LU2IN002 - Introduction à la programmation orientée-objet

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

Cours de ce vendredi : Sabrina Tollari
(email: Sabrina.Tollari@lip6.fr)

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

SCIENCES SORBONNE UNIVERSITÉ

Cours 6 – vendredi 21 octobre 2022

Plan du cours

Programme du jour

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

$\Longrightarrow$ 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 ?

```java
public class Point {
   private double x, y;
   public Point(double x, double y) {
      this.x = x; this.y = y;
   }
   public void move(double x, double y) {
      this.x = x; this.y = y;
   }
   public String toString() {
      return "("+x+","+y+")";
   }
}
```


Héritage : syntaxe : mot clef extends

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

```java
public class PointNomme extends Point {
  private String name;

  public PointNomme(double x, double y, String name) {
      super(x, y);
      this.name = name;
  }
  public String toString() {
    return "PointNomme " + name + " " + super.toString() ;
  }
}
```

- 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

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

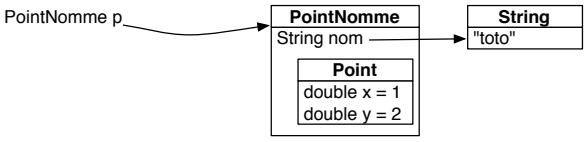
```
7 pnA.move(7,8); // méthode de Point
8 System.out.println(pnA); // PointNomme A (7.0,8.0)
```


Héritage et représentation mémoire

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

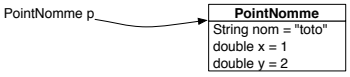
- Un objet PointNomme "englobe" un objet Point

```
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



Remarques / vocabulaire

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

Mot clef super

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 PointNommme(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 :

```
public class Point {
    private double x,y;
    public Point(){
        x=0; y=0;
    }
    ...
```

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

```
public class PointNomme extends Point {
    private String name;
    public PointNomme(String name) {
        super(); // optionnel
        this.name = name;
    }
    ...
```


Héritage et constructeurs : exemple

```
public class Animal {
  private String nom;
  public Animal(String nom) {
     this.nom = nom;
  }
  public String getNom() {
     return nom;
  }
}
public class Poule
              extends Animal {
  private static int cpt = 0;
  public Poule() {
     super("Poule"+(++cpt));
  }
}
public class Renard
              extends Animal {
  private String couleur;
  public Renard(String nom,
              String couleur) {
     super(nom);
     this.couleur=couleur;
  }
```

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

```
Poule p1=new Poule();
Poule p2=new Poule();
Renard r=new Renard("Rox","roux");
System.out.println(p1.getNom());
System.out.println(p2.getNom());
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

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

Diagramme de classe détaillé

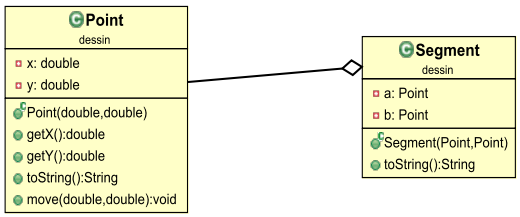
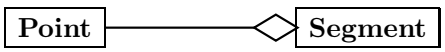


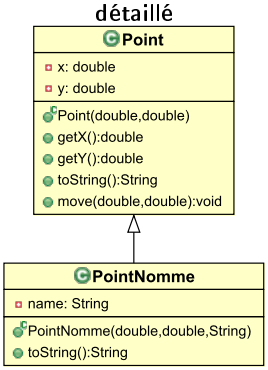
Diagramme de classe simplifié



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.

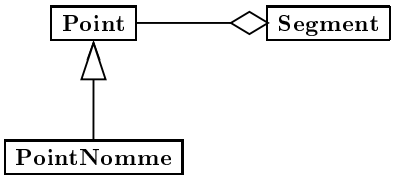
Diagramme de classe détaillé




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

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

Diagramme de classe simplifié

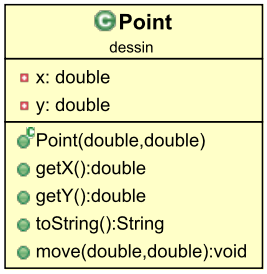


Rappel : UML Client *vs* Fournisseur

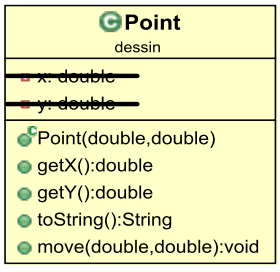
Plusieurs types de diagrammes pour plusieurs usages :

- **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

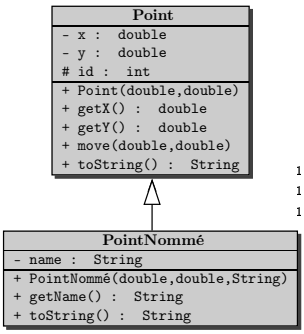
Nouveau niveau d'accès : `protected`

- `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




```java
public class Point {
  private double x,y;
  protected final int id;
  ...
}
public class PointNomme extends Point {
  ...
  public void methode(double d){
    int toto = id; // ou super.id;
    ...
  }
}
```

Variable/Méthode protected

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

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

Héritage : propriétés

Si B hérite de A, implicitement, B hérite :

- des **méthodes publiques** 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 contructeurs **publics, privés ou protégés** de A

Plan du cours

Subsomption

Si la classe B hérite de la classe A :

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

```java
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

```java
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

Subsomption

- Un `PointNomme` EST UN `Point`

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

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

```java
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

```java
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

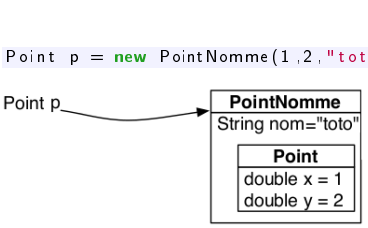
```java
Point p = new PointNomme(1,2,"toto"); // subsomption
// Soit getName() méthode de PointNomme
p.getName(); // -> ERREUR compilation
             // pas accessible à partir d'une variable Point
             // un Point n'est pas un PointNomme
```

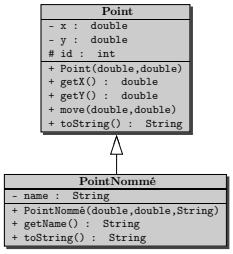
Role du compilateur :

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

Subsomption : visions compilateurs *vs* JVM

```java
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.

Subsomption : arguments de méthode

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

★ Et pour les arguments de méthodes ?

```
public class Truc {
    public void maMethode(Point p){
        ...
    }
}
// main
Truc t = new Truc();
t.maMethode(new Point(1,2));
t.maMethode(new PointNomme(1,2, "toto")); // OK
t.maMethode(new ClasseHeritantDePoint()); // OK
```

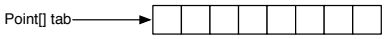
Idée :

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 !

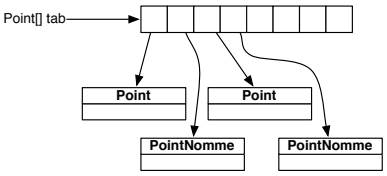
Polymorphisme : application aux tableaux

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

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




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

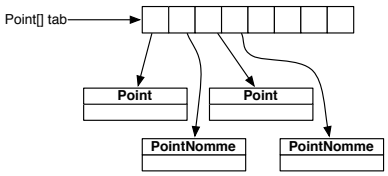


Polymorphisme : application aux tableaux

★ Peut-on bouger tous les points du tableau ?

Oui, car la méthode `move` :

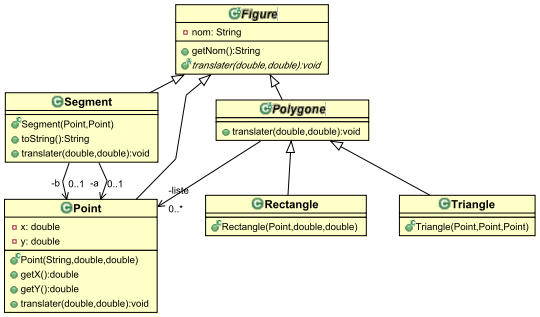
- est dans `Point`
- est héritée par `PointNomme`




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

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


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

Limites du polymorphisme

△ 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`

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

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

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

On l'étudiera plus tard...

Role du compilateur :

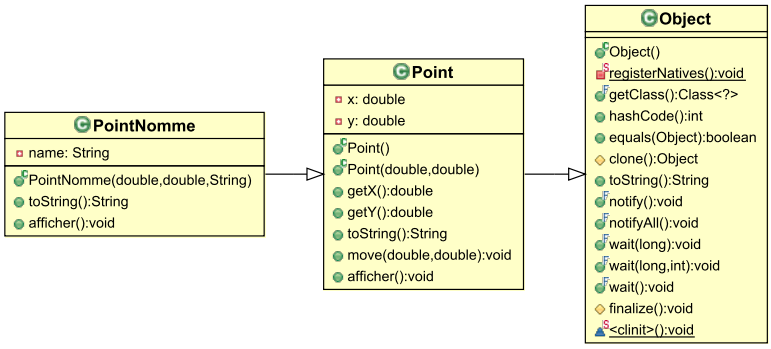
il vérifie le type des variables est les possibilités offertes par celles-ci.

La classe `Object`

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

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

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

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

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

```
Object[] tab = new Object[10];
tab[0] = "toto";
tab[1] = 10; // --> conversion implicite en Integer
tab[2] = new Point(1,2);
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**

Plan du cours

Surcharge

Définition

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

```
public class Point {
...
    public void move(double dx, double dy){
        x+=dx; y+=dy;
    }
    public void move(double dx, double dy, double scale){
        x+=dx*scale; y+=dy*scale;
    }
    public void move(int dx, int dy){
        x+=dx; y+=dy;
    }
    public void move(Point p){
        x+=p.x; y+=p.y;
    }
```


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

```
public class Point {
...
    public void move(double dx, double dy){ // 1
        x+=dx; y+=dy;
    }
    public void move(double dx, double dy, double scale){ // 2
        x+=dx*scale; y+=dy*scale;
    }

Point p = new Point(1,2);
p.move(3, 1); // présélection de 1
p.move(3, 1, 0.5); // présélection de 2
```


Surcharge : les limites

- Interdiction d'avoir des signatures identiques

```
    public void move(double dx, double dy){
        x+=dx; y+=dy;
    }

    // Meme signature pour le compilateur
    //      -> ERREUR de compilation
    public void move(double a, double b){
        x+=a; y+=b;
    }
```

- Pas de prise en compte du type de retour

```
    // Meme signature pour le compilateur
    //      -> ERREUR de compilation
    public Point move(double dx, double dy){
        x+=dx; y+=dy;
        return this;
    }
```

Redéfinition

Définition

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

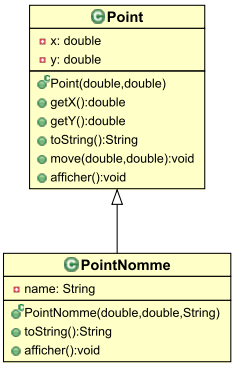
```java
public class Point {
    ...
    public void afficher(){ // 1
        System.out.println("Je suis un Point");
    }
}
////
public class PointNomme extends Point {
    ...
    public void afficher(){ // 2
        System.out.println("Je suis un PointNomme");
    }
}
```

- 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

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



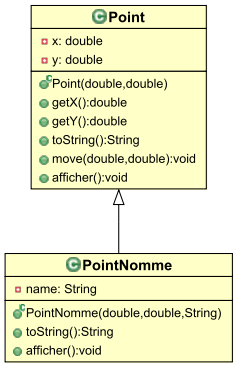
Dans la classe `Point`, une seule méthode correspond à la signature `afficher()`
Affichage de :

```
Je suis un Point
```


Redéfinition : exemples de fonctionnement

Cas 2 : résolution d'une ambiguité

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

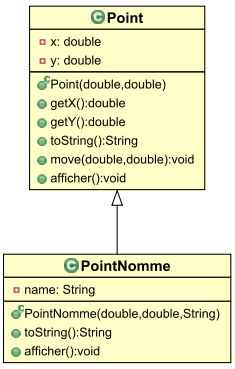


- Dans la classe `PointNomme`, deux méthodes correspondent à la signature `afficher()` :
  - une dans `Point`
  - une dans `PointNomme`
- La JVM choisit, au moment de l'exécution du programme en fonction du type de l'instance de `pn`, la méthode la plus proche
- Affichage de :
  `Je suis un PointNomme`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

Redéfinition : exemples de fonctionnement

Cas 3 : Surcharge + **subsomption**

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



- **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

```java
public class PointNomme extends Point {
    ...
    public void afficher(){
        System.out.println("Je suis un PointNomme " +
            "de coordonnées : "+ super.getX()+ " " +
             super.getY());
    }
}
```

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

```java
public class PointNomme extends Point {
    ...
    public void affichageGlobal(){
        afficher(); // -> Je suis un PointNomme
        this.afficher(); // -> Je suis un PointNomme
        super.afficher(); // -> Je suis un Point
    }
}
```


Redéfinition et usage de super 2/3

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

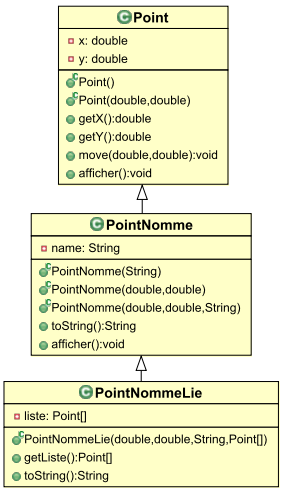
```java
// classe Point
public String toString() {
    return "("+x+","+y+")";
}
// classe PointNomme
public String toString() {
    return "PointNomme " + name + " " + super.toString() ;
}
```

1. Quels sont les affichages en sortie du code suivant :

```java
Point p = new Point(1,2);
PointNomme pnA = new PointNomme(3,4,"A");
Point pPnB = new PointNomme(5,6,"B");
System.out.println(p.toString());
// (1,2)
System.out.println(pnA.toString());
// PointNomme A (3,4)
System.out.println(pPnB.toString());
// PointNomme B (5,6) // toString() de l'objet, pas de la variable
```

2. 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
```