

LU2IN002 - Introduction à la programmation orientée-objet

Responsable de l'UE : Christophe Marsala
(email: Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(email: Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 6 – lundi 17 octobre 2022

PLAN DU COURS

- 1 Héritage : syntaxe
 - Rappels des principes de la POO
 - Les mots clefs extends et super
 - Diagramme de classes UML et héritage
 - Niveau d'accès : protected
- 2 Héritage : subsomption et polymorphisme
- 3 Héritage : surcharge / redéfinitions

PROGRAMME DU JOUR

- 1 Héritage : syntaxe
 - Rappels des principes de la POO
 - Les mots clefs extends et super
 - Diagramme de classes UML et héritage
 - Niveau d'accès : protected
- 2 Héritage : subsomption et polymorphisme
 - Principe de subsomption
 - Polymorphisme : application aux tableaux
 - La classe Object
- 3 Héritage : surcharge / redéfinitions
 - Surcharge
 - Redéfinition
 - Redéfinition et usage de super

PRINCIPES ORIENTÉS OBJETS

Principe 1 : Encapsulation

- Rapprochement données (attributs) et traitements (méthodes)
- Protection de l'information (private/public)

Principe 2 : Composition/Agrégation

- Un objet de la classe A **est composé** d'objets de la classe B
- Classe A **AVOIR** des Classe B

Principe 3 : Héritage

- Un objet de la classe B **est un** objet de la classe A aussi
- Classe B **ETRE** une Classe A

⇒ La classe B **hérite** de la classe A

EXEMPLE : POINT ET POINTNOMME

Problème

On a déjà écrit une classe Point qui est bien adaptée à notre programme, mais on veut maintenant que certains points aient en plus un nom. Comment faire pour :

- éviter de modifier la classe Point ?
- ne pas recopier le code de la classe Point dans une autre classe ?

```
1 public class Point {
2     private double x, y;
3     public Point(double x, double y) {
4         this.x = x; this.y = y;
5     }
6     public void move(double x, double y) {
7         this.x = x; this.y = y;
8     }
9     public String toString() {
10        return "("+x+", "+y+")";
11    }
12 }
```

EXEMPLE : POINT ET POINTNOMME

Bouger un point

```
1 Point p=new Point(1,2);
2 System.out.println(p); // (1.0,2.0)
3 p.move(3,4);
4 System.out.println(p); // (3.0,4.0)
```

Comment bouger un point nommé ?

```
5 PointNomme pnA=new PointNomme(5,6,"A");
6 System.out.println(pnA); // PointNomme A (5.0,6.0)
```

- ★ Peut-on appeler une méthode de la classe Point à partir de la variable pnA qui est de type PointNomme ?

- Oui, car grâce à l'héritage PointNomme est un Point

```
7 pnA.move(7,8); // méthode de Point
8 System.out.println(pnA); // PointNomme A (7.0,8.0)
```

HÉRITAGE : SYNTAXE : MOT CLEF extends

- ★ Comment faire hériter la classe PointNomme de la classe Point ?

```
1 public class PointNomme extends Point {
2     private String name;
3
4     public PointNomme(double x, double y, String name) {
5         super(x, y);
6         this.name = name;
7     }
8     public String toString() {
9         return "PointNomme_" + name + "_" + super.toString();
10    }
11 }
```

- Deux nouveaux mots-clefs :
 - ◆ extends dans la signature de la classe
 - ◆ super (explications plus tard)

Erreur courante

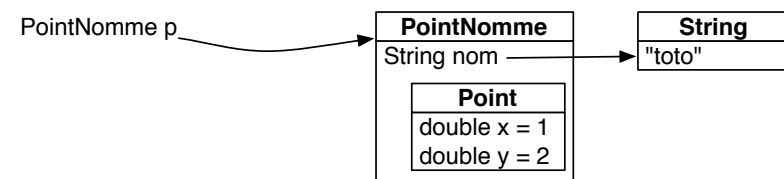
Attention à ne pas réécrire les attributs et les méthodes de la classe Point dans la classe PointNomme

HÉRITAGE ET REPRÉSENTATION MÉMOIRE

- ★ Comment représenter un objet en mémoire quand il y a de l'héritage ?

- Un objet PointNomme "englobe" un objet Point

```
PointNomme p = new PointNomme(1, 2, "toto");
```



Remarque : on peut aussi utiliser une représentation simplifiée

- △ cette dernière représentation donne une vision trompeuse... donc à éviter



La classe PointNomme **hérite** / **étend** / **dérive** de la classe Point
On dit que :

- Point est la **classe mère** (ou **super-classe**) de la classe PointNomme
- PointNomme est une **classe fille** (ou **sous-classe**) de Point

Remarques :

- Une classe fille ne peut hériter que d'une seule classe
- Une classe peut avoir de nombreuses classes filles
- La classe fille "connaît" sa classe mère
- La classe mère ne "connaît" pas ses classes filles

this = référence vers l'objet courant

Quand on est dans la classe fille :

super = permet d'accéder aux attributs, méthodes et constructeurs de la classe mère **qui ne sont pas privés**

- **super.maVariable** :
 - ◆ accès à la variable maVariable de la classe mère
- **super.maMéthode(...)** :
 - ◆ appel à la méthode de même signature de la classe mère
- **super(...)** ;
 - ◆ appel au constructeur de la classe mère qui a la même signature

Remarque : d'autres informations sur super slide 41.

HÉRITAGE ET CONSTRUCTEURS

★ Que doit faire le constructeur d'une classe fille ?

Principe : 2 étapes

- 1 appeler le constructeur de la **classe mère**
- 2 initialiser les variables d'instance de la **classe fille**

```
1 public class PointNomme extends Point {
2     private String name;
3     public PointNomme(double x, double y, String name) {
4         super(x, y); // appel au constructeur de la classe mère
5         this.name = name; // init. variable d'instance fille
6     }
}
```

L'appel au constructeur de la classe mère **super(...)** :

- △ doit toujours être la première instruction du constructeur
- △ ne peut être utilisé que dans un constructeur

HÉRITAGE ET CONSTRUCTEURS

△ Il faut choisir un constructeur de la classe mère qui existe

```
1 public class Point {
2     private double x, y;
3     public Point(double x, double y) {
4         this.x = x; this.y = y;
5     }
6     public Point() {
7         this(Math.random(), Math.random());
8     }
9 }
10 public class PointNomme extends Point {
11     private String name;
12     public PointNomme(double x, double y, String name) {
13         super(x, y); // OK Point(double, double) existe
14         this.name = name;
15     }
16     public PointNomme(String name) {
17         super(); // OK Point() existe
18         this.name = name;
19     }
20     public PointNomme(double val, String name) {
21         super(val); // Erreur le constructeur Point(double) n'existe pas dans la mère
22         this.name = name;
23     } // On pourrait faire par exemple : this(val, val, name);
}
```

HÉRITAGE ET CONSTRUCTEURS

△ Quand une classe hérite d'une autre classe, penser à mettre le `super(...)` au début de **chaque constructeur** de la classe fille

Cas particulier S'il existe un constructeur accessible et sans argument dans la classe mère :

```
1 public class Point {
2     private double x,y;
3     public Point(){
4         x=0; y=0;
5     }
6     ...
```

alors il n'est pas obligatoire d'écrire `super();`, car cela est fait automatiquement par Java :

```
7 public class PointNomme extends Point {
8     private String name;
9     public PointNomme(String name) {
10         super(); // optionnel
11         this.name = name;
12     }
13     ...
```

HÉRITAGE ET CONSTRUCTEURS : EXEMPLE

```
1 public class Animal {
2     private String nom;
3     public Animal(String nom) {
4         this.nom = nom;
5     }
6     public String getNom() {
7         return nom;
8     }
9 }
10 public class Poule
11     extends Animal {
12     private static int cpt = 0;
13     public Poule() {
14         super("Poule"++cpt);
15     }
16 }
17 public class Renard
18     extends Animal {
19     private String couleur;
20     public Renard(String nom,
21         String couleur) {
22         super(nom);
23         this.couleur=couleur;
24     }
25 }
```

Le constructeur ...

- ... de la classe mère Animal prend 1 paramètre
- ... de la classe fille Poule prend 0 paramètre
- ... de la classe fille Renard prend 2 paramètres

```
31 Poule p1=new Poule();
32 Poule p2=new Poule();
33 Renard r=new Renard("Rox","roux");
34 System.out.println(p1.getNom());
35 System.out.println(p2.getNom());
36 System.out.println(r.getNom());
```

Affiche :

```
Poule1
Poule2
Rox
```

RAPPEL : UML : COMPOSITION/AGRÉGATION

- La relation de composition/agrégation est représentée par une **ligne avec un losange** du côté de la classe qui agrège

```
1 public class Segment{
2     private Point a,b;
3     ...
```

Diagramme de classe détaillé

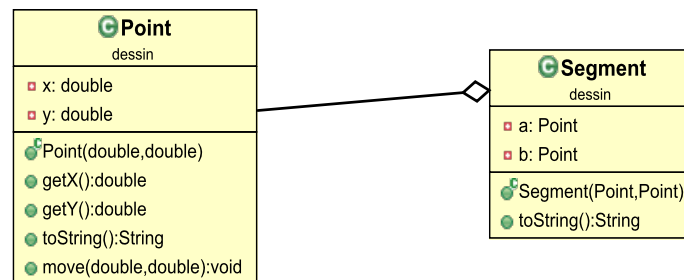
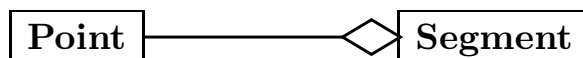


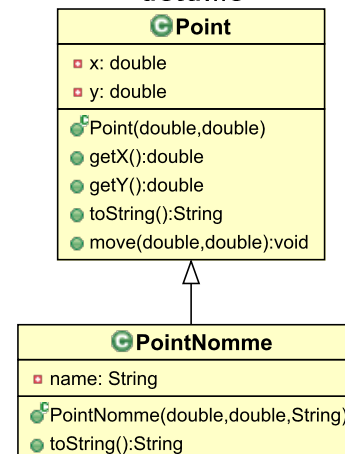
Diagramme de classe simplifié



UML : HÉRITAGE

- La relation d'héritage est représentée par une **ligne fléchée orientée de la classe fille vers la classe mère**. Le bout de la flèche est un **triangle vide**.

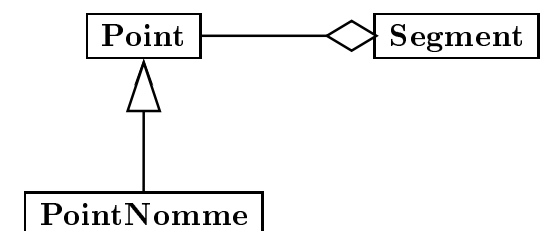
Diagramme de classe détaillé



`PointNomme pn = new PointNomme(1,2,"A");`

- pn est aussi un Point (accès à `move`, `getX`, `getY`...)

Diagramme de classe simplifié

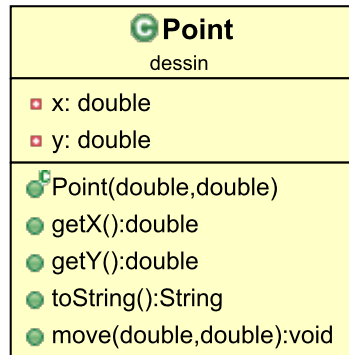


RAPPEL : UML CLIENT vs FOURNISSEUR

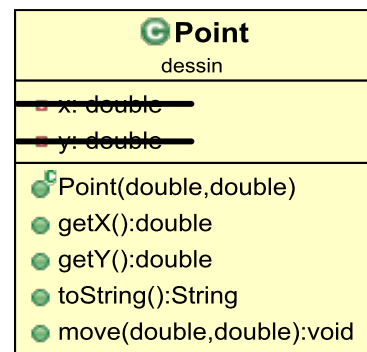
Plusieurs types de diagrammes pour plusieurs usages :

- **Vue fournisseur** : représente tous les attributs, constructeurs et méthodes
- **Vue client** : représente seulement les attributs, constructeurs et méthodes **public**

Vue fournisseur



Vue client



NOUVEAU NIVEAU D'ACCÈS : protected

Attribut/méthode :

- **public** : visible partout
- **private** : visible dans la classe uniquement

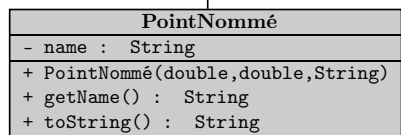
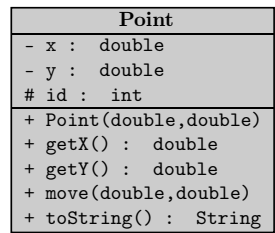
Nouveau niveau d'accès : **protected**

- **protected** :
 - ◆ visible dans la classe
 - ◆ visible dans les classes filles et descendantes
 - ◆ mais pas dans les autres classes

Dans quel cas l'utiliser ?

- quand on veut que les classes filles puissent avoir accès à un attribut/méthode, mais pas les autres classes
- cas assez rare

NOUVEAU NIVEAU D'ACCÈS : protected



```

1 public class Point {
2     private double x,y;
3     protected final int id;
4     ...
5 }
6 public class PointNomme extends Point {
7     ...
8     public void methode(double d){
9         int toto = id; // ou super.id;
10        ...
11    }
12 }
    
```

Variable/Méthode **protected**

- Accès depuis la classe
- Accès depuis les classes filles
- Pas d'accès depuis une classe quelconque

HÉRITAGE : PROPRIÉTÉS

Si B hérite de A, implicitement, **B hérite** :

- des méthodes **publics** de A exceptés les constructeurs publics
- des méthodes **protégées** de A exceptés les constructeurs protégés
- d'un attribut **super** du type de la super-classe (A)
 - ◆ **super** référence la *partie de B* qui correspond à A

En revanche, **B n'hérite pas** :

- des attributs **privés** de A
- des méthodes **privées** de A
- des constructeurs **publics, privés ou protégés** de A

Une classe ⇒ 3 visions possibles : développeur, **héritier**, client

- 1 Héritage : syntaxe
- 2 Héritage : subsomption et polymorphisme
 - Principe de subsomption
 - Polymorphisme : application aux tableaux
 - La classe Object
- 3 Héritage : surcharge / redéfinitions

Si la classe B **hérite** de la classe A :

- les méthodes de A peuvent être invoquées sur un objet de la classe B

```
PointNomme pnA=new PointNomme(5,6,"A");
pnA.move(7,8); // méthode de Point
```

- **Subsomption** : dans toute expression qui attend un A, on peut utiliser un B à la place

```
Point p = new PointNomme(1,2,"toto");
```

△ la variable est un Point, l'objet est un PointNomme

Polymorphisme = exploitation de la **subsomption**

Un PointNomme **EST UN** Point ⇒

on peut le traiter comme tel

SUBSUMPTION

- Un PointNomme **EST UN** Point

```
Point p = new PointNomme(1,2,"toto");
```

- Mais un Point **N'EST PAS** un PointNomme

```
PointNomme pn=new Point(2,3); // -> ERREUR compilation
```

- Par héritage, les méthodes de la classe mère sont accessibles à partir d'une variable de la classe fille

```
PointNomme pnA=new PointNomme(5,6,"A");
pnA.move(7,8); // méthode de Point
```

- Mais les méthodes de la classe fille ne sont pas accessibles à partir d'une variable de la classe mère

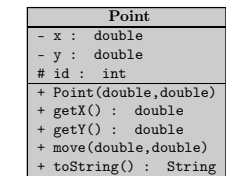
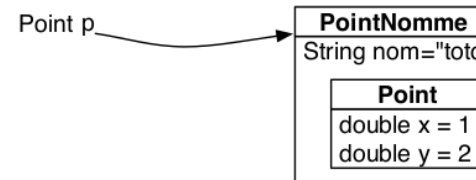
```
Point p = new PointNomme(1,2,"toto"); // subsomption
// Soit getName() méthode de PointNomme
p.getName(); // -> ERREUR compilation
// pas accessible à partir d'une variable Point
// un Point n'est pas un PointNomme
```

Role du compilateur :

il vérifie le type des **variables** est les possibilités offertes par celles-ci.

SUBSUMPTION : VISIONS COMPILATEURS vs JVM

```
Point p = new PointNomme(1,2,"toto");
```



Compilateur

La **variable** p est de type **Point** :
seule les méthodes de **Point** sont accessibles :

- p.getX(); p.getY(); // OK
- p.getName();
⇒ **Erreur de compilation**
(méthode inconnue dans la classe Point)

JVM

L'**instance** référencée par p est de type **PointNomme**

- En cas d'appel à toString(), c'est bien la **méthode de PointNomme** qui est invoquée.

SUBSUMPTION : ARGUMENTS DE MÉTHODE

```
1 Point p1 = new Point(1,2);
2 Point p2 = new PointNomme(1,2, "toto"); // OK
3 Point p3 = new ClasseHeritantDePoint(); // OK
```

★ Et pour les arguments de méthodes ?

```
1 public class Truc {
2     public void maMethode(Point p){
3         ...
4     }
5 }
6 // main
7 Truc t = new Truc();
8 t.maMethode(new Point(1,2));
9 t.maMethode(new PointNomme(1,2, "toto")); // OK
10 t.maMethode(new ClasseHeritantDePoint()); // OK
```

Idée :

Comme tous les descendants de Point sont des Point...

⇒ Toutes les informations utiles/nécessaires et toutes les méthodes clientes sont disponibles

⇒ aucun problème technique en perspective !

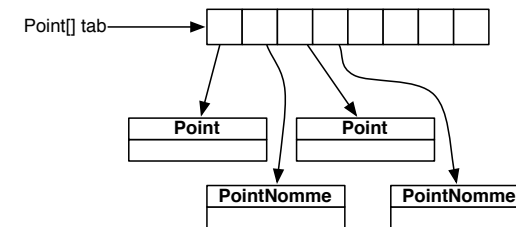
POLYMORPHISME : APPLICATION AUX TABLEAUX

★ Application classique : un tableau de points qui contient des points et des points nommés

```
1 Point[] tab = new Point[10]; // OK, 10 variables de type Point
2                               // aucun objet Point créé
```



```
3 for(int i=0; i<tab.length; i++) {
4     if ( i%2 == 0 )
5         tab[i] = new Point(Math.random()*10, Math.random()*10);
6     else
7         tab[i] = new PointNomme(Math.random()*10,
8                                 Math.random()*10, "toto"+i);
9 }
```

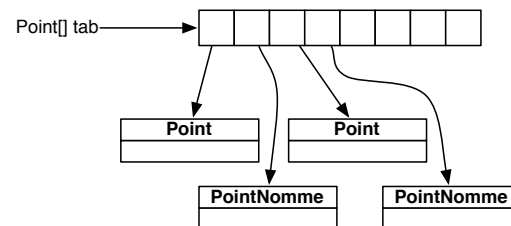


POLYMORPHISME : APPLICATION AUX TABLEAUX

★ Peut-on bouger tous les points du tableau ?

Oui, car la méthode move :

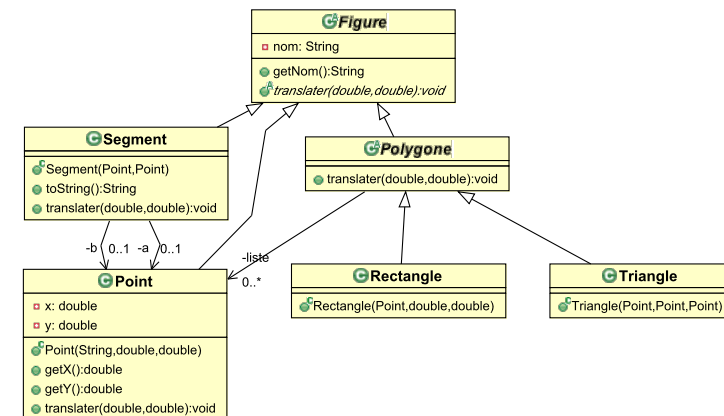
- est dans Point
- est héritée par PointNomme



```
1 // par exemple, procédure de Figure (méthode de classe)
2 public static void translaterTout(Point[] tab,
3                                   double tx, double ty) {
4     for(int i=0; i<tab.length; i++)
5         tab[i].move(tx, ty);
6 }
```

```
11 // variante
12 public static void translaterTout(Point[] tab,
13                                   double tx, double ty) {
14     for(Point p : tab)
15         p.move(tx, ty);
16 }
```

EXEMPLE PLUS COMPLEXE



On peut :

- créer un tableau de Figure
- remplir le tableau avec des segments, points, rectangles, triangles
- translater toutes les figures du tableau (la méthode translater est dans Figure)

⇒ Bien réfléchir aux opérations à effectuer sur les tableaux

△ On ne peut invoquer **que** les méthodes de la classe mère

- Exemple : si le type est Figure, on ne peut invoquer que les méthodes de Figure, même si l'objet est un Point

```
1 Figure [] tabFig = new Figure[10];
2 tabFig[0]=new Point();
3 tabFig[0].translater(2,2); // OK variable de type Figure
4 tabFig[0].getX(); // ERREUR variable de type Figure, pas Point
```

- Pour pouvoir accéder aux méthodes de la classe fille, il faut caster :

```
5 ((Point)tabFig[0]).getX(); // OK
```

On l'étudiera plus tard...

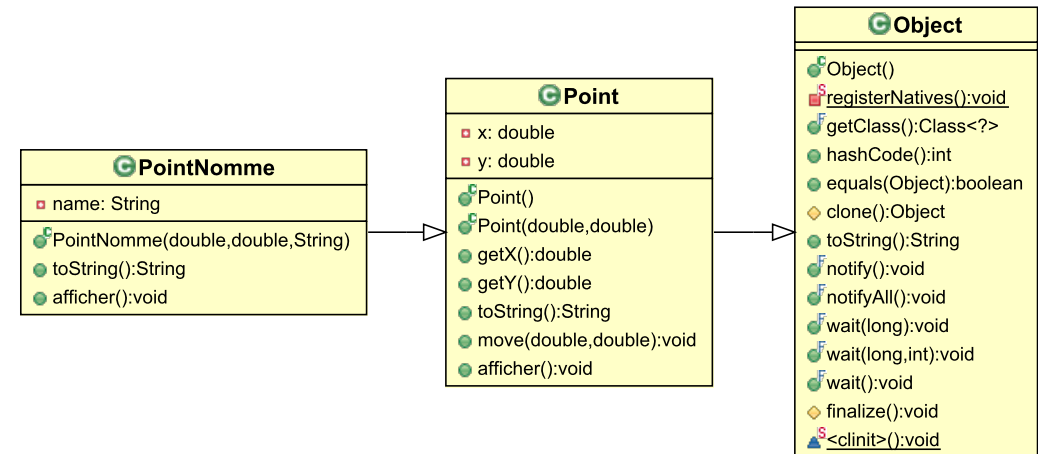
Role du compilateur :

il vérifie le type des **variables** est les possibilités offertes par celles-ci.

Classe standard

Toutes les classes dérivent de la classe **Object** de JAVA

- contient les méthodes standards toString(), equals(...)



Cet héritage est implicite, pas de déclaration dans la signature

LA CLASSE Object (SUITE)

- Une variable Object peut contenir une référence d'instance quelconque

```
1 Object o = new Point(1,2);
2 Object o2 = new PointNomme(2,3,"toto");
```

- ... Mais on ne peut (presque) rien faire sur o et o2

```
3 System.out.println(o.toString()); // OK
4 System.out.println(o.getX()); // KO
5 System.out.println(o.getY()); // KO
6 System.out.println(o2.getName()); // KO
```

- Création d'un tableau/ArrayList contenant "n'importe quoi"

```
7 Object [] tab = new Object[10];
8 tab[0] = "toto";
9 tab[1] = 10; // —> conversion implicite en Integer
10 tab[2] = new Point(1,2);
11 tab[3] = new PointNomme(2,3,"toto");
```

CONCLUSION ET LIMITES

- Le principe de **subsomption** est ce qui fait l'intérêt de l'héritage :
 - ◆ stockage d'instances hétérogènes dans des structures de données (type tableau/liste),
 - ◆ application des méthodes en *batch* sur toutes ces données.
- ... **A condition d'avoir bien réfléchi à l'architecture**, aux méthodes communes des différentes classes !
- Enfin, attention à ne pas s'emmêler :
 - ◆ si A EST UN B alors B n'est pas un A
 - ◆ **La subsomption ne marche que dans un sens**

- 1 Héritage : syntaxe
- 2 Héritage : subsomption et polymorphisme
- 3 Héritage : surcharge / redéfinitions
 - Surcharge
 - Redéfinition
 - Redéfinition et usage de super

Définition

- Même nom de méthode **MAIS** argument(s) différent(s)
- Le type de retour n'est pas considéré pour différencier les méthodes

```

1 public class Point {
2     ...
3     public void move(double dx, double dy){
4         x+=dx; y+=dy;
5     }
6     public void move(double dx, double dy, double scale){
7         x+=dx*scale; y+=dy*scale;
8     }
9     public void move(int dx, int dy){
10        x+=dx; y+=dy;
11    }
12    public void move(Point p){
13        x+=p.x; y+=p.y;
14    }

```

SURCHARGE : QUI FAIT QUOI

Le compilateur (pré)-sélectionne les méthodes :

- Ces méthodes sont totalement différentes pour le compilateur qui analyse le type des paramètres
- Elles peuvent être indifféremment dans la classe fille ou dans la classe mère

```

1 public class Point {
2     ...
3     public void move(double dx, double dy){ // 1
4         x+=dx; y+=dy;
5     }
6     public void move(double dx, double dy, double scale){ // 2
7         x+=dx*scale; y+=dy*scale;
8     }
9
10    Point p = new Point(1,2);
11    p.move(3, 1); // présélection de 1
12    p.move(3, 1, 0.5); // présélection de 2

```

SURCHARGE : LES LIMITES

- Interdiction d'avoir des signatures identiques

```

1 public void move(double dx, double dy){
2     x+=dx; y+=dy;
3 }
4
5 // Meme signature pour le compilateur
6 // → ERREUR de compilation
7 public void move(double a, double b){
8     x+=a; y+=b;
9 }

```

- Pas de prise en compte du type de retour

```

1 // Meme signature pour le compilateur
2 // → ERREUR de compilation
3 public Point move(double dx, double dy){
4     x+=dx; y+=dy;
5     return this;
6 }

```

REDÉFINITION

Définition

Redéfinition d'une méthode de **même signature** dans la classe fille

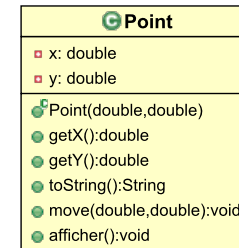
```
1 public class Point {
2     ...
3     public void afficher(){ // 1
4         System.out.println("Je suis un Point");
5     }
6 }
7 ////
8 public class PointNomme extends Point {
9     ...
10    public void afficher(){ // 2
11        System.out.println("Je suis un PointNomme");
12    }
13 }
```

- Pas de problème à la compilation
- A l'exécution, la JVM décide de la méthode à invoquer en fonction du type de l'instance appelante

REDÉFINITION : EXEMPLES DE FONCTIONNEMENT

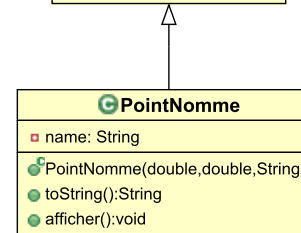
Cas 1 : facile

```
1 public static void main(String[] args) {
2     Point p = new Point(1, 2);
3     p.afficher();
4 }
```



Dans la classe Point, une seule méthode correspond à la signature afficher()
Affichage de :

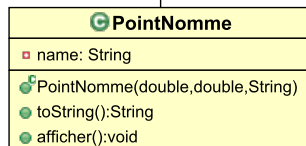
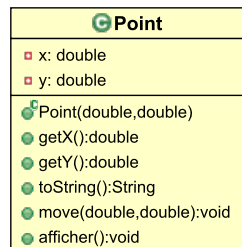
Je suis un Point



REDÉFINITION : EXEMPLES DE FONCTIONNEMENT

Cas 2 : résolution d'une ambiguïté

```
1 public static void main(String[] args) {
2     PointNomme pn = new PointNomme(1, 2, "toto");
3     pn.afficher();
4 }
```

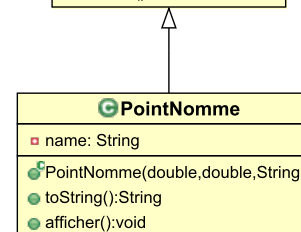
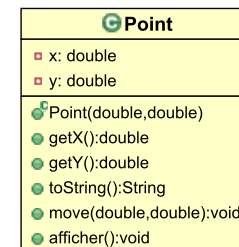


- Dans la classe PointNomme, **deux méthodes** correspondent à la signature afficher() :
 - ◆ une dans Point
 - ◆ une dans PointNomme
- La JVM choisit, **au moment de l'exécution** du programme en fonction du type de l'instance de pn, la méthode la plus proche
- Affichage de :
Je suis un PointNomme
car **l'objet est un PointNomme**

REDÉFINITION : EXEMPLES DE FONCTIONNEMENT

Cas 3 : Surcharge + subsomption

```
1 public static void main(String[] args) {
2     Point p = new PointNomme(1, 2, "toto"); // subsomption
3     p.afficher(); // ???
4 }
```



- **Compilation** : type des variables uniquement
 - ◆ variable p est un Point
 - ◆ afficher() existe dans Point ⇒ OK
- **Exécution** : JVM regarde le **type de l'objet** référencé par p
 - ◆ p référence un PointNomme
 - ◆ recherche de afficher() dans PointNomme
- Affichage de :
Je suis un PointNomme
car **l'objet est un PointNomme**

REDÉFINITION ET USAGE DE super 1/3

Le mot clef **super** permet :

- de **préciser** que l'on utilise des informations de la super-classe

```
1 public class PointNomme extends Point {
2     ...
3     public void afficher(){
4         System.out.println("Je suis un PointNomme" +
5             "de coordonnées: " + super.getX() + " " +
6             super.getY());
7     }
8 }
```

- de **forcer** le programme à aller chercher une méthode dans la super-classe (**obligatoire**)

```
1 public class PointNomme extends Point {
2     ...
3     public void affichageGlobal(){
4         afficher(); // -> Je suis un PointNomme
5         this.afficher(); // -> Je suis un PointNomme
6         super.afficher(); // -> Je suis un Point
7     }
8 }
```

REDÉFINITION ET USAGE DE super 2/3

L'un des usages les plus classiques concerne toString() :

```
1 // classe Point
2 public String toString() {
3     return "("+x+", "+y+")";
4 }
5 // classe PointNomme
6 public String toString() {
7     return "PointNomme_" + name + "_" + super.toString() ;
8 }
```

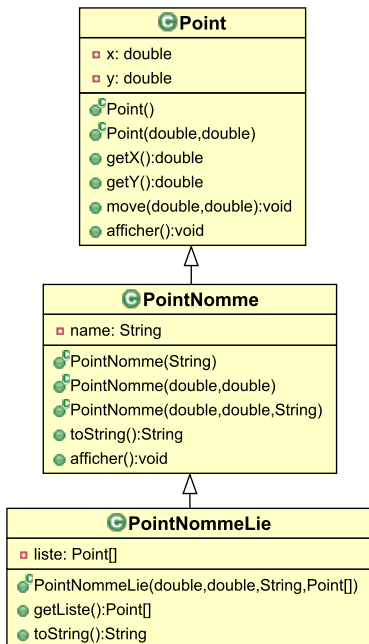
- Quels sont les affichages en sortie du code suivant :

```
21 Point p = new Point(1,2);
22 PointNomme pnA = new PointNomme(3,4,"A");
23 Point pPnB = new PointNomme(5,6,"B");
24 System.out.println(p.toString());
25 // (1,2)
26 System.out.println(pnA.toString());
27 // PointNomme A (3,4)
28 System.out.println(pPnB.toString());
29 // PointNomme B (5,6) // toString() de l'objet, pas de la variable
```

- Que se passerait-il si on oublie le super à la ligne 7 ?

◆ Un appel récursif...

REDÉFINITION ET USAGE DE super 3/3



Ajout d'une classe "petite-fille"
PointNommeLie pour un point lié à
d'autres dans l'espace (graphe)

Dans la classe PointNommeLie :

```
1 toString(); // OK PointNommeLie
2 super.toString(); // OK PointNomme
3 super.super.toString(); // syntaxe interdite
```

△ Avec super, on ne peut remonter
que de un niveau

```
4 getX(); // OK, existe ici par
5 // héritage de Point
6 super.getX(); // OK existe dans PointNomme
7 // par héritage de Point
```

Les méthodes redéfinies dans la classe
mère **bloquent** l'accès aux versions de la
classe "grand-mère".

OUVERTURE DE LA REDÉFINITION

Définition

Il est possible d'**augmenter la visibilité** d'une méthode dans la classe
fille mais **pas de la réduire**

Exemple :

- dans Point

```
1 protected Point maMethode(){ return new Point (...); }
```

- dans PointNomme

```
2 // pour éviter le cast
3 protected PointNomme maMethode(){ return new PointNomme (...); }
```

- dans PointNommeLie

Ouverture de la redéfinition possible (méthode devient **public**)

```
4 public PointNommeLie maMethode(){ return new PointNommeLie (...); }
```

Mais pas de réduction de la visibilité possible

```
5 private PointNommeLie maMethode(){ return new PointNommeLie (...); }
6 // ERREUR : redéfinition : ne peut pas devenir private
```