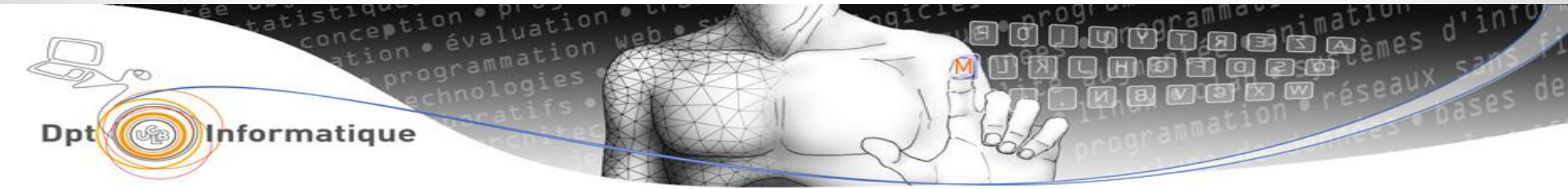
Lyon 1

L'héritage & Polymorphisme

Hamid LADJAL
hamid.ladjal@univ-lyon1.fr
hamid.ladjal@liris.cnrs.fr



L'héritage en Java

Héritage

L'héritage constitue l'un des fondements de la Programmation Orientée Objets

Intérêt

- Raccourcir les temps d'écriture et de mise au point du code d'une application, il est intéressant de pouvoir réutiliser du code déjà écrit

Réutilisation par des classes clientes

- Soit une classe **A** dont on a le code compilé
- Une classe **C** veut réutiliser la classe **A**
- Elle peut créer des instances de **A** et leur demander des services
- On dit que la classe **C** est une classe cliente de la classe **A**

Réutilisation avec modifications

- Souvent, cependant, on souhaite modifier en partie le comportement de **A** avant de le réutiliser

- Le comportement de **A** convient, sauf pour des détails qu'on aimerait changer

- Ou alors, on aimerait ajouter une nouvelle fonctionnalité à **A**

Réutilisation avec modifications du code source

- On peut copier, puis modifier le code source de **A** dans des classes **A1**, **A2**,...

- **Problèmes :**

– on n'a pas toujours le code source de **A**

– les améliorations futures du code de **A** ne seront pas dans les classes **A1**, **A2**,…

Réutilisation par l'héritage

- L'héritage existe dans tous les langages objet à classes
- L'héritage permet d'écrire une classe **B**

– qui se comporte dans les grandes lignes comme la classe **A**

– mais avec quelques différences sans toucher au code source de **A**

- On a seulement besoin du code compilé de **A**

Réutilisation par l'héritage

- Le code source de **B** ne comporte que ce qui a changé par rapport au code de **A**
- On peut par exemple
  - ajouter de nouvelles méthodes
  - modifier certaines méthodes

Vocabulaire

- La classe **A** s'appelle une classe mère, classe parente ou **super-classe**

- La classe **B** qui hérite de la classe **A** s'appelle une classe fille ou sous-classe

Exemple d'héritage en Java - classe mère

```
class Point
{
public void initialise (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 (« Le point à les coordonnées" + x + "  " +
y) ;
}
private int x, y ;
}
```

Exemple d'héritage en Java - classe fille

```
class Pointcol extends Point   // classePointcol derivée de Point
{
  public void colore (byte couleur)
{ this.couleur = couleur ;
}
private byte couleur ;
}
```

La mention ***extends*** **Point** précise au compilateur que la classe ***Pointcol*** est une classe dérivée de ***Point***

Exemple d'héritage en Java - classe fille

Nous pouvons déclarer des variables de type ***Pointcol*** et créer des objets de ce type de manière usuelle, par exemple :

Pointcol pc ; // pc contiendra une référence a un objet de type Pointcol

Pointcol pc2 = new Pointcol() ; // pc2 contient la référence a un objet de
// type **Pointcol** crée en utilisant le constructeur par défaut

pc = new Pointcol() ;

Un **objet** de type *Pointcol* peut alors faire appel :

- aux méthodes publiques de *Pointcol*, ici ***colore*** ;
- mais aussi aux méthodes publiques de ***Point*** : ***initialise***, ***deplace*** et ***affiche***.

Exemple d'héritage en Java - classe fille

```
class Pointcol extends Point   // classePointcol derivée de Point
{
  public void colore (byte couleur)
{ this.couleur = couleur ;
}
private byte couleur ;
}
```

La mention ***extends*** **Point** précise au compilateur que la classe ***Pointcol*** est une classe dérivée de ***Point***

Exemple d'héritage en Java

```
// classe de base
class Point
{ public void initialise (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 (« Coordonees sont " + x
+ " " + y) ;
}
private int x, y ;
}
// classe derivee de Point
class Pointcol extends Point
{
public void colore (byte couleur)
{
this.couleur = couleur ;}
private byte couleur ;
}
```

```
// classe utilisant Pointcol
public class TstPcol1
{
public static void main (String args[])
{
Pointcol pc1 = new Pointcol() ;
pc1.affiche() ;

pc1.initialise (1, 2) ;
pc1.colore ((byte)5) ;

pc1.affiche() ;
pc1.deplace (2, 3) ;

pc1.affiche() ;
Point p2 = new Point() ;
p2.initialise (15, 19) ;
p2.affiche() ;
}
}
```

Solution:
0 0
1 2
3 5
15 19

Accès d'une classe dérivée aux membres de sa classe de base

Une classe dérivée n'accède pas aux membres privés

Une méthode d'une classe dérivée n'a pas accès aux membres privés de sa classe de base.

On considère la classe *Pointcol* :

```
==========================================
void affichec() // méthode affichant les coordonnées et la couleur
{
System.out.println ("Je suis en " + x + " " + y) ;
System.out.println (" et ma couleur est : " + couleur) ;}
==========================================
```

La méthode affichec de Pointcol n'a pas accès aux champs privés x et y de sa classe de base.

Accès d'une classe dérivée aux membres de sa classe de base

Une méthode d'une classe dérivée a accès aux membres publics de sa classe de base.

Ainsi, pour écrire la méthode ***affichec***, nous pouvons nous appuyer sur la méthode *affiche* de *Point* en procédant ainsi :

```
==================================
public void affichec ()
{ affiche() ;
System.out.println (" et ma couleur est : " + couleur) ;
}
\\Une méthode d'affichage d'un objet de type Pointcol
==================================
```

On notera que l'appel *affiche()* dans la méthode *affichec* est en fait équivalent à :
this.affiche() ;
Autrement dit, il applique la méthode *affiche* à l'objet (de type *Pointcol*) ayant appelé la méthode *affichec*.

Accès d'une classe dérivée aux membres de sa classe de base

```
class Point
{ public void initialise (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 ("Je suis en " + x + " " + y) ;
}
private int x, y ;
}
```

```
class Pointcol extends Point
{ public void colore (byte couleur)
{ this.couleur = couleur ;
}
public void affichec ()
{ affiche() ;
System.out.println (" et ma couleur
est : " + couleur) ; }
public void initialisec (int x, int y, byte couleur)
{ initialise (x, y) ;
this.couleur = couleur ;
}
private byte couleur ;}
```

```
public class TstPcol2
{
public static void main (String args[])
{
Pointcol pc1 = new Pointcol() ;
pc1.initialise (3, 5) ;
pc1.colore ((byte)3) ;
pc1.affiche() ; // attention, ici affiche
pc1.affichec() ; // et ici affichec

Pointcol pc2 = new Pointcol() ;
pc2.initialisec(5, 8, (byte)2) ;
pc2.affichec() ;
pc2.deplace (1, -3) ;
pc2.affichec() ;
}}
```

Accès d'une classe dérivée aux membres de sa classe de base

```
public class TstPcol2
{ public static void main (String args[])
{
Pointcol pc1 = new Pointcol() ;
pc1.initialise (3, 5) ;
pc1.colore ((byte)3) ;
pc1.affiche() ; // attention, ici affiche
pc1.affichec() ; // et ici affichec

Pointcol pc2 = new Pointcol() ;
pc2.initialisec(5, 8, (byte)2) ;
pc2.affichec() ;
pc2.deplace (1, -3) ;
pc2.affichec() ;
}
}
```

```
Je suis en 3 5
Je suis en 3 5
et ma couleur est : 3
Je suis en 5 8
et ma couleur est : 2
Je suis en 6 5
et ma couleur est : 2
```

Construction et initialisation des objets dérivés

Appels des constructeurs :

Dans les exemples précédents, nous avons volontairement choisi des classes sans constructeurs

```
class Point
{ .....
public Point (int x, int y)
{ .....
}
private int x, y ;
}
```

```
class Pointcol extends Point
{ .....
public Pointcol (int x, int y, byte
couleur)
{ .....
}
private byte couleur ;
}
```

Tout d'abord, il faut savoir que :

En Java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.

Construction et initialisation des objets dérivés

Ainsi, le constructeur de *Pointcol* pourrait :

- initialiser le champ *couleur* (accessible, car membre de *Pointcol*),
- appeler le constructeur de *Point* pour initialiser les champs *x* et *y*.

Il faut respecter une **règle imposée par Java :**

Si un constructeur d'une **classe dérivée** appelle un constructeur d'une **classe de base**, il doit obligatoirement s'agir **de la première instruction du constructeur** et ce dernier est désigné par le mot clé

super.

Construction et initialisation des objets dérivés

Dans notre cas, voici ce que pourrait être cette instruction :

super (x, y) ; // appel d'un constructeur de la classe de base, auquel
// on fournit en arguments les valeurs de x et de y

D'où notre constructeur de *Pointcol* :

```
public Pointcol (int x, int y, byte couleur)
{ super (x, y) ;                    // obligatoirement comme première instruction
this.couleur = couleur ;
}
```

Construction et initialisation des objets dérivés

Voici un petit programme complet

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

```
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
{ super (x, y) ; // obligatoirement comme
premiere instruction
this.couleur = couleur ;
}
public void affichec ()
{ affiche() ;
System.out.println (" et ma couleur est : " +
couleur) ;}
private byte couleur ;
}
```

```
public class TstPcol3
{ public static void main (String args[])
{
Pointcol pc1 = new Pointcol(3, 5, (byte)3) ;
pc1.affiche() ;

pc1.affichec()
Pointcol pc2 = new Pointcol(5, 8, (byte)2) ;

pc2.affichec() ;
pc2.deplace (1, -3) ;
pc2.affichec() ;

}}
```

```
Je suis en 3 5
Je suis en 3 5
et ma couleur est : 3
Je suis en 5 8
et ma couleur est : 2
Je suis en 6 5
et ma couleur est : 2
```

Construction et initialisation des objets dérivés

```
public class TstPcol3
{ public static void main (String args[])
{
Pointcol pc1 = new Pointcol(3, 5, (byte)3) ;
pc1.affiche() ;    // attention, ici affiche

pc1.affichec()   // et ici affichec
Pointcol pc2 = new Pointcol(5, 8, (byte)2) ;

pc2.affichec() ;
pc2.deplace (1, -3) ;
pc2.affichec() ;

}}
```

```
Je suis en 3 5
Je suis en 3 5
et ma couleur est : 3
Je suis en 5 8
et ma couleur est : 2
Je suis en 6 5
et ma couleur est : 2
```

Appel du constructeur d'une classe de base dans un constructeur d'une classe dérivée

1- La classe de base ne possède aucun constructeur

Il reste possible d'appeler le constructeur par défaut dans la classe dérivée, comme dans :

```
class A
{ ..... // aucun constructeur
}
class B extends A
{ public B (...) // constructeur de B
{ super() ; // appelle ici le pseudo-constructeur par défaut de A
.....
}
}
```

L'appel ***super()*** est ici superflu, mais il ne nuit pas. En fait, cette possibilité s'avère pratique lorsque l'on définit une classe dérivée sans connaître les détails de la classe de base.

La classe dérivée ne possède aucun constructeur

la classe de base devra :

- soit posséder un constructeur public sans argument, lequel sera alors appelé,
- soit ne posséder aucun constructeur ; il y aura appel du pseudo-constructeur par défaut.

Exemple 1

```
class A
{ public A() { ..... } // constructeur 1 de A
public A (int n) { ..... } // constructeur 2 de A
}
class B extends A
{ ..... // pas de constructeur
}
B b = new B() ; // construction de B --> appel de constructeur 1 de A
```

La construction d'un objet de type *B* entraîne l'appel du constructeur sans argument de *A*.

La classe dérivée ne possède aucun constructeur

Exemple 2

```
class A
{ public A(int n) { ..... } // constructeur 2 seulement
}
class B extends A
{ ..... // pas de constructeur
}
```

Erreur de compilation car le constructeur par défaut de *B* cherche à appeler un constructeur sans argument de *A*. Comme cette dernière dispose d'au moins un constructeur, il n'est plus question d'utiliser le constructeur par défaut de *A*.

La classe dérivée ne possède aucun constructeur

Exemple 3

```
class A
{ ..... // pas de constructeur
}
class B extends A
{ ..... // pas de constructeur
}
```

Cet exemple ressemble au précédent, avec cette différence que *A* ne possède plus de constructeur.
Aucun problème ne se pose plus. La création d'un objet de type *B* entraîne l'appel du constructeur par défaut de *B* , qui appelle le constructeur par défaut de *A*.

Initialisation d'un objet dérivé

la création d'un objet fait intervenir plusieurs phases :

- allocation mémoire,
- initialisation par défaut des champs,
- initialisation explicite des champs,
- exécution des instructions du constructeur.

La généralisation à un objet d'une classe dérivée est assez intuitive.
Supposons que :
class B extends A { ..... }

La création d'un objet de type *B* se déroule en 6 étapes.

Initialisation d'un objet dérivé

1. Allocation mémoire pour un objet de type *B*
2. Initialisation par défaut de tous les champs de *B* aux valeurs "nulles" habituelles.

3. Initialisation explicite, s'il y a lieu, des champs hérités de *A* ; éventuellement, exécution des blocs d'initialisation de *A*.

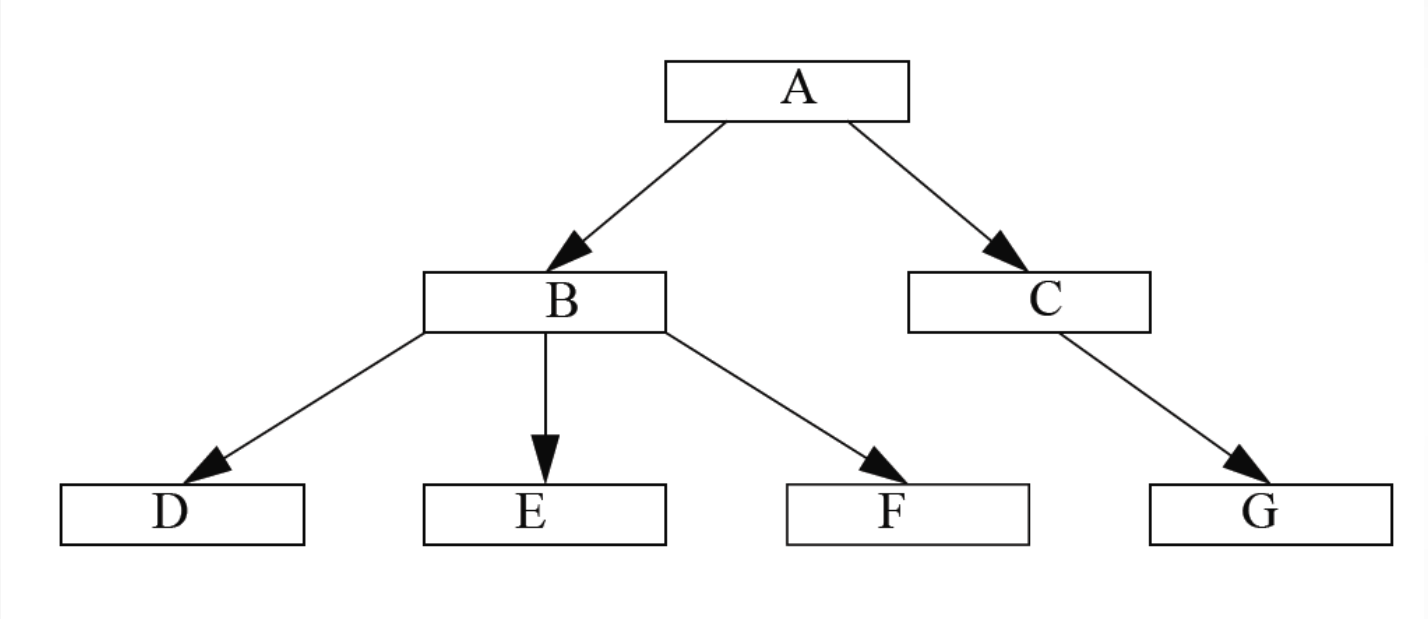
4. Exécution du corps du constructeur de *A*.

5. Initialisation explicite, s'il y a lieu, des champs propres à *B* ; éventuellement, exécution des blocs d'initialisation de *B*.

6. Exécution du corps du constructeur de *B*.

Dérivations successives

• une même classe peuvent être dérivées plusieurs classes différentes,
• les notions de classe de base et de classe dérivée sont relatives puisqu'une classe dérivée peut, à son tour, servir de classe de base pour une autre.



D est dérivée de *B*, elle-même dérivée de *A*. On dit souvent que *D* dérive de *A*. On dit aussi que *D* est une **sous-classe** de *A* ou que *A* est une **super-classe** de *D*. Parfois, on dit que *D* est une descendante de *A* ou encore que *A* est une ascendante de *D*.

Naturellement, *D* est aussi une descendante de *B*. Lorsqu'on a besoin d'être plus précis, on dit alors que *D* est une descendante directe de *B*.

Redéfinition et surdéfinition de membres

- Pas d'héritage multiple, pour des raisons de performances et de complexité (en maintenance). À la place, notion d'interface.

- Il existe une classe au sommet de la hiérarchie, Object. Sans mot-clé extends, le compilateur met automatiquement extends Object.

- De la même manière que l'on peut assigner à une variable int un byte, on peut déclarer une variable de type Object et y stocker une référence à une instance de toute sous classe d'Object.

Exemple 2

Mot-clé extends

```
class Point3D extends Point {
int z;
Point3D(int x, int y, int z) {
this.x = x;
this.y = y;
this.z = z;
}
Point3D() {
Point3D(-1, -1, -1);
}}
```

Exemple 2

super réfère aux variables d'instance et aux constructeurs de la super classe.

```
class Point3D extends Point {
int z;
Point3D(int x, int y, int z) {
super(x, y);                    // Appel de Point(x,y).
this.z = z;
}
}
```

Cet appel au constructeur de la classe mère doit être la 1ière ligne du constructeur.

Redéfinition et surdéfinition de membres

La notion de redéfinition de méthode

Nous avons vu qu'un objet d'une classe dérivée peut accéder à toutes les méthodes publiques de sa classe de base:

```
class Point
{ .....
public void affiche()
{ System.out.println ("Je suis en " + x + " " + y) ;
}
private int x, y ;
}
```

```
class Pointcol extends Point
{ ..... // ici, on suppose qu'aucune méthode ne se nomme affiche
private byte couleur ;
}
Point p ; Pointcol pc ;
```

Redéfinition et surdéfinition de membres

La notion de redéfinition de méthode

L'appel *p.affiche()* fournit tout naturellement les coordonnées de l'objet *p* de type *Point*.

L'appel *pc.affiche()* fournit également les coordonnées de l'objet *pc* de type *Pointcol*, mais bien entendu, il n'a aucune raison d'en fournir la couleur.

Pour cela nous avons introduit dans la classe *Pointcol* une méthode *affichec* affichant à la fois les coordonnées et la couleur d'un objet de type *Pointcol*.

```
public void affichec ()
{ affiche() ;
System.out.println (" et ma couleur est : " + couleur) ;
}
```

il paraît logique de chercher à leur attribuer le même nom.

Possibilité existe en Java  ***redéfinition de méthode***

Redéfinition et surdéfinition de membres

La notion de redéfinition de méthode

```
====================
class Pointol extends Point
{ public void affiche()
{ affiche() ;
System.out.println (" et ma couleur est " + couleur) ;
}.....
}
====================
```

l'appel *affiche()* provoquerait un appel récursif de la méthode *affiche* de *Pointcol*

Il faut donc préciser qu'on souhaite appeler non pas la méthode *affiche* de la classe *Pointcol*, mais la méthode *affiche* de sa classe de base. Il suffit pour cela d'utiliser le mot **clé *super***

```
public void affiche()
{ super.affiche() ; // appel de la methode affiche de la super-classe
System.out.println (" et ma couleur est " + couleur) ;
}
```

La notion de redéfinition de méthode

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

```
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
{ super (x, y) ; // obligatoirement comme
premiere instruction
this.couleur = couleur ;
}
public void affiche ()
{ super.affiche() ;
System.out.println (" ma couleur est : " +
couleur) ;  }
private byte couleur ; }
```

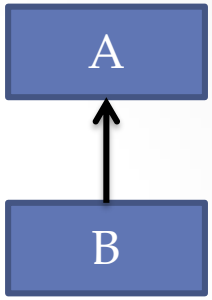
Exemple de redéfinition de la méthode affiche *dans une classe dérivée* Pointcol

```
public class TstPcoleur
{ public static void main (String args[])
{ Pointcol pc = new Pointcol(10, 22,
(byte)3) ;
pc.affiche() ;    // affiche de Pointcol
pc.deplace (2, -1) ;
pc.affiche() ;
}
}
```

```
Position en 10 22
ma couleur est : 3
Position en 12 21
ma couleur est : 3
```

Surdéfinition et héritage

```
class A
{ public void f (int n) { ..... }
.....
}
class B extends A
{ public void f (float x) { ..... }
.....
}
A a ; B b ;
int n ; float x ;
.....
a.f(n) ;     // appelle f (int) de A
a.f(x) ;     // erreur de compilation : une seule méthode acceptable
                    (f(int) de A    et on ne peut pas convertir x de float en int)
b.f(n) ;     // appelle f (int) de A
b.f(x) ;     // appelle f(float) de B
```



la recherche d'une méthode ne se fait qu'en remontant la hiérarchie d'héritage, jamais en la descendant... C'est pourquoi l'appel *a.f(x)* ne peut être satisfait, malgré la présence dans *B* d'une fonction *f* qui conviendrait

Utilisation simultanée de surdéfinition et de redéfinition

```
class A
{ .....
public void f (int n) { ..... }
public void f (float x) { ..... }
}
class B extends A
{ .....
public void f (int n) { ..... } // redefinition de f(int) de A
public void f (double y) { ..... } // surdefinition de f (de A et de B)
}
A a ; B b ;
int n ; float x ; double y ;
.....
a.f(n) ;  // appel de f(int) de A (mise en jeu de surdefinition dans A)
a.f(x) ;  // appel de f(float) de A (mise en jeu de surdefinition dans A)
a.f(y) ;  // erreur de compilation
b.f(n) ;  // appel de f(int) de B (mise en jeu de redefinition)
b.f(x) ;  // appel de f(float) de A (mise en jeu de surdefinition dans A et B)
b.f(y) ;  // appel de f(double) de B (mise en jeu de surdefinition dans A et B)
```

Exemple complet : Héritage Ville-→ Capitale

```
class Ville
{
private String nom; //le nom ne sera accessible que par la classe Ville, et pas par la classe Capitale
protected int nbHab; //le nombre d'habitant sera accessible par la classe Capitale

public Ville(String leNom){
nom = leNom.toUpperCase( ); //ainsi tous le noms de ville seront en majuscule
nbHab = -1;                 //-1 signifie que le nombre d'habitant est inconnu
}

public Ville (String leNom, int leNbHab){
nom = leNom.toUpperCase( );    \\ Retourne le texte de chaine en majuscules
if (leNbHab< 0)
{
System.out.println("Un nombre d'habitant doit être positif.");
nbHab = -1;}
else
nbHab = leNbHab;
}
```

Suite Exemple

```
public String getNom()
{return nom;}
//pas d'accesseur en écriture pour le nom → il est impossible de changer le nom d'une ville
public int getNbHab( )
{return nbHab;}

public void setNbHab(int nvNbHab){
if (nvnbHab < 0)
System.out.println("Un nombre d'habitant doit être positif. La modification n'a pas été prise en compte");
else
nbHab = nvNbHab;}

public String presenteToi(){
String presente = "Ville "+ nom +" nombre d'habitants ";
if (nbHab = = -1)
presente = presente + "inconnu";
else
presente = presente + " = " + nbHab;
return presente;}}
```

Suite Exemple

```
class Capitale extends Ville
{
private String pays;
public Capitale(String leNom, String lePays) //constructeurs
{
super(leNom); //appel du constructeur de Ville. nbHab est initialisé à –1 par
                //ce constructeur
pays = lePays;}

public Capitale(String leNom, String lePays, int leNbHab)
{super(leNom, leNbHab);
pays = lePays;
}
public String getPays( )        //accesseurs supplémentaires
{return pays;}
public void setPays(String nomPays)
{pays = nomPays;}

public String presenteToi( ) //méthode presenteToi( ) redéfinie
{String presente = super.presenteToi( );     //super.nom_de_la_methode()
presente = presente + " Capitale de "+ pays;
return presente;}}
```

Suite Exemple

```
//Classe de test
public class testVilles
{
public static void main(String args[])
{
Ville v1 = new Ville("Paris", 1300000);
Ville v2 = new Ville("Lyon");

Capitale c1 = new Capitale("Paris", "France", 150000000);
Capitale c2 = new Capitale("Hanoï.", "Vietnam");
Capitale c3= new Capitale("Hanoï.", " Vietnam " , 8 000 000);

System.out.println(v1.presenteToi( ));
System.out.println(v2.presenteToi( ));
System.out.println(c1.presenteToi( ));
System.out.println(c2.presenteToi( ));
System.out.println(c3.presenteToi( ));
}
}
```

Suite Exemple

Résultats:

Ville Paris nombre d'habitants = 1500000

Ville Lyon nombre d'habitants inconnu

Ville Paris nombre d'habitants 10000000 Capitale de France

Ville Hanoi nombre d'habitants inconnu Capitale de Vietnam
Ville Hanoi nombre d'habitants 8 000000 Capitale de Vietnam

Polymorphisme en JAVA

Polymorphisme

- On peut caractériser le polymorphisme en disant qu'il permet de manipuler des objets sans en connaître (tout à fait) le type.

- Le polymorphisme peut être vu comme la capacité de choisir dynamiquement la méthode qui correspond au type réel de l'objet.

Exemple:
On pourra construire un tableau d'objets les uns étant de type *Point*, les autres étant
de type *Pointcol* (dérivé de *Point*) et appeler la méthode *affiche* pour chacun des objets du tableau.

Chaque objet réagira en fonction de son propre type.

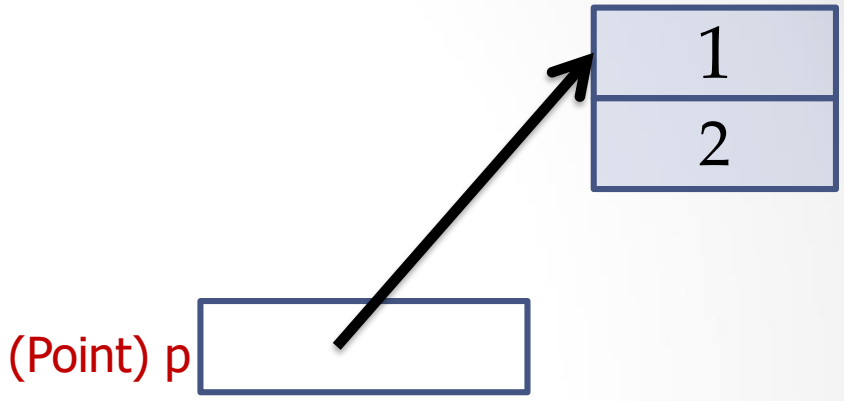
Les bases du polymorphisme

Exemple:
les classes *Point* et *Pointcol* sont censées disposer chacune d'une méthode *affiche*, ainsi que des constructeurs habituels.

```
class Point
{ public Point (int x, int y) { ..... }
public void affiche () { ..... }
}
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
public void affiche () { ..... }
}
Point p1 ;
p1 = new Point (1, 2) ;
p1= new Pointcol (11, 22, (byte)2) ;
```

Les bases du polymorphisme

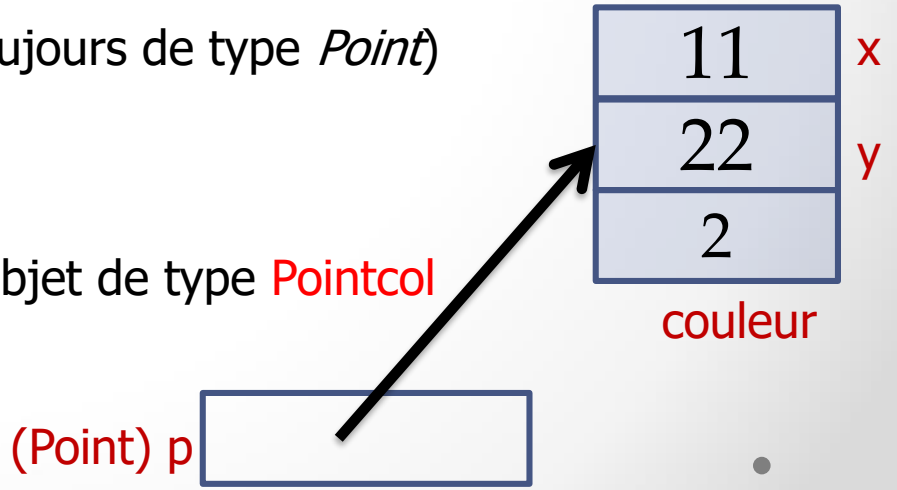
```
class Point
{ public Point (int x, int y) { ..... }
public void affiche () { ..... }
}
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
public void affiche () { ..... }
}
Avec ces instructions :
Point p1 ;
p1 = new Point (1, 2) ;
```



Java autorise ce genre d'affectation (*p* étant toujours de type *Point*)

p1 = new Pointcol (11, 22, (byte)2) ;

// p1 de type Point contient la référence a un objet de type Pointcol



Les bases du polymorphisme

Java permet d'affecter à une variable objet non seulement la référence à un objet du type correspondant, mais aussi une référence à un objet d'un type dérivé

Considérons maintenant ces instructions :

```
Point p = new Point (3, 5) ;
p.affiche () ; // appelle la méthode affiche de la classe Point

p = new Poincol (4, 8, 2) ;
p.affiche () ; // appelle la méthode affiche de la classe Pointcol
```

Dans la dernière instruction, la variable ***p*** est de type ***Point***, alors que l'objet référencé par *p* est de type ***Pointcol***. L'instruction ***p.affiche()*** appelle alors la méthode *affiche* de la classe ***Pointcol***

Les bases du polymorphisme

le polymorphisme en Java se traduit par :

- la compatibilité par affectation entre un type classe et un type ascendant.

- la ligature dynamique des méthodes.

Le polymorphisme permet d'obtenir un comportement adapté à chaque type d'objet, sans avoir besoin de tester sa nature de quelque façon que ce soit

Polymorphisme

Exemple1 de polymorphisme

```
class Point
{private int x, y ;

public Point (int x, int y)
{ this.x = x ; this.y = y ;
}
public void deplace (int dx, int dy)
{ x += dx ; y += dy ;
}
public void affiche ()
{
System.out.println (« les cord" + x +
" " + y) ;
}
}
```

Classe mère

```
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
{ super (x, y) ; // obligatoirement
comme première instruction
this.couleur = couleur ;
}
public void affiche ()
{ super.affiche() ;
System.out.println (" et la couleur est : " +
couleur) ;
}
private byte couleur ;
}
```

Classe dérivée

Polymorphisme

```
public class Poly
{ public static void main (String args[])
{
Point p = new Point (3, 5) ;
p.affiche() ; // appelle affiche de Point
Pointcol pc = new Pointcol (4, 8, (byte)2) ;
p = pc ; // p de type Point, reference un objet de type Pointcol
p.affiche() ; // on appelle affiche de Pointcol
p = new Point (5, 7) ; // p reference a nouveau un objet de type Point
p.affiche() ; // on appelle affiche de Point
}
}
```

```
les cord 3 5
les cord 4 8
et la couleur est : 2
les cord 5 7
```

Polymorphisme

```
class Point
{ public Point (int x, int y)
{ this.x = x ; this.y = y ;
}
public void affiche ()
{ System.out.println ("Je suis en " + x + " " + y) ;
}
private int x, y ;}
```

```
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
{ super (x, y) ; // obligatoirement comme
                 // premiere instruction
this.couleur = couleur ;
}
public void affiche ()
{ super.affiche() ;
System.out.println (" et ma couleur est : " +
couleur) ;
}
private byte couleur ;}
```

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

Point [] tabPts = new Point [4] ;
tabPts [0] = new Point (0, 2) ;
tabPts [1] = new Pointcol (1, 5, (byte)3) ;
tabPts [2] = new Pointcol (2, 8, (byte)9) ;
tabPts [3] = new Point (1, 2) ;
for (int i=0 ; i< tabPts.length ; i++)
 tabPts[i].affiche() ;}}
```

Polymorphisme

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

Point [] tabPts = new Point [4] ;
tabPts [0] = new Point (0, 2) ;
tabPts [1] = new Pointcol (1, 5, (byte)3) ;
tabPts [2] = new Pointcol (2, 8, (byte)9) ;
tabPts [3] = new Point (1, 2) ;
for (int i=0 ; i< tabPts.length ; i++)
 tabPts[i].affiche() ;
}
}
```

```
Je suis en 0 2
Je suis en 1 5
et ma couleur est : 3
Je suis en 2 8
et ma couleur est : 9
Je suis en 1 2
```

Exemple

Autre situation où l'on exploite le polymorphisme

```
class Point
{ public Point (int x, int y)
{ this.x = x ; this.y = y ;}
public void affiche ()
{ identifie() ;
System.out.println (" Mes coordonnees sont : " + x + " " + y) ;}
public void identifie ()
{ System.out.println ("Je suis un point ") ;}
private int x, y ;}
```

```
Je suis un point
Mes coordonnees sont : 0 2

Je suis un point colore de couleur 3
Mes coordonnees sont : 1 5

Je suis un point colore de couleur 9
Mes coordonnees sont : 2 8

Je suis un point
Mes coordonnees sont : 1 2
```

```
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
{ super (x, y) ;
this.couleur = couleur ;
}
public void identifie ()
{ System.out.println ("Je suis un point
colore de couleur " + couleur) ;
}
private byte couleur ;
}
```

```
public class TabHeterogene2
{ public static void main (String args[])
{
Point [] tabPts = new Point [4] ;
tabPts [0] = new Point (0, 2) ;
tabPts [1] = new Pointcol (1, 5, (byte)3) ;
tabPts [2] = new Pointcol (2, 8, (byte)9) ;
tabPts [3] = new Point (1, 2) ;
for (int i=0 ; i< tabPts.length ; i++)
tabPts[i].affiche() ;
}}
```

Autre situation où l'on exploite le polymorphisme

```
public class TabHeterogene2
{ public static void main (String args[])
{ Point [] tabPts = new Point [4] ;
tabPts [0] = new Point (0, 2) ;
tabPts [1] = new Pointcol (1, 5, (byte)3) ;
tabPts [2] = new Pointcol (2, 8, (byte)9) ;
tabPts [3] = new Point (1, 2) ;
for (int i=0 ; i< tabPts.length ; i++)
tabPts[i].affiche() ;
}}
```

```
Je suis un point
Mes coordonnees sont : 0 2
Je suis un point colore de couleur 3
Mes coordonnees sont : 1 5
Je suis un point colore de couleur 9
Mes coordonnees sont : 2 8
Je suis un point
Mes coordonnees sont : 1 2
```