```

```
    }
```

```
}
```

```
class test {
```

```
    void calcul(){
```

```
    void calcul() throws PasDeSolution {
```

```
        Equation eq = new Equation(1,0,1) ;
```

```
        Eq.resultat() ;
```

```
    }
```

```
    // Cette méthode doit déclarer la propagation de l'exception PasDeSolution que  
    Eq.resultat() peut déclencher, car elle ne la traite pas localement.
```

```
}
```



CAPTURE DES EXCEPTIONS : CATCH, TRY ET FINALLY

- Un gestionnaire d'exception est défini en :
 - Définissant quelles instructions sont surveillées par ce gestionnaire, en plaçant celles-ci dans un bloc d'instructions préfixé par le mot clé **try**.
 - Définissant un ou plusieurs blocs de traitement d'exception préfixés par le mot-clé **catch**.



```

class PasDeSolution extends Exception { ..... }

class Equation { /* Equation du second degré ax2+bx+c */
    private double a, b, c ;

    public Equation(double a, double b, double c) { this.a = a ; this.b = b ; this.c = c ; }

    public double resultat() throws PasDeSolution { // Cette méthode propage
une exception

        double discriminant = b*b-4*a*c ;
        if (discriminant < 0) throw new PasDeSolution() ;
        return ( b + Math.sqrt(discriminant) ) / (2 * a) ;
    }
}

class test {
    void calcul() {
        try {
            Equation eq = new Equation(1,0,1) ;
            double resultat = Eq.resultat() ;
            System.out.println("Resultat = " + resultat) ;
        }
        catch ( PasDeSolution e ) {
            System.out.println("Pas de solutions") ;
        }
    }
}

```



```
try {  
    ... // 1  
}  
catch (Exception e) { /* permet de capturer tout type d'exception */  
    ... //2  
}
```

Il est aussi possible de mettre plusieurs bloc `catch` :

```
try {  
    ...  
}  
catch (IOException e) { /* permet de capturer tout type d'exception d'entrée/sorties */  
    ...  
}  
catch (PasDeSolution e) {  
    ...  
}
```

LE BLOC FINALLY

- Il est possible de définir un bloc **finally** qui, contrairement au **catch**, n'est pas obligatoire, et qui permet de spécifier du code **qui sera exécuté dans tous les cas, qu'une exception survienne ou pas.**
- *Ce code sera exécuté même si une instruction **return** est exécutée dans le catch ou le try !*



```
try {  
    // Ouvrir un fichier  
    // Lire et écrire des données  
}  
catch ( IOException e ) { /* permet de capturer tout type d'exception d'entrée/sorties */  
    System.out.println(e) ;  
    return ;  
}  
finally {  
    // Fermeture du fichier  
}  
...// autres traitements sur les données lues...
```

- Nous constatons dans cet exemple qu'un objet Exception peut être directement affiché avec la méthode System.out.println().
- L'affichage consiste en une chaîne de caractères expliquant la nature de cette exception.



LA MÉTHODE PRINTSTACKTRACE()

```
catch ( IOException e ) { /* permet de capturer tout type  
    d'exception d'entrée/sorties */  
    System.out.println("Erreur du type : " + e) ;  
    e.printStackTrace() ;  
}
```

- Il est aussi possible en appliquant la méthode **printStackTrace()** sur l'objet exception,
 - d'afficher la pile des appels de méthodes remontées depuis le déclenchement de cette exception



EXERCICE

```
1- public class EntierNaturel
{
    private int Nbr ;
    // constructeur
    public EntierNaturel (int Nbr) throws
ExceptNeg
    {
        // il existe une exception lorsque le nombre
        donné est négatif
        // on peut générer une exception qu'on peut
        appeler ExceptNeg
        if (Nbr < 0) throw new ExceptNeg(Nbr) ;
        this.Nbr = Nbr ;
    }
    public int getNbr () { return Nbr ;}
}
```

```
2- public class ExceptNeg extends
Exception
{
    private int value;
    // constructeur
    public ExceptNeg (int value)
    { this.value = value ;}
    public int getValue() { return value;}
}
```



```

3- public class Testentiernaturel
{
public static void main (String args[])
{
// on peut tester en changeant les valeurs tels
que 100, -50...
int n;
Scanner C = new Scanner(System.in);
System.out.println("donner un entier
naturel : ") ;
n = C.nextInt() ;
try
{
EntierNaturel Nbr1 = new EntierNaturel (n);
System.out.println("le nombre naturel est
valide et =" + Nbr1.getNbr()) ;
// la question à penser : pourquoi on a utilisé
getNbr et non pas Nbr1.Nbr ?
}
catch ( ExceptNeg e)
{
System.out.println ("*****erreur de
construction du nombre***** ") ;}
finally
{
System.exit(-1) ;}
}

```

```

4- public class EntierNaturel
{ ...
public static EntierNaturel somme (EntierNaturel
Nbr1, EntierNaturel Nbr2) throws ExceptSom,

ExceptNeg
{
int op1 = Nbr1.Nbr; int op2= Nbr2.Nbr;
int S = op1 + op2 ;
if (S > Integer.Max_Value) throw new ExceptSom
(op1,op2);
EntierNaturel res = new EntierNaturel (S);
return res;
}
public static EntierNaturel diff (EntierNaturel Nbr1,
EntierNaturel Nbr2) throws ExceptDiff, ExceptNeg
{
int op1 = Nbr1.Nbr; int op2= Nbr2.Nbr;
int D = op1 - op2 ;
if (D<0) throw new ExceptDiff (op1,op2);
// autre façon en utilisant classe anonyme
return new EntierNaturel(D);
}
}

```



5 - **public class ExceptSom extends Exception**

```
{  
    public int value1, value2;  
    // constructeur  
    public ExceptSom (int value1, int value2)  
    { this.value1 = value1 ;  
      this.value1 = value1 ;}  
}
```

public class ExceptDiff extends Exception

```
{  
    public int value1, value2;  
    // constructeur  
    public ExceptDiff (int value1, int value2)  
    { this.value1 = value1 ;  
      this.value1 = value1 ;}  
}
```

//une autre façon est de créer une classe Except_op et des sous classes ExceptSom et ExceptDiff, et appeler super(value1, value2)

6 - On peut définir une classe : class ExceptNaturel extends Exception { }

Les autres seront des sous-classes :

```
class ExceptNeg extends ExceptNaturel { .....}  
class Except_op extends ExceptNaturel { .....}  
class ExceptSom extends Except_op { .....}  
class ExceptDiff extends Except_op { .....}
```

public class Test2entiernaturel

```
{  
    public static void main (String args[])  
    {  
        // on peut tester en changeant les valeurs tels que 100, -50...  
        int n1, n2;  
        System.out.println("donner un premier entier naturel : ") ;  
        n1 = Clavier.lireInt() ;  
        System.out.println("donner un second entier naturel : ") ;  
        n2 = Clavier.lireInt() ;  
        try{  
            EntierNaturel Nbr1 = new EntierNaturel (n1);  
            EntierNaturel Nbr2 = new EntierNaturel (n2);  
            EntierNaturel Som = EntierNaturel.somme (n1,n2) ;  
            EntierNaturel Di = EntierNaturel.diff (n1,n2) ;  
        }  
        // exception sans différenciation  
        catch ( ExceptNaturel e)  
        { Sytem.out.println ("****erreur entier naturel***** ") ;}  
        // test avec différenciation des exceptions  
        try {  
            EntierNaturel Nbr1 = new EntierNaturel (n1);  
            EntierNaturel Nbr2 = new EntierNaturel (n2);  
            EntierNaturel Som = EntierNaturel.somme (n1,n2) ;  
            EntierNaturel Di = EntierNaturel.diff (n1,n2) ;  
        }  
        catch ( ExceptNeg e)  
        { Sytem.out.println ("erreur construction entier  
                             naturel "+e.getValue() ) ; }  
        catch ( ExceptSom e)  
        { Sytem.out.println ("erreur somme des entiers  
                             naturels "+e.value1 + " "+e.value2 ) ; }  
        catch ( ExceptDiff e)  
        { Sytem.out.println ("erreur difference des entiers  
                             naturels "+e.value1 + " "+e.value2 ) ; }  
        Finally {System.exit(-1) ;} } }
```