

## 2i002 - Héritage

Vincent Guigue



Le cours est inspiré de sources diverses: L. Denoyer, F Peschanski,...

## Principe 1 : Encapsulation

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

## Principe 2 : Agrégation/Association

- Classe A **A UN** Classe B
- Classe A **UTILISE** Classe B

## Principe 3 : Héritage

- Class B **EST UN** Classe A

## Idée de l'héritage

Spécialiser une classe, ajouter des fonctionnalités dans une classe  
Hériter tout le comportement d'une classe existante

- **1 classe de base = plusieurs spécialisations possibles**
  - Animaux → vache, chien, mouton...
  - Hiérarchisation possible : Animaux → AnimauxAiles → Papillon
- **Ne pas modifier le code existant**
  - Point → PointNomme : un point avec un nom
  - Ne pas modifier la classe de base
- **Mais ne pas faire de copier coller**
  - Hériter le comportement d'une classe

# EXEMPLES & CONTRE EXEMPLES

Pour les cas suivants : dire si les relations sont des relations de type **Agrégation/Association** ou **Héritage** :

- Salle de Bains et Baignoire
- Piano et JoueurPiano
- Personne, Enseignant et Etudiant
- Cercle et ellipse
- Entier et Réel

# EXEMPLES & CONTRE EXEMPLES

Pour les cas suivants : dire si les relations sont des relations de type **Agrégation/Association** ou **Héritage** :

- Salle de Bains et Baignoire
- Piano et JoueurPiano
- Personne, Enseignant et Etudiant
- Cercle et ellipse
- Entier et Réel

# EXEMPLES & CONTRE EXEMPLES

Pour les cas suivants : dire si les relations sont des relations de type **Agrégation/Association** ou **Héritage** :

- Salle de Bains et Baignoire
- Piano et JoueurPiano
- Personne, Enseignant et Etudiant
- Cercle et ellipse
- Entier et Réel

# EXEMPLES & CONTRE EXEMPLES

Pour les cas suivants : dire si les relations sont des relations de type **Agrégation/Association** ou **Héritage** :

- Salle de Bains et Baignoire
- Piano et JoueurPiano
- Personne, Enseignant et Etudiant
- Cercle et ellipse
- Entier et Réel

# EXEMPLE : POINT ET POINTNOMME

## Problème

On veut implémenter deux classes :

- ① Point en 2 dimensions
- ② Point en 2 dimensions **qui possède un nom**

```
1  public class Point { // (1) Type / nom de classe
2      private double x;    // (2) Attributs
3      private double y;
4      public Point(double x, double y) { // (3) Constructeurs
5          this.x = x;
6          this.y = y;
7      }
8      public double getX() { // (4) Accesseurs
9          return x;
10     }
11     public double getY() {
12         return y;
13     }
14     // Ca continue ...
```

# EXEMPLE : POINT ET POINTNOMME

```
1 // (4) méthodes ( traitements )
2 public double calculeDistance( Point p2 ) {
3     double dx = Math.abs(p2.getX() - getX());
4     double dy = Math.abs(p2.getY() - getY());
5     return Math.sqrt(dx*dx+dy*dy);
6 }
7
8 public void move( double tx , double ty ) {
9     deplace(x+tx , y+ty );
10 }
11 // (4) méthodes privées
12 private void deplace( double x , double y ) {
13     this.x=x; this.y=y;
14 }
15 // (5) méthodes standards
16 public String toString() {
17     return "(" +x+ "," +y+ ")";
18 }
```

## HÉRITAGE : SYNTAXE

```
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  
9     public String toString() {  
10        return "PointNomme[" + name + "] " +  
11            super.toString() + "]";  
12    }  
13 }
```

- Mot clé **extends** dans la signature de la classe
- Mot clé **super** (à voir plus tard)

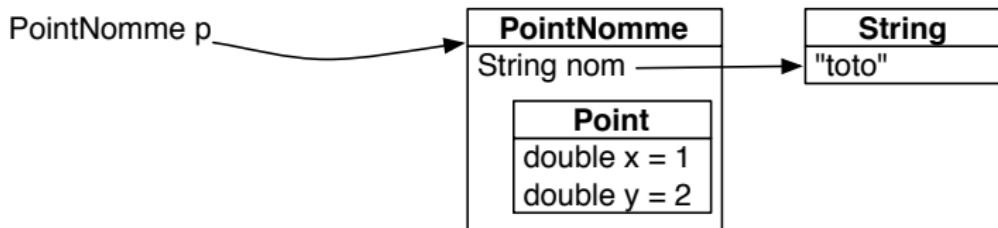
### Erreur courante

Attention à ne pas dupliquer les attributs : la classe fille étend la super-classe, elle *contient* la super-classe

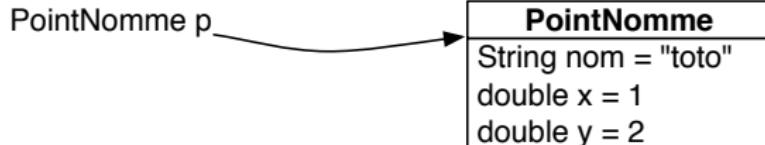
# REPRÉSENTATION MÉMOIRE

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

- ① Représentation complète : tous les objets sont séparés, on représente explicitement la séparation entre attributs de la classe et de la super-classe



- ② Représentation usuelle simplifiée :



Attention, la représentation de la `String` est limite...

# CONSTRUCTION D'UNE INSTANCE DE LA CLASSE FILLE

Idée :

- ① construire une instance de la classe mère
- ② initialiser les attributs relatifs à 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); // première instruction, obligatoirement  
5         this.name = name; // init. attributs de la classe fille  
6     }  
}
```

- **Règle générale** : choisir un constructeur dans la super-classe et l'appeler avec `super(...)`
- **Exception** : si la super-classe a un constructeur sans argument, l'appel à `super()` est implicite

## CAS PARTICULIER : super()

S'il existe un constructeur accessible & sans argument dans la super-classe :

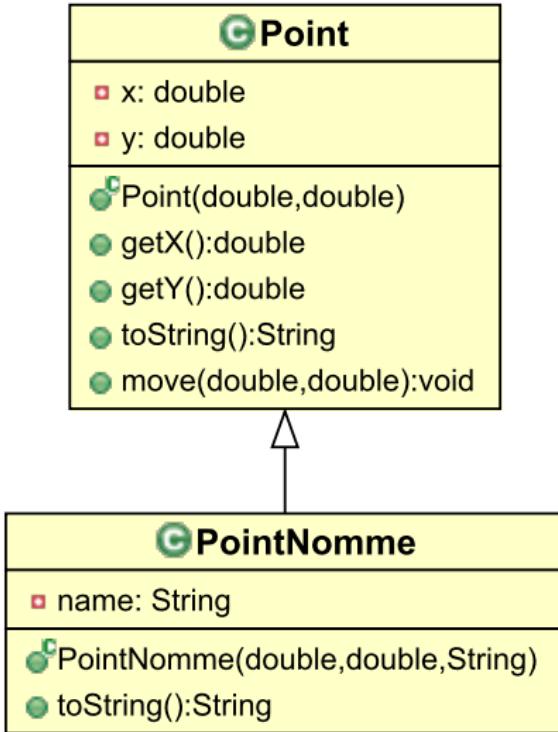
```
1 public class Point {  
2     private double x,y;  
3     public Point(){  
4         x=0; y=0;  
5     }  
6     ...
```

Alors, les deux écritures suivantes sont équivalentes :

```
1 public class PointNomme  
2             extends Point {  
3     private String name;  
4  
5     public PointNomme(String name)  
6         super();  
7         this.name = name;  
8     }  
9     ...
```

```
1 public class PointNomme  
2             extends Point {  
3     private String name;  
4  
5     public PointNomme(String name) {  
6         this.name = name;  
7     }  
8     ...
```

# REPRÉSENTATION DES LIENS UML



- Extension des capacités/propriétés d'un objet
- `PointNomme p = new PointNomme(1, 2, "totot");`
- `p` est un `PointNomme`
- `p` est un `Point` (accès à `getX()`, `getY()`...)

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

- des **méthodes publiques** de A exceptés les constructeurs publiques
- 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 publiques, privés ou protégés** de A

# VISION CLIENT

PointNomme
<b>PointNomme(x, y, nom)</b>
getX() : double
getY() : double
move(tx :double, ty :double) : void
toString() : String
<b>getNom() : String</b>

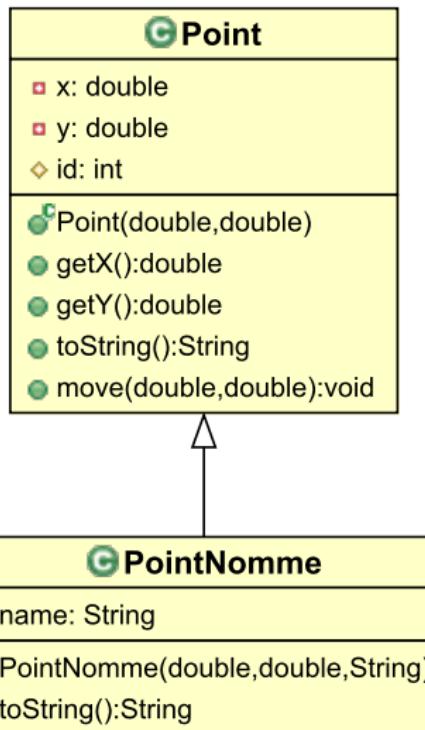
```
1 PointNomme p =
2     new PointNomme(1,2, "p1");
3
4 System.out.println(p.getX());
5 // méthode définie dans Point
6 // héritée dans PointNomme
7 // utilisée de manière transparente
```

- Méthodes publiques de Point (mais pas les constructeurs)
- Pas de vision sur les données private

## protected

- Niveau intermédiaire
- Les attributs et méthodes **protected** ne sont pas visibles de l'extérieur mais sont visibles dans les classes filles
- **public** : visible partout, dans la classe et chez le client (main, autres classes...)
- **protected** : visible dans la classe, dans les classes filles mais nulle part ailleurs.
- **private** : visible dans la classe seulement

## EXEMPLE D'ACCÈS PROTECTED



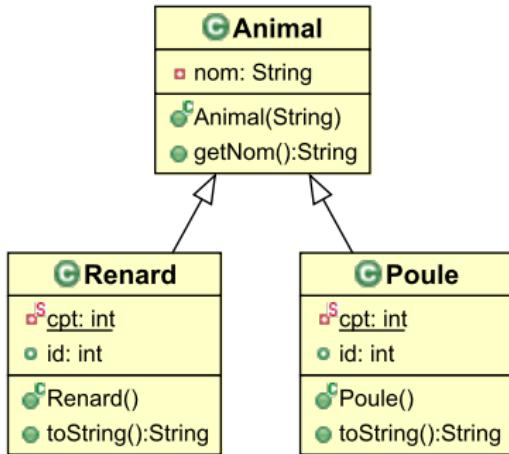
```
1 public class Point {  
2     private double x,y;  
3     // les classes dérivées y ont accès  
4     protected int id;  
5     ...  
6 }  
7 public class PointNomme {  
8     ...  
9     void methode(double d){  
10         int toto = id; // ou super.id;  
11         ...  
12     }  
13 }
```

## Variable/Méthode **protected**

- Accès depuis la classe
  - Accès depuis les classes filles
  - Pas d'accès depuis l'extérieur

Une classe  $\Rightarrow$  3 visions possibles : développeur, héritier, client

# CAS PARTICULIER : ARGUMENTS PAR DÉFAUT



- Le constructeur de la super-classe a des arguments
- Les constructeurs des classes filles non...
  - On donne des arguments par défaut

```

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
11 public class Poule extends Animal{
12     private static int cpt = 0;
13     public int id ;
14
15     public Poule() {
16         super("poule");
17         id = cpt++;
18     }
19     public String toString(){
20         return String.format("%s%02d" ,
21                             getNom(), id);
22     }
23 }
  
```

The provided Java code defines the **Animal** class with a private attribute `nom` and a constructor that takes a `String` parameter. It also defines the **Poule** class, which extends **Animal**. The **Poule** class has a private static variable `cpt` initialized to 0, a public attribute `id`, and its own constructor that calls the **super** constructor with the string "poule" and increments the `cpt` counter to assign a unique `id` to each instance. The **Poule** class also overrides the `toString()` method to format the output as "`getNom()%02d`".

# CONCLUSION

- Un nouveau paradigme de réflexion, au cœur de la POO
- Des éléments de syntaxe à maîtriser
- ... To be continued (**subsomption, surcharge & redéfinition**)

# 2i002 - Héritage : principe de subsomption

Vincent Guigue



Le cours est inspiré de sources diverses: L. Denoyer, F Peschanski,...

# POLYMORPHISME

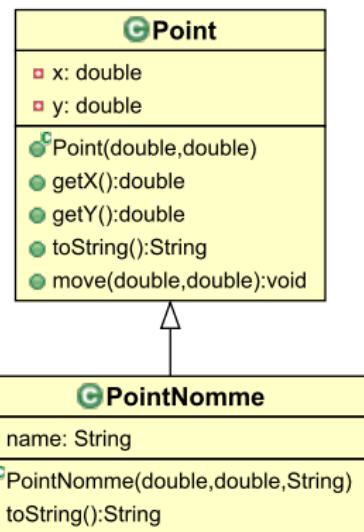
Si la classe B hérite de la classe A :

- le type B « EST-UN » le type A
- les méthodes de A peut-être invoquée sur une instance de la classe B (+ transitivité A hérite de SA, SA hérite de SSA, etc.)
- **Subsomption** : Dans toute expression « qui attend » un A (type A), je peux « placer » un B à la place

Polymorphisme = exploiter la subsomption

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

Un PointNomme EST UN Point ⇒  
Je peux le traité comme tel



# POLYMORPHISME : PRÉSENTATION MÉMOIRE

Point
□ x: double
□ y: double

Point(double,double)
● getX():double
● getY():double
● toString():String
● move(double,double):void



PointNomme
□ name: String
● PointNomme(double,double,String)
● toString():String

## ○ Syntaxe :

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

## ○ Limite

```
1 p.getNom(); // -> ERREUR de compilation
```

## Role du compilateur :

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

# POLYMORPHISME : PRÉSENTATION MÉMOIRE

C Point	
□	x: double
□	y: double
⊕	Point(double,double)
⊕	getX():double
⊕	getY():double
⊕	toString():String
⊕	move(double,double):void



C PointNomme	
□	name: String
⊕	PointNomme(double,double,String)
⊕	toString():String

## ○ Syntaxe :

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

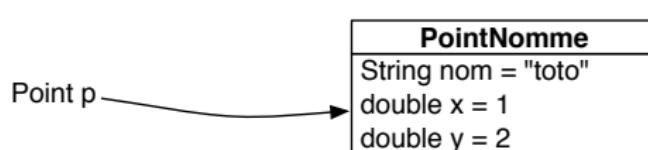
## ○ Limite

```
1 p.getNom(); // -> ERREUR de compilation
```

## Role du compilateur :

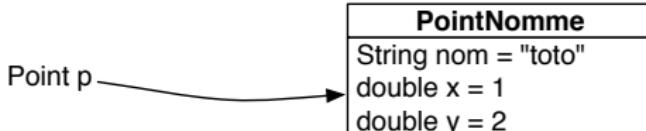
il vérifie le type des **variables** est les possibilités offertes par celles-ci.

## Représentation mémoire :



⇒ bien distinguer le type des **instances** et des **variables**

```
1 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(); //...
- p.getNom();  
⇒ **Erreur de compilation**  
(méthode inconnue dans la  
classe **Point**)

## JVM

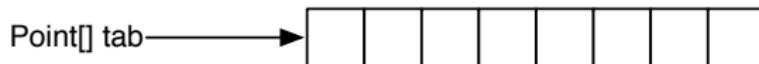
L'**instance** référencée par **p** est de  
type **PointNomme**

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

# POLYMORPHISME : PASSAGE AUX TABLEAUX

Application classique sur les tableaux de concepts abstraits :

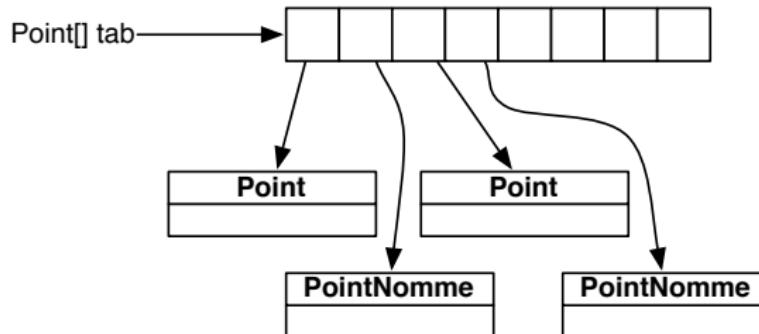
```
1 Point [] tab = new Point[10]; // OK,  
2 // il s'agit de 10 variables de type Point
```



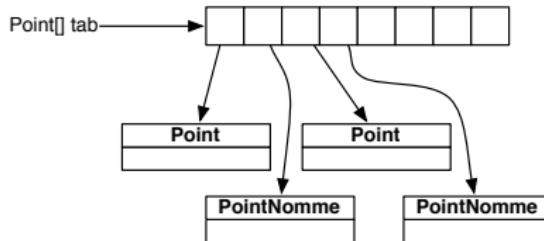
# POLYMORPHISME : PASSAGE AUX TABLEAUX

Application classique sur les tableaux de concepts abstraits :

```
1 Point [] tab = new Point [10]; // OK,  
2 // il s'agit de 10 variables de type Point  
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 }
```

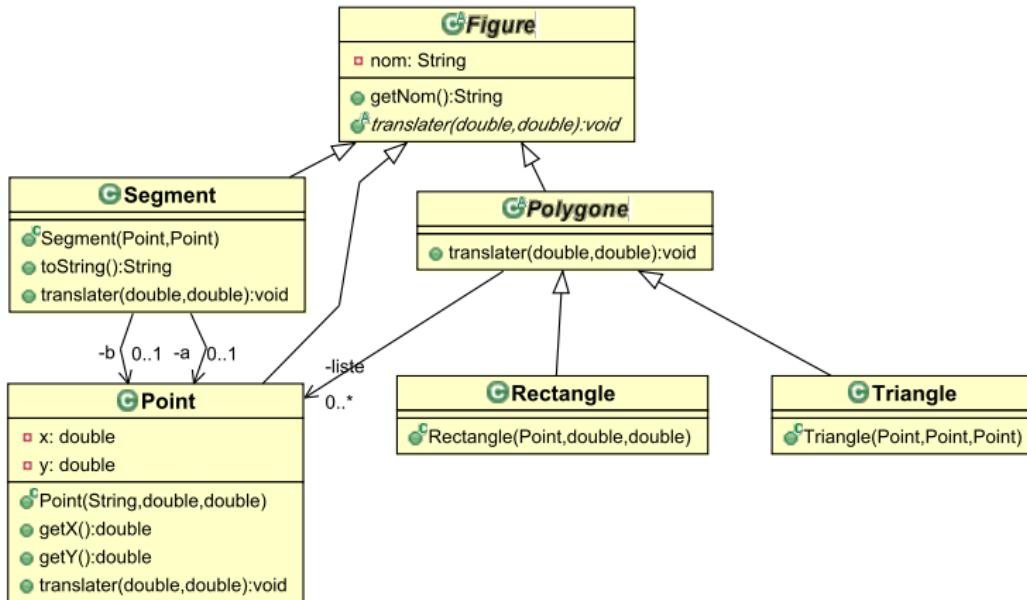


# POLYMORPHISME : APPLICATION SUR UN TABLEAU



```
1 // par exemple, procédure de Figure (méthode de classe)
2 public static void translaterTout(Point[] pts,
3                                     double tx, double ty) {
4     for(int i=0;i<pts.length;i++)
5         pts[i].translater(tx,ty);
6 }
7
8 // variante
9
10 public static void translaterTout(Point[] pts,
11                                     double tx, double ty) {
12     for(Point p : pts)
13         p.translater(tx,ty);
14 }
```

# EXEMPLE PLUS COMPLEXE



- ⇒ Bien réfléchir aux opérations à effectuer sur les tableaux
- recherche/affichage d'un nom
  - translation

# LIMITES DU POLYMORPHISME

- On ne peut qu'invoquer les méthodes du super-type
  - ex. : si le type est Figure, on ne peut invoquer que les méthodes de Figure, même si l'instance est un Point, un Carré, etc.

```
1 Figure fig1 = new Carré(2,1,4,5);
2 fig1.translater(2,2); // OK type Figure
3 double x = fig1.calculerSurface(); // KO type Figure !
4
5 Figure fig2 = new Point(2,1);
6 fig2.translater(2,2); // OK type Figure
7 fig2.getX(); // KO type Figure !
```

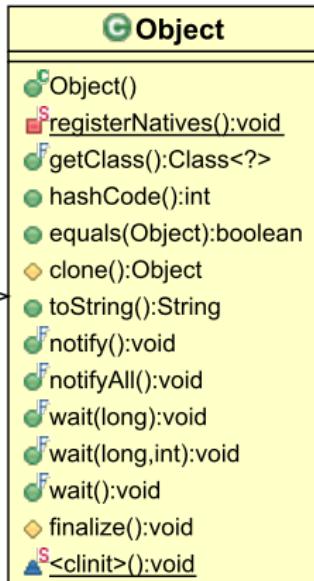
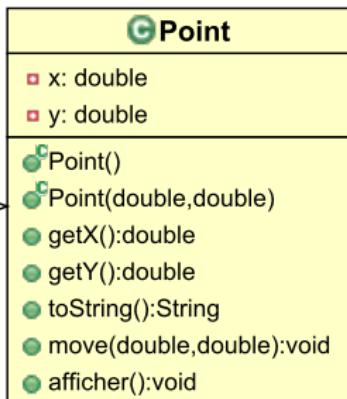
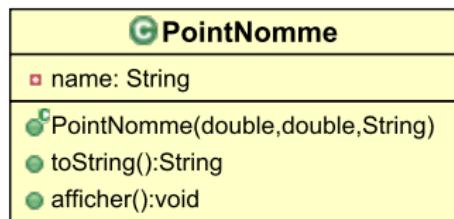
Role du compilateur :

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

# CLASSE OBJECT

## Classe standard

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



Cet héritage est implicite, pas de déclaration dans la signature

## OBJET, SUITE

- Avec une variable Object, je peux référencer n'importe quelle instance :

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

- ... Mais je ne peux (presque) rien faire sur o,o2

```
1 System.out.println(o.toString()); // OK
2 System.out.println(o.getX()); // KO
3 System.out.println(o.getY()); // KO
4 System.out.println(o2.getNom()); // KO
```

- Je peux donc créer un tableau/ArrayList avec n'importe quoi dedans :

```
1 Object[] tab = new Object[10];
2 tab[0] = "toto";
3 tab[1] = 10; // autoboxing
4 tab[2] = new Point(1,2);
```

# ARGUMENTS DE MÉTHODE

```
1 // dans n'importe quelle classe, par exemple Truc
2 public void maMethode(Point p){
3     ...
4 }
5
6 // invocations possibles:
7 Truc t = new Truc();
8 t.maMethode(new Point(1,2));
9 t.maMethode(new PointNomme(1,2, "toto"));
10 t.maMethode(new ClasseHeritantDePoint());
```

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 !

# LE CAS DE LA MÉTHODE equals

```
1 // Exemple de la classe Point
2 public boolean equals(Object obj) {
3     if (this == obj)
4         return true;
5     if (obj == null)
6         return false;
7     if (getClass() != obj.getClass())
8         return false;
9     Point other = (Point) obj;
10    if (x != other.x)
11        return false;
12    if (y != other.y)
13        return false;
14    return true;
15 }
```

Tous les objets sont comparables entre eux !

On peut donc rechercher un objet en particulier dans un tableau exploitant le polymorphisme :

```
1 ArrayList<Object> tab =
2             new ArrayList<Object>();
3 tab.add( "toto" ); tab.add(10);
4 tab.add( new Point(1,2));
5 tab.add( new PointNomme(2,3, "titi" );
6
7 tab.contains( new Point(1,2)); // true
8 // méthode exploitant equals
```

Question suivante : comment passer d'un Object (générique) à une classe spécifique (ici : Point) ?  
⇒ réponse dans le cours sur l'opérateur cast

## 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 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.**

# 2i002 - Héritage : surcharge / redéfinitions

Vincent Guigue



Le cours est inspiré de sources diverses: L. Denoyer, F. Peschanski, ...

# SURCHARGE

## Définition

Même nom de fonction, arguments différents

```
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    }
```

Rappel : dans la classe Point, vous avez accès aux attributs privés des autres instances de Point

# SURCHARGE : QUI FAIT QUOI

Le compilateur (pré)-sélectionne les méthodes :

- Ces méthodes sont totalement différentes (pour le compilateur qui analyse le type des arguments)
- Elles peuvent être indifféremment dans la classe ou la super-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

- Pas de prise en compte du type de retour
- 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  }  
10  
11 // Meme signature pour le compilateur  
12 //      -> ERREUR de compilation  
13 public Point move(double dx, double dy){  
14     x+=dx; y+=dy;  
15     return this;  
16 }
```

# REDÉFINITION :

## Définition

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

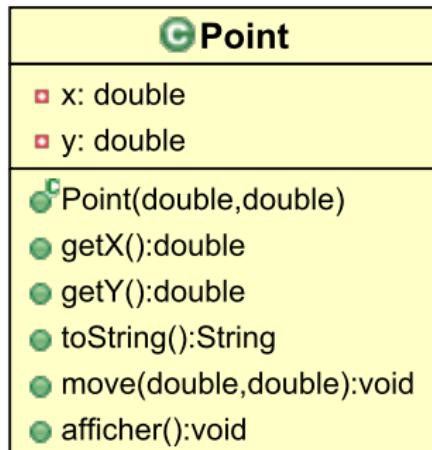
```
1 public class Point {  
2 ...  
3     public void afficher(){ // 1  
4         System.out.println("Je suis un Point");  
5     }  
6 }  
////  
7 public class PointNomme extends Point {  
8 ...  
9     public void afficher(){ // 2  
10        System.out.println("Je suis un PointNomme");  
11    }  
12 }  
13 ...  
14 }
```

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

# EXEMPLES DE FONCTIONNEMENT 1/3

## Cas 1 : (facile)

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



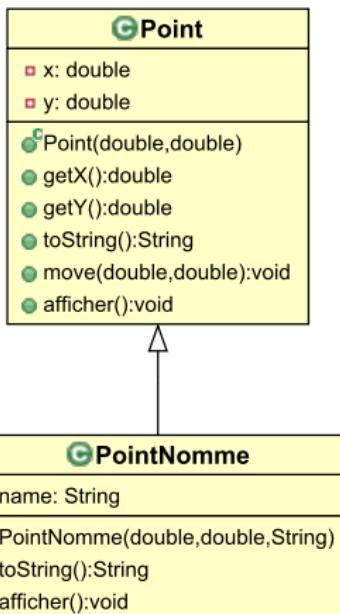
Dans la classe Point, une seule méthode correspond à la signature **afficher()**  
Affichage de :

```
1 Je suis un Point
```

## EXEMPLES DE FONCTIONNEMENT 2/3

### Cas 2 : (résolution d'une ambiguïté)

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



Dans la classe **PointNomme**, **deux méthodes** correspondent à la signature **afficher()** (méthodes accessibles : dans la classe et dans les classes parentes)

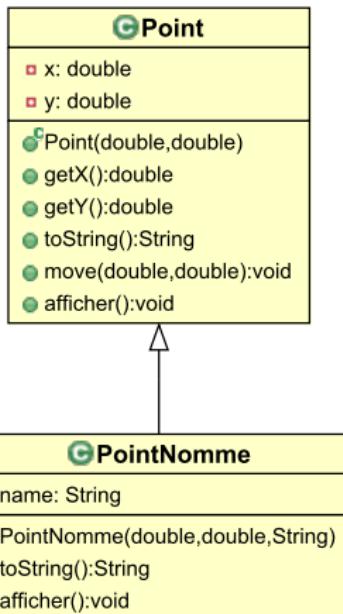
- Une dans **Point**
- Une dans **PointNomme**

La JVM choisit, **au moment de l'exécution** du programme en fonction du type de l'instance de **p**, la méthode la plus proche

# EXEMPLES DE FONCTIONNEMENT 2/3

## Cas 2 : (résolution d'une ambiguïté)

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

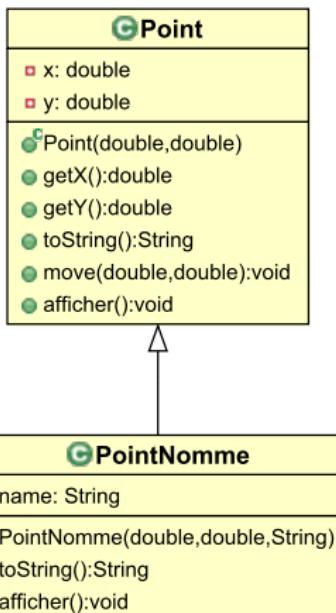


Affichage de :  
Je suis un PointNomme

# EXEMPLES DE FONCTIONNEMENT 3/3

## 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 :** (javac) = type des variables uniquement

- `p = Point`
- `void afficher()` existe dans `Point`  
⇒ OK

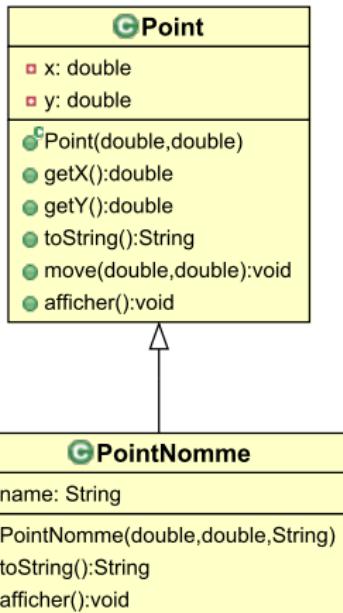
**Execution :** JVM regarde le **type de l'instance** référencée par `p`

- `p` référence un `PointNomme`
- recherche de `void afficher()` dans `PointNomme`

# EXEMPLES DE FONCTIONNEMENT 3/3

## Cas 3 : Surcharge + subsomption

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



Affichage de :  
Je suis un PointNomme

# USAGE DE super 1/3

Le mot clé **super** permet :

- de **préciser** que l'on va chercher des informations dans la super-classe (dans ce cas, **super** est **optionnel**)

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

- 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 }
```

# USAGE DE super 2/3

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

```
1 // classe Point
2     public String toString() {
3         return "(" + x + "," + y + ")";
4     }
5
6 // classe PointNomme
7     public String toString() {
8         return "PointNomme[" + name + " " + super.toString() + "]";
9     }
```

① Quels sont les affichages en sortie du code suivant :

```
1 Point p = new Point(1,2);
2 PointNomme pn = new PointNomme(3,4, "toto");
3
4 System.out.println(p.toString());
5 System.out.println(pn.toString());
```

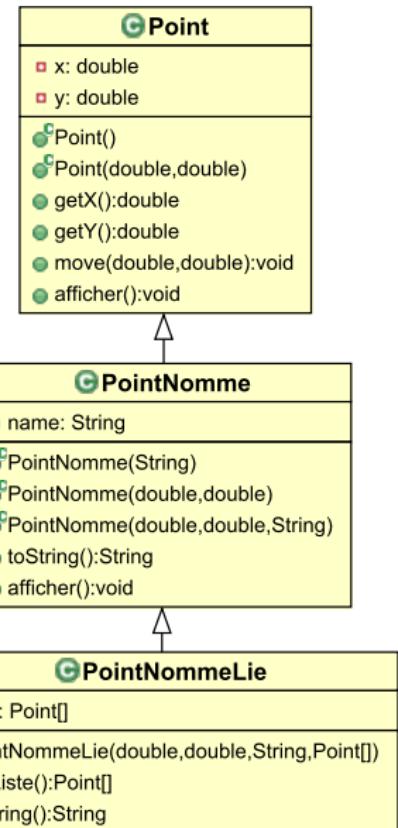
② Que se passerait-il si on oublie le `super` ?

# USAGE DE super 3/3

Ajout d'une classe pour un Point lié à d'autres dans l'espace (système de graphe)

```
1 // dans PointNommeLie
2
3 toString(); // OK local
4 super.toString(); // OK super-classe
5 super.super.toString(); // syntaxe interdite
6
7 getX(); // OK, existe ici par
8     // héritage de Point
9 super.getX(); // OK existe dans PointNommeLie
10    // par héritage de Point
```

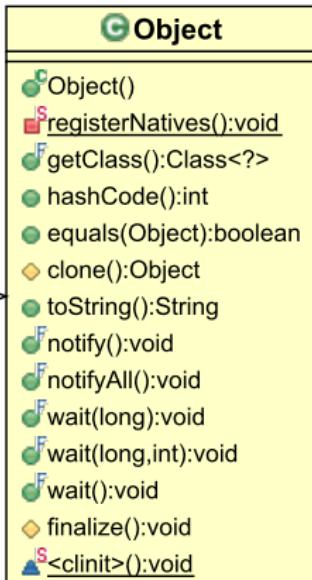
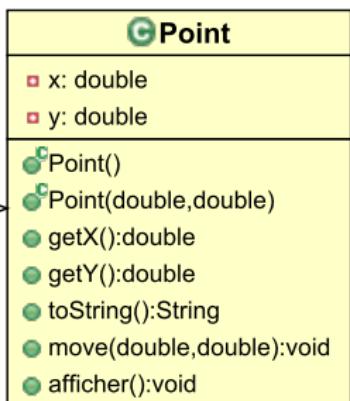
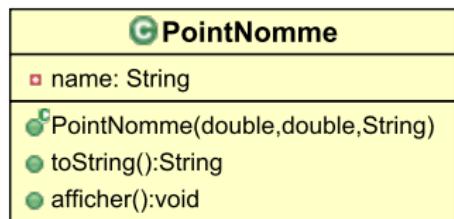
Les méthodes redéfinies dans la super-classes *empêchent* l'accès aux versions de la génération supérieure.



# CLASSE OBJECT

## Classe standard

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



Cet héritage est implicite, pas de déclaration dans la signature

# MÉTHODES STANDARDS

Dans le cadre de l'UE, nous nous intéresserons à certaines méthodes dites *standards*

- String `toString()`
- boolean `equals(Object)`

Sur la classe basique suivante :

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

Les opérations suivantes sont possibles :

```
1 Point p1 = new Point(1,2); Point p2 = new Point(1,2);  
2 System.out.println(p1.toString());  
3 System.out.println(p1.equals(p2));  
4 System.out.println(p1.equals(p1));
```

Quelles sont les sorties associées ?

## RAPPELS SUR boolean equals(Object o)

- Par défaut, equals existe mais teste l'égalité référentielle, ce qui n'est pas intéressant...
- Redéfinition = faire en sorte de tester les attributs  
Un processus en plusieurs étapes :
  - ① Vérifier s'il y a égalité référentielle :
    - si true renvoyer true
  - ② Vérifier le type de l'Object o (cf prochain cours)
  - ③ Convertir l'Object o dans le type de la classe (idem)
  - ④ Vérifier l'égalité entre attributs

```
1     public boolean equals(Object obj) {  
2         if (this == obj) return true;  
3         if (obj == null) return false;  
4         if (getClass() != obj.getClass()) return false;  
5         Point other = (Point) obj;  
6         if (x != other.x) return false;  
7         if (y != other.y) return false;  
8         return true;  
9     }
```

# REDÉFINITION ET CO-VARIANCE DU RÉSULTAT

## [E. CHAILLOUX]

Depuis la version 1.5, il est possible dans le cadre d'une redéfinition de préciser le type de retour. Le type du résultat de la méthode redéfinie est en relation de sous-typage avec celui défini dans la sur-classe. On parle de co-variance du type résultat.

### Exemple :

- dans Object

```
1 protected Object clone(){...}
```

- dans Point

```
1 // pour éviter les cast
2 protected Point clone(){return new Point(...);}
```

- dans PointNomme

```
1 protected PointNomme clone(){return new PointNomme(...);}
```

NB : il s'agit d'une règle générale mais je ne connais qu'une application...

# 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 Object

```
1 protected Object clone(){...}
```

- dans Point

```
1 // pour éviter les cast
2 protected Point clone(){return new Point(...);}
```

- dans PointNomme

```
1 // ouverture de la redéfinition
2 public PointNomme clone(){return new PointNomme(...);}
3 // private PointNomme clone(){return new PointNomme(...);}
4 // Ne compile pas
```