

# Héritage

Cours de l'héritage sous Java

Version 0.75

Djaafri

# Plan

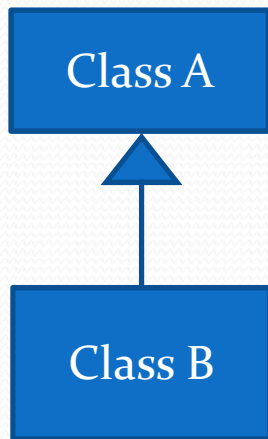
## Héritage

- Définitions
- L'arbre de toutes les classes
- Redéfinition des méthodes
  - Surcharge et masquage
  - Redéfinition des méthodes
- Héritage et constructeurs
- Classe Object
- Le polymorphisme
  - Casting entre classes
- Méthodes et classes finales
- Méthodes abstraites
- Classes abstraites
- Interfaces

# Définitions (1/3)

- L'héritage est un mécanisme offert par les langages OO permettant de créer une nouvelle classe - dite classe fille ou sous-classe - à partir d'une (ou plusieurs) autre(s) classe(s) - dite classe mère ou super classe - en partageant ses attributs et ses méthodes.
- L'héritage exprime la relation « est un » entre une sous classe et une super classe.
- Java permet seulement d'hériter une seule classe => héritage simple

# Définitions (2/ 3)



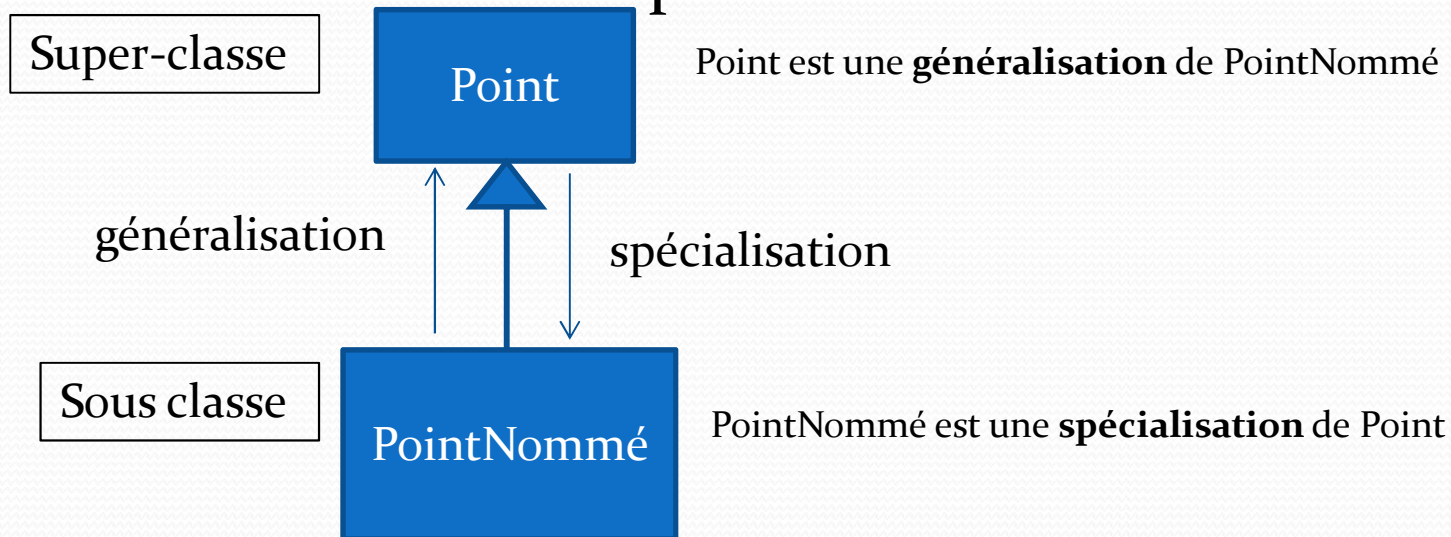
B hérite de A=>

En plus des membres définis dans la classe B, les objets de B possèdent les attributs et méthodes définis dans A

- A est la classe mère et B la classe fille.
- la classe B hérite de la classe A
- la classe B est une sous-classe de la classe A
- la classe A est la super-classe de la classe B

# Définitions (3/ 3)

- L'héritage exprime la relation de généralisation entre une super classe et une sous classe
- L'héritage exprime la relation de spécialisation entre une sous classe et une super classe



# Intérêts de l'héritage

- **Spécialisation, enrichissement** : une nouvelle classe réutilise les attributs et les opérations d'une classe en y ajoutant et/ou des opérations particulières à la nouvelle classe
- **Redéfinition** : une nouvelle classe redéfinit les attributs et opérations d'une classe de manière à en changer le sens et/ou le comportement pour le cas particulier défini par la nouvelle classe
- **Réutilisation** : évite de réécrire du code existant et parfois on ne possède pas les sources de la classe à hériter

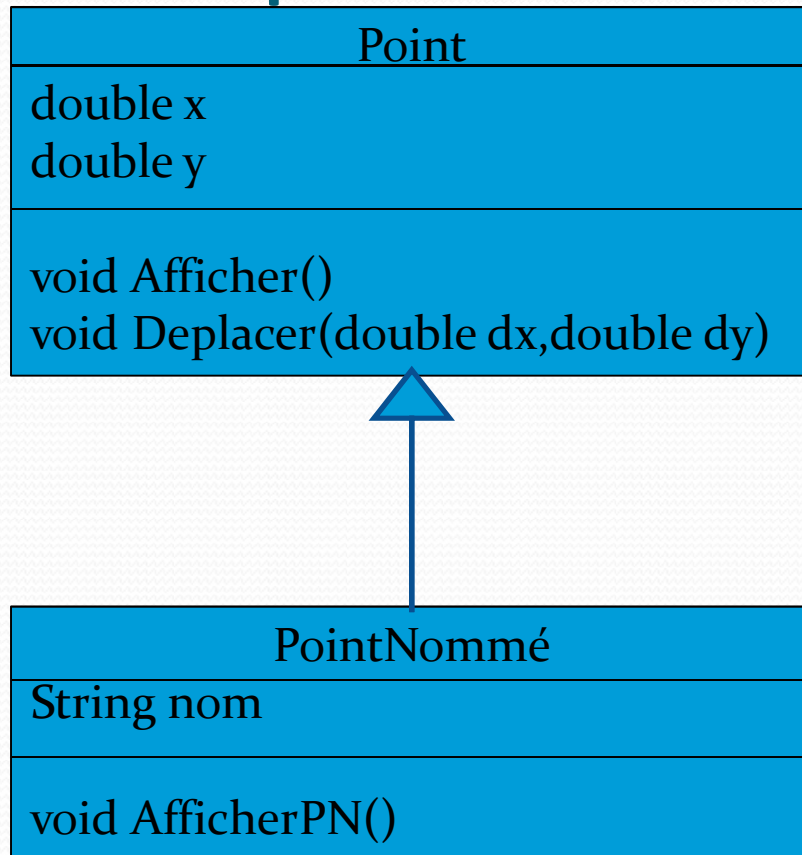
# Héritage et Java

- Java ne permet que l'héritage simple
- Utilisation du mot clé **extends** pour réaliser l'héritage entre deux classes.
- Si la classe B hérite de la classe A alors, on écrit le code suivant :

```
class A {  
// déclaration des membres de A  
}
```

```
classe B extends A{  
// déclaration des membres de B  
}
```

# Exemple: Classe PointNommé (1/3)



- On veut créer une classe représentant un point nommé à partir de la classe **Point** => utilisation de l'héritage
- Un objet **PointNommé** est un objet **Point** à lequel on rajoute un attribut **nom** et une méthode **AfficherPN()**

# Exemple: Classe PointNommé (2/3)

```
class Point {  
    double x, y;  
    void deplacer( double x, double y){  
        this.x = this.x + x;  
        this.y = this.y + y;  
    }  
    void afficher() {  
        System.out.println("(" + x + ", " + y + ")");  
    }  
}
```

```
class PointNommé extends Point {  
    String nom;  
    void afficherPN() {  
        System.out.println("Point:"  
        + nom + "(" + x + ", " + y + ")");  
    }  
}  
  
class Test {  
    public static void main(){  
        PointNommé pn= new PointNommé (...);  
        pn.deplacer(2,4);  
        pn.affiche();  
        pn.affichePN();  
    }  
}
```

# Redéfinition des membres(1/2)

## concepts

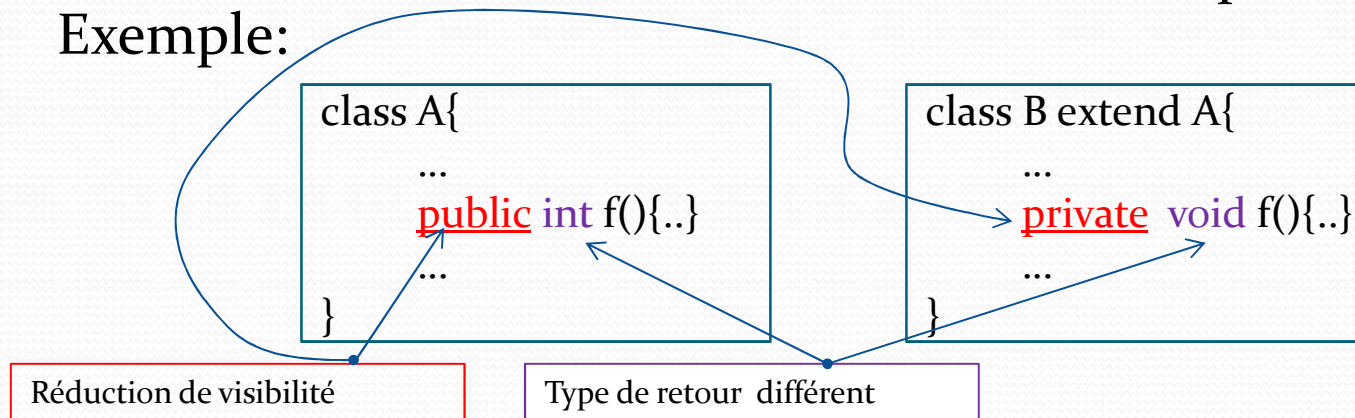
- La classe héritière peut redéfinir des membres avec le même nom que la classe mère. Deux cas se présentent :
  - Signature différente et nom identique entre membres classe fille et classe mère:  
=>Il s'agit de cas de surcharge; la classe fille traite les membres redéfinis comme deux membre de la même classe, on parle de **surcharge** ou **overloading**
  - Signature identique entre méthodes de la classe fille et la classe mère, on parle de **surdéfinition** ou **overriding**:  
=>Il y a masquage : dans la classe fille , les membres redéfinis masque nt les membres de la classe mère

# Redéfinition des membres(2/2)

## les contraintes

- Java exige les conditions suivantes lors de la surdéfinition des méthodes dans une sous classe:
- Le type de retour doit être le même ou une spécialisation du type de retour de la super-classe.
- La méthode surdéfinie dans la sous classe ne doit pas réduire la visibilité de de la méthode définie dans la super-classe.

Exemple:



La surdéfinition de la méthode f() présente **deux erreurs de compilation**: réduction de la visibilité, changement du type de retour

# Redéfinition des membres: exemple

```
class A{
    int q,t;
    void f(){..}
    void p(int a){..} // visibilité par défaut
    int g(){..}
}
class B extends A{
    int q;//surdéfinition de q dans la sous classe
    int g(int a){..} //surcharge de la méthode g()
    private void p(int a){..} /*redéfinition non acceptée- réduction de
visibilité*/
    void f(){..} /*redéfinition de la méthode f(), donc masquage de
f() de la super classe*/
}
```

# Redéfinition des membres: exemple

- comment accéder aux membres masqués à l'intérieur d'une sous classe? => Utilisation du mot clé **super**
- **super** permet d'accéder aux membres de la super classe, de la même manière que l'on accède aux attributs de la classe elle-même à l'aide du mot-clé **this**.
- Dans ces conditions, à l'intérieur d'une méthode de la classe B

```
System.out.print(q); //attribut q de l'objet en cours déclaré dans B
```

```
System.out.print(super.q); //attribut q de l'objet déclaré dans A
```

```
System.out.print(t); //attribut t déclaré dans A
```

```
f(); //méthode f() de l'objet b déclaré dans B
```

```
super.f(); //méthode f() de l'objet b déclaré dans A
```

```
System.out.print(g()); //méthode g() de l'objet b déclaré dans A
```

```
System.out.print(g(3)); //méthode g(int a) de l'objet b déclaré dans B
```

```
}
```

```
}
```

# Retour sur la classe PointNommé

```
class Point {  
    double x, y;  
    void deplacer( double x, double y){  
        this.x = this.x + x;  
        this.y = this.y + y;  
    }  
    void afficher() {  
        System.out.println("(" + x + ", " + y + ")");  
    }  
}
```

```
class PointNommé extends Point {  
    String nom;  
    void afficher() {  
        System.out.println("Point:" + nom);  
        super.afficher();  
    }  
}  
  
class Test {  
    public static void main(){  
        PointNommé pn= new PointNommé (...);  
        pn.deplacer(2,4);  
        pn.afficher();  
    }  
}
```

- Redéfinir la méthode **afficher()** de Point
- Éviter d'utiliser les propriété de la super class (Point) à l'intérieur de PointNommé
- Réutilisation du code de la méthode **afficher()**=> utilisation de **super.afficher()**

# Héritage et constructeurs(1/3)

- L'initialisation d'un objet d'une sous-classe implique son initialisation en tant qu'objet de la super-classe.
- Tout constructeur de la sous-classe commence nécessairement par l'appel d'un constructeur de la super-classe.
- l'appel du constructeur de la super classe se fait par le biais du mot clé **super(...)**
- **super(...)** correspond à un constructeur de la super-classe avec liste d'arguments compatible.

```
class A{  
    ...  
    A(..){  
    ..}  
    ...  
}
```

```
class B extend A{  
    ...  
    B(..){  
    super(..)  
    ..}  
    ...  
}
```

```
class C extend B {  
    ...  
    C(){  
    super(..)  
    ..}  
    ...  
}
```

# Héritage et constructeurs(2/3)

- Si aucun appel à un constructeur n'est définie dans la sous classe, alors un appel implicite (sans écriture de code) au constructeur **super()** est réalisé par la sous classe
- si aucun constructeur n'est déclaré dans la super-classe alors **super()** correspond au constructeur par défaut

```
class A{
    ...
    //aucuns
    constructeurs
    ...
}
```

```
class B extend A{
    ...
    B(..){
        super();// appel
        facultatif
    }
    ..}
    ...
}
```

```
class C extend B {
    ...
    C(){
        super(..)
    }
    ..}
    ...
}
```

# Héritage et constructeurs(3/3)

- Exemple :

```
class Point {
    double x, y;
    Point(double x, double y){
        this.x=x; this.y=y;
    }
    void deplacer( double x, double y){
        this.x = this.x + x;
        this.y = this.y + y;
    }
    void afficher() {
        System.out.println("(" +x+" "+y+"");
    }
}
```

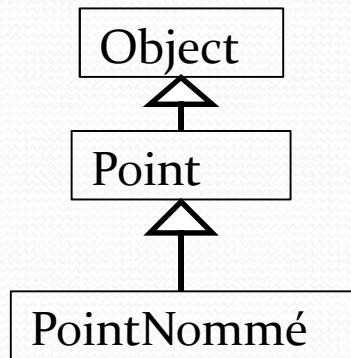
```
class PointNommé extends Point {
    String nom;
    PointNommé (double x, double y, String nom){
        super(x,y);/*Obligatoirement la première
instruction*/
        this.nom=new String(nom);
    }
    void afficher() {
        System.out.println("Point:" +nom);
        super.afficher();
    }
}
class Test {
    public static void main(){
        PointNommé pn= new PointNommé (3,4, "A" );
        pn.deplacer(2,4);
        pn.afficher();
    }
}
```

# La classe « Object »

- La classe **Object** de la bibliothèque standard de Java est la classe de plus haut niveau dans la hiérarchie d'héritage
- Toute classe hérite directement ou indirectement de la classe **Object**
- Une classe qui ne définit pas de clause **extends** hérite de la classe **Object**
- Toutes les méthodes contenues dans **Object** sont accessibles à partir de n'importe quelle classe car par héritage successif toutes les classes héritent d'Object.

Objects	
Class	getClass()
String	toString()
boolean	equals(Object o)
int	hashCode()
...	

# La classe « Object »: exemple de redéfinition de méthodes



Avant  
redéfinition de  
méthodes

```
public static void main(String [] args){  
    PointNommé p1=new PointNommé(3,5,"x");  
    System.out.println(p1); /* toString() méthode de  
                             la classe Object, affichage :  
                             PointNommé@152b6651 */  
}
```

`println(p1.toString())`

redéfinition de  
méthodes

```
class Point {  
    ...  
    public String toString(){  
        return ("+x+", "+y+");  
    }  
    ...  
}
```

```
Class PointNommé extends Point {  
    ...  
    public String toString(){  
        return "point:"+nom+Super.toString();  
    }  
    ...  
}
```

Après redéfinition de  
méthodes

```
public static void main(String [] args){  
    Point p1=new Point(2,-4);  
    System.out.println(p1); //affiche: (2,-4)  
    PointNommé p2=new PointNommé(3,5,"x");  
    System.out.println(p2); // affichage :  
    point : x(3,5)  
}
```

`println(p2.toString())`

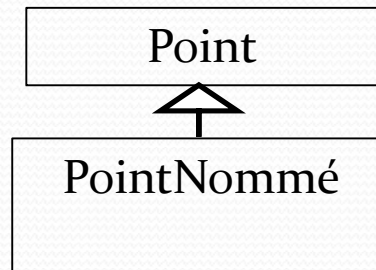
# Méthodes et classes finales

- Utilisation du mot-clé **final**
  - Méthode : interdire une éventuelle redéfinition d'une méthode  
=> `public final void afficherNP(){...}`
  - Classe : interdire toute spécialisation ou héritage de la classe concernée  
=> `public final class PointNommé extends Point {...}`
  - A titre d'exemple la classe **String** est **final**

# Polymorphisme

## surclassement

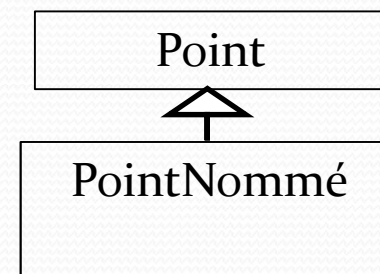
- Une classe B qui hérite de la classe A peut être vue comme un sous-type (sous ensemble) du type défini par la classe A
  - Le type PointNommé est un sous ensemble du type Point
  - L'ensemble des PointNommé est inclus dans l'ensemble des Point.



# Polymorphisme

## surclassement

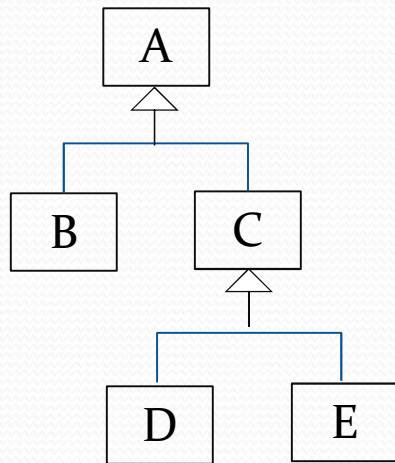
- Une instance de B peut donc être vue comme instance de la super classe de B qui est la classe A
- Cette relation est directement supportée par le langage JAVA :
  - à une référence déclarée de type A il est possible d'affecter une valeur qui est une référence vers un objet de type B (surclassement ou upcasting)
  - `Point p=new PointNommé(...);`



# Polymorphisme

## surclassement

- En plus général, à une référence d'un type donné, on peut affecter n'importe quel objet dont le type effectif est un sous-type directe ou indirecte du type de la déclaration.
- On parle de référence ou objet surclassé



```
A a;  
a=new C();  
a=new B();  
a=new E();  
a=new D();
```

```
C c;  
c=new D();//c surclassé  
c=new E();  
c=new A();  
c=new B();
```

# Polymorphisme

## surclassement

- Questions: comment se comporte java avec le surclassement?

# Polymorphisme

## surclassement

- À la compilation :
  - Lorsqu'un objet est « surclassé », il est vu par le compilateur comme un objet du type de la référence utilisée pour le désigner
  - Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence

```
class Test {  
    public static void main(String args[]){  
        //déclaration d'un objet p de type Point  
        Point p=new PointNommé(2,3,4, "x");  
        //utilisation de méthodes de la classe Point  
        p.deplacer(2,2);  
        p.afficher();  
        //utilisation d'un méthode de la classe PointNommé  
        p.changerNom("x1"); //erreur, la méthode  
        changerNom(..) n'existe pas dans la classe Point  
    }  
}
```

Il faut vérifier le type de la référence : dans l'exemple c'est **Point**, seules les méthodes la classe **Point** seront disponibles

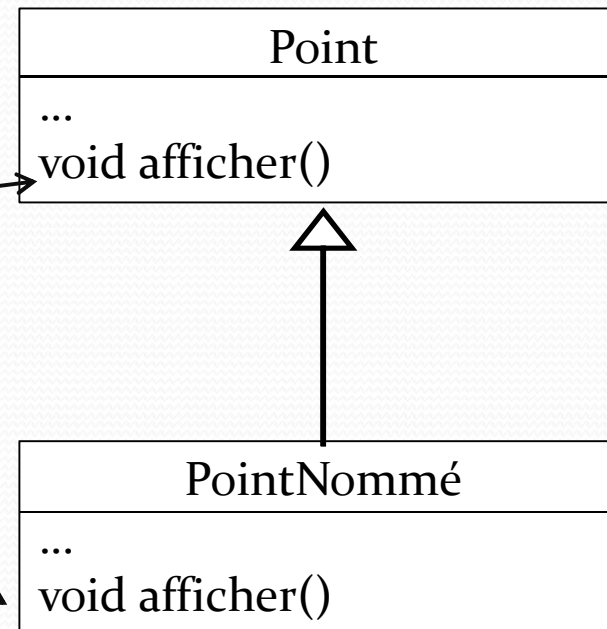
# Polymorphisme

## Lien dynamique

- Quelle méthode **afficher()** va être exécutée? Celle de la classe **Point** ou celle de la classe **PointNommé**?

```
...  
Point p=new PointNommé(2.3,4, "x");  
...  
p.afficher();  
...
```

L'objet *p* initialise les attributs de la classe *PointNommé*



C'est la méthode **afficher()** de **PointNommé** qui est appelée. Puis elle appelle (par `super...`) la méthode de la super-classe

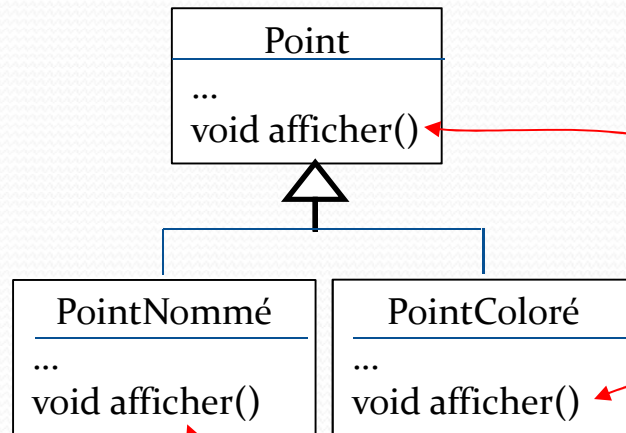
# Polymorphisme

## Lien dynamique

- A l'exécution:
  - S'il y a redéfinition d'une méthode, et que cette dernière est accédée par un objet « surclassé », c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est invoquée et exécutée.
  - Le lien avec la méthode effective est réalisé lors de l'exécution, on parle de lien dynamique ou liaison tardive.

# Polymorphisme

## surclacement- intérêt



```
//-----
Point[] tabPoint=new Point[3];
tabPoint[0]=new Point();
tabPoint[1]=new PointNommé();
tabPoint[2]=new PointColoré();

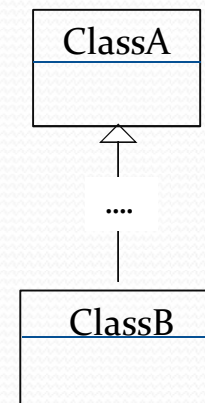
for(int i=0; i<tabPoint.length; i++){
    tabPoint[i].afficher();
}
//tabPoint[0].afficher();
//tabPoint[1].afficher();
//tabPoint[2].afficher();
```

Sélection dynamique (lors de l'exécution) de la méthode à appliquer, selon le type réel de l'objet

- En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, le polymorphisme permet une programmation beaucoup plus générique. Le développeur n'a pas à savoir, lorsqu'il programme une méthode, le type précis de l'objet sur lequel la méthode va s'appliquer. Il lui suffit de savoir que cet objet implémentera la méthode.[wikipedia]

# Polymorphisme sousclassement

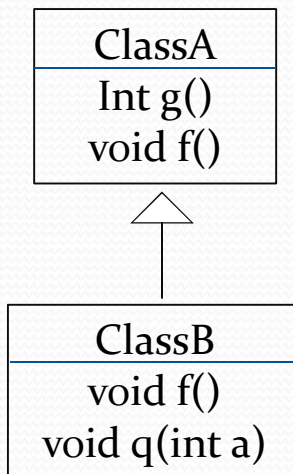
- Le sous-classement force un objet à « libérer » les fonctionnalités (méthodes) cachées par le surclassement
- Comment ?=> Conversion de type explicite (cast), même syntaxe du cast en types primitifs.
- Dans quelle condition la conversion explicite d'objets fonctionne ?
  - Si ClasseA obj1 = ... ; ClassB obj2=(ClassB)obj1  
Alors ClassA est une super classe de la classeB  
Et ClassB est le type effectif de obj1



# Polymorphisme

## Sousclassement- Opérateur instanceof

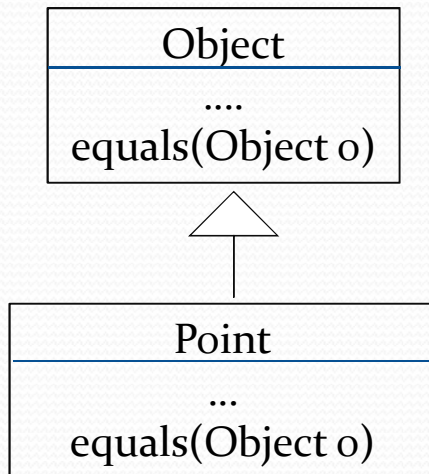
- S'assurer du type effectif d'un objet donné en utilisant l'opérateur `instanceof`
  - `obj instanceof ClassB` ← Retourne **true** ou **false**



```
ClassA b,a;
a=new ClassA();
b=new ClassB();// surclassement de l'objet «b»
//-----
int x=b.g();
b.f();
b.q(5);//méthode cachée par le surclassement
ClasseB bb=(ClassB)b;//sous classement de l'objet« bb »
//-----
aa=(ClassB)a;//
bb.q(5);//méthode libéré par le sousclassement
```

# Polymorphisme

## redéfinition de la méthode equals



```
public class Point extends Object {
    public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point p= (Point)o;
        return (this.x == p.x && this.y== p.y);
    }
    ...
}
```

```
public class Test {
    public static void main(String [] args) {
        Point p1,p2,p3;
        p1=new Point(3,4); p2=new Point(3,4); p3=new Point(3,-10);
        String s="Bonjour";
        System.out.println("p1 égale à p2 :"+p1.equals(p2));
        System.out.println("p1 égale à p2 :"+p1.equals(p3));
        System.out.println("p1 égale à p2 :"+p1.equals(s));
    }
}
```

```
Console X
<terminated> Point (1) [Java App
p1 égale à p2 :true
p1 égale à p2 :false
p1 égale à p2 :false
```

The screenshot shows a console window with the following output:

# ce qu'il reste à voir...

- Les classes abstraites sous java
  - Les méthodes abstraites
- Les interfaces java