2I002
HÉRITAGE
SURCHARGE / REDÉFINITIONS

Vincent Guigue & Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ
LIP6
MLIA Machine Learning & Deep Learning for Information Access

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`

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 ////////////////////////////////////////////
11
12 Point p = new Point(1,2);
13 p.move(3, 1);  // preselection de 1
14 p.move(3, 1, 0.5); // preselection 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  }
```

Définition

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

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

- RAS à la compilation:

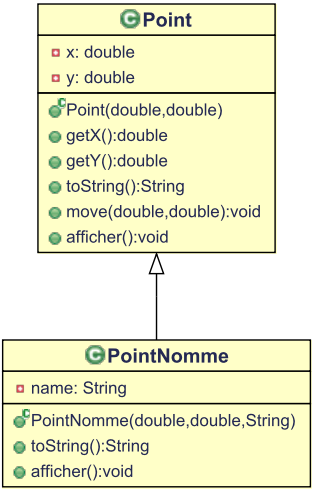
```
1 Point p       = new Point(1, 3);
2 PointNomme pn = new PointNomme(1, 2, "toto");
3 Point p2      = new PointNomme(1, 3, "titi");
```

```
4 p.afficher();
5 pn.afficher();
6 p2.afficher();
```

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

Exemples de fonctionnement - Cas 1/3 (facile)

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



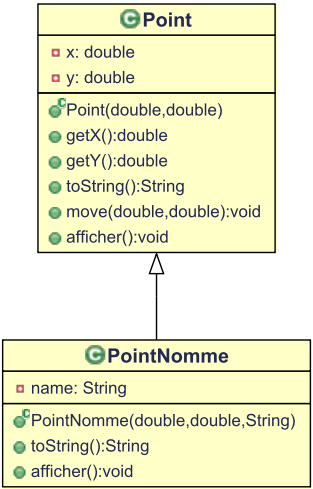
Dans la classe Point, une seule méthode correspond à la signature `afficher()`

Affichage de :

```
Je suis un Point
```

Exemples de fonctionnement - Cas 2/3 (résolution d'une ambiguité)

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



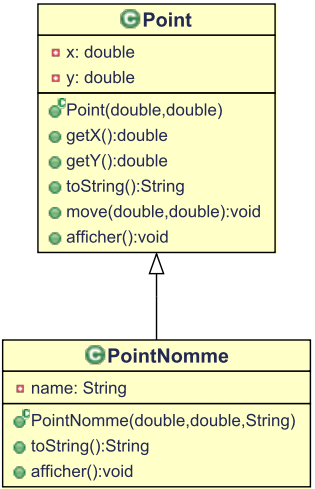
Dans la classe PointNomme, deux méthodes correspondent à la signature `afficher()` (méthodes accessibles: dans la classe et dans les classes parentes)

- 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 - Cas 2/3 (résolution d'une ambiguité)

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

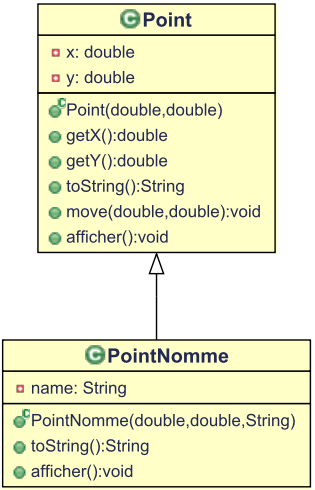


Affichage de :

```
Je suis un PointNomme
```

Exemples de fonctionnement - Cas 3/3 Surcharge + subsomption

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



Compilation: (javac) = type des variables uniquement

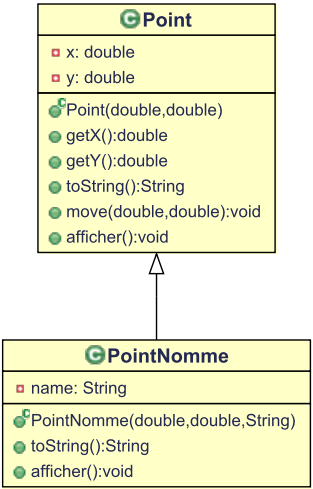
- `p = Point`
- `void afficher()` existe dans `Point` ⇒ OK

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

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

Exemples de fonctionnement - Cas 3/3 Surcharge + subsomption

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



Affichage de :

```
Je suis un PointNomme
```

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␣coordonnees␣:␣"+ super.getX()+ "␣" + super.getY());
5   }
6 }
```

- 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=" + 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());
```

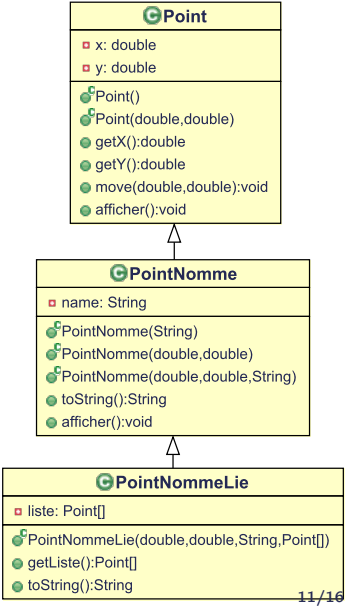
2. Que se passerait-il si on oublie le `super`?

Usage de super 3/3

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

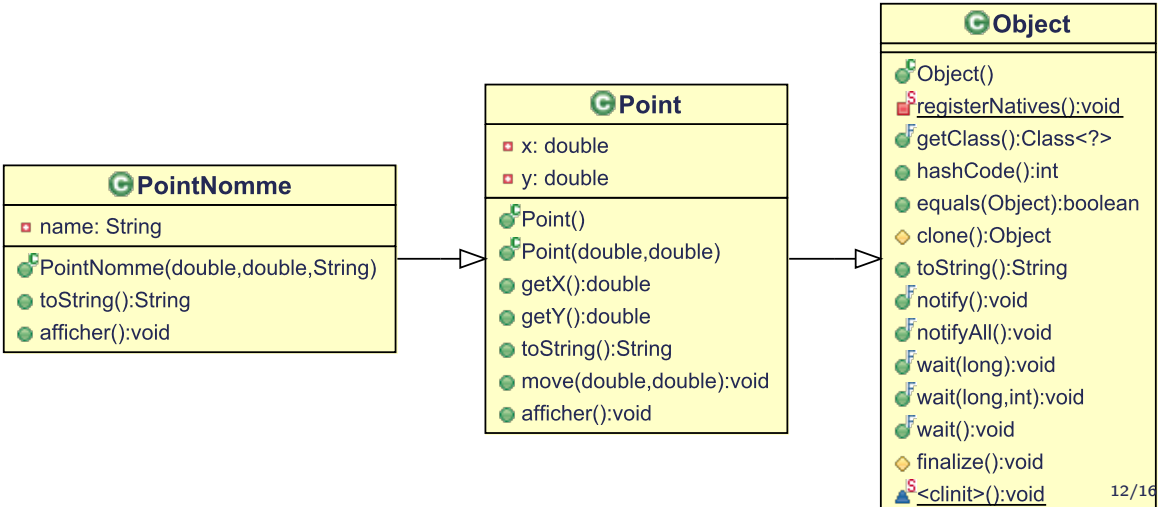
```
1  // dans PointNommeLie
2
3  toString(); // OK local
4  super.toString(); // OK super-classe
5  super.super.toString(); // syntaxe interdite
6
7  getX(); // OK, existe ici par
8          // heritage de Point
9  super.getX(); // OK existe dans PointNomme
10         // par heritage de Point
```

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



Classe standard

Toutes les classes dérivent de la classe Object de JAVA



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

```java
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;
    Point other = (Point) obj;
    if (x != other.x) return false;
    if (y != other.y) return false;
    return true;
}
```

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 définie dans la sur-classe. On parle de co-variance du type résultat.

Exemple:

- dans `Object`

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

- dans `Point`

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

- dans `PointNomme`

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

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 eviter les cast
2   protected Point clone(){return new Point(...);}
```

- dans `PointNomme`

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