Technologie Objet

—

Héritage – Classes abstraites – Réutilisation

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Sciences du Numérique

Sommaire

Sommaire

Exemple introductif

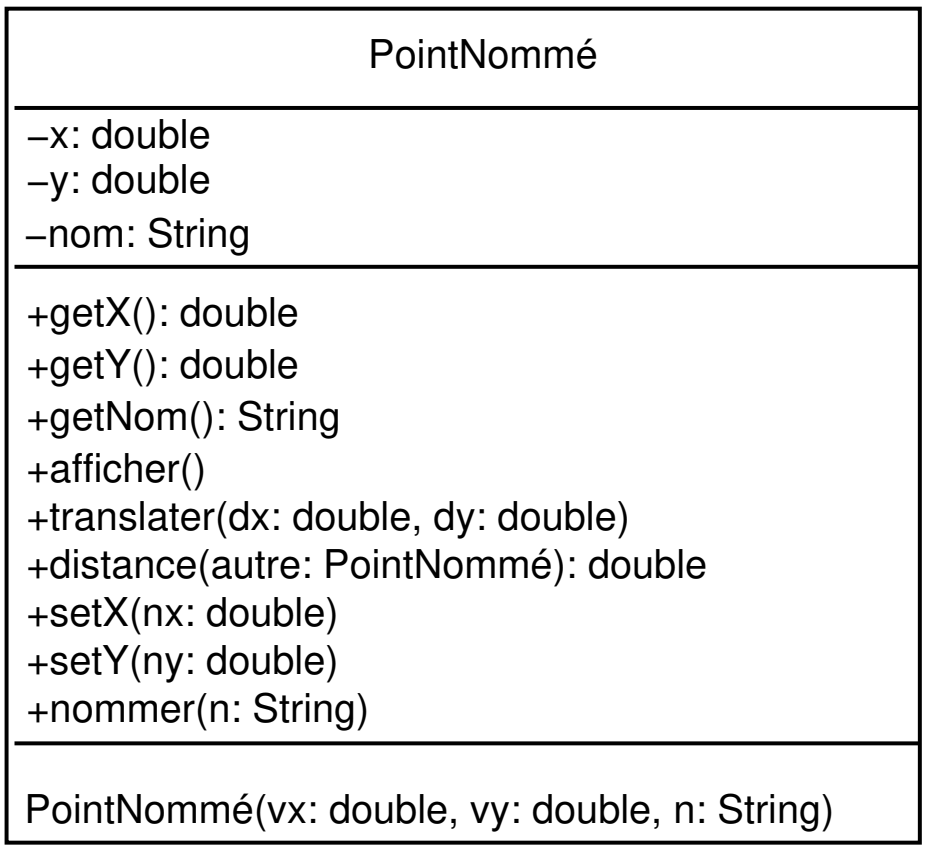
Exercice 1 : Définition d'un point nommé
Un point nommé est un point, caractérisé par une **abscisse** et une **ordonnée**, qui possède également un **nom**. Un point nommé peut être **translaté** en précisant un déplacement suivant à l'axe des X (abscisses) et un déplacement suivant l'axe des Y (ordonnées). 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 comparez 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)

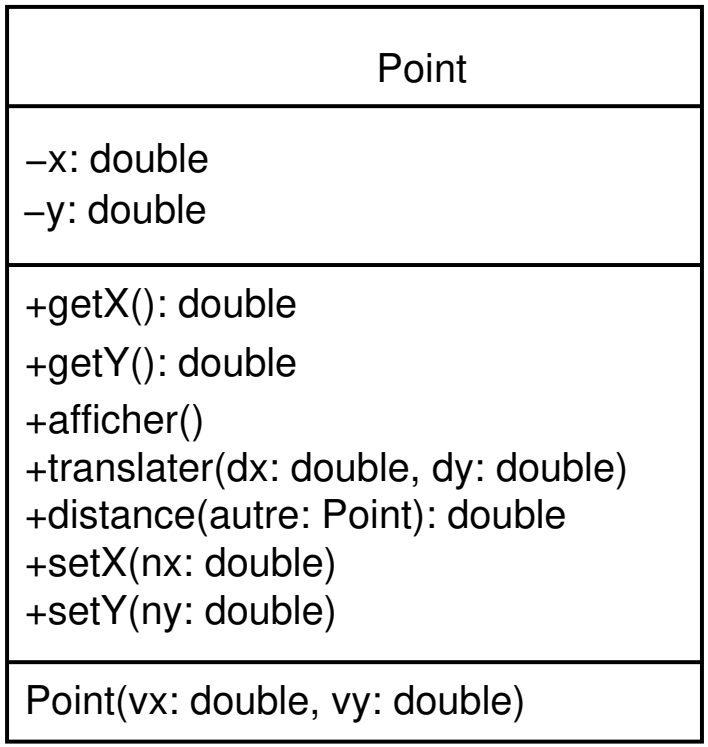
Rien de nouveau : on applique toujours la même démarche !

Une solution : Écrire directement la classe PointNommé



Ça marche... Mais on se souvient que l'on a déjà écrit une classe Point, très proche !

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 (mauvaise solution !)

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

Évaluons cette solution :

- 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 ?

Évaluons cette première solution

Que faire si on se rend compte que le calcul de la distance est faux ?

- Il faudra faire la **même correction deux fois** dans Point et PointNommé !

Comment ajouter une coordonnée z ?

- Il faudra faire la **même évolution deux fois**, sur Point et PointNommé !

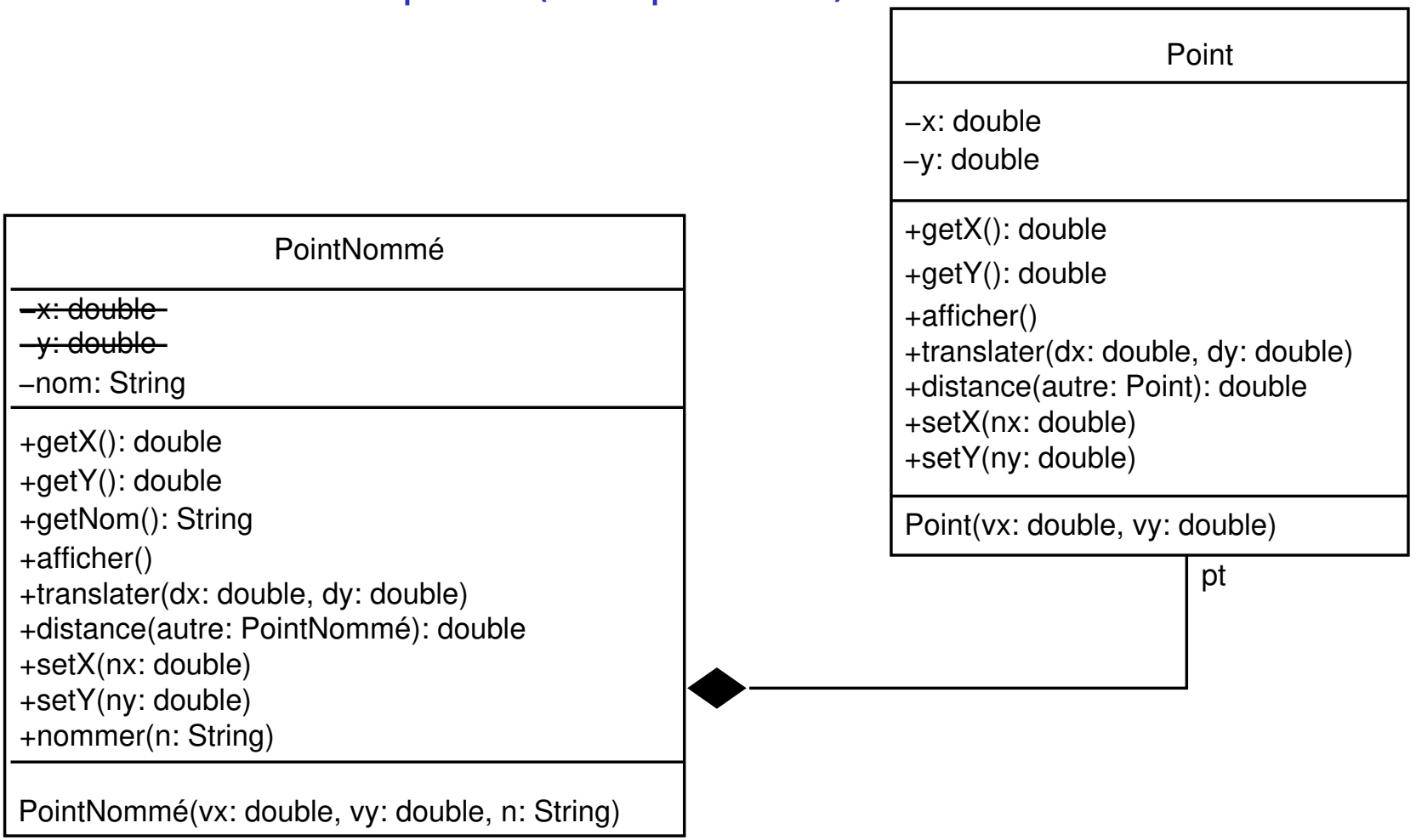
Peut-on calculer la distance entre un point et un point nommé ?

- On ne peut pas !
- Sauf à surcharger la méthode distance dans Point :

```
double distance(PointNommé autre) { ... }
```

- Ce qui crée une dépendance de Point sur PointNommé ! Mauvaise idée !
  - Que faudra-t-il faire si on ajoute PointPondéré, PointColoré, etc. ?
  - Modifier Point ! Encore et encore !
  - C'est contraire à l'extensibilité !
- Et le code est le même que la méthode distance(Point).

Solution 2 : Utiliser le point (composition)



- La classe PointNommé a deux attributs : nom et pt
- Cette solution pourrait être utilisée mais n'est pas optimale ici

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

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

$\Longrightarrow$ Pas de code dupliqué !

- Pour corriger une erreur dans distance, il suffit de corriger Point

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 (`afficher` est adaptée pour faire apparaître le nom)

On parle de **délégation** car une partie des fonctionalités de PointNommé est déléguée à Point

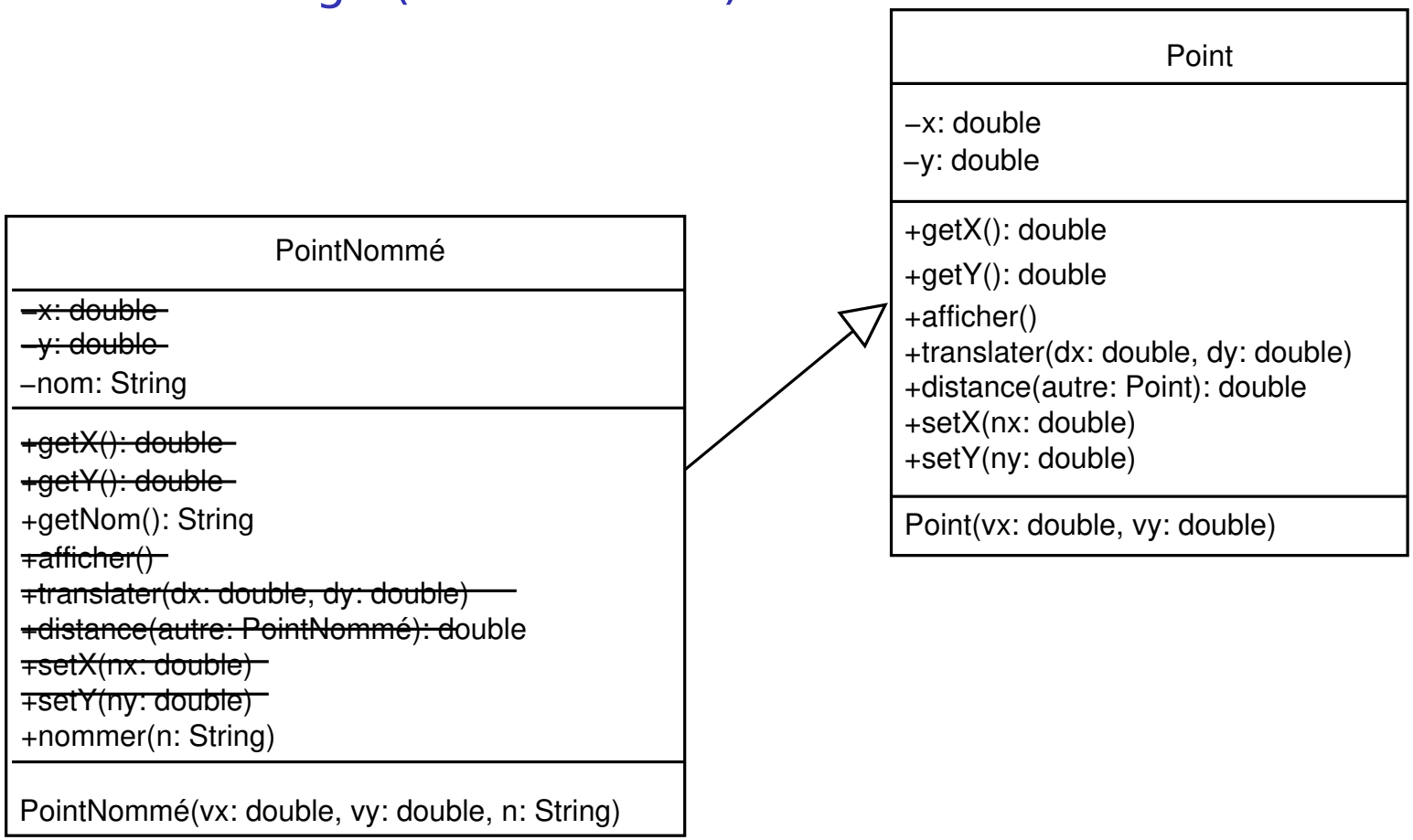
Toujours impossible de calculer la distance entre Point et PointNommé.

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

Important : On constate qu'un point nommé est un point particulier :

- Un point nommé **est un** point... qui a un nom.
- Plus précisément, on pourrait utiliser un point nommé partout ou un point est attendu.
- Ceci nous permet d'appliquer la solution 3 (la bonne ici !)

Solution 3 : L'héritage (la bonne ici !)



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 :

- définir une nouvelle classe comme **spécialisation** d'une classe existante ;
  - Ajout de nouvelles fonctionnalités
  - Exemple : PointNommé spécialise Point (ajoute la notion de nom)
- définir une nouvelle classe comme **généralisation** de classes existantes
  - Factorisation des fonctionnalités communes à plusieurs classes
  - Exemple : Point généralise PointNommé, PointPondéré, PointColoré...
- définir une **relation de sous-typage** entre classes (principe de substitution [1])

Conséquence :

- copie virtuellement les éléments de classes existantes dans la sous-classe (héritage)

  Attention : ne devrait jamais être la raison pour recourir à l'héritage [2]

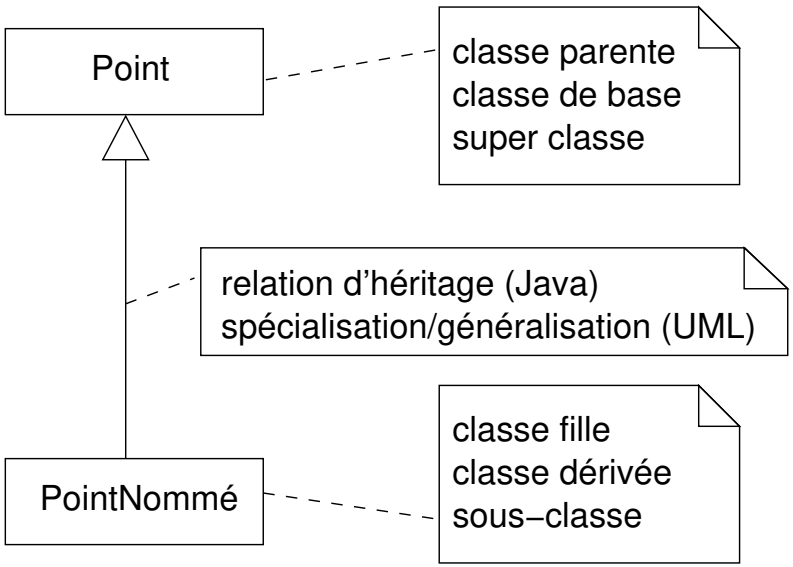
Moyen :

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

1. Barabara Liskov. Déjà vu avec les interfaces. C'est le même !
2. Si héritage implique sous-typage comme en Java.

Notation et vocabulaire

Notation UML



Notation en Java

```
public class PointNommé
    extends Point       // héritage
{
    ...
}
```

Attention : La flèche est fermée : marque du sous-typage, comme pour une réalisation.
Une flèche ouverte correspondrait à une relation d'association avec sens de navigation !

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  }   }
```

- PointNommé hérite de Point, donc PointNommé est une sous-type Point
- On n'a défini que ce qui est spécifique à PointNommé

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 PointNommé on peut définir afficher(String) :

```
public void afficher(String préfixe) {
    System.out.println(préfixe);        // afficher le préfixe
    this.afficher();                    // utiliser l'autre afficher, sans paramètre
}
```

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 objet `PointNommé`, il y a un `Point`... et un nom.
- le Point doit être initialisé avec un constructeur de la classe Point

Syntaxe : **`super`** suivi des paramètres effectifs pour appeler 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, comme **`this`**`(...)`
- 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 !

Remarque : Code très proche de la solution 2 (composition), T. 11 : **`super`** vs pt !

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.
- `afficher()` n'affiche pas le nom du PointNommé.

  $\Rightarrow$ Il faut l'adapter...

Redéfinition de méthode

Redéfinition : Une 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``.afficher()` : appeler la méthode `afficher()` définie dans la super-classe (Point)

```
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 définie dans la sous-classe (mêmes signatures !)

L'annotation @Override (Java5)

Principe : @Override exprime l'intention du programmeur de redéfinir une méthode de la super-classe et permet donc au compilateur de contrôler qu'il s'agit bien d'une redéfinition !

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 : Est-ce que ce sont bien des redéfinitions ?

L'annotation @Override (Java5) (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
```

Conseils :

1. Toujours utiliser l'annotation `@Override` quand on (re)définit une méthode de manière à rendre son intention explicite pour le compilateur et les lecteurs !
2. Mettre aussi `@Override` devant la définition, dans une réalisation, des méthodes d'une interface.

Un programme qui compile dans erreur est équivalent au même programme sans `@Override`.
Mais pourquoi prendre le risque de ne pas mettre `@Override` ? Le goût du risque ?

Principe de substitution

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

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

- Tout ce qu'on peut demander à un point peut être demandé à un point nommé.

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

- On ne peut ni demander le nom, ni changer le nom d'un point !

```
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();  // Possible ?  Affichage ?
5  q = pn1; q.afficher();  // Possible ?  Affichage ?
6
7  PointNomme qn;  // poignée sur un PointNommé
8  qn = p1;  qn.afficher(); // Possible ?  Affichage ?
9  qn = pn1; qn.afficher(); // Possible ?  Affichage ?
```

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

Réponses

```
q = p1;          // OK : q et p1 ont même type (Point)
q.afficher();    // (3, 4) : on exécute la méthode afficher() de Point


q = pn1;         // OK : q de type Point et pn1 de type PointNommé, sous-type de Point
q.afficher();
    // A:(30, 40) : on exécute la méthode afficher() de PointNommé (liaison dynamique)
```

- C'est la même *liaison dynamique* que celle vue sur les interfaces !
  - Liaison statique : le type de la poignée permet de sélectionner la signature
  - Liaison dynamique : son implantation est cherchée dans le type réel de l'objet attaché à la poignée
- La différence ici, c'est que afficher() a un code dans Point et PointNommé
- Voir T. 29 pour l'intérêt de ce mécanisme !

```
qn = p1;   // Erreur : Point, type de p1, n'est pas sous-type de PointNommé, type de qn


qn = pn1;        // OK : même type pour qn et pn1 (PointNommé)
qn.afficher();   // A:(30, 40), méthode afficher() de PointNommé
```

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

Pour chaque instruction du programme principal suivant, indiquer :

1. si elle est acceptée par le compilateur,
2. et, dans l'affirmative, ce qui sera affiché.

```
1  class A {
2      void m()      { System.out.println("A.m()");    }
3      void m(int n) { System.out.println("A.m(int)"); }
4  }
5
6  class B extends A {
7      void m()         { System.out.println("B.m()");         }
8      void m(String s) { System.out.println("B.m(String)");   }
9  }
10
11 class Exemple {
12     public static void main(String[] args) {
13         A c = new B();      A a = new A();      B b = new B();
14         c.m();              a.m();              b.m();
15         c.m(1);             a.m(1);             b.m(1);
16         c.m("abc");         a.m("abc");         b.m("abc");
17     }
18 }
```

Solution

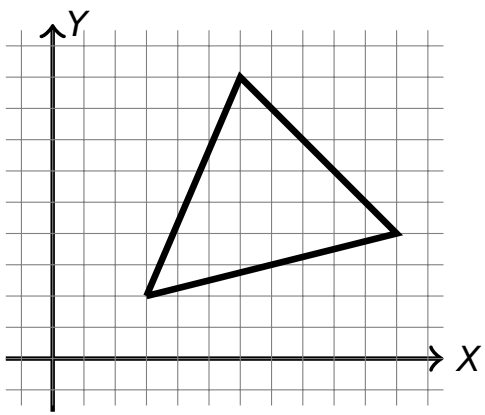
```
A c = new B();  // OK car B est un sous-type de A (héritage)
    // type apparent de c = A => résolution de la surcharge dans A
    // type réel de c = B, donc la méthode sera cherchée dans la classe B
c.m();  // Résolution surcharge : m(), seule candidate, donc OK
        // Liaison dynamique : m() dans B ? Celle de B (redéfinition) :  "B.m()"
c.m(1); // Résolution surcharge : m(int), seule candidate, donc OK
        // Liaison dynamique : m(int) dans B ? Celle de A => "A.m(int)"
c.m("abc");     // Résolution surcharge : pas de méthode m(String) dans A => Erreur !


A a = new A();   // OK mêmes types
    // type apparent de a = A => mêmes résultats que ci-dessus pour résolution surcharge
    // type réel de a = A, donc la méthode sera cherchée dans la classe A
a.m();       // "A.m()"
a.m(1);      // "A.m(int)"
a.m("abc");  // Erreur de compilation : pas de m(String) dans A !


B b = new B(); // OK mêmes types
    // type apparent de b = B => résolution surcharge dans B
    // type réel de b = B => exécution de la méthode de B
b.m();  // Surcharge : m(), Liaison dynamique : m() dans B => "B.m()"
b.m(1); // Surcharge : m(int), Liaison dynamique : m(int) dans B ? celle de A => "A.m()"
b.m("abc");  // Surcharge : m(String), Liaison dynamique : m(String) de B => "B.m(String)"
```

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

Solution

- Initialement, la classe Segment s'appuie sur Point et ne connait que Point.
- On a déjà travaillé :
  - Ajout de PointNommé qui spécialise (hérite) de Point et redéfinit la méthode afficher()
- Il suffit donc de remplacer la ligne 4 :

```
PointNommee p1 = new PointNommee("A", 3, 2);
```

- On peut créer un Segment à partir de Point et PointNommé
  - car PointNommé hérite de Point, donc sous-type
- quand on fait `s12.afficher()`, on appelle `afficher()` sur `extrémité1` et `extrémité2` (accepté car du type Point qui contient afficher()). La liaison dynamique fait que :
  - type réel de `extrémité1` = PointNommé ⇒ afficher() de PointNommé est appelée : `A:(3,2)`
  - type réel de `extrémité2` = Point ⇒ afficher() de Point est appelée : `(6, 9)`
- **Conclusion :**
  - On n'a fait aucun changement sur Segment.
  - Segment ne connaissait pas et ne connait toujours pas PointNommé.
  - Et pourtant un segment peut être créé avec des Point ou des PointNommé
  - Et quand on affiche un Segment, la méthode afficher() adaptée au type du point sera appelée
  - C'est l'intérêt de la liaison dynamique !
  - Elle permet l'**extensibilité** ! Ici, facilite l'ajout de nouvelles variétés de Point

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 le point q
if (q instanceof PointNommé) {          // 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 (q instanceof ...) {
    ...
} else {                // Ce n'est donc qu'un point !
    // afficher q comme un Point
}
```

Solution : Définir/spécifier une méthode (ici afficher) dans le type le plus général (Point) et la redéfinir dans les sous-classes (PointNommé, PointColoré...) !

Question : Que penser du premier commentaire (*afficher le point q*) ?

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.
   - *C'est au programmeur de penser à le faire !*
   - Si ce n'est pas le cas, il ne faut pas utiliser l'héritage !
   - Ce ne doit jamais être pour récupérer ce qui est dans une classe [3]
   - Vérifier le sous-typage : est-ce que la phrase « Sous-classe **est un** Super-classe » sonne juste ?
     « Un point nommé est un point » : Oui.
     « Un cercle est un point » : Non !
2. Est-ce qu'il y a des méthodes de la super-classe à adapter ?
   - *C'est au programmeur de penser à se poser la question !*
   - Redéfinir les méthodes à adapter !
   - Attention, il y a des règles sur la redéfinition (voir programmation par contrat et T. 71)
3. Enrichir la classe des attributs et méthodes supplémentaires.
   - En général, on n'oublie pas. C'est souvent la raison d'être de la nouvelle classe.
4. Tester la classe
   - Comme toute classe !
   - Elle doit aussi réussir les programmes de test de sa super-classe (sous-typage !).

3. Sauf pour les langages qui dissocient héritage et sous-typage.

Comment ajouter un nouveau type de point, PointPondéré ?

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 l'applciation
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.

```
// On doit connaître les types de points pour les créer
Point p1 = new PointNommee("A", 3, 2);
Point p2 = new Point(6, 9);
Point p3 = new PointPondéré(10.5, 11, 4);

// Dans le reste on les manipule comme des Point
// sans avoir à connaître leur type réel !
// Merci sous-typage, principe de substitution et laison dynamique !
```

Suite du programme du T. 24

```
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()   // ????
```

L'affectation ligne 11 est refusée : le type apparent de qn est PointNommé, il ne peut donc pas être initialisé avec q dont le type apparent est Point !

On sait que le type réel de q est PointNommé. Comment faire quand même l'affectation ?

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 problème de l'**affectation renversée** résolu en Java par le « transtypage » :

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

Attention : Ce transtypage est vérifié 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

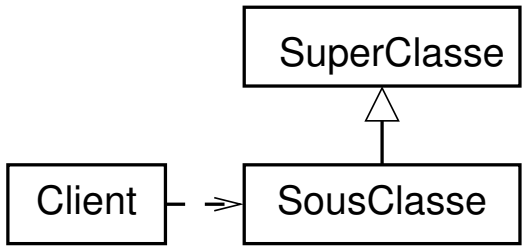
En Java, si une classe n'a pas de classe parente, elle hérite implicitement de la classe Object. C'est l'**ancêtre commun** à toutes les classes.

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).
- et quelques autres...

Conséquence : En Java toute classe est un descendant de Object. Il faut donc se demander si les méthodes de Object doivent être redéfinies (toString, equals, etc.)

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

Questions :

1. Dans un objet de type B, y a-t-il 2 ou 3 attributs ?
2. Qu'affiche l'exécution de ce programme ?

Résultats

Il y a bien 3 attributs dans un objet de type B

Le résultat de l'exécution donne :

```
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

- Classe abstraite
- Méthode abstraite
- Constructeurs
- Intérêt
- Exemple : les points
- Exemple : schéma mathématique
- Notation UML

Classe abstraite

Définition : Une classe abstraite est une classe qui ne permet pas de créer d'objet.

Syntaxe : On utilise le modifieur **abstract** devant **class**.

Exemple :

```
1  abstract class ClasseAbstraite {
2  }
3
4  class Main {
5      public static void main(String[] args) {
6          ClasseAbstraite c = new ClasseAbstraite();
7      }
8  }
```

```
ClasseAbstraite.java:6: error: ClasseAbstraite is abstract; cannot be instantiated
                ClasseAbstraite c = new ClasseAbstraite();
                                    ^
1 error
```

Remarque : Elle devra nécessairement avoir des sous-classes !

Méthode abstraite

Définition : Une méthode abstraite (ou retardée) est une méthode dont le code n'est pas donné. On doit utiliser le mot-clé **`abstract`** (dans les modifieurs).

```
/** ... */
abstract void uneMethodeAbstraite();
```

- On les a déjà vues dans les interfaces.
- Dans une classe le mot-clé **`abstract`** est obligatoire
- Le « ; » remplace le code de la méthode.
- Le commentaire de documentation est essentiel (d'autant que le code n'est pas donné !)
- Une classe qui contient une méthode abstraite doit obligatoirement être déclarée abstraite.
  - Les sous-classes devront définir les méthodes abstraites où elles seront elles-mêmes abstraites.

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

Constructeurs et classe abstraite

Une classe abstraite peut définir des constructeurs.

Exemple :

```
// Commentaires omis et indentation non respectée (gain de la place)
public abstract class ElementNomme {
        private String nom;

        public ElementNomme(String nom)   { this.nom = nom; }
        public String getNom()            { return this.nom; }
        public void setNom(String nom)    { this.nom = nom; }
}
```

Question : Une classe abstraite ne permet pas de créer d'objets. À quoi sert le constructeur ?

Réponse : Il sert toujours à initialiser l'objet. Il sera appelé quand on créera un objet à partir d'une sous-classe car le constructeur d'une sous-classe doit obligatoirement appeler un constructeur de la super-classe.

Ceci garantira de ne pas oublier d'initialiser d'initialiser l'état correspondant à la super-classe.

Intérêt d'une classe abstraite

1. Factoriser du code (attributs et méthodes) commun aux sous-classes
   - les sous-classes peuvent redéfinir les méthodes (sauf si **`final`**)
2. Imposer un comportement aux sous-classes en définissant une méthode **`final`**
   - Une méthode **`final`** ne peut être redéfinie pas une sous-classe
3. Obliger les sous-classes à définir les méthodes abstraites (pour donner le code adéquat)
   - Sinon les sous-classes devront être déclarées abstraites

Exercice 3 Quel est l'intérêt d'utiliser une méthode abstraite plutôt que de définir une méthode avec un code vide (ou affichant un message d'erreur) qui serait redéfinie dans les sous-classes ?

Réponse à l'exercice 3

- Supposons que la classe A a une méthode m() définie avec un code vide ou qui signale une erreur au lieu d'être déclarée abstraite
- Supposons qu'on oublie de redéfinir m() dans la classe B qui hérite de A
- Pour identifier l'erreur, il faut
  1. écrire un programme qui crée un B et appelle la méthode m() dessus
  2. constater l'erreur signalée.

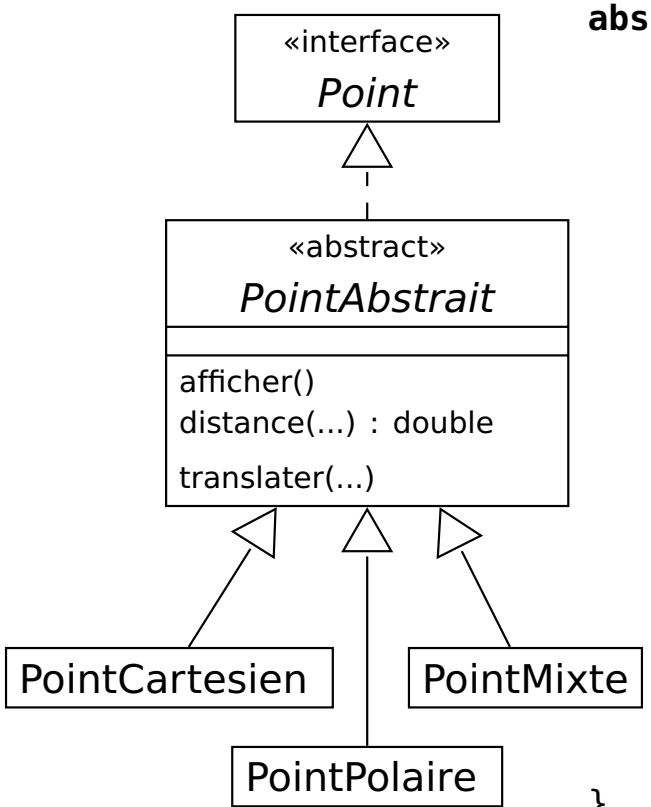
  $\Rightarrow$ Détection aléatoire et tardive !
- Si m() avait été déclarée abstraite dans A, le compilateur aurait refusé de compiler B
  - en signalant que m() de A est abstraite dans B
  - et que la classe B doit être déclarée abstraite

  $\Rightarrow$ Le programmeur sait qu'il a oublié de définir m()
  - ou oublié de déclarer la classe B abstraite (**`abstract`**)

  $\Rightarrow$ L'erreur étant signalée par le compilateur, elle ne peut pas être ratée !

Exemple : Les différentes réalisations de Point

Nous avions spécifié un Point (interface) et proposé plusieurs réalisations (PointCartésien, PointPolaire...). On constate que ces réalisations ont le même code pour les méthodes afficher, distance et translater. On peut le factoriser dans une classe abstraite.




```
abstract public class PointAbstrait implements Point {

    @Override public void afficher() {
        System.out.print("(" + this.getX() + ","
                + this.getY() + ")");
    }

    @Override public double distance(Point autre) {
        double dx2 = Math.pow(autre.getX() - this.getX(), 2);
        double dy2 = Math.pow(autre.getY() - this.getY(), 2);
        return Math.sqrt(dx2 + dy2);
    }

    @Override public void translater(double dx, double dy) {
        this.setX(this.getX() + dx);
        this.setY(this.getY() + dy);
    }
}
```

Remarque : Les *default methods* de Java8 permettraient de le définir dans l'interface Point.

Exemple : schémas mathématiques

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 ?

Modéliser le schéma

Partant du programme du T. 29, on veut mettre les trois segments et le barycentre dans un même tableau. On appelle X le type des éléments du tableau.

```
1  X[] schema = new X[10]; // 10 suffisant pour cet exemple !
2  int nb = 0;             // taille du schéma (nombre de constituants)
3
4  // Construire le schéma
5  schema[nb++] = s12;
6  schema[nb++] = s23;
7  schema[nb++] = s31;
8  schema[nb++] = barycentre;
```

Quelles sont les contraintes sur X pour que ce code fonctionne ?

- Les affections `schema[nb++] = s##` impliquent que Segment doit être un sous-type de X
- L'affectation `schema[nb++ = barycentre|` implique que Point doit être un sous-type de X
- **Conclusion :** X généralise Segment et Point

Questions :

- Quels sont les membres (attributs et opérations) de X ?
- Peut-on définir des constructeurs sur X ?
- Comment appeler X ?

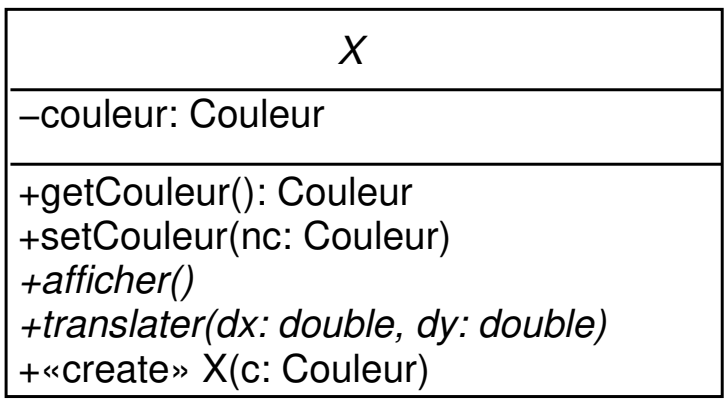
Définition de X

Quels sont les membres (attributs et opérations) de X ?

- Ce qui est commun à Point et Segment (généralisation)
- C'est-à-dire les attributs et méthodes qui apparaissent dans Point et Segment
- Donc, pensons aux points et segments vus en TP : l'attribut couleur, son accesseur, son modifieur, translater et afficher.

Peut-on définir des constructeurs sur X ?

- Oui, un pour initialiser la couleur !



Questions :

- Quel est le code de translater ?
- Maintenant qu'on sait mieux ce qu'est X, comment l'appeler ?

Le code (et le nom) de X

Quel est le code de translater ?

- On ne sait pas l'écrire, car on ne sait pas ce qu'est un X.
- On sait translater un Segment ou un Point, pas un X.

$\Rightarrow$ translater est donc une méthode abstraite ! Les sous-classes devront la définir.

Maintenant qu'on sait mieux ce qu'est X, comment l'appeler ?

- J'espère que vous avez essayé de trouver des noms...
- Si vous pensez Schéma, Dessin, etc. Ce n'est pas bon. X n'est pas plusieurs éléments mais un seul : soit un Point, soit un Segment, etc.
- Si on pense Objet, Element, c'est possible mais vague, trop général.
- Prenons ObjetGéométrique !

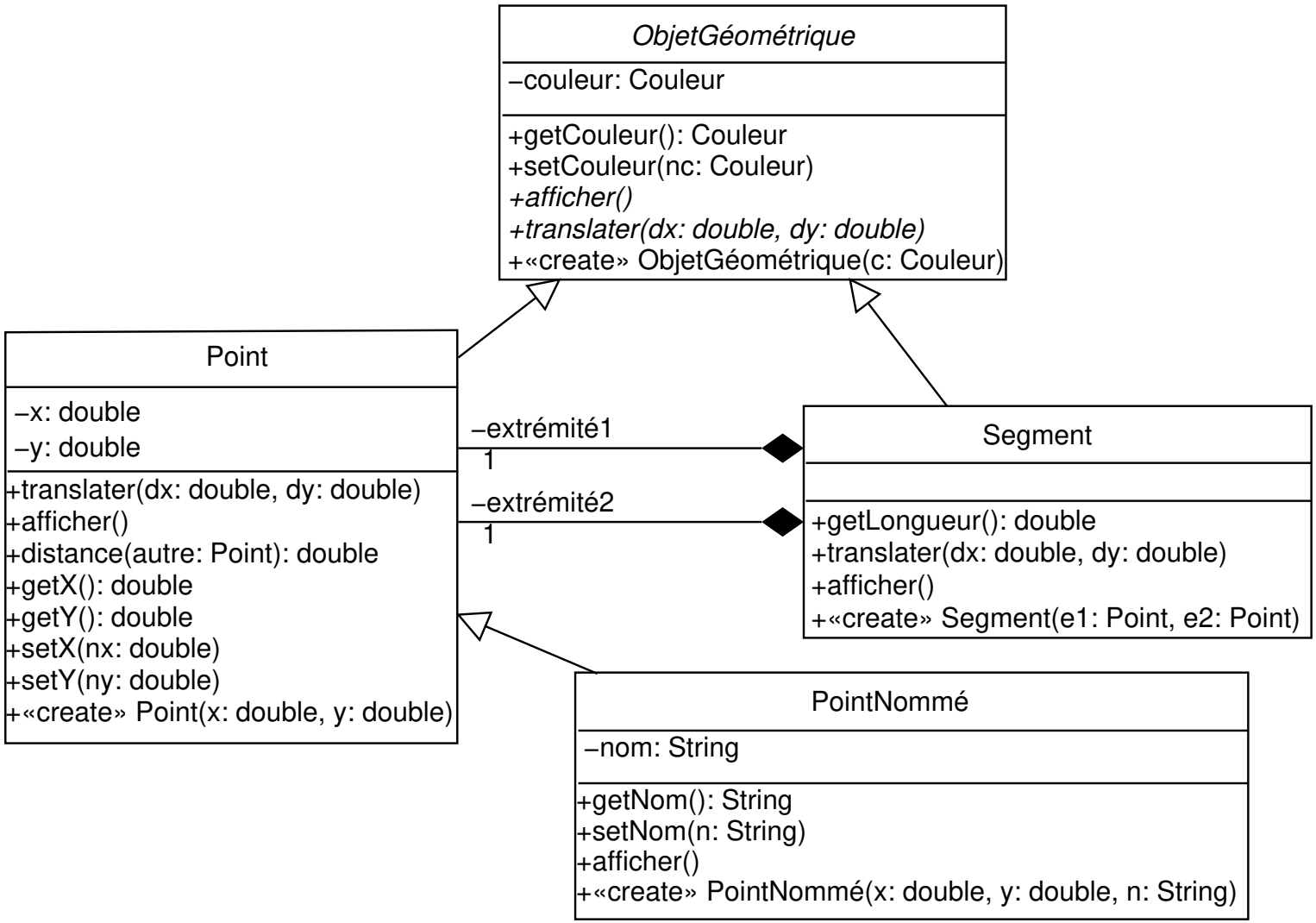
On a tout pour écrire le code de la classe ObjetGéométrique...

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 : Peu de choses définies. Une telle classe est-elle réellement utile ?
Réponse T. 56

Architecture : diagramme de classe UML



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 !
  - On peut également utiliser la contrainte `{abstract}`
  - ou le stéréotype «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 ?
    réponse en TD...

Autres questions de l'exercice 4, T. 49

Comment faire pour translater un schéma

```
// translater le schéma de dx et dy
for (int i = 0; i < nb; i++) {
    schema[i].translater(dx, dy);
}
```

- Ça fonctionne :
  - liaison statique : translater existe sur ObjetGéométrique et
  - liaison dynamique : c'est la version de Point, Segment... qui sera appelée
- Si on avait choisi Object au lieu de ObjetGéométrique :
  - On aurait pu créer le schéma : Point et Segment sont des Object !
  - On n'aurait pas pu le translater : pas de méthode translater sur Object
  - C'est l'intérêt de spécifier une méthode même si on ne sait pas l'implanter (cf interfaces)

Comment faire si on veut ajouter un Polygone ?

- On définit une nouvelle classe par spécialisation de ObjetGéométrique !

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

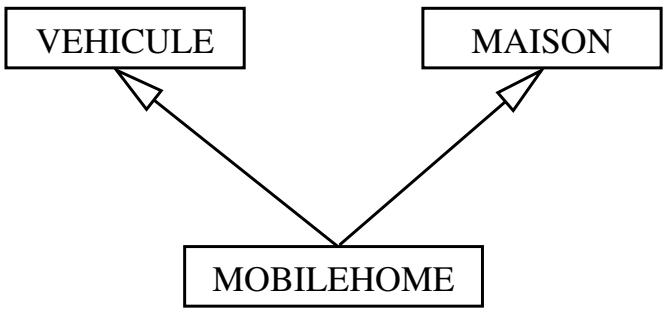
- On spécifie une méthode abstraite « agrandir » sur ObjetGéométrique
- On la définit dans toutes les sous-classes de ObjetGéométrique
  - Si on oublie, erreur de compilation : méthode agrandir abstraite et classe non déclarée abstraite

Sommaire

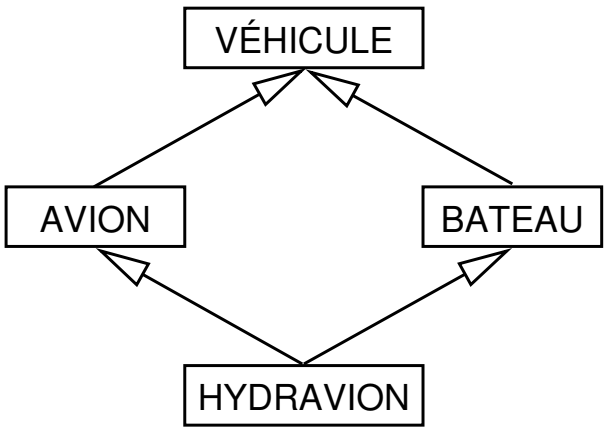
Héritage multiple

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 !

Sous-typage multiple

Il est **important d'avoir du sous-typage multiple** : un type est sous-type de plusieurs types

Exemple : La classe Point

- On veut considérer un Point comme un ObjetGéométique, un Dessinable (on peut le dessiner), un Comparable, etc.
  - pour utiliser un Point là où l'un de ces types est attendu !
- Un Dessinable n'est pas forcément Comparable, un Comparable pas forcément Dessinable.

  $\Rightarrow$ Il ne peut donc pas y avoir de relation de sous-typage entre les deux.
- Donc Point (ou un de ses super-types) doit être sous-type de Comparable **et** Dessinable.

  $\Rightarrow$ On a donc bien besoin de sous-typage multiple.

Solution en Java : Passer par des interfaces

- Une classe peut réaliser un nombre quelconque d'interfaces
- Une interface peut hériter de plusieurs interfaces

Conséquence : Le code qui aurait pu être factorisé dans une classe sera dupliqué.

- On ne peut pas définir PointNommé comme un héritage de `Point` et `ÉlémentNommé` (T. 45).
- Le code de `ÉlémentNommé` est donc dupliqué dans Point !

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

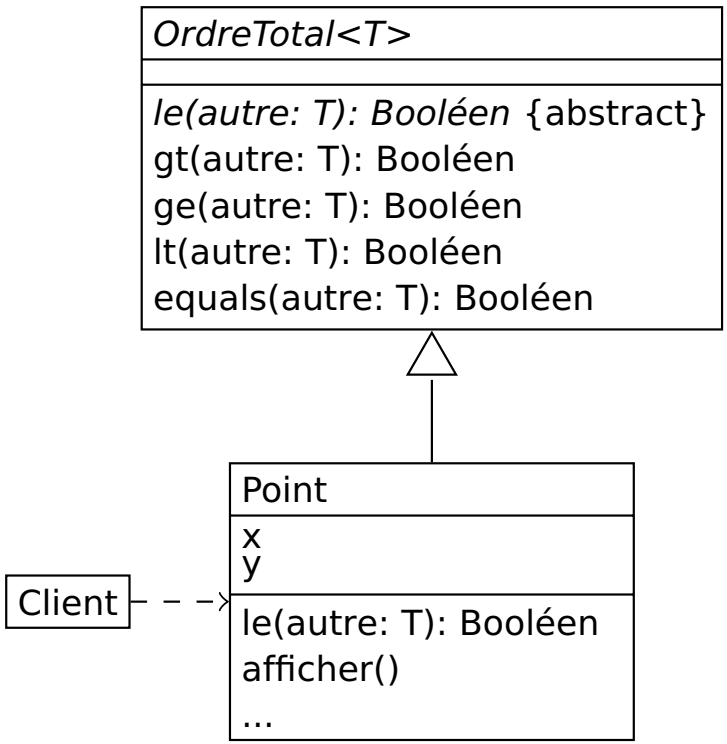
Interface vs classe abstraite

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

Remarque : Depuis Java8, les interfaces permettent aussi de factoriser du code grâce aux méthodes par défaut (*default methods*).

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

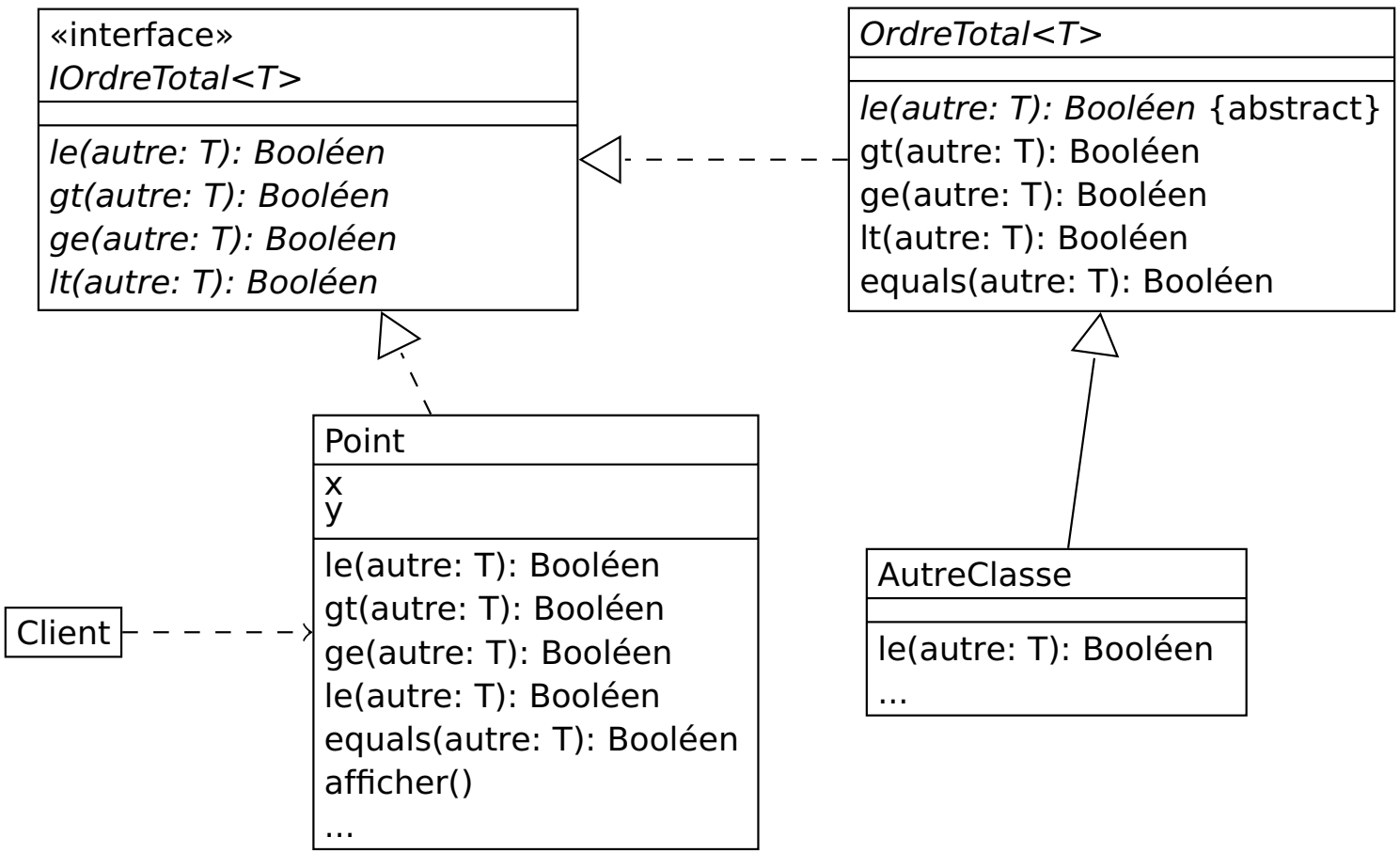
La classe abstraite OrdreTotal

```
/** 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` ? Réponse en TD...

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 `OrdreTotal` réalise `IOrdreTotal` et définit les opérateurs (et equals)
  `OrdreTotal` 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 hérite d'`ObjetGéométrique` et doit donc réaliser `IOrdreTotal` et :
  - définir tous les opérateurs de comparaison (fastidieux, respecter la cohérence)
  - penser à définir equals pour qu'elle soit cohérente avec la relation d'ordre
- **Corolaire :** Éviter des méthodes redondantes dans une interface

```
public interface Comparable {   // une seule méthode, facile à utiliser
    /** Compare (this - o) par rapport à 0. */
    public int compareTo(Object o);
} // La documentation de Comparable demande de redéfinir equals
```

- **Meilleure solution avec Java8 :** Les méthodes **`default`** des interfaces.

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 de délégation* (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 : « Que choisir entre délégation et héritage ? »

Choisir entre sous-typage et délégation

Règles simples : (voire simplistes)

- « a » $\Longrightarrow$ association (ou agrégation), donc délégation
- « est composé de » $\Longrightarrow$ composition (ou agrégation), donc délégation
- « est un » $\Longrightarrow$ sous-typage (héritage, réalisation)

  Attention : « est un ... et ... » (délégation) $\neq$ « est un ... ou ... » (sous-typage)
  - Une cercle **est un** centre **et** un rayon : en fait délégation !
  - Un objet géométrique **est un** point **ou** un segment : c'est bien du sous-typage !

Remarque : ÊTRE, c'est AVOIR un peu !
On peut *toujours* remplacer l'héritage par la délégation (mais on perd le sous-typage)

- On peut définir PointNommé par composition de Point (voir T. 11)

L'inverse est faux. AVOIR, ce n'est pas toujours ÊTRE !

- Un conducteur **a** une voiture mais un conducteur **n'est pas** une voiture

Deux règles :

- si on veut utiliser le polymorphisme $\Longrightarrow$ héritage (en fait sous-typage)
- si on veut pouvoir changer dynamiquement le comportement des objets
  $\Longrightarrow$ délégation (poignée) associée au sous-typage

Exercice 5 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` de B.
- 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 !