Chapitre III : Concepts avancé de l'orienté objet

- La super-classe Object.
- La généricité.
- Les classes Abstraites
- Les interfaces.
- Traitement des exceptions

La super-classe Object

• **La super-classe Object :** C'est une classe dont dérive **implicitement** toute classe Java (Toutes les classe java sont dérivées **implicitement** de la classe **Object**).

• si on définit la classe Point par:

```
class Point{
…
}
```

tout se passe comme si on le fait explicitement par:

```
class Point extends Object{
…
}
```

La super-classe Object

• **Utilisation d'une référence de type Object :** Une variable de type ***Object*** peut être utilisée pour référencer un objet de type quelconque :

Exemple :

```
…
Point p = new Point(…);
Pointcol pc = new Point(…);
Fleur f = new Fleur(…);
Object o;
…
o=p;          // affectation légale (principe de polymorphisme)
o=pc;         // affectation légale (principe de polymorphisme)
o=f;          // affectation légale (principe de polymorphisme)
```

Utilité : Cette possibilité peut être utilisée pour manipuler des objets dont on ne connaît pas le type exact.

La super-classe Object

• **Utilisation de méthodes de la classe Object :** la classe Object dispose de quelques méthodes qu'on peut soit utiliser telles quelles, soit redéfinir.

- La méthode **toString** : elle fournit une chaîne de caractère contenant le nom de la classe concernée et l'adresse de l'objet en hexadécimal(précédée de @)

```
public class TestString{
      public static void main(String[] args) {
        Point p1 = new Point(1,2);
        Point p2 = new Point(2,7);
        System.out.println("l'objet p1 : "+p1.toString());
        System.out.println("l'objet p2 : "+p2.toString());
    }
}
class Point{
    public Point(int x,int y){
        this.x=x;  this.y=y;
    }
    int x; int y;
}
```

Le programme affiche :

```
l'objet p1 : Point@fbb7cb
l'objet p2 : Point@1df8b99
```

La super-classe Object

• La méthode **equals** : elle permet de comparer les adresses des deux objets.

```
Exemple :
...
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
boolean b = p1.equals(p2);
System.out.prinln(b);  // affiche false, puisque p1 et
                       // p2 ayant des adresses différentes
...
```


La généricité (programmation générique)

• La généricité c'est un mécanisme introduite par le JDK 5.0, qui permet d'écrire un programme unique utilisable avec des objets de type quelconque.

• Elle permet de sécuriser certaine utilisation dans la P.O.O (les collection: les listes, les ensembles,..).

• **Exemple 1: Généricité à un seule paramètre de type.**

1. **Définition de la classe générique :**

```
class Couple <T>
{
    public Couple(T premier,T second)
    {
        x=premier;
        y=second;
    }
    public void afficher()
    {
        System.out.println("Premier valeur "+x+" Second valeur "+y);
    }
    T getpremier()
    {
        return x;
    }
    private T x,y;
}
```

Paramètre de type qui sera remplacé par **Objet** lors la compilation

La généricité (programmation générique)

2. **Utilisation de la classe :**

Équivalent à : `Integer o1 = new Integer(2);`

On doit préciser le type correspondant à T, le type T doit être une classe(Casting implicite de **Object** vers **Integer** lors l'utilisation)

```
public class Generique {

    public static void main(String[] args) {
        Integer o1=2;
        Integer o2=5;

        Couple<Integer> c1 = new Couple<Integer>(o1,o2);
        c1.afficher();
        Couple<Double> c2 = new Couple<Double>(2.0,12.5);
        c2.afficher();

        Double p=c2.getpremier();
        System.out.println("Premier élément du couple : "+p);
    }
}
```

L'exécution de ce programme nous affiche :

```
Premier valeur 2 Second valeur 5
Premier valeur 2.0 Second valeur 12.5
Premier élément du couple : 2.0
```


La généricité (programmation générique)

•Exemple 2: Classe générique à plusieurs paramètres de type

1. **Définition de la classe générique :**

```
class Couple <T,U>
{
    public Couple(T premier,U second)
    {
        x=premier;  y=second;
    }
    public void afficher()
    {
        System.out.println("Premier valeur "+x+" Second valeur "+y);
    }
    public T getpremier()
    {
        return x;
    }
    public U getsecond()
    {
        return y;
    }
    private T x;
    private U y;
}
```

La généricité (programmation générique)

•**Exemple 2: Classe générique à plusieurs paramètres de type**

1. **Utilisation de la classe générique :**

```
public class Generique {

    public static void main(String[] args) {
        Integer o1=2;
        Double o2=5.25;

        Couple <Integer,Double> c1 = new Couple<Integer,Double>(o1,o2);
        c1.afficher();
        Integer o3 = 5;
        Couple <Integer,Double> c2 = new Couple<Integer,Double>(o3,o2);
        c2.afficher();
       System.out.println("Prem: "+c2.getpremier()+" Sec:"+c2.getsecond());
    }
}
```

Le programme affiche :

```
Premier valeur 2 Second valeur 5.25
Premier valeur 5 Second valeur 5.25
Prem: 5 Sec:5.25
```


La généricité (programmation générique)

• **Remarque : Limitation portant sur les classes génériques :**

- **On ne peut pas instancier un objet d'un type paramétré :** ainsi on ne peut pas écrire

```
Class Exemple <T>
{
T x;
...
Void f(...)
{
...
x = new T();
...
}
...
}
```

La généricité (programmation générique)

• **Héritage et généricité :** disposant d'une classe générique donnée par:

```
Class C <T> {…}
```

- Nous pouvons dériver une classe générique D à partir de C en conservant le paramètre de type T ainsi:

```
class D <T> extends C <T> {…}
```

- il en irait de même avec :

```
class D <T,U> extends C<T,U> {…}
```

- La classe dérivée utilise les mêmes paramètres de type que la classe de base, en ajoutant de nouveaux :

```
class D <T,U> extends C<T> {…}
```

- La classe dérivée introduit des limitations sur un ou plusieurs des paramètres de type de la classe de base :

```
class D <T extends Number> extends C<T> {…}
```

- La classe de base n'est pas générique, la classe dérivée l'est (X : est une classe)

```
class D <T> extends X {…}
```

- Ces situations sont incorrectes :

```
class D extends C<T> {…} //incorrecte
class D <T> extends C <T extends Number> {…} // incorrecte
```


La généricité (programmation générique)

• **Préservation du polymorphisme:** La généricité ne remet pas en cause le polymorphisme ainsi :

```
class A<T>
{void f(){...}
}

class B<T> extends A<T>
{void f(){...}
}

class C<T,U> extends A<T>
{void f(){...}
}
...
A<Integer> a;
B<Integer> b;
C<Integer, String> c;
...
a=b; a.f();
a=c; a.f();
```

Les classes Abstraites

• **Les classes abstraites:** une classe abstraite est une classe qui **ne permet pas d'instancier** des objets. Elle ne peut servir que de classe de base pour une dérivation. Elle se déclare ainsi:

```
abstract class A
{
  …
}
```

• Dans une classe abstraite, on peut trouver des méthodes et des champs, dont héritera toute classe dérivée. Mais on peut trouver des méthodes dites abstraites dont on ne fournit que la signature et le type de la valeur de retour, ainsi:

```
abstract class A
{
...
public void f(...){...}
...
public abstract void g(int n);
...
}
```

f est définie dans la classe A

g n'est pas définie dans A, on ne fournit que l'entête

Les classes Abstraites

• On pourra déclarer une variable de type A :

```
A a;
```

• On peut pas instancier l'objet a, ainsi on ne peut pas écrire :

```
a = new A(...);
```

• Si on dérive de A une classe B qui définit la méthode abstraite g :

```
class B extends A
{
...
public void g(int n){...}
...
}
```

Ici on définit la méthode g

• On pourra alors instancier un objet de de type B et même affecter sa référence à une variable de type A :

```
A a = new B(...);
```

Ou bien, `B b = new B();`

Les classes Abstraites

• Une méthode abstraite doit obligatoirement être déclarée ***public***, ce qui est logique puisque sa vocation est d'être redéfinie dans une classe dérivée.

• Dans l'entête d'une méthode déclarée abstraite, les noms d'arguments doivent figurer, ainsi :

```
class A
{
...
public abstract void f(int);
...
}
```

Erreur , nom d'argument obligatoire

Les classes Abstraites

• Une classe dérivée d'une classe abstraite n'est pas obligée de redéfinir toutes les méthodes abstraites dans sa classe de base. Dans ce cas, elle reste simplement abstraite (il est nécessaire de mentionner **`abstract`** dans sa déclaration).

```
abstract class A
{
...
public abstract void f();
public abstract void g(int n);
...
}
abstract class B extends A
{
...
public void f(){...}
...
}
```

`abstract` obligatoire

Définition de f pas de définition de g

• une classe dérivée d'une classe non abstraite peut être déclarée abstraite et peut contenir des méthodes abstraite.

• une classe abstraite peut comporter un ou plusieurs constructeurs, mais ils ne peuvent pas être abstraits.

Les classes Abstraites

- **Exemple complet :**

```
abstract class Affichage{
    abstract public void affiche();
}
class Entier extends Affichage{
    public Entier(int n){
        valeur = n;
    }
    public void affiche(){
        System.out.println("Valeur entière = "+valeur);
    }
    private int valeur;
}
class Flottant extends Affichage{
    public Flottant(float f){
        valeur = f;
    }
    public void affiche(){
        System.out.println("Valeur Flottante = "+valeur);
    }
    private float valeur;
}
```

Un classe abstraite contenant une seule méthode abstraite

Redéfinition de la méthode abstraite affiche()

Redéfinition de la méthode abstraite affiche()

Les classes Abstraites

- **Exemple complet :**

```
public class ClasseAbstraite {
    public static void main(String[] args) {
     Affichage a1 = new Entier(5);
     Affichage a2 = new Flottant(15.5f);
     Affichage a3 = new Entier(2);

     a1.affiche();
     a2.affiche();
     a3.affiche();
    }
}
```

L'exécution du programme nous affiche :

```
Valeur entière = 5
Valeur Flottante = 15.5
Valeur entière = 2
```

Les interfaces

• une interface définit les en-têtes d'un certain nombre de méthodes, ainsi que les constantes. En effet comme les classes abstraites :

- les interfaces pourront se dériver.
- on pourra utiliser des variables de type interfaces.

De plus :

- une classe pourra implémenter plusieurs interfaces.
- la notion d'interface va se superposer à celle de dérivation, et non s'y substituer.

• nous définissons une interface en utilisant le mot-clé **`interface`** :

```
interface I{
...
void f(int n);
void g();
...
}
```

• une classe **A** peut implémenter l'interface **I** en utilisant le mot-clé **`implements :`**

```
class A implements I{
...
public void f(int n){..};
public void g(){...};
...
}
```

La classe **A** doit redéfinir les méthodes **f** et **g** prévues dans l'interface **I**

Les interfaces

• Une même classe peut implémenter plusieurs interfaces :

```
interface I1{
void f(int n);
}
interface I2{
void g();
}
class A implements I1,I2{
public void f(int n){...}
public void g(){...}
}
```

La classe **A** doit **obligatoirement** redéfinir les méthodes **f** et **g** prévues dans les interfaces **I1,I2**

• nous pouvons définir des variables de type interface :

```
interface I{.....}
.....
I i;
```

i est une référence à un objet d'une classe implémentant l'interface **I.**

Les interfaces

- **Exemple complet :**

```
interface Affichage{
    void afficher();
}
class Entier implements Affichage{
    public Entier(int n)
    {
        valeur = n;
    }
    public void afficher(){
        System.out.println("Valeur entière = "+valeur);
    }
    private int valeur;
}
class Flottant implements Affichage{
    public Flottant(float f){
        valeur = f;
    }
    public void afficher(){
        System.out.println("Valeur flottante = "+valeur);
    }
    private float valeur;
}
```


Les interfaces

- **Exemple complet :**

```
public class InterfaceExemple {
  public static void main(String[] args) {
        Affichage a1,a2,a3;

        a1 = new Entier(12);
        a2 = new Flottant(12.4f);
        a3 = new Flottant (11.33f);

        a1.afficher();
        a2.afficher();
        a3.afficher();
    }
}

Le programme nous affiche :

Valeur entière = 12
Valeur flottante = 12.4
Valeur flottante = 11.33
```

Les interfaces

• La clause **implements** est totalement indépendante de l'héritage, ainsi une classe dérivée peut implémenter une interface ou plusieurs :

```
interface I{
    void f(int n);
    void g();
}
class A {.......}
Classe B extends A implements I
{
...
// les méthodes f et g doivent soit être déjà définies dans A,
// soit définies dans B
...
}
```

•Nous pouvons rencontrer de même cette situation :

```
interface I1 {......}
interface I2 {......}
class A implements I1{......}
class B extends A implements I2{.......}
```


Les exceptions

• Exception : condition anormale survenant lors de l'exécution.

• Lorsqu'une exception survient, un objet représentant cette exception est créé.

• Cet objet est jeté (thrown) dans la méthode ayant provoqué l'erreur.

• Cette méthode peut choisir :

  • de gérer l'exception elle-même,

  • de la passer sans la gérer.

• Les exceptions peuvent être générées :

  • par l'environnement d'exécution Java,

  • manuellement par du code.

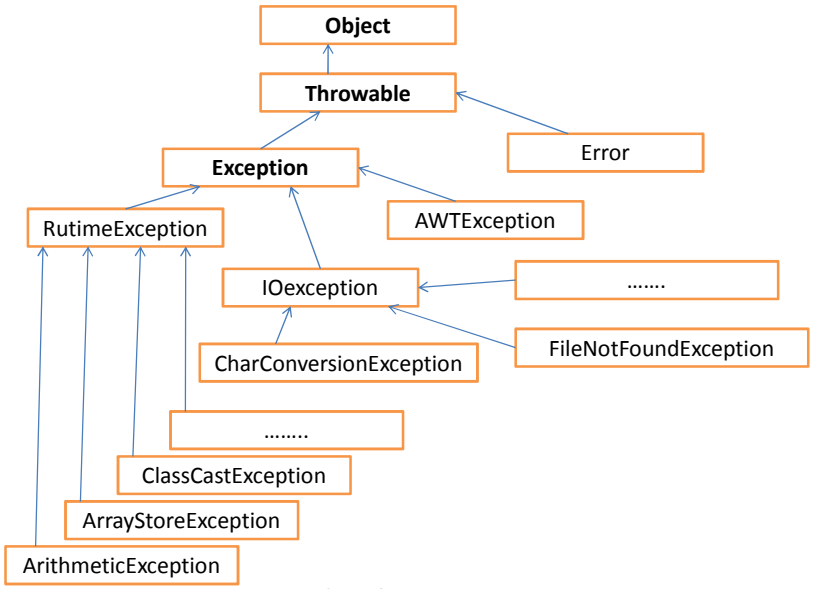
• Les exceptions jetées (ou levées) par l'environnement d'exécution résultent de violations des règles du langage ou des contraintes de cet environnement d'exécution.

• Il y a 5 mots clés d'instructions dédiées à la gestion des exceptions :

try, catch, throw, throws , finally.

Les exceptions

• **Hiérarchie des exceptions :** quelques classes d'exception



Les exceptions

• **Exemple 1 : Division par zéro**

```
public class Excep_Exemple1 {

    public static void main(String[] args) {
        int i=4;
        int j =0;
        int x;
        x = i/j;
        System.out.println("x = "+x);
        System.out.println("Fin programme");
    }
}
```

• Si on exécute le programme, un message d'erreur s'affiche :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
       at Excep_Exemple1.main(Excep_Exemple1.java:8)
Java Result: 1
```

Les exceptions

• **Exemple 2 : traitement des exceptions**

```
public class Excep_Exemple2 {
 public static void main(String[] args) {
  int i=4;
  int j =0;
  int x;
  try
  {
    x = i/j;
    System.out.println("x = "+x);
  }
  catch(Exception err)
  {
    System.out.println("Division par zéro");
    System.out.println("Message System : " + err.getMessage());
  }
  System.out.println("Fin programme");
 }}
```

L'instruction qui crée l'exception

Appel du gestionnaire d'exception **Exception**

• l'exécution du programme, nous affiche :

```
Division par zéro
Message System : / by zero
Fin programme
```


Les exceptions

• **Exemple 3 :** Même comportement que l'exemple 2 si on remplace Exception par ArithmeticException (Polymorphisme).

```
public class Excep_Exemple3 {
 public static void main(String[] args) {
  int i=4;
  int j =0;
  int x;
  try
  {
    x = i/j;
    System.out.println("x = "+x);
  }
  catch(ArithmeticException err)
  {
    System.out.println("Division par zéro");
    System.out.println("Message System : " + err.getMessage());
  }
  System.out.println("Fin programme");
 }}
```

Appel du gestionnaire d'exception **ArithmeticException (Sous classe de Exception)**

• l'exécution du programme, nous affiche aussi:

```
Division par zéro
Message System : / by zero
Fin programme
```

Les exceptions

• **Exemple 4 :** Utilisation de **throw** , **throws**

```
public class Excep_Exemple4 {
 public static void main(String[] args) {
        try
        {   double y = calcul(1,0);
            System.out.println("y=  "+y);
        }
        catch(Exception erreur)
        {
         System.out.println("Message : "+erreur.getMessage());
        }
        System.out.println("Fin programme");
    }
  public static double calcul(double i,double j) throws Exception
    {   int x;
        if (j==0) throw new Exception("Division par 0");
        x=i/j;
        return x;
    }}
```

On doit citer le type d'exception à créer

Création manuelle d'une exception par une méthode

•l'exécution du programme, nous affiche aussi:

```
Message : Division par 0
Fin programme
```


Les exceptions

• **Exemple 5 :** **Création de nos propos Gestionnaires d'Exception**

1. Creation des classes Point, ConsException, DepException :

```
class Point{
public Point(int x,int y) throws ConsException{
    if (x<0 || y<0) throw new ConsException("Erreur de Construction");
        this.x = x;  this.y = y;
    }
public void deplacer(int dx,int dy) throws DepException{
    if ((x+dx)<0 || (x+dy)<0) throw new DepException("Erreur de déplacement");
        this.x = this.x + dx;  this.y = this.y + dy;
    }
    private int x; private int y;
}


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

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

Les exceptions

• **Exemple 5 : Création de nos propos Gestionnaires d'Exception**

2. Utilisation des classes Point , ConsException, DepException :

```
public class Excep_Exemple5 {
 public static void main(String[] args) {

  try{
      Point p1 = new Point(1,2);
      p1.deplacer(1, 1);
      p1.deplacer(-5, -4);
      Point p2 = new Point(1,3);
     }
  catch(ConsException e){System.out.println(e.getMessage());
        }
  catch (DepException e){System.out.println(e.getMessage());
        }
  System.out.println("Fin du programme");
  }
}
```

• **Le programme nous affiche :**

```
Erreur de déplacement
Fin du programme
```


Les exceptions

• **Exemple 6 : Utilisation du mot clé : `finally`**

Utilisation des classes Point , ConsException, DepException de l'exemple 5

```
public class Excep_Exemple6 {
    public static void main(String[] args) {

     try{
            Point p1 = new Point(1,2);
            p1.deplacer(1, 1);
            p1.deplacer(5, 4);
            Point p2 = new Point(1,3);
        }
    catch(ConsException e){ System.out.println(e.getMessage());
        }
    catch (DepException e){ System.out.println(e.getMessage());
        }
    finally {System.out.println("La partie finally");}
    System.out.println("Fin du programme");
    }
}
```

• **Le programme nous affiche :**

```
La partie finally
Fin du programme
```