NFP121 — Programmation Avancée
—
Héritage – Classes abstraites – Réutilisation

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Télécommunications & Réseaux

Sommaire

Sommaire

Exemple introductif

Exercice 1 : Définition d'un point nommé

Un point nommé est un point, caractérisé par une **abscisse** et une **ordonnée**, qui possède également un **nom**. Un point nommé peut être **translaté** en précisant un déplacement suivant à l'axe des X (abscisses) et un déplacement suivant l'axe des Y (ordonnées). On peut obtenir sa **distance** par rapport à un autre point. Il est possible de **modifier** son abscisse, son ordonnée ou son nom. Enfin, ses caractéristiques peuvent être **affichées**.

1.1 Modéliser en UML cette notion de point nommé.

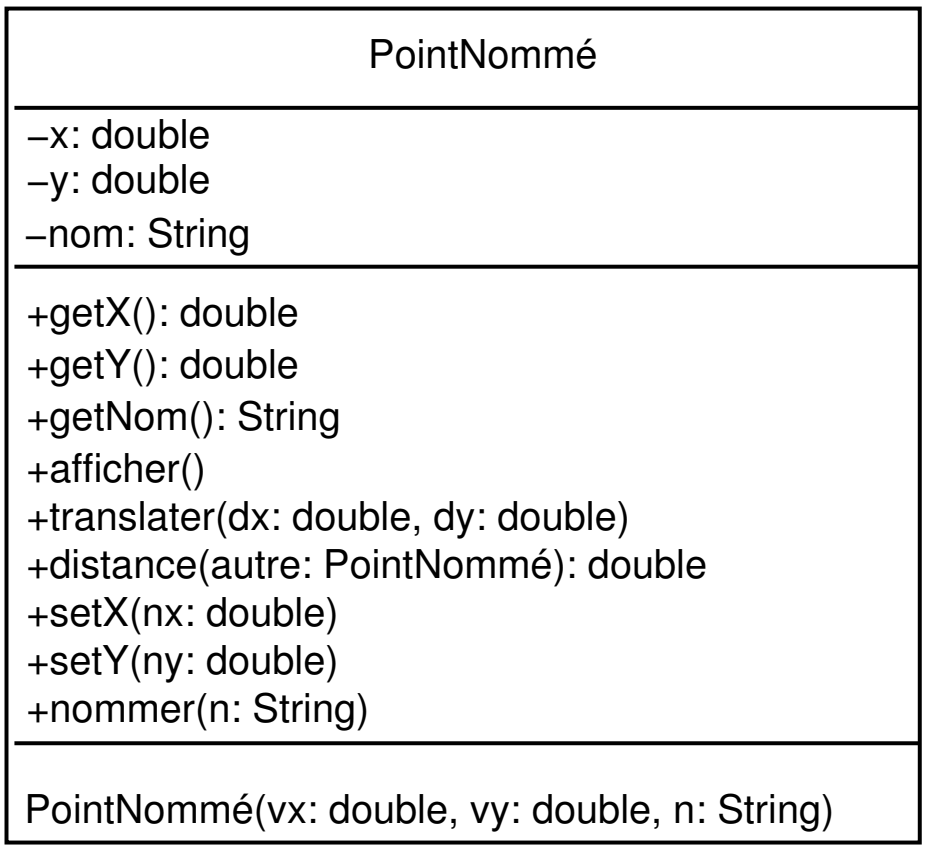
1.2 Une classe Point a déjà été écrite. Que constatez-vous quand vous comparer le point nommé et la classe Point des transparents suivants ?

1.3 Comment écrire la classe PointNommé ?

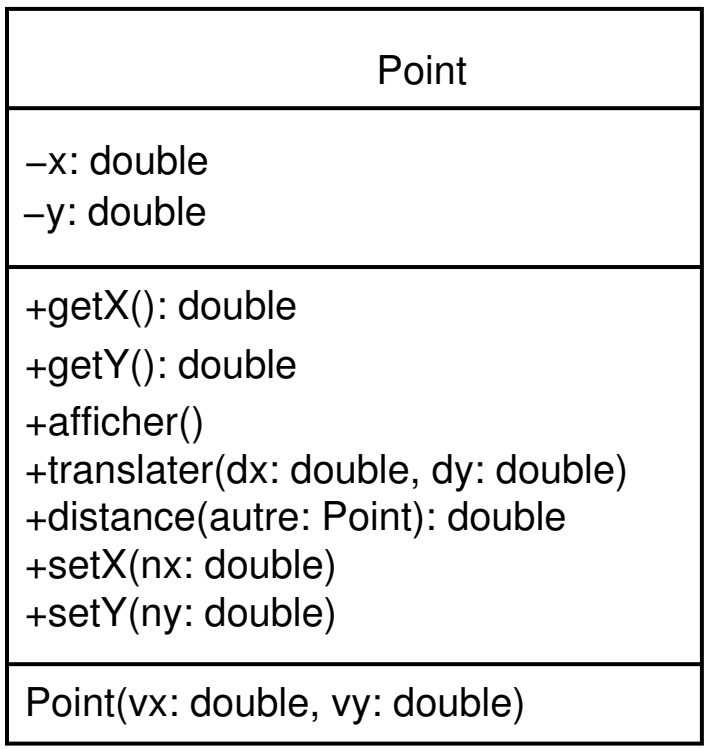
Modélisation de la classe PointNommé

PointNommé
requêtes
x : double
y : double
nom : String
distance(autre : PointNommé) : double
commandes
afficher
translater(dx : double, dy : double)
setX(nx : double)
setY(ny : double)
nommer(n : String)

Une solution : Écrire directement la classe PointNommé



La classe Point existe !



La classe Point

```
1  public class Point {    // commentaires volontairement omis !
2      private double x;           // abscisse
3      private double y;           // ordonnée
4
5      public Point(double vx, double vy) { this.x = vx; this.y = vy; }
6
7      public double getX()               { return this.x; }
8      public double getY()               { return this.y; }
9      public void setX(double vx)        { this.x = vx; }
10     public void setY(double vy)        { this.y = vy; }
11
12     public void afficher() {
13         System.out.print("(" + this.x + "," + this.y + ")");
14     }
15     public double distance(Point autre) {
16         double dx2  = Math.pow(autre.x - this.x, 2;
17         double dy2  = Math.pow(autre.y - this.y, 2;
18         return Math.sqrt(dx2 + dy2);
19     }
20     public void translater(double dx, double dy) {
21         this.x += dx;
22         this.y += dy;
23     }
24     ...
25 }
```

Solution 1 : Copier les fichiers

Il suffit (!) de suivre les étapes suivantes :

1. Copier Point.java dans PointNommé.java

```
cp Point.java PointNommé.java
```

2. Remplacer toutes les occurrences de Point par PointNommé.
   Par exemple sous vi, on peut faire :

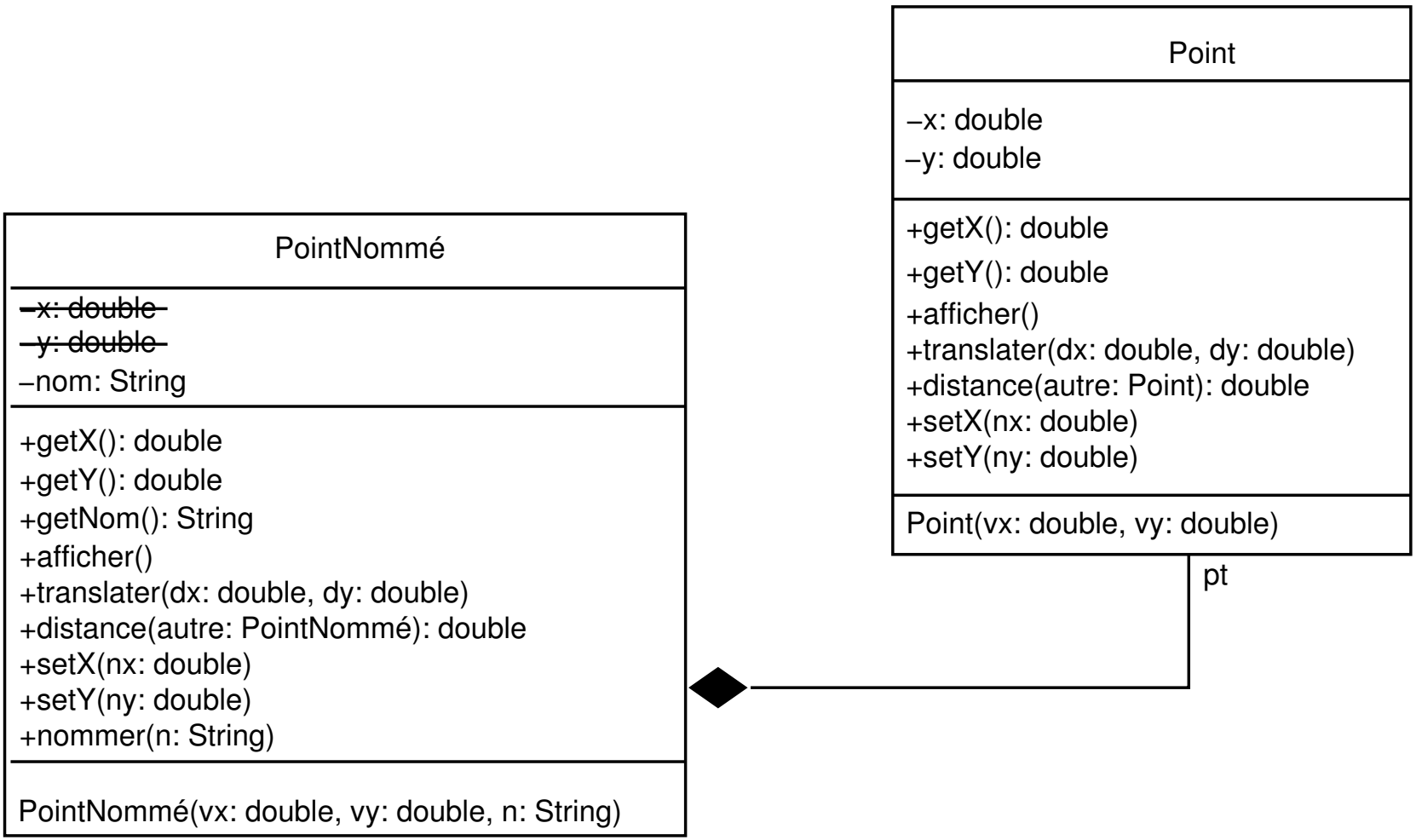
```
:%s/\<Point\>/PointNommé/g
```

3. Ajouter les attributs et méthodes qui manquent (`nom` et `nommer`)
4. Adapter le constructeur

Beaucoup de problèmes :

- Que faire si on se rend compte que le calcul de la distance est faux ?
- Comment ajouter une coordonnée z ?
- Peut-on calculer la distance entre un point et un point nommé ?

Solution 2 : Utiliser le point (composition)



Le code Java correspondant

```
1  public class PointNomme {       // commentaires volontairement omis !
2      private Point pt;
3      private String nom;
4
5      public PointNomme(double vx, double vy, String nom) {
6          this.pt = new Point(vx, vy);
7          this.nom = nom;
8      }
9
10     public double getX()        { return this.pt.getX(); }
11     public double getY()        { return this.pt.getY(); }
12     public String getNom()      { return this.nom; }
13
14     public void translater(double dx, double dy) {
15         this.pt.translater(dx, dy);
16     }
17
18     public void afficher()      {       // adaptée
19         System.out.print(this.nom + ":");
20         this.pt.afficher();
21     }
22
23     public double distance(PointNomme autre) {
24         return this.pt.distance(autre.pt);
25     }
26     ...
27 }
```

Discussion sur la solution 2 (utilisation)

Le code de la classe Point est **réutilisé** dans PointNommé

$\Longrightarrow$ Pas de code dupliqué !

Il faut **définir toutes les méthodes** dans PointNommé :

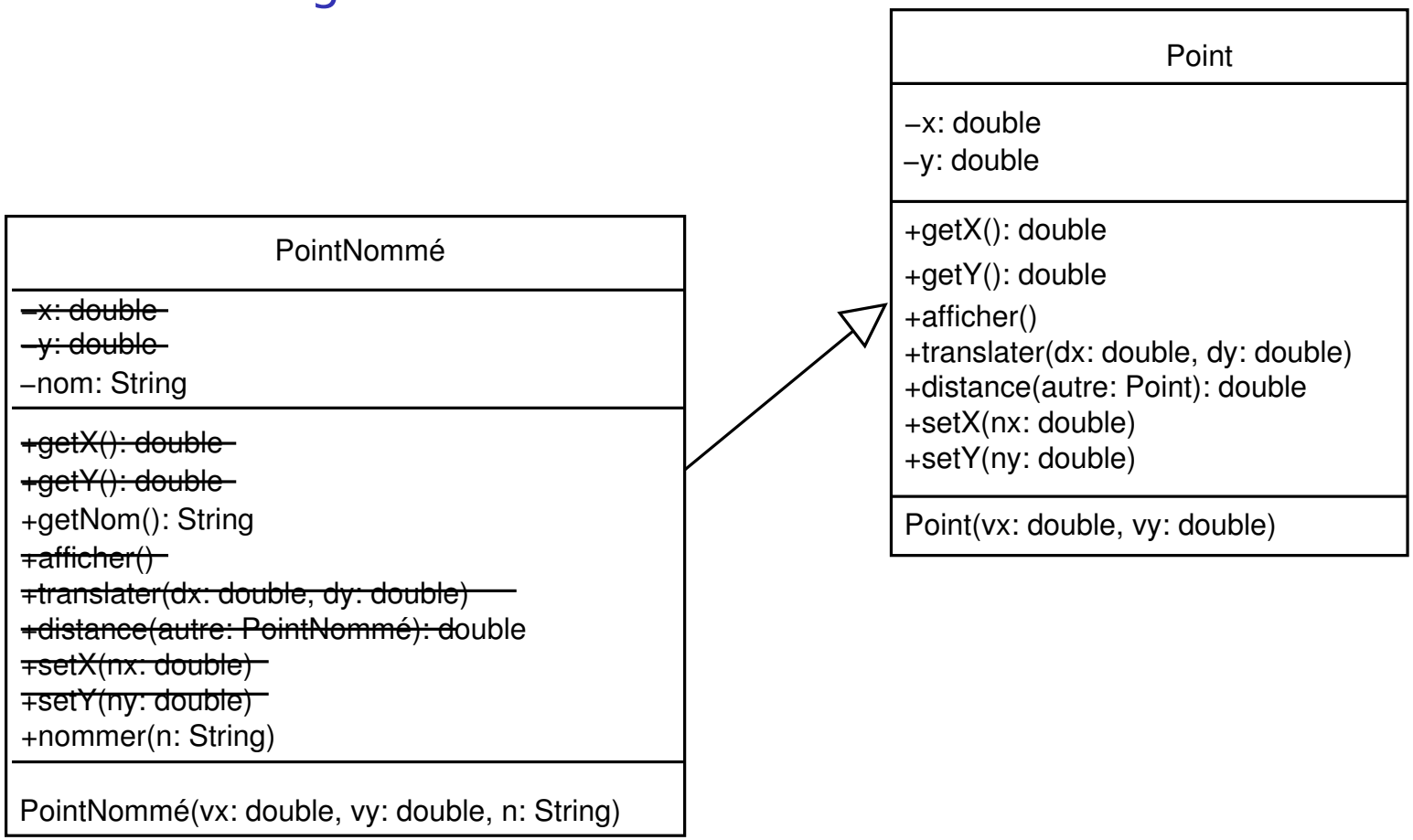
- les nouvelles méthodes sont codées dans PointNommé (`getNom`, `nommer`)
- celles présentes dans point sont appliquées sur l'attribut `pt`
- il est possible de les adapter (par exemple `afficher` est adaptée)

Toujours impossible de calculer la distance entre Point et PointNommé.

- La surcharge est une solution mais elle oblige à modifier la classe Point !

Remarque : On constate qu'un point nommé est un point particulier.
Un point nommé **est** un point... qui a un nom.

Solution 3 : L'héritage



En Java, hériter c'est :

1. **définir un sous-type** : PointNommé est un sous-type de Point
2. récupérer dans PointNommé (sous-classe) les éléments de Point (super-classe)

La relation d'héritage

Buts : plusieurs objectifs (non liés) :

- définir une nouvelle classe :
  - comme **spécialisation** d'une classe existante ;
  - comme **généralisation** de classes existantes (factorisation des propriétés et comportements communs) ;
- définir une **relation de sous-typage** entre classes (substitutionalité) ;
- copier virtuellement les caractéristiques de classes existantes dans la définition d'une nouvelle classe (héritage).

Moyen :

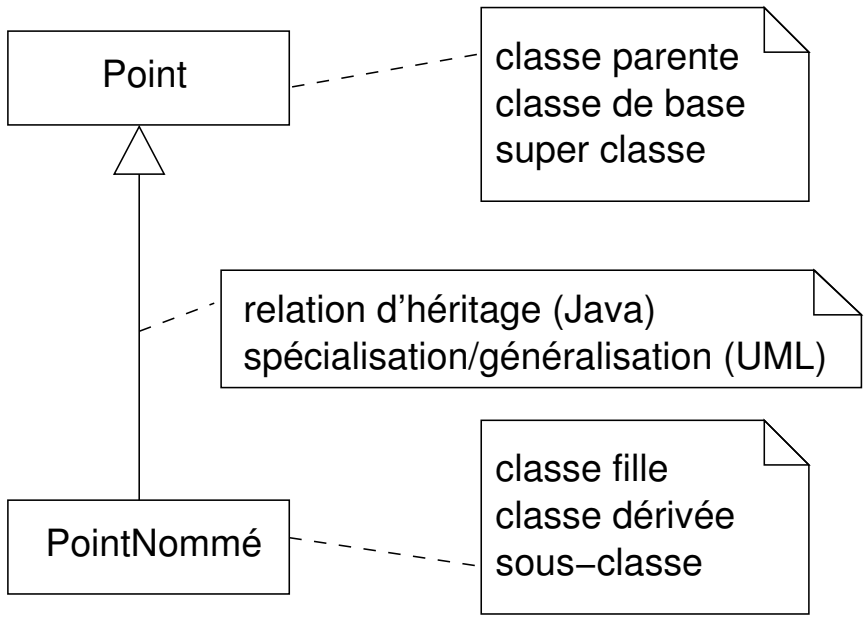
- La relation d'héritage en Java
- Appelée relation de généralisation/spécialisation en UML

Exemples :

- PointNommé est une spécialisation de Point (un point avec un nom).
- Point est une généralisation de PointNommé, PointPondéré, PointColoré...

Notation et vocabulaire

Notation UML



Notation en Java

```
public class PointNommé
    extends Point                    // relation d'héritage
{ ... }
```

Vocabulaire : On parle d'ancêtres et de descendants (transitivité de la relation d'héritage)

La classe PointNommé (version initiale)

```
1  /** Un point nommé est un point avec un nom.  Ce nom peut être changé. */
2  public class PointNomme
3      extends Point       // héritage (spécialisation d'un point)
4  {
5    private String nom;
6
7    /** Initialiser à partir de son nom et de ses coordonnées cartésiennes */
8    public PointNomme(String sonNom, double vx, double vy) {
9        super(vx, vy);  // appel au constructeur de Point (1ère instruction)
10       this.nom = sonNom;
11   }
12   /** Le nom du point nommé */
13   public String getNom() {
14       return this.nom;
15   }
16   /** changer le nom du point */
17   public void nommer(String nouveauNom) {
18       this.nom = nouveauNom;
19   }
20 }
```

Remarque : L'héritage évite de faire du copier/coller (toujours une mauvaise idée).

Enrichissement

Dans la sous-classe, on peut ajouter :

- de nouveaux attributs (conseil : prendre de nouveaux noms !) ;

```
private String nom;
```

- de nouvelles méthodes.

```
public String getNom() { ... }
public void nommer(String nouveauNom) { ... }
```

Remarque : Ajouter une nouvelle méthode s'entend au sens de la *surcharge*.
Par exemple, sur le PointNommé on peut ajouter :

```
public void afficher(String préfixe) {
    System.out.println(préfixe);        // afficher le préfixe
    this.afficher();                    // utiliser le afficher « classique »
}
```

qui surcharge la méthode `afficher()` de Point.

Héritage et constructeurs

Règle : Tout constructeur d'une sous-classe doit **obligatoirement** appeler un des constructeurs de la super-classe.

Justification : Hériter d'une classe, c'est récupérer son état qui doit donc être initialisé. Dans un `PointNommé`, il y a un `Point`... et un nom.

En pratique : On utilise **super** suivi des paramètres effectifs permettant au compilateur de sélectionner un constructeur de la super-classe.

```
public PointNomme(String sonNom, double vx, double vy) {
   super(vx, vy); // appel au constructeur de Point (1ère instruction)
   this.nom = sonNom;
}
```

- L'appel à **super** doit être la première instruction du constructeur !
- La super-classe est **toujours initialisée avant** la sous-classe.
- Si aucun appel à **super** n'est fait, le compilateur appelle automatiquement le constructeur par défaut de la super-classe : **super**().
  $\Longrightarrow$ D'où le danger de définir un constructeur par défaut !

Utilisation de la classe PointNommé

Exemple d'utilisation la classe PointNommé :

```
1  PointNomme pn = new PointNommé("A", 1, 2,);
2          // créer un point "A" de coordonnées (1,2)
3  pn.afficher();                  // (1,2)  méthode de Point
4  pn.translater(-1, 2);           //        méthode de Point
5  pn.afficher();                  // (0,4)  méthode de Point
6  pn.nommer("B");                 // méthode de PointNomme
7  String n = pn.getNom();         // méthode de PointNomme
8  pn.afficher("Le point est");    // méthode de PointNomme (surcharge)
9          //  Le point est (0, 4)
```

- La classe PointNommé a bien **hérité** de toutes les caractéristiques de Point.
- **Exercice 2** Lister toutes les caractéristiques de la classe PointNommé. Préciser les droits d'accès (voir T. 33).
- `afficher()` n'affiche pas le nom du PointNommé.

Redéfinition de méthode

Redéfinition : La sous-classe peut donner une nouvelle **version** (nouveau corps) à une méthode définie dans la super-classe (**adaptation** du comportement).

Exemple : PointNommé peut (doit !) **redéfinir** « afficher » pour afficher aussi le nom du point.

Remarque : Les deux méthodes correspondent à la même opération `afficher()`.

```
/** Afficher le point nommé sous la forme nom:(x,y) */
@Override public void afficher() {
    System.out.print(getNom() + ":");
    super.afficher();   // utilisation du afficher de Point
}
```

super : appeler la méthode de la super classe !

```
pn.afficher();          // A:(0,4)  méthode de PointNommé !
```

Attention : Un client (ou une sous-classe) n'aura accès qu'à la redéfinition.
La version de la superclasse n'est plus accessible !

Ne pas confondre surcharge et redéfinition :

- **surcharge :** deux méthodes différentes qui ont le même nom (et donc des signatures différentes)
- **redéfinition :** méthode de la super-classe (re)définie dans la sous-classe (mêmes signatures !)

L'annotation `@Override`

Principe : `@Override` exprime l'intention du programmeur de redéfinir une méthode de la superclasse.

Intérêt : Le compilateur vérifie qu'il s'agit bien d'une redéfinition.

```
1  class A {
2      void uneMethodeAvecUnLongNom(int a) {}
3  }
4  class B1 extends A {
5      @Override
6      void uneMethodeAvecUnLongNom(Integer a) {}
7  }
8  class B2 extends A {
9      @Override
10     void uneMethodeAvecUnLongNon(int a) {}
11 }
12 class B3 extends A {
13     @Override
14     void uneMethodeAvecUnLongNom(int a) {}
15 }
```

Question : Quels problèmes sont signalés par le compilateur ?

L'annotation `@Override` (2)

```
1  UtilisationOverride.java:5: error: method does not override or implement a method
       from a supertype
2      @Override
3      ^
4  UtilisationOverride.java:9: error: method does not override or implement a method
       from a supertype
5      @Override
6      ^
7  2 errors
```

Conseil : Toujours utiliser l'annotation `@Override` quand vous (re)définissez une méthode de manière à rendre votre intention explicite pour le compilateur et les lecteurs !

Attention : Les annotations ont été introduites en Java 5.

Conseil : Mettre aussi `@Override` devant la définition, dans une réalisation, des méthodes d'une interface.

Principe de substitution

Principe de substitution : L'instance d'un descendant peut être utilisée partout là où un ancêtre est déclaré.

Justification intuitive : Tout ce qui peut être demandé à la super-classe peut aussi l'être à la sous-classe.

Attention : L'inverse est faux. L'instance d'un ancêtre ne peut pas être utilisée où un descendant est déclaré.

```
1  Point p1 = new Point(3, 4);
2  PointNomme pn1 = new PointNomme("A", 30, 40);
3  Point q;         // poignée sur un Point
4  q = p1;  q.afficher();  // ?????
5  q = pn1; q.afficher();  // ?????
6
7  PointNomme qn;  // poignée sur un PointNommé
8  qn = p1;  qn.afficher(); // ?????
9  qn = pn1; qn.afficher(); // ?????
```

Remarque : Le principe de substitution est vérifié à la compilation.

Résolution d'un appel polymorphe

```
T p;              // Déclaration de la poignée (type apparent : T)
p = new X(...); // Affectation de la poignée (type réel : X)
...
p.m(a1, ..., an);        // Appel de la méthode m(...) sur p
```

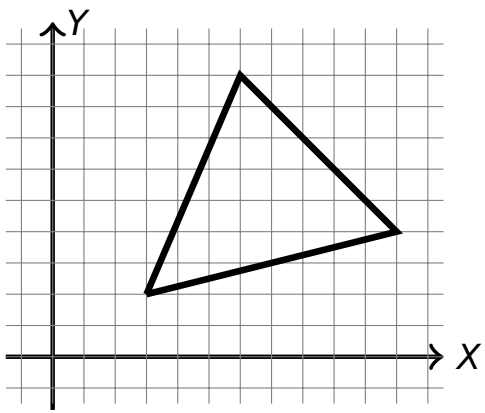
1. **Résolution de la surcharge** (*liaison statique*).
   - **But :** Identification de la signature de la méthode à exécuter.
   - La classe du type apparent de la poignée (classe T) doit avoir une méthode m(T_1, ..., T_n) dont les paramètres correspondent en nombre et en type aux paramètres effectifs a_1, ..., a_n.
   - **Si** pas exactement une signature trouvée **Alors** *erreur de compilation* !
   - **Remarque :** Le principe de substitution et les conversions automatiques sont utilisés sur les paramètres !
2. **Liaison dynamique** (à l'exécution, généralement).
   - **But :** Le système choisit la **version** de m(T_1, ..., T_n) à exécuter : *celle qui est dans la classe X*.
   - C'est la dernière (re)définition rencontrée partant du type T et descendant vers le type réel de l'objet attaché à la poignée p (X).

Exercice : intérêt de la liaison dynamique

```
1  public class ExempleSchema1 {
2     public static void main(String[] args) {
3        // Créer les trois segments
4        Point p1 = new Point(3, 2);
5        Point p2 = new Point(6, 9);
6        Point p3 = new Point(11, 4);
7        Segment s12 = new Segment(p1, p2);
8        Segment s23 = new Segment(p2, p3);
9        Segment s31 = new Segment(p3, p1);
10
11       // Créer le barycentre
12       double sx = p1.getX() + p2.getX() + p3.getX();
13       double sy = p1.getY() + p2.getY() + p3.getY();
14       Point barycentre = new Point(sx / 3, sy / 3);
15
16       // Afficher le schéma
17       System.out.println("Le schéma est composé de : ");
18       s12.afficher();      System.out.println();
19       s23.afficher();      System.out.println();
20       s31.afficher();      System.out.println();
21       barycentre.afficher();  System.out.println();
22    }
23 }
```

Résultat de l'exécution :

```
Le schéma est composé de :
[(3.0, 2.0)-(6.0, 9.0)]
[(6.0, 9.0)-(11.0, 4.0)]
[(11.0, 4.0)-(3.0, 2.0)]
(6.666666666666667, 5.0)
```



Exercice 3 On souhaite pouvoir nommer certains points du schéma. Par exemple, on veut appeler "A" le point (3, 2).

Liaison tardive : réaliser le principe du choix unique

Intérêt : Éviter des structures de type « choix multiples » :

- Les alternatives sont traitées par le compilateur (sûreté : pas d'oubli).
- L'ajout d'une nouvelle alternative est facilité (extensibilité).

Sans liaison dynamique, on teste explicitement le type d'un objet pour choisir le traitement :

```
Point q = ...; // Afficher les caractéristiques de q
if (q instanceof PointNomme) {          // tester le type réel
    // afficher q comme un PointNomme
} else if (q instanceof PointColoré ) { // tester le type réel
    // afficher q comme un PointColoré
} else if (...) {    ...
} else {                // Ce n'est donc qu'un point !
    // afficher q comme un Point
}
```

Solution : Définir/spécifier une méthode dans le type le plus général et la redéfinir dans les sous-classes !

Question : Que penser du premier commentaire .

Principe du choix unique (B. Meyer) :

> *« Chaque fois qu'un système logiciel doit prendre en compte un ensemble d'alternatives, un et un seul module du système doit en connaître la liste exhaustive ».*

Comment définir une classe par spécialisation ?

1. Vérifier qu'il y a bien sous-typage.
   - Si ce n'est pas le cas, il ne faut pas utiliser l'héritage !
2. Est-ce qu'il y a des méthodes de la super-classe à adapter ?
   - redéfinir les méthodes à adapter !
   - attention, il y a des règles sur la redéfinition (voir programmation par contrat et T. 59)
3. Enrichir la classe des attributs et méthodes supplémentaires.
4. Tester la classe
   - Comme toute classe !
   - Elle doit aussi réussir les programmes de test de sa super-classe.

Comment ajouter un nouveau type de point ?

PointPondéré par exemple

1. Choisir de faire une spécialisation de la classe Point.
   - Il y a bien sous-typage : on veut construire un segment à partir de PointPondéré...
   - Une nouvelle classe à côté des autres
   - On ne risque pas de casser le système
2. Écrire la classe
   - Redéfinir la méthode afficher
   - Ajouter masse, getMasse et setMasse
   - Tester (y compris en tant que Point)
3. Intégrer dans le système en proposant de créer des PointPondérés

Principe du choix unique : Seule la partie de l'application qui s'occupe de la création des points connait les types de points.

Suite du programme du T. 23

```
1  Point p1 = new Point(3, 4);                        //      type    type
2  PointNomme pn1 = new PointNomme("A", 30, 40);      //    apparent  réel
3  Point q;        // poignée sur un Point                 Point   null
4  q = p1;  q.afficher();  // (3,4)                                 Point
5  q = pn1; q.afficher();  // A:(30,40)                             PN
6
7  PointNomme qn;  // poignée sur un PointNommé            PN      null
8  qn = p1;  qn.afficher(); // INTERDIT !                           Point
9  qn = pn1; qn.afficher(); // A:(30,40)                           PN
10
11 qn = q; // Possible ? (Le type réel de q est PointNommé)
12 qn.afficher()   // ????
```

Transtypage et interrogation dynamique de type

```
Point p = new PointNomme("A", 1, 2);
PointNomme q;
q = p;  // Interdit par le compilateur
```

Le compilateur interdit cette affectation car il s'appuie sur les types apparents.

Or, un PointNommé est attaché à p. L'affectation aurait donc un sens.

C'est le pb de l'**affectation renversée** résolu en Java par le « transtypage » :

```
q = (PointNomme) p;     // Autorisé par le compilateur
```

Attention : Ce transtypage est vérifiée à l'exécution (ClassCastException).

Interrogation dynamique de type : opérateur **`instanceof`**

```
if (p instanceof PointNomme) {
  ((PointNomme) p).nommer("B");
}
```

```
if (p instanceof PointNomme) {
  PointNomme q = (PointNomme) p;
  q.nommer("B");
}
```

Le modifieur **`final`**

Définition : Le modifieur **`final`** donne le sens d'immuable, de non modifiable.

Il est utilisé pour :

- une *variable locale* : c'est une constante (la variable ne peut être affectée qu'une seule fois) ;
- un *attribut d'instance ou de classe* (l'attribut ne peut être affecté qu'une seul fois) ;
- une *méthode* : la méthode ne peut pas être redéfinie par une sous-classe. Elle n'est donc pas polymorphe ;
- une *classe* : la classe ne peut pas être spécialisée. Elle n'aura donc aucun descendant (aucune de ses méthodes n'est donc polymorphe).

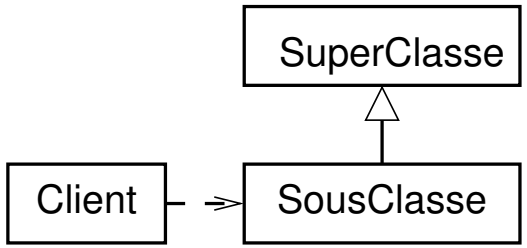
La classe Object

En Java, si une classe n'a pas de classe parente, elle hérite implicitement de la classe Object. C'est l'**ancêtre commun** à toutes les classes.

Elle contient en particulier les méthodes :

- `protected void finalize();`
  - Méthode appelée lorsque le ramasse-miettes récupère la mémoire d'un objet.
- `public String toString();`
  - représentation de l'objet sous forme d'une chaîne de caractère.
  - Elle est utilisée dans print, println et l'opérateur de concaténation + par l'intermédiaire de String.valueOf(Object).
- `public boolean equals(Object obj);`
  - Égalité logique de `this` et `obj`.
  - Attention, l'implantation par défaut utilise == (égalité physique).

Héritage et droits d'accès



Qu'est ce que la sous-classe peut utiliser de la super-classe ?

- tout ce qui est **`public`**
- tout ce qui est **`protected`**, **c'est l'intérêt de `protected`**
- ce qui est droit d'accès paquetage si elle est dans le paquetage de SuperClasse
- ce qui est **`private`** lui est **inaccessible** !

  Conclusion : *Une méthode d'une sous-classe a accès aux caractéristiques de la super-classe déclarées* ***`public`****,* ***`protected`*** *et éventuellement à celles de niveau paquetage.*

Qu'est ce que le client peut utiliser de la sous-classe ? Déja vu !

- ce qui est public
- ce qui est **`protected`** ou *paquetage* si Client dans le paquetage de SousClasse

En particulier, que devient le droit d'accès d'une méthode héritée ?

- par défaut, il reste inchangé
- peut être augmenté par la SousClasse (si elle y accède !)

Héritage et droits d'accès : Exemple

```
1  class SuperClasse {
2      public void m1() { }
3      protected void m2() { }
4      protected void m3() { }
5      private void m4() { }
6  }
7
8  class SousClasse extends SuperClasse {
9      public void m3() {  // droit d'accès augmenté
10         super.m3(); // la méthode m3 dans SuperClasse (voir « redéfinition »)
11     }
12     public void g() {
13         // peut utiliser m1, m2, m3
14         // ne peut pas utiliser m4 (car private)
15     }
16 }
17
18 class Client {
19     SousClasse f;
20     void m() {
21         // peut utiliser f.m1, f.m3 et g
22         // ne peut pas utiliser f.m2 ni f.m4
23     }
24 }
```

Attributs, héritage et liaison dynamique

```
1  public class AttributsEtHeritage {
2      public static void main(String[] args) {
3          A a = new A();
4          B b = new B();
5          A c = new B();
6          System.out.println("a.a1 = "       + a.a1);
7          System.out.println("b.a1 = "       + b.a1);
8          System.out.println("a.a2 = "       + a.a2);
9          System.out.println("b.a2 = "       + b.a2);
10         System.out.println("((A) b).a2 = " + ((A) b).a2);
11         System.out.println("c.a2 = "       + c.a2);
12         System.out.println("((B) c).a2 = " + ((B) c).a2);
13     }
14 }
15
16 class A {
17     int a1 = 1;
18     int a2 = 2;
19 }
20
21 class B extends A {
22     int a2 = 4;
23 }
```

Résultats de l'exécution

```
1  a.a1 = 1
2  b.a1 = 1
3  a.a2 = 2
4  b.a2 = 4
5  ((A) b).a2 = 2
6  c.a2 = 2
7  ((B) c).a2 = 4
```

Règles :

- Il n'y a pas de redéfinition possible des attributs, seulement du masquage.
- L'attribut est sélectionné en fonction du type apparent de l'expression.

Conseil : Éviter de donner à un attribut un nom utilisé dans sa super-classe.

Remarque : Si les attributs sont privés, le problème ne se pose pas !

Sommaire

- Exemple introductif
- Méthode retardée
- Classe abstraite
- Notation UML
- Intérêt des classes abstraites

Exemple introductif

Exercice 4 Étant données les classes Point, PointNommé, Segment, Cercle... on souhaite formaliser la notion de schéma.

4.1 Comment peut-on modéliser cette notion de schéma en Java ?

4.2 Comment faire pour afficher ou translater un tel schéma ?

4.3 Comment faire si on veut ajouter un Polygone ?

4.4 Comment faire si on veut pouvoir agrandir (effet zoom) le schéma ?

Méthode retardée (ou abstraite)

Définition : Une méthode retardée (ou abstraite) est une méthode dont on ne donne pas le code.
Cette méthode est notée **`abstract`** (modifieur).

Exemple : Un objet géométrique peut être affiché et déplacé mais si on ne connaît pas le type d'objet, on ne sait pas écrire le code de ces méthodes

```
/** Afficher les caractéristiques de l'objet géométrique. */
abstract public void afficher();

/** Translater l'objet.
 *  @param dx déplacement suivant l'axe des X
 *  @param dy déplacement suivant l'axe des Y */
abstract public void translater(double dx, double dy);
```

Remarque : Les méthodes d'une interface sont abstraites.

Intérêt : Obliger une sous-classe à définir la méthode (et à donner le code adéquat).

Attention : **`abstract`** est incompatible avec **`final`** ou **`static`** : une méthode retardée est nécessairement une méthode d'instance polymorphe !

Classe abstraite : la classe ObjetGéométrique

```
1  /** Modélisation de la notion d'objet géométrique. */
2  abstract public class ObjetGéométrique {
3      private java.awt.Color couleur;     // couleur de l'objet
4
5      /** Construire un objet géométrique.
6       * @param c la couleur de l'objet géométrique */
7      public ObjetGéométrique(java.awt.Color c)   { this.couleur = c; }
8
9      /** Obtenir la couleur de cet objet.
10      *  @return la couleur de cet objet */
11     public java.awt.Color getCouleur()       { return this.couleur; }
12
13     /** Changer la couleur de cet objet.
14      *  @param c nouvelle couleur */
15     public void setCouleur(java.awt.Color c)    { this.couleur = c; }
16
17     /** Afficher sur le terminal les caractéristiques de l'objet.  */
18     abstract public void afficher();
19
20     /** Translater l'objet géométrique.
21      *  @param dx déplacement en X
22      *  @param dy déplacement en Y */
23     abstract public void translater(double dx, double dy);
24 }
```

Question : Une telle classe est-elle réellement utile ?

Classe abstraite

Classe abstraite : Classe dont on ne peut pas créer d'instance (**`abstract`**)

Règle : Une classe qui possède une méthode retardée (propre ou héritée) est nécessairement une classe abstraite.

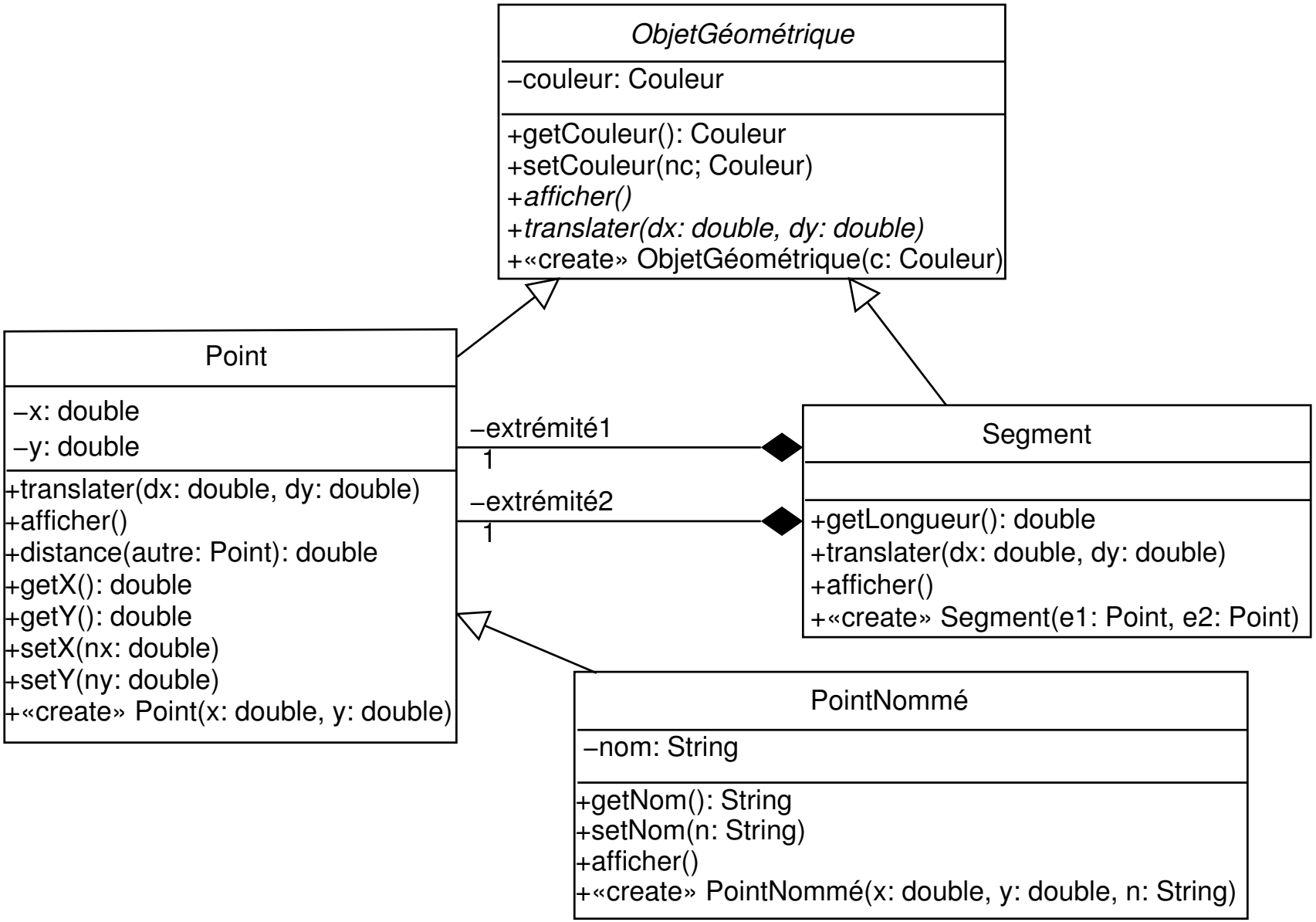
Conséquence : Une classe abstraite ne pouvant pas être instanciée, elle devra donc avoir des descendantes qui définiront les méthodes retardées.

Remarque : Une classe peut être abstraite même si toutes ses méthodes sont définies.

Constructeurs : Une classe abstraite peut avoir des constructeurs et il est recommandé d'en définir (s'ils ont un sens).

Question : Quand ces constructeurs seront exécutés ?

Exercice 5 Quel est l'intérêt d'utiliser une méthode retardée plutôt que de définir une méthode avec un code vide (ou affichant un message d'erreur) qui serait redéfinie dans les sous-classes ?


ObjetGéométrique
–couleur: Couleur
+getCouleur(): Couleur
+setCouleur(nc; Couleur)
+afficher()
+translater(dx: double, dy: double)
+«create» ObjetGéométrique(c: Couleur)
Point
–x: double
–y: double
+translater(dx: double, dy: double)
+afficher()
+distance(autre: Point): double
+getX(): double
+getY(): double
+setX(nx: double)
+setY(ny: double)
+«create» Point(x: double, y: double)
–extrémité1
1
–extrémité2
1
Segment
+getLongueur(): double
+translater(dx: double, dy: double)
+afficher()
+«create» Segment(e1: Point, e2: Point)
PointNommé
–nom: String
+getNom(): String
+setNom(n: String)
+afficher()
+«create» PointNommé(x: double, y: double, n: String)

Commentaires sur la notation UML

- Essayer de placer les classes parentes au-dessus des sous-classes.
- Ne pas confondre la relation de généralisation/spécialisation avec une relation d'association avec sens de navigation !
- Le nom des classes abstraites et des méthodes retardées sont notés en italique... Ce qui n'est pas toujours très facile/visible !
  Remarque : On peut également utiliser la contrainte `{abstract}`.
- Dans une sous-classe UML, on ne fait apparaître que les méthodes qui sont définies ou redéfinies dans cette sous-classe.
  - `Point` définit `afficher` et `translater` qui étaient spécifiées dans `ObjetGéométrique`.
  - Dans `PointNommé` on ne fait apparaître ni `translater` ni `distance` car ce sont celles de `Point`. En revanche, `afficher` est redéfinie.
- La relation entre `Segment` et `Point` est une relation de composition.
  Question : Comment la réaliser en Java ?

Intérêt des classes abstraites

- **Factoriser** le comportement de plusieurs classes même si on n'est pas capable de le coder complètement. Ce comportement peut alors être utilisé.

  Exercice 6 Définir une classe Groupe (d'objets géométriques).
  - Groupe est-elle abstraite ?
  - Peut-on mettre un Groupe dans un Groupe ? Comment faire ?

- **Classifier** les objets, par exemple les objets géométriques généraux, fermés, ouverts, etc.
- Permettre au compilateur de **vérifier** que la définition des méthodes retardées héritées est bien donnée dans les sous-classes.

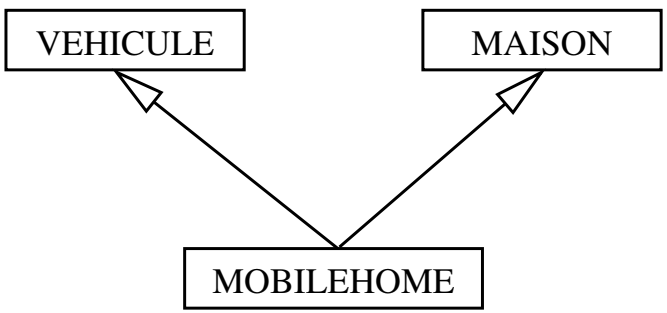
Remarque : Une sous-classe d'une classe abstraite peut être concrète ou abstraite suivant qu'elle donne ou non une définition à toutes ses méthodes (propres et héritées).

Sommaire

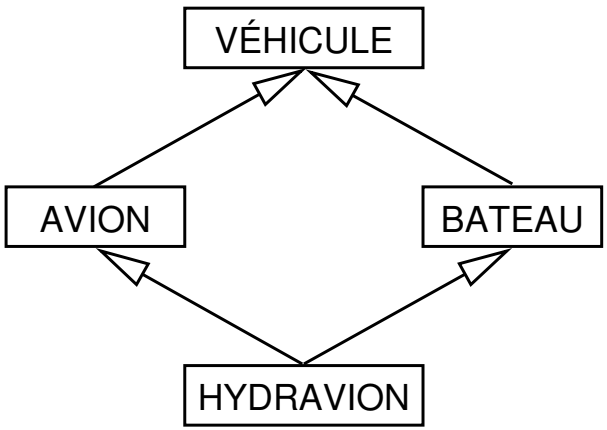
Héritage multiple

Définition : On dit qu'il a héritage multiple si une classe hérite d'au moins deux classes (parentes).

Exemple : Un mobilehome et un hydravion.



héritage multiple



héritage répété

Problèmes posés : Que deviennent les attributs et méthodes présents dans les deux classes parentes ? Représentent-ils la même notion (fusion) ou des notions différentes (duplication).

Solution choisie en Java : Interdire l'héritage multiple !
Et utiliser les interfaces à la place !

Exercice 7

7.1 Comment faire pour écrire une méthode de tri d'un tableau qui soit générale ?

7.2 Comment faire pour écrire une méthode permettant de dessiner sur un écran graphique tous les éléments d'un tableau ?

7.3 Comment faire pour qu'un objet, un point par exemple, puisse apparaître à la fois comme élément du premier tableau et du deuxième tableau ?

Héritage multiple sur les interfaces

Héritage multiple : Une interface peut spécialiser plusieurs interfaces.

```
interface I extends I1, I2 { ...  }
```

Attention : Deux méthodes ayant même nom et même signature sont considérées comme la même méthode.
$\Longrightarrow$ C'est une **résolution syntaxique** du problème de l'héritage multiple : - (

```
1  public interface Affichable {
2      /** Afficher avec un décalage de indentation espaces. */
3      void afficher(int indentation);
4  }
```

```
1  public interface MultiAffichable {
2      /* Afficher plusieurs fois. */
3      void afficher(int nb);
4  }
```

```
1  public class PbHeritageInterface implements Affichable, MultiAffichable {
2      public void afficher(int entier) { // Que doit faire afficher ?
3          System.out.println("Entier = " + entier);
4      }
5  }
```

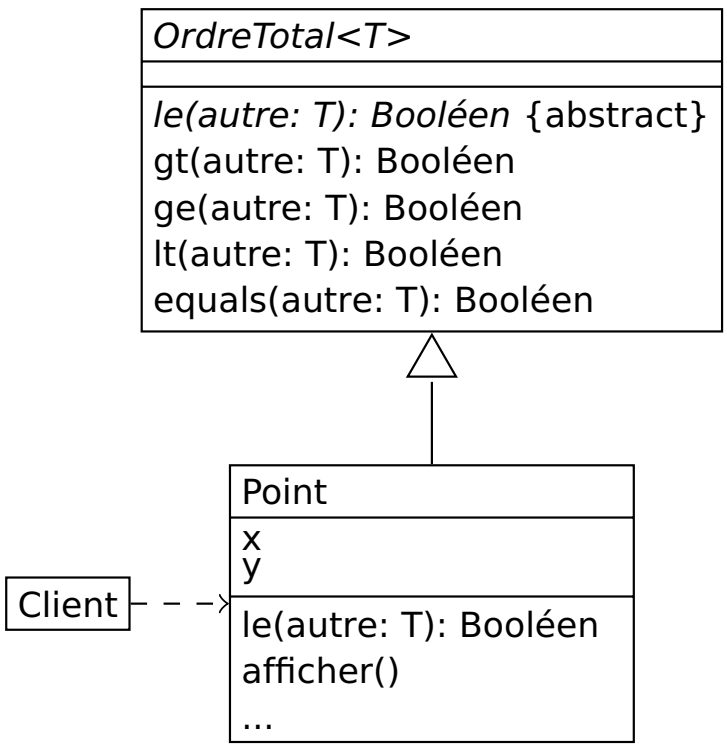
Sommaire

Interface vs classe abstraite

interface	**classe abstraite**
Notion abstraite $\Longrightarrow$ Ne peut pas être instanciée	
Spécifie un comportement	
Oblige à définir des classes (réalisation ou sous-classe)	
− Aucun code ne peut être écrit	+ Factorisation de code possible (attributs, méthodes)
	+ Peut imposer une méthode aux sous-classes (**`final`**)
+ On garde la possibilité d'hériter	− Obligation d'hériter de cette classe

Intérêt d'une classe abstraite

Ordre totale avec une classe abstraite



`OrdreTotal`

- elle est **abstraite**
- tous les opérateurs de comparaison sont spécifiés
- seule `le` est retardée
- les autres opérateurs sont définies à partir de le
- la redéfinition de `equals` garantit sa cohérence
- ces méthodes sont **`final`** (cohérence garantie)

`Point`

- hérite de `OrdreTotal` (obligatoire)
- ne doit définir que `le`
- récupère le code des autres opérateurs
- y compris `equals` (cohérent avec le !)

`Client`

- il a accès à tous les opérateurs de comparaison

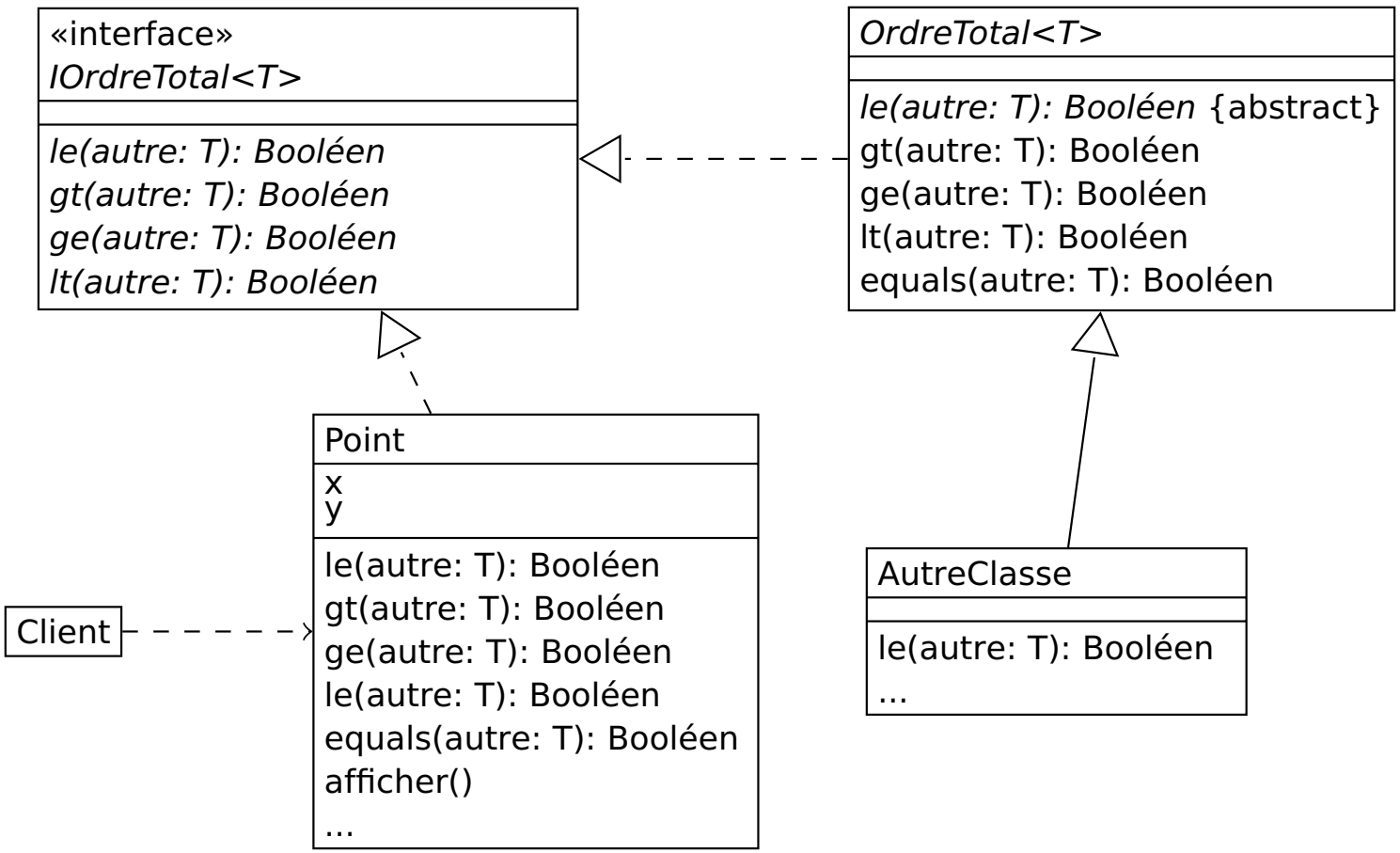
La classe abstraite OrdreTotal

```
/** Comparer des éléments de type T avec une relation d'ordre total */
public abstract class OrdreTotal<T> {
    /** Inférieur ou égal (lesser equal) */
    public abstract boolean le(T a);

    /** Supérieur ou égal (greater equal) */
    final public boolean ge(T a)        { return a.le(this); }

    /** Strictement supérieur (greater than) */
    final public boolean gt(T a)        { return ! this.le(a); }

    /** Strictement inférieur (lesser than) */
    final public boolean lt(T a)        { return ! a.le(this); }

    final public boolean equals(T a)
                            { return this.le(a) && a.le(this); }
}
```

Attention : Pour être correct, l'opérateur « `le` » doit être réflexif, transitif et antisymétrique.
Question : La méthode `equals` est-elle vraiment celle d'`Object` ?

Limite des classes abstraites

Problème : `Point` hérite déjà de `ObjetGéométrique`. Que faire ?
(On veut conserver la relation d'héritage sur ObjetGéométrique)



Discussion

- Ajouter une interface pour `OrdreTotal` : `IOrdreTotal`
- La classe abstraite réalise `IOrdreTotal` et définit les opérateurs (et equals)
  Elle factorise le code qui peut l'être
- Une classe choisit entre :
  - **hériter de la classe abstraite :**
    - récupére le code factorisé dans l'interface
    - ne peut plus hériter d'une autre classe
  - **réaliser l'interface :**
    - doit définir `toutes` les méthodes
    - garde sa possibilité d'hériter d'une autre classe
- Ici, la classe Point doit :
  - définir tous les opérateurs de comparaison (fastidieux, respecter la cohérence)
  - penser à définir equals pour qu'elle soit cohérente avec la relation d'ordre
- **Conséquence :** Éviter des méthodes redondantes dans une interface
  D'où l'interface `Comparable` de Java :

```
interface Comparable {
    /** Comparer cet objet - l'objet o par rapport à 0. */
    public int compareTo(Object o);
}
```

Remarque : La documentation de `Comparable` demande de redéfinir `equals`.

Sommaire

Réutilisation

La réutilisation se fait en Java au travers des classes.
Il s'agit de réutiliser des classes déjà écrites.
Il y a deux possibilités pour qu'une classe A « réutilise »une classe B :

- la *relation d'utilisation* (association, agrégation ou composition) :

```
class A {
    B b;        // poignée sur un objet de B
    ...
}
```

  Exemple : La classe segment (ré)utilise la classe Point.

- la *relation* d'héritage (spécialisation) :

```
class A extends B {     // A spécialise B
    ...
}
```

  Exemple : La classe PointNommé (ré)utilise la classe Point.

La question est alors : « Quoi choisir entre utilisation et héritage ? »

Choisir entre héritage et utilisation

Règles simples :

- « a » $\Longrightarrow$ association (ou agrégation)
- « est composé de » $\Longrightarrow$ composition (ou agrégation)
- « est un » $\Longrightarrow$ héritage
  Attention : « est un ... et ... » (utilisation) $\neq$ « est un ... ou ... » (héritage)

Remarque : ÊTRE, c'est AVOIR un peu !
On peut *toujours* remplacer l'héritage par l'utilisation.
L'inverse est faux. AVOIR, ce n'est pas toujours ÊTRE !

Deux règles :

- si on veut utiliser le polymorphisme $\Longrightarrow$ héritage
- si on veut pouvoir changer dynamiquement le comportement des objets
  $\Longrightarrow$ utilisation (poignée) associée à l'héritage

Exercice 8 Comment modéliser une équipe de football ?

Sommaire

Contraintes sur la (re)définition

Respect de la sémantique : La redéfinition d'une méthode doit préserver la sémantique de la version précédente : la nouvelle version doit fonctionner au moins dans les mêmes cas et faire au moins ce qui était fait (cf programmation par contrat).

Preuve : Une méthode `f(B b, ...)` travaille sur une classe polymorphe B.

- Cette classe B contient au moins une méthode polymorphe g.
- L'auteur de f ne connaît que B (*a priori*). Il utilise donc la spécification de B pour savoir comment appeler g et ce qu'elle fait.
- En fait, la méthode f est appelée avec un objet de la classe A sous-classe de B (principe de substitution) et redéfinissant g.
- En raison de la liaison tardive, c'est donc la version de g de la sous-classe A qui est appelée.

Conclusion : la version de g dans A doit fonctionner dans les cas prévus dans la super-classe B et faire au moins ce qui était prévu dans B.

Test : Une sous-classe doit réussir les tests de ses classes parentes.

Constructeur et méthode polymorphe

Règle : Un constructeur ne devrait pas appeler de méthode polymorphe.

Preuve : Considérons une classe A dont l'un de ses constructeurs utilise une méthode polymorphe m.

- Puisque m est une méthode polymorphe, elle peut être redéfinie dans une classe B sous-classe de A.
- La redéfinition de m dans B peut utiliser un attribut de type objet `attr` ajouté dans B (donc non présent dans A).
- L'ordre d'initialisation des constructeurs fait que le constructeur de A est exécuté avant celui de B, donc `attr` est **`null`**. Or le constructeur de A exécute `m`, donc la version de B (liaison tardive) qui utilise `attr` non encore initialisé !