Chapitre II : Concepts de base de l'orienté objet

- Notions de l'orienté objet
- Java : Généralité
- La notion de classe.
- La notion de constructeur.
- Affectation et comparaison d'objets.
- Le ramasse-miettes (Garbage Collector).
- Champs et méthodes de classe (static).
- Surdéfinition des méthodes.
- Autoréférence : le mot clé this.
- L'héritage.
- Le polymorphisme.

Notions de l'orienté objet

• **La P.O.O** (**P**rogrammation **O**rienté **O**bjet) est fondée sur la programmation structuré, elle contribue à la fiabilité des logiciels et elle facilite la réutilisation de code existant. Elle introduit de nouveaux concepts (Objet, encapsulation, classe, héritage).

• **Les concepts d'objet et d'encapsulation** :

➢Un programme met en œuvre différents ***objets***. Chaque objet associe des ***données (champs)*** et des ***méthodes*** agissant exclusivement sur les données de l'objet.

➢Les données d'un objet ne sont accessible directement qu'à travers les méthodes, qui jouent ainsi le rôle d'interface obligatoire → Encapsulation des données.

Avantage : Modifier la structure des données d'un objet n'a d'incidence que sur l'objet lui-même.

• **Le concept de classe :**

➢ Une classe est la description d'un ensemble d'objets ayant une structure de données commune est disposant des mêmes méthodes. En P.O.O , un objet est une instance de sa classe.

➢ les objets de même classe : structure des données (champs) commune, les valeurs des champs sont propre à chaque objet, les méthode communes.

Notions de l'orienté objet

- **L'héritage :** il permet de définir une nouvelle classe à partir d'une classe existante à laquelle on ajoute de nouvelles données et de nouvelles méthodes.
  - **Avantage :** facilite la réutilisation de produit existant.
  - une classe C peut hériter de B, qui elle-même hérite de A.
- **Polymorphisme :** une classe peut redéfinir certaines des méthodes hérités de sa classe de base.
- un programme orienté objet est formé d'une ou plusieurs classes et il instanciera des objets.
- il y a plusieurs langage orienté objets : Java, C++, C#, PHP, Python, Ada, Smalltalk,… → La syntaxe change mais le concept objet est le même.
- Langage utilisé : Java.
  - est un langage de programmation oriente objet.
  - crée par James Gosling et Patrick Naughton (Sun).
  - présenté officiellement le 23 mai 1995.
  - indépendant de la machine employée pour l'exécution

Java : Généralité

• Premier programme (Exemple introductif):

```
public class PremierProgramme {


    public static void main(String[] args) {
        System.out.println("Mon Premier Programme");
    }
}
```

• Le programme doit être sauvegardé avec le nom : PremierProgramme**.java.**

• Le programme doit être compilé pour produire un code intermédiaire '**Bytecodes**'. → Fichier PremierProgramme**.class** obtenu.

• on pourra lancer l'exécution des Bytecodes par l'intermédiaire de la machine virtuelle Java **JVM.**

Java : Généralité

- Les sorties:

```
public class DeuximeProgramme {
    public static void main(String[] args) {
        int n;
        double x;
        n=5;
        x= 2*n + 1.5;
        System.out.println("n= " + n);
        System.out.println("x= "+x);
        double y;
        y= n*x + 12;
        System.out.println ("valeur de y: " + y);
    }
}
```

- Résultats Affiché par ce programme :

n= 5

x= 11.5

valeur de y: 69.5

Java : Généralité

- Les entrées : lire un entier puis afficher sa valeur :

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class TroisiemeProgramme {
    public static void main(String[] args) {
        int x = 0;
        String ligne_lu =null;
        try
        {
            InputStreamReader lecteur = new InputStreamReader (System.in);
            BufferedReader entree = new BufferedReader(lecteur);
            ligne_lu=entree.readLine();
        }
        catch(IOException err)
        {
            System.exit(0);
        }

        try
        {
            x = Integer.parseInt(ligne_lu);
        }
        catch(NumberFormatException err)
        {
            System.exit(0);
        }

        System.out.println("x = "+ x );
    }
}
```

Java : Généralité

- Les entrées : lire un float puis afficher sa valeur :

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class TroisiemeProgramme {
    public static void main(String[] args) {
        float y = 0;
        String ligne_lu =null;
        try
        {
            InputStreamReader lecteur = new InputStreamReader (System.in);
            BufferedReader entree = new BufferedReader(lecteur);
            ligne_lu=entree.readLine();
        }
        catch(IOException err)
        {
            System.exit(0);
        }

        try
        {
            y = Float.parseFloat(ligne_lu);
        }
        catch(NumberFormatException err)
        {
            System.exit(0);
        }

        System.out.println("y = "+ y );
    }
}
```

Java : Généralité

- Les entrées : lire un double puis afficher sa valeur :

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class TroisiemeProgramme {
    public static void main(String[] args) {
        double z = 0;
        String ligne_lu =null;
        try
        {
            InputStreamReader lecteur = new InputStreamReader (System.in);
            BufferedReader entree = new BufferedReader(lecteur);
            ligne_lu=entree.readLine();
        }
        catch(IOException err)
        {
            System.exit(0);
        }

        try
        {
            z = Double.parseDouble(ligne_lu);
        }
        catch(NumberFormatException err)
        {
            System.exit(0);
        }

        System.out.println("z = "+ z );
    }
}
```

La notion de classe

• Une classe comporte des champs (données) et des méthodes dont l'objectif est de créer des objets.

• **Exemple :** la classe **Point** destiné à manipuler les points d'un plan.

Les champs : deux coordonnées x , y déclarées par :

```
private int x;
private int y;
```

Le mot ***private*** précise l'encapsulation de données (les champs ne seront pas accessible à l'extérieur de la classe).

Les méthodes : trois méthodes

• **`intialise`** : pour attribuer des valeurs aux coordonnées.

• **`deplace`** : pour modifier les coordonnées.

• **`affiche`** : pour afficher un point.

La notion de classe

• La définition de la méthode **`intialise`** :

```
public void intialise (int abs, int ord)
{ x=abs;
y=ord;
}
```

• l'en-tête précise:

- Le nom de la méthode : intialise.
- Le mode d'accès : public (la méthode est accessible depuis un programme quelconque)
- Les paramètres sur lesquels on fournit des valeurs lors d'un appel *abs* et *ord*.
- Le type de valeur de retour *void* (la méthode ne fournit aucun résultat).

• *La signature d'une méthode :* Une méthode est caractérisée par sa ***signature*** :

- son nom.
- la liste des types des paramètres, dans l'ordre.

• Dans une classe deux méthodes différentes ne peuvent pas avoir la même signature.

La notion de classe

• La définition complète de la classe Point

```
public class Point {
    public void intialise(int abs, int ord)
    {
        x=abs;
        y=ord;
    }
    public void deplace(int dx, int dy)
    {
        x+=dx;
        y+=dy;
    }
    public void affiche()
    {
        System.out.println("x = "+x+"y = "+y);
    }
    private int x;
    private int y;
}
```

• Le premier identificateur **public** signifie que le code source de la classe sera sauvegardé dans un fichier (Point.java).

La notion de classe

• la classe Point peut être utilisée par une méthode d'une autre classe pour créer (instancier) des objets:

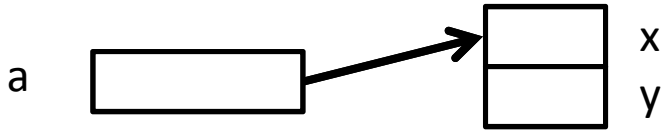
```
public class Testpoint {
    public static void main(String[] args) {
    Point a;
    a = new Point();
    a.intialise(5, 4);
    a.deplace(2, 3);
    a.affiche();
    Point b = new Point();
    b.intialise(2, 3);
    b.affiche();
    }
}
```

• La classe Testpoint doit être sauvegardé dans un fichier Testpoint.java.

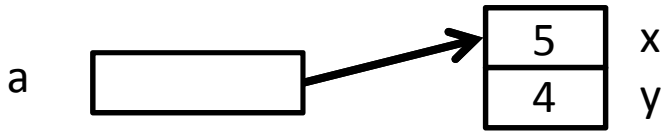
La notion de classe

• La déclaration : **Point a**; réserve un emplacement pour une ***référence*** à un objet de type point.

• L'affectation : **a = new Point();** crée un emplacement pour un objet de type **Point** et affecte sa référence dans **a**.

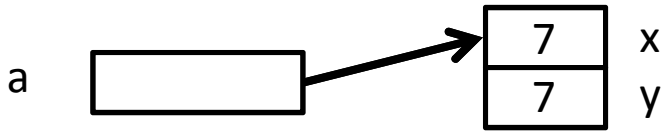


• l'instruction **a.intialise(5,4);** permet d'initialiser les champs de l'objet a.



La notion de classe

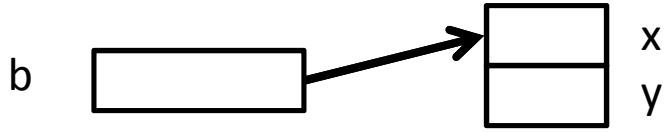
- L'instruction **a.deplace(2, 3);** modifie les champs de l'objet a, ainsi :



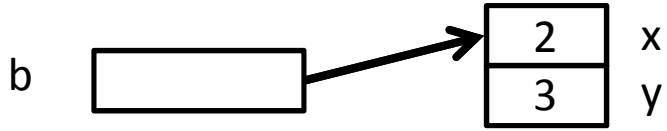
- L'instruction **a.affiche();** affiche le contenu des champs d l'objet a, ainsi :

x=7 y = 7

- L'instruction **Point b = new Point();** réserve un emplacement pour une référence à un objet de type **Point** de de créer un emplacement de cet objet.



- l'instruction **b.intialise(2,3);** permet d'initialiser les champs de l'objet b.



- L'instruction **b.affiche();** affiche le contenu des champs de l'objet b, ainsi :

x=2 y = 3

La notion de constructeur

• Un constructeur est une méthode, sans valeur de retour, pourtant le même nom que la classe. Il peut disposer d'un nombre quelconque d'arguments.

• **exemple :**

```
public class Point {
    public Point(int abs, int ord)
    {
        x=abs;
        y=ord;
    }
    public void deplace(int dx, int dy)
    {
        x+=dx;
        y+=dy;
    }
    public void affiche()
    {
        System.out.println("x = "+x+"y = "+y);
    }
    private int x;
    private int y;
}
```

• Pour créer un objet **a** de type **Point** :

```
Point a = new Point (1,3)
```

La notion de constructeur

• Un constructeur ne fournit aucune valeur de retour (aucun type ne doit figurer devant son nom). Ainsi, on ne peut pas écrire :

```
public int Point(int abs, int ord)
```

• Une classe peut ne disposer pas d'aucun constructeur, mais si elle possède un , on doit l'utiliser pour la création des objets.

• Un constructeur ne peut pas être appelé directement depuis une autre méthode. Ainsi, on ne peut pas écrire :

```
a.Point(3,7);
```

• Une classe peut avoir plusieurs constructeurs (sur-définition des méthodes)

• un constructeur peut être déclaré privé (**`private`**), dans ce cas, la classe doit posséder au moin un autre constructeur **`public`**.

• Dés qu'un objet est crée, et avant l'appel du constructeur, ses champs sont initialisés par défaut «Nulle » (les champs de type : boolean → « false», entier et flottant →0, objet→null, char→null).

• **(init. implicte → init. explicite →appel corps du constructeur)**

La notion de constructeur

- **Exemple :**

```
public class Constr {
        public static void main(String[] args) {
            A a = new A();
            a.aff();
    }
}
class A{
    public A(){
        b=5;
    }
    public void aff(){
        System.out.println("a= "+a+" b= "+b);
    }
    private int a;
    private int b=3;
}
```

- Résultats affichés :

```
a= 0 b= 5
```

Affectation et comparaison d'objets

- **Exemple 1 :**

```
public class Constr {
        public static void main(String[] args) {
            A a1 = new A(1,2);
            A a2 = new A(3,4);
            a1=a2;
            a1.aff();
            a2.aff();
        }
    }
    class A{
        public A(int x1,int x2){
            a=x1;
            b=x2;
        }
        public void aff(){
            System.out.println("a= "+a+" b= "+b);
        }
        private int a;
        private int b;
    }
```

- Ce programme affiche :

```
a= 3 b= 4
a= 3 b= 4
```

Affectation et comparaison d'objets

- **Exemple 2 :**

```
public class Constr {
        public static void main(String[] args) {
            A a1 = new A(1,2);
            A a2 = new A(3,4);
            A c;
            c=a1; a1=a2; a2=c;
            a1.aff();
            a2.aff();
        }
    }
    class A{
        public A(int x1,int x2){
            a=x1; b=x2;
        }
        public void aff(){
            System.out.println("a= "+a+" b= "+b);
        }
        private int a;
        private int b;
    }
```

- Ce programme affiche :

```
a= 3 b= 4
a= 1 b= 2
```

Affectation et comparaison d'objets

- **Exemple 3 :**

```
public class Constr {
        public static void main(String[] args) {
            A a1 = new A(1,2);
            A c = null;
            if (c == null) System.out.println("objet c null");
            a1 = c;
            if (a1 == null) System.out.println("objet a1 null");
    }
}
class A{
    public A(int x1,int x2){
        a=x1;
        b=x2;
    }
    public void aff(){
        System.out.println("a= "+a+" b= "+b);
    }
    private int a;
    private int b;
}
```

- Ce programme affiche :

```
objet c null
objet a1 null
```

Le ramasse-miettes (Garbage Collector)

- Est un mécanisme de gestion automatique de la mémoire, son principe :
  - A tout instant, on connaît le nombre de références à un objet donné (Java gère toujours un objet par référence)
  - Lorsqu'il n'existe plus aucune référence sur un objet, on est certain que le programme ne pourra plus y accéder. Il est donc possible de libérer l'emplacement correspondant qui pourra être utilisé pour une autre chose.

❑ On peut créer un objet sans en conserver la référence, ainsi:

```
(new Point(2,1)).affiche();
```

Un objet est crée dont on affiche les coordonnées, dès la fin de l'instruction, l'objet devient candidat au **Garbage Collector.**

Champs et méthodes de classe (static)

• **Champs statique (déclarés avec le mot `static`):** Champs qui, au lieu d'exister dans chacune des instances de la classe, n'existent qu'en un seul exemplaire pour toutes instances d'une même classe (données globales partagés par toutes les instances).

• **Exemple :**

```
public class ChStatique {
    public static void main(String[] args) {
     NombreObjet n1 = new NombreObjet();
     NombreObjet n2 = new NombreObjet();
     NombreObjet n3 = new NombreObjet();
    }
}
class NombreObjet{
    public NombreObjet(){
        nbre++;
        System.out.println("objet num "+ nbre);
    }
    private static int nbre=0;
}
```

• Le programme affiche :

```
objet num 1
objet num 2
objet num 3
```

Champs et méthodes de classe (static)

• **Méthodes statiques (déclarés avec le mot `static`):** Méthodes qui peuvent être appelées indépendamment de tout objet de la classe (ex. la méthode main).

• **Exemple 1:**

```
public class ChStatique {
    public static void main(String[] args) {
     NombreObjet n1 = new NombreObjet(); n1.affiche();
     NombreObjet n2 = new NombreObjet(); n2.affiche();
     NombreObjet n3 = new NombreObjet(); NombreObjet.affiche();
    }
}
class NombreObjet{
    public NombreObjet(){ nbre++;}
    public static void affiche(){
        System.out.println("objet num "+ nbre);
    }
    private static int nbre=0;
}
```

• Le programme affiche :

```
objet num 1
objet num 2
objet num 3
```

Champs et méthodes de classe (static)

- **Exemple 2:**

```
public class ChStatique {
    public static void main(String[] args) {
     NombreObjet n1 = new NombreObjet();
     n1.affiche();
     NombreObjet n2 = new NombreObjet();
     n2.affiche(); n2.modifier(10);
     n1.affiche();
    }
}
class NombreObjet{
    public NombreObjet(){ nbre++;}
    public static void affiche(){
        System.out.println("objet num "+ nbre);
    }
    public static void modifier(int m){ nbre = m;}
    private static int nbre=0;
}
```

- Le programme affiche :

```
objet num 1
objet num 2
objet num 10
```

Champs et méthodes de classe (static)

- **Remarque 1 : Une méthode de classe (Méthode statique) ne pourra en aucun cas agir sur des champs non-statique, puisque elle n'est liée à aucun objet en particulier.**

- **Remarque 2 : un champs statique peut être utilisé par une méthode non-statique.**

Surdéfinition des méthodes

- Surdéfinition (surcharge) des méthodes est la possibilité que plusieurs méthodes peuvent avoir un même nom, pour peu que le nombre et le type de leurs arguements permettent au compilateur d'effectuer son choix.
- **Exemple :**

```
class Point{
    public Point(int abs,int ord)
    { x=abs;  y=ord ; }
    public void deplacer(int dx,int dy)
    {  x+=dx; y+=dy; }
    public void deplacer(int dx)
    {  x+=dx; }
    public void deplacer(short dx)
    {  x+=dx;    }
    private int x;
    private int y;
}
```

Surdéfinition des méthodes

- **Remarque :**

  ❑ On peut surdéfinir des méthodes de classe (Statiques) , de la même manière qu'on surdéfinit des méthodes usuelles.

  Exemple:
  ```
  static int calculer(int x,float y)
  static int calculer(int x)
  ```

  ❑ Le type de la valeur de retour d'une méthode n'intervient pas dans le choix d'une méthode surdéfinie.

  Exemple : on ne peut pas surdéfinir
  ```
  int calculer(int a,int b)
  float calculer(int a,int b)
  ```

  ❑ Cas d'ambiguité :
  ```
  int calculer(int a, byte b)
  int calculer(byte a,int b)
  ```

  Appel :

  Les deux argument passés à la méthode sont de type **byte**

Surdéfinition des méthodes

- Les constructeurs peuvent être surdéfinis comme n'importe quelle autre méthode.
- **Exemple :**

```
class Point{
    public Point(){
        x = 0;
        y = 0;
    }
    public Point(int a1,int a2){
        x = a1;
        y = a2;
    }
    public void afficher(){
        System.out.println("x= "+x+" y= "+y);
    }
    private int x;
    private int y;
}
```

Autoréférence : le mot clé this

• **this :** peut être utilisé comme une référence à l'objet en cours d'utilisation pour lever une ambiguïté

```
class Additionneur{
    public Additionneur(int val1,int val2)
    {
       this.val1=val1;
       this.val2=val2;
    }
    public Additionneur(int val){
        val1 = val;
        val2 = val;
    }
    public int calculerSomme(){
        return val1+val2;
    }

    private int val1;
    private int val2;
}
```

Autoréférence : le mot clé this

- **this :** peut être utilisé pour appeler un autre constructeur.

```
class Additionneur{
    public Additionneur(int val1,int val2)
    {
       this.val1=val1;
       this.val2=val2;
    }
    public Additionneur(int val){
    this(val,val);
    System.out.println("Constructeur appelé");
    }
    public int calculerSomme(){
        return val1+val2;
    }

    private int val1;
    private int val2;
}
```

La référence *this* doit être la première

Autoréférence : le mot clé this

- **this :** D'autres méthodes peuvent utiliser la référence *this*

```
class Additionneur{
    public Additionneur(int val1,int val2)
    {
       this.val1=val1;
       this.val2=val2;
    }
    public Additionneur(int val){
        this(val,val);
    }
    public int calculerSomme(){
        System.out.println("Calcul somme");
        return this.val1+this.val2;
    }
    private int val1;
    private int val2;
}
```

Autoréférence : le mot clé this

• **this :** lorsqu'une référence à l'instance courante doit être passée en paramètre à une méthode

```
class Additionneur{
    public Additionneur(int val1,int val2)
    {
       this.val1=val1;
       this.val2=val2;
    }
    public void modifier(Additionneur c)
    {
        c.val1=0;
        c.val2=10;
    }
    public void appelmodif(){
        modifier(this);
    }
    private int val1;
    private int val2;
}
```

Autoréférence : le mot clé this

- **Remarque : La référence this est utilisable seulement à l'intérieur des methodes d'objet(Méthode non statique)**

L'héritage

- C'est l'un des **fondements** de la programmtion orienté objet (POO).
- Il est à l'origine de **réutilisation** des composants logiciels que sont les classes.
- Il permet de définir une nouvelle classe, dite **classe dérivée** à partir d'une classe existante dite **classe de base**.
- La classe dérivée **hérite** des fonctionnalités de la classe de base (champs et méthodes) → **la classe dérivée** pourra **modifier** ou **compléter** sans remettre en question la classe de base.
- Il est possible de définir à partir d'une classe de base autant de classes dérivée.
- Une classe dérivée pourra à son tour servir de classe de base pour une nouvelle classe dérivée.

L'héritage

- **Exemple :**
- **La classe de base (super-classe, classe ascendante)**

```
class Point{
    public Point(int x,int y)
    {
        this.x = x;
        this.y = y;
    }
    public void deplacer(int dx,int dy)
    {
        x+=dx;
        y+=dy;
    }
    public void afficher()
    {
       System.out.println("x = "+x+"y= "+y);
    }
    private int x;
    private int y;
}
```

L'héritage

- **Exemple :**
- **La classe dérivée (sous-classe, classe déscendante)**

```
class Pointcol extends Point{
    public Pointcol(int x,int y,byte col)
    {
        super(x,y);
        this.col = col;
    }
    public void affichercol()
    {
        afficher();
        System.out.println("Couleur = "+col);
    }
    byte col;
}
```

Appel du constructeur de la classe de base doit être **la première instruction**

La classe dérivé utilise uniquement les méthodes et champs publiques de la classe de base

L'héritage

- **Exemple :**
- **la classe principale (qui contient la méthode main)**

```
public class Heritage {
    public static void main(String[] args) {
        Pointcol pc1,pc2;
        pc1=new Pointcol(1,2,(byte)12);
        pc1.afficher();
        pc1.affichercol();
        pc2=new Pointcol(3,9,(byte)45);
        pc1.deplacer(1, -2);
        pc1.affichercol();
    }
}
```

- **arès l'exécution nous pouvons voir :**

```
x = 1y= 2
x = 1y= 2
Couleur = 12
x = 2y= 0
Couleur = 12
```

L'héritage

• **Redéfinition des méthodes :** La classe dérivée peut redéfinir des méthodes figurant dans la classe de base avec le même nom , la même signature et le même type de valeur de retour.

• **Exemple : La classe de base Point (Voir page 28):**

```
class Pointcol extends Point{
    public Pointcol(int x,int y,byte col)
    {
        super(x,y);
        this.col = col;
    }
    public void afficher()
    {
        super.afficher();
        System.out.println("Couleur = "+col);
    }
    private byte col;
}
```

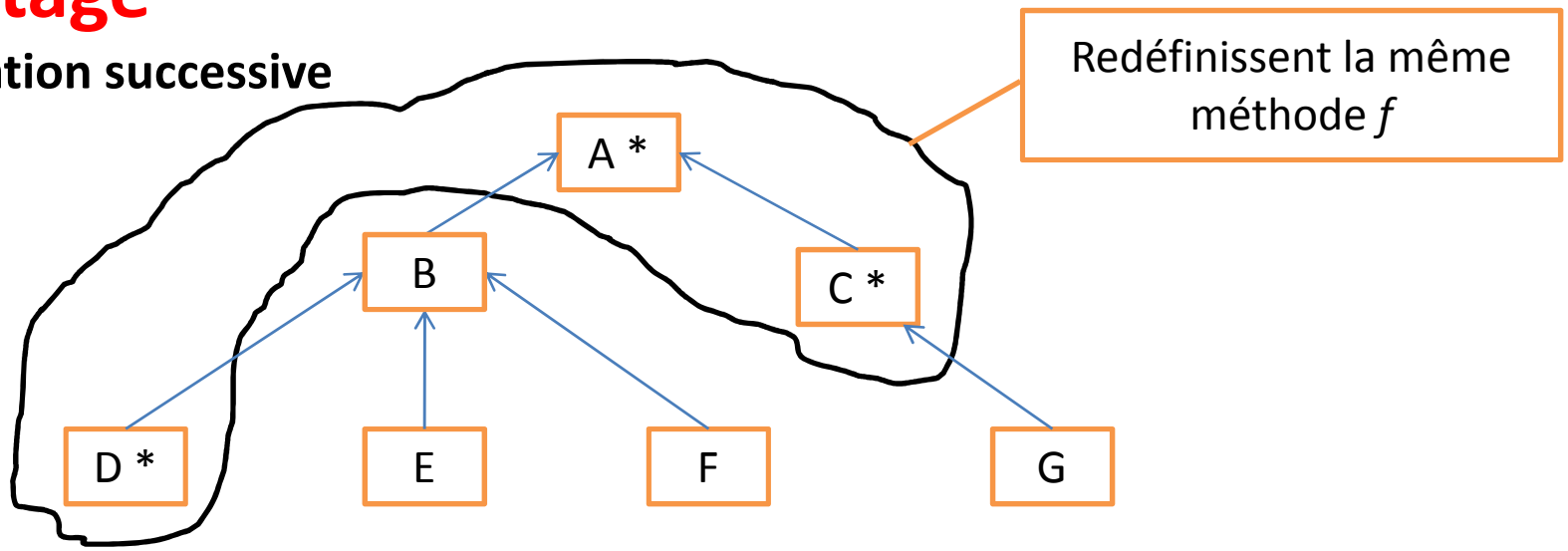
L'héritage

- **Surdéfinition des méthodes :** La classe dérivée peut surdéfinir une méthode figurant dans la classe de base.
- **Exemple :**

```
class A{
public void f(int n){...}
...
}
Class B extends A{
Public void f(float p){...}
...
}
...
A a; B b;
...
int x;float y;
...
a.f(x);
b.f(x);
b.f(y);
a.f(y); → la classe de base ne peut pas utiliser une
méthode de la classe dérivée
```

L'héritage

- **Dérivation successive**



- D est dérivée de B , elle-même dérivée de A→ D est dérivée de A.
- classe A: utilise la méthode *f* de A
- classe B: utilise la méthode *f* de A
- classe C: utilise la méthode *f* de C
- classe D: utilise la méthode *f* de D
- classe E: utilise la méthode *f* de A
- classe F: utilise la méthode *f* de A
- classe G: utilise la méthode *f* de C

Le polymorphisme

•Le polymorphisme est la possibilité qu'un objet peut avoir plusieurs formes.

•Concept puissant en P.O.O qui complète l'héritage.

• Permet de manipuler des objet sans en connaître le type.

• Il permet d'affecter à une variable objet (référence) , la référence à un objet d'un type dérivé.

• Donc, il ya une conversion implicite (légale) d'une référence à un type classe T en une référence à type ascendant de T → compatibilité par affectation entre un type classe et un type ascendant.

Le polymorphisme

- **Exemple : classe de base**

```
class Point{
    public Point(int x,int y)
    {
       this.x = x;
       this.y = y;
    }
    public void deplacer(int dx,int dy)
    {
        x+=dx;
        y+=dy;
    }
    public void afficher()
    {
        System.out.println("x = "+x+"y= "+y);
    }
    private int x;
    private int y;
}
```

Le polymorphisme

- **Exemple : la classe dérivée**

```
class Pointcol extends Point{
    public Pointcol(int x,int y,byte col)
    {
        super(x,y);
        this.col = col;
    }
    public void afficher()
    {
        super.afficher();
        System.out.println("Couleur = "+col);
    }
    private byte col;
}
```

Le polymorphisme

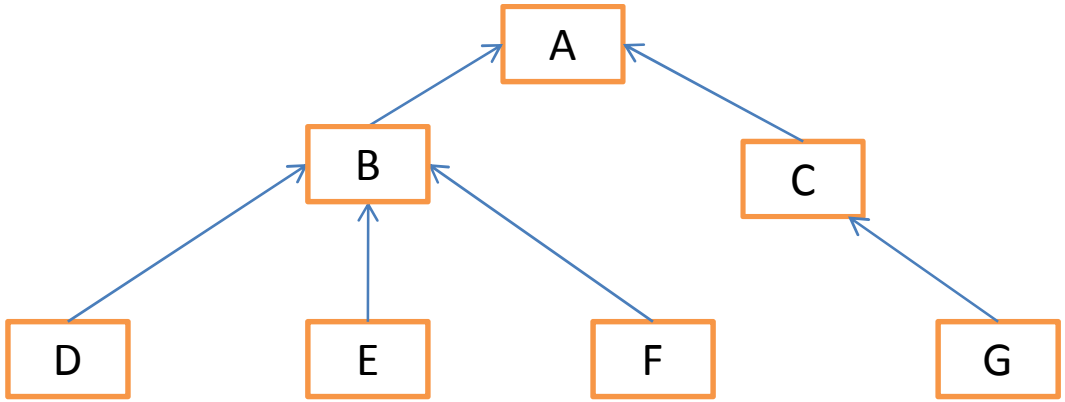
- **Exemple : la classe principale qui contient la méthode *main***

```
public class Polymorphisme{
    public static void main(String args[])
    {
      Point p = new Point (3,5);
      p.afficher();
      Pointcol pc = new Pointcol (4,8,(byte)2);
      p=pc;
      p.afficher();
      p=new Point(5,7);
      p.afficher();
    }}
```

```
L'exécution du programme affiche :
x = 3y= 5
x = 4y= 8
Couleur = 2
x = 5y= 7
```

Le polymorphisme

- Généralisation à plusieurs classes :




```
Avec les déclaration :
A a; B b; C c; D d; E e; F f; G g;

Les affectation suivantes sont légales :
a=b; b=d; b=e; b=f; a=c; c=g; a=g; a=f; a=e; a=d;

Les affectation suivante ne sont pas légales:
b=a; c=a; e=f; d=a;...
```

Le polymorphisme

- **Conversion implicite des arguments :**
  - **Cas 1 :**

```
class A
   { public void identifie()
     {System.out.println("Objet de type A");
     }
   }
Class B extends A
{... //Pas de redéfinition de indentifie
}
Class Util
{ public void f(A a)
  { a.identifie();
  }
}
...
A a = new A(); B b = new B(); U u = new U();
u.f(a); // affiche "Objet de type A"
u.f(b); // affiche "Objet de type A"
```

Le polymorphisme

- **Conversion implicite des arguments :**
  - **Cas 2 :**

```
class A
   { public void identifie()
     {System.out.println("Objet de type A");
     }
   }
Class B extends A
{    public void identifie()
     {System.out.println("Objet de type B");
     }
}
Class Util
{ public void f(A a)
  { a.identifie();
  }
}
...
A a = new A(); B b = new B(); U u = new U();
u.f(a); // affiche "Objet de type A"
u.f(b); // affiche "Objet de type B"
```

Le polymorphisme

- **Conversion implicite des arguments :**
  - **Cas 3 :**

```
class A{...}
Class B extends A{...}
Class Util
{ public void f(int p, B b){...}
  public void f(float x,A a){...}
}
...
A a = new A(); B b = new B(); U u = new U();
int n; float x;

u.f(n,b); // appel de f(int, B) sans conversion.
u.f(x,a); // appel de f(float,A) sans conversion.
u.f(n,a); // appel de f(float, A) avec conversion de n en float.
u.f(x,b); // appel de f(float, A) avec conversion de b en A.
```

Le polymorphisme

- **Conversion implicite des arguments :**
  - **Cas 4 :**

```
class A{...}
Class B extends A{...}
Class Util
{ public void f(int p, A a){...}
  public void f(float x,B b){...}
}
...
A a = new A(); B b = new B(); U u = new U();
int n; float x;

u.f(n,a); // appel de f(int, A) sans conversion.
u.f(x,b); // appel de f(float,B) sans conversion.
u.f(n,b); // erreur, cas d'Ambiguïté entre f(int, A) et f(float,B).
u.f(x,a); // erreur, aucune fonction ne convient.
```

Le polymorphisme

- **Conversion explicite de référence :** nous pouvons forcer le compilateur à réaliser des conversion explicite en utilisant l'opérateur de *cast*

```
class Point{
...
}
Class Pointcol extends Point{
...
}
...
Point p;
Pointcol pc1 = new Pointcol();
Pointcol pc2;
p=pc1;
pc2=p; // erreur , manque de compatibilité
Pc2 = (Pointcol)P // accepté par le compilateur en utilisant
l'opérateur de cat (Conversion explicite)
```