2i002 - Héritage : les classes abstraites

Vincent Guigue - vincent.guigue@lip6.fr

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

Nouveaux concepts

- **Classe abstraite**
  - Non implémentable
  - Seules les classes filles seront implémentables
  - Ex : Animal (ABS) $\rightarrow$ Poule, Renard, ...
- **Fonction abstraite**
  - Seulement dans les classes abstraites
  - Contient une signature mais pas de code
  - Ex : dans Animal, `String getEspece()`

Classe abstraite

Définition

- Représente un objet qui ne peut pas être instancié
- Un concept unificateur qui permet de factoriser du code pour toutes les classes qui hériteront
- Introduction de la notion de contrat : toutes les classes filles devront *gérer* ce qui est décidé par la classe mère (signature de méthode abstraite)

```
1 public abstract class Figure {
2 ...
3     // signature seulement pour les méthodes abstraites
4     public abstract String getType();
5 }
```

- On ne peut pas créer d'instance `Figure`
- Des classes vont hériter de Figure, elles devront implémenter `public abstract String getType();`

Propriétés des classes abstraites

- Les classes abstraites peuvent avoir des attributs
  - Ex : une figure géométrique (abstraite), peut être localisée par des coordonnées
- Les classes abstraites peuvent avoir des méthodes
  - Ex : la méthode déplacer peut changer les coordonnées précédentes
- Les classes abstraites peuvent avoir des méthodes abstraites
  i.e. : des méthodes dont on donne seulement la signature. Elles seront forcément implémentées pour les classes concrètes qui héritent.
  - Ex : `String getType()`

Idées

Les classes abstraites sont pensées pour leurs descendantes, les classes filles qui en seront dérivées

(Retour) sur les bonnes pratiques

Développement à long terme

modification d'un projet existant = ajout d'une classe

Idée :

Structurer un projet avec des classes abstraites =

- les classes filles possèdent des fonctionnalités dès leur création
  - factorisation du code
- Ajout de **contraintes** sur les classes fille
  - **Plus facile** à développer (classe fille = canevas à remplir)
  - **Contrat** sur les fonctionnalités (garanties)
  - Garanties sur des **classes qui n'existent pas encore** : facilités d'évolution du code
- Usage du polymorphisme
  - Tableau hétérogène $\Rightarrow$ + de possibilités

Etude de cas

On souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des figures géométriques. Il faudra pouvoir intégrer différents types de figures :

- Figures simples : Point, Segment, Droite
- Polygones : Triangle, Carré, Losange, Rectangle, Parallélépipède
- Courbes : Béziers, Lignes brisées, Cercles, Ellipses

et le logiciel pourra être étendu pour en ajouter de nouvelles figures. Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (figures fermées), et bien sûr afficher les figures.

Etude de cas : Identification des concepts/-classes

On souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des **figures géométriques**. Il faudra pouvoir intégrer différents types de figures :

- **Figures simples** : **Point**, **Segment**, **Droite**
- **Polygones** : **Triangle**, **Carré**, **Losange**, **Rectangle**, **Parallélépipède**
- **Courbes** : **Béziers**, **Lignes brisées**, **Cercles**, **Ellipses**

et le logiciel pourra être étendu pour en ajouter de nouvelles figures. Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (**figures fermées**), et bien sûr afficher les figures.

Etude de cas : Identification des concepts/-classes concrètes

Classes concrètes

Les classes concrètes sont les classes que l'on souhaitera effectivement **construire en mémoire**

On souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des **figures géométriques**. Il faudra pouvoir intégrer différents types de figures :

- **Figures simples** : Point, Segment, Droite
- **Polygones** : Triangle, Carré, Losange, Rectangle, Parallélépipède
- **Courbes** : Béziers, Lignes brisées, Cercles, Ellipses

et le logiciel pourra être étendu pour en ajouter de nouvelles figures. Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (**figures fermées**), et bien sûr afficher les figures.

Etude de cas : Identification des concepts/-classes concrètes

Classes abstraites

Les classes concrètes sont les classes que l'on souhaitera ne pas **construire en mémoire**

On souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des figures géométriques. Il faudra pouvoir intégrer différents types de figures :

- Figures simples : Point, Segment, Droite
- Polygones : Triangle, Carré, Losange, Rectangle, Parallélépipède
- Courbes : Béziers, Lignes brisées, Cercles, Ellipses

et le logiciel pourra être étendu pour en ajouter de nouvelles figures. Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (figures fermées), et bien sûr afficher les figures.

Identification des traitements

On souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des figures géométriques. Il faudra pouvoir intégrer différents types de figures :

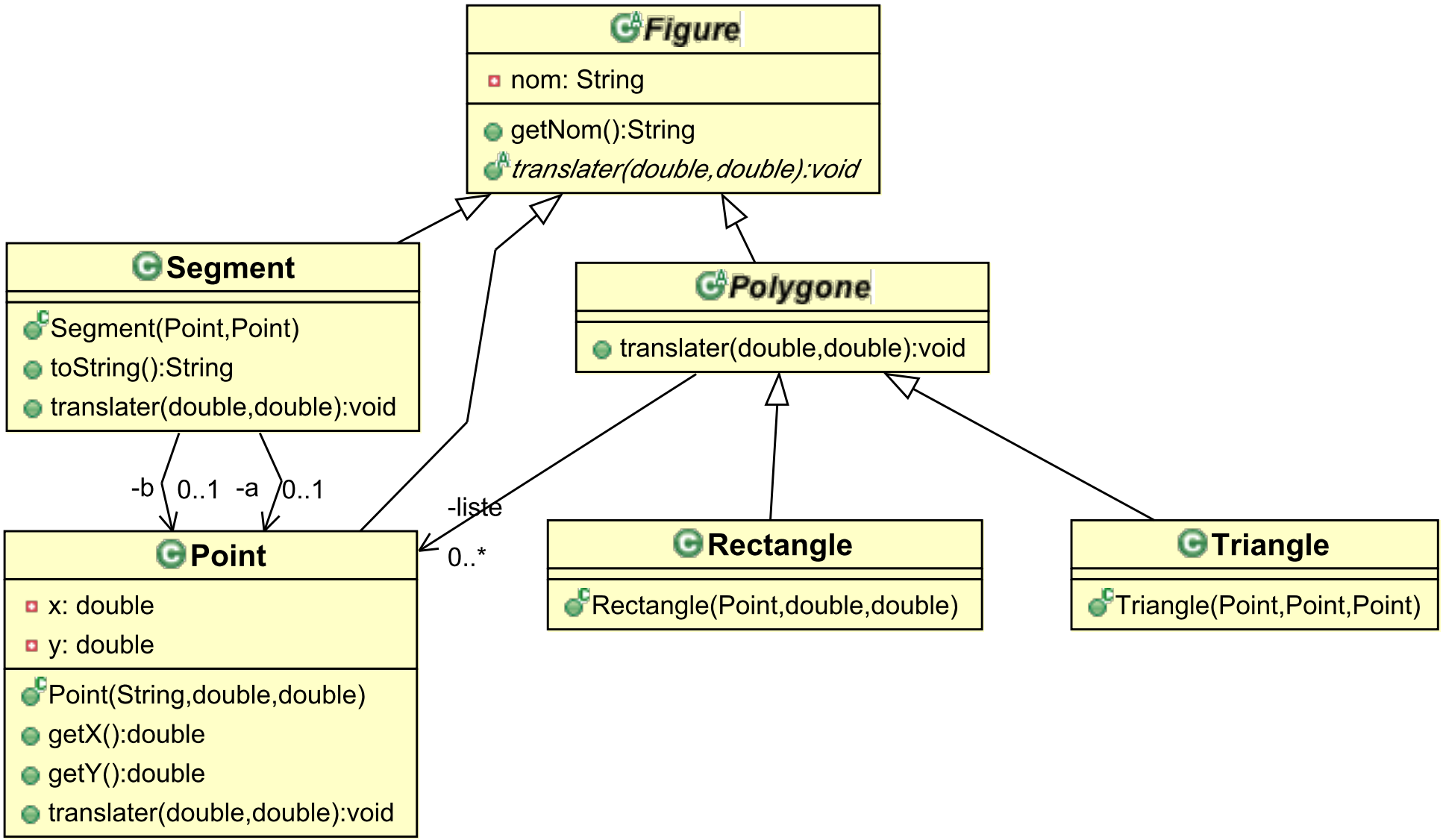
- Figures simples : Point, Segment, Droite
- Polygones : Triangle, Carré, Losange, Rectangle, Parallélépipède
- Courbes : Béziers, Lignes brisées, Cercles, Ellipses

et le logiciel pourra être étendu pour en ajouter de nouvelles figures. Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (figures fermées), et bien sûr afficher les figures.

Résultat : modèle objet

- **Concepts abstraits :** Classes abstraites + héritage éventuel
  - Figure, Polygone hérite de Figure, Courbe hérite de Figure
- **Concepts concrets :** Classes concrètes + héritage
  - Point hérite de Figure (idem Segment, Droite, LigneBrisee)
  - Triangle hérite de Polygone (idem Carre, Rectangle, etc.)
  - etc. (exercice)
- **Traitements = Méthodes** (privilégier les classes mères)
  - afficher et translater dans Figure
  - calculer la distance dans Point
  - calculer la longueur dans Segment
  - mettre à l'échelle, calculer la surface dans Polygone

Résultat : modèle objet



Code Java : Classe mère Figure

```java
public abstract class Figure {
   private String nom; // (1) attributs
        // (2) Constructeur protégé (pas d'instances)
   protected Figure(String nom) {
     this.nom = nom;
   }
   public String getNom() { // (3) méthode concrète
     return x;
   }
   public String toString() {
     return nom+": ";
   }
   //  (4) méthodes abstraites
   // (implémentation dans les sous-classes)
   public abstract void translater(double tx, double ty);
}
```

Contenu d'une classe abstraite

- Attributs privés (comme d'habitude)
- Constructeurs protégés
  - on ne peut pas construire d'instance mais il faut pouvoir initialiser les attributs depuis une sous-classe
- Méthodes implémentées
  - publiques : invocables depuis l'extérieure (vision client)
  - protégées : invocables dans les sous-classes (vision héritier)
  - privées : invocables dans la classe uniquement (vision fournisseur)
- Méthodes abstraites publiques ou protégées
  - pas d'implémentation (ex. calculer la surface d'une figure)

Contrat minimal de l'héritier

Si une classe B hérite d'une classe A il faut **impérativement** :

- **implémenter des constructeurs pour la sous-classe B**
  - première instruction du constructeur : appel à un constructeur de A via super
    - Il faut construire la partie de B « qui est un » A (+ pas d'accès aux attributs privés de A)
    - Exemple : dans Point(x,y) appel de super(nom) vers Figure
- **implémenter toutes les méthodes abstraites de la classe mère A**
  - exemple : Point hérite de figure et doit donc implémenter translater
  - cas particulier : si B est également abstraite

Usages croisés, polymorphisme

- Classe abstraite `Figure`
  - Constructeur sans argument
  - méthode abstraite `void translater (Point p)`
  - méthode abstraite `String getType ()`
- Classe `Point extends Figure`

- Est-ce possible de faire un usage croisé des classes ?
- Que doit-on faire dans `Point` ?
- Que penser des lignes suivantes :

```
1 Figure f ;
```

Usages croisés, polymorphisme

- Classe abstraite `Figure`
  - Constructeur sans argument
  - méthode abstraite `void translater (Point p)`
  - méthode abstraite `String getType ()`
- Classe `Point extends Figure`

- Est-ce possible de faire un usage croisé des classes ?
- Que doit-on faire dans `Point` ?
- Que penser des lignes suivantes :

```
1 Figure f ; OK
2 f  = new Figure();
```

Usages croisés, polymorphisme

- Classe abstraite `Figure`
  - Constructeur sans argument
  - méthode abstraite `void translater (Point p)`
  - méthode abstraite `String getType ()`
- Classe `Point extends Figure`

- Est-ce possible de faire un usage croisé des classes ?
- Que doit-on faire dans `Point` ?
- Que penser des lignes suivantes :

```
1 Figure f ; OK
2 f  = new Figure(); KO
3 Figure f2 = new Point(1,2);
```

Usages croisés, polymorphisme

- Classe abstraite `Figure`
  - Constructeur sans argument
  - méthode abstraite `void translater (Point p)`
  - méthode abstraite `String getType ()`
- Classe `Point extends Figure`

- Est-ce possible de faire un usage croisé des classes ?
- Que doit-on faire dans `Point` ?
- Que penser des lignes suivantes :

```
1 Figure f ; OK
2 f  = new Figure(); KO
3 Figure f2 = new Point(1,2); OK
4 Point p = new Point(2,3);  OK
5 p.translater(new Point(1,1));
```

Usages croisés, polymorphisme

- Classe abstraite `Figure`
  - Constructeur sans argument
  - méthode abstraite `void translater (Point p)`
  - méthode abstraite `String getType ()`
- Classe `Point extends Figure`

- Est-ce possible de faire un usage croisé des classes ?
- Que doit-on faire dans `Point` ?
- Que penser des lignes suivantes :

```
1 Figure f ; OK
2 f  = new Figure(); KO
3 Figure f2 = new Point(1,2); OK
4 Point p = new Point(2,3);  OK
5 p.translater(new Point(1,1)); OK
6 f2.translater(p);
```

Usages croisés, polymorphisme

- Classe abstraite `Figure`
  - Constructeur sans argument
  - méthode abstraite `void translater (Point p)`
  - méthode abstraite `String getType ()`
- Classe `Point extends Figure`

- Est-ce possible de faire un usage croisé des classes ?
- Que doit-on faire dans `Point` ?
- Que penser des lignes suivantes :

```
1 Figure f ; OK
2 f  = new Figure(); KO
3 Figure f2 = new Point(1,2); OK
4 Point p = new Point(2,3);  OK
5 p.translater(new Point(1,1)); OK
6 f2.translater(p); OK
7 p.translater(f2);
```

Usages croisés, polymorphisme

- Classe abstraite `Figure`
  - Constructeur sans argument
  - méthode abstraite `void translater (Point p)`
  - méthode abstraite `String getType ()`
- Classe `Point extends Figure`

- Est-ce possible de faire un usage croisé des classes ?
- Que doit-on faire dans `Point` ?
- Que penser des lignes suivantes :

```
1 Figure f ; OK
2 f  = new Figure(); KO
3 Figure f2 = new Point(1,2); OK
4 Point p = new Point(2,3);  OK
5 p.translater(new Point(1,1)); OK
6 f2.translater(p); OK
7 p.translater(f2); KO
```

Un dynamisme étonnant

- Classe abstraite `Figure`
  - Constructeur sans argument
  - méthode abstraite `void translater (Point p)`
  - méthode abstraite `String getType ()`
  - méthode `String toString()` (1)
- Classe `Point extends Figure`
  - méthode `String toString()` (2)

1. Je peux faire appel à `getType` dans la classe `Figure` !

```
1 // Dans Figure
2 public void afficher(){
3   System.out.println("Je␣suis␣de␣type:␣" + getType());
4 }
```

2. Code mystère :

```
1 // Dans figure
2 public void afficher2(){
3   System.out.println(toString());
4 }
5 // dans le main
6 Point p = new Point(1,2); // ou Figure p
7 p.afficher2(); // (1) ou (2)
```

Un dynamisme étonnant

- Classe abstraite `Figure`
  - Constructeur sans argument
  - méthode abstraite `void translater (Point p)`
  - méthode abstraite `String getType ()`
  - méthode `String toString()` (1)
- Classe `Point extends Figure`
  - méthode `String toString()` (2)

1. Je peux faire appel à getType dans la classe `Figure` !

```
1 // Dans Figure
2 public void afficher(){
3   System.out.println("Je␣suis␣de␣type:␣" + getType());
4 }
```

2. Code mystère :

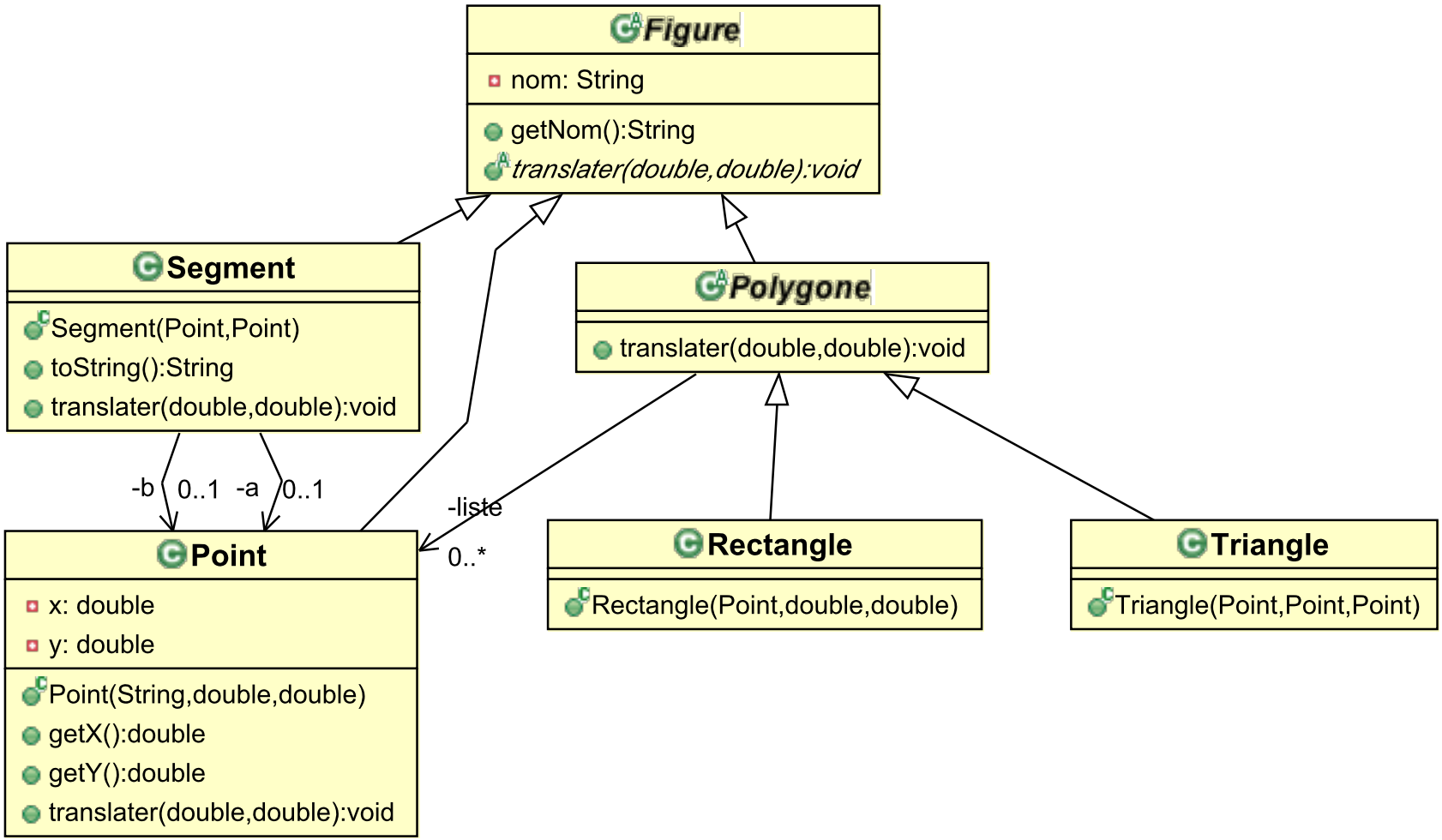
```
1 // Dans figure
2 public void afficher2(){
3   System.out.println(toString());
4 }
5 // dans le main
6 Point p = new Point(1,2); // ou Figure p
7 p.afficher2(); // (1) ou (2) ⇒ (2)!!
```

2i002 - Conversion des types (cast...)

Vincent Guigue - vincent.guigue@lip6.fr

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

Exemple du logiciel de dessin



Récupérer le type d'une instance dynamiquement

Cas amusant :

le type des instances est parfois (souvent) inconnu

Exemple :

```
1 Figure f;
2 if(Math.random()>0.5)
3   f = new Point(2,3);
4 else
5   f = new Segment(new Point(1,2), new Point(5,3));
6
7 // pour le compilateur f est de type Figure
8 // pour la JVM, quel est le type de l'instance f ???
```

Récupérer le type d'une instance dynamiquement

- Les limites du polymorphisme sont imposées par le compilateur qui vérifie (statiquement) le type des variables

1) Contournement = connaitre le type des instances en cours de programme (dynamiquement)
2) Revenir au type de l'instance pour accéder aux méthodes spécifiques = faire un cast sur la variable

Exemple :

```
1 Figure f;
2 if(Math.random()>0.5)
3   f = new Point(2,3);
4 else
5   f = new Segment(new Point(1,2), new Point(5,3));
6
7 // pour le compilateur f est de type Figure
8 // pour la JVM, quel est le type de l'instance f ???
```

1) Récupération des informations : instanceof

variable **instanceof** *NomClasse*

$\Rightarrow$ retourne un `boolean`

```
1  Figure f;
2  if(Math.random()>0.5)
3    f = new Point(2,3);
4  else
5    f = new Segment(new Point(1,2), new Point(5,3));
6
7  if(f instanceof Point)
8     System.out.println("C'est un Point");
9  else
10    System.out.println("C'est un Segment");
```

- Syntaxe particulière
- Souvent, il existe d'autre moyen de faire...
  Globalement, sauf exception :

  `instanceof` = mauvaise programmation

Procédure qui spécialise ses traitements par type de figure

```
1  public static void afficheType(Figure f) {
2    if(f instanceof Point)
3        System.out.println("C'est un Point");
4    else if(f instanceof Segment)
5        System.out.println("C'est un Point");
6    else if(f instanceof Rectangle)
7        System.out.println("C'est un Rectangle");
8      // etc ... un cas par figure !
```

Que se passe-t-il si on ajoute un nouveau type de Figure ?

Il faut toucher au code utilisateur ! ! !

$\Rightarrow$ l'outil existe, mais souvent, il faut mieux **ne pas l'utiliser** !

instanceof : Solution

Code spécifique dans les classes :

- dans Figure :

```
1 public abstract String getTypeFigure();
```

- dans Point :

```
1 public String getTypeFigure() { return "Point"; }
```

- dans Rectangle :

```
1 public String getTypeFigure() { return "Rectangle"; }
```

- Code générique (éventuellement en dehors des classes) :

```
1    public static void afficheType(Figure f)
2    {
3      System.out.println("C'est un "+f.getTypeFigure());
4     }
```

INSTANCEOF : ATTENTION À LA HIERARCHIE

```java
public static void main(String[] args) {
    Point p1 = new Point("toto", 0, 2);
    Point p2 = new Point( "toto 2", 3, 2);
    Figure f = new Segment(p1, p2);

    if(f instanceof Point)
        System.out.println("f est un Point");
    if(f instanceof Segment)
        System.out.println("f est un Segment");
    if(f instanceof Figure)
        System.out.println("f est une Figure");
}
```

Le programme suivant retourne :

```
f est un Segment
f est une Figure
```

Le résultat est logique : il faut comprendre `instanceof` comme «EST UN ?»

Alternative à instanceof : getClass()

getClass : Class

- Méthode de `Object`
- S'utilise sur une instance (syntaxe différente de `instanceof`)
- Retourne la classe de l'instance

```
1 Figure f = new Segment(p1, p2);
2 System.out.println("f est de type : "+f.getClass());
3 // retour :
4 // f est de type : class cours2.Segment
```

Usage classique pour comparer le type de deux instances :

```
1 // soit deux Objets obj1 et obj2
2 if (obj1.getClass() != obj2.getClass())
3   ...
```

2) Modifier le type d'une variable : Cast

Cast = 2 modes de fonctionnement

- Conversion sur les types basiques : le codage des données change. Souvent implicite dans votre codage...

```
1 double d = 1.4;
2 int i = (int) d;  // i=1
```

- Conversion dans les hiérarchies de classes :

  la variable est modifiée, l'instance est inchangée

```
1 Figure f = new Segment(p1, p2);
2 Segment s = (Segment) f;
3 // Pour vérifier que vous avez compris:
4 // donner un diagramme mémoire
```

- Utile pour accéder aux méthodes spécifiques d'une instance
- Dangereux : aucun contrôle du compilateur...

Cast : limite

Un système peu sécurisé à la compilation :

```
1 Point p1 = new Point("toto", 0, 2);
2 Point p2 = new Point( "toto␣2", 3, 2);
3 Figure f = new Segment(p1, p2);
4
5 Figure f2 = (Figure) p1;
```

Cast : limite

Un système peu sécurisé à la compilation :

```
1 Point p1 = new Point("toto", 0, 2);
2 Point p2 = new Point( "toto␣2", 3, 2);
3 Figure f = new Segment(p1, p2);
4
5 Figure f2 = (Figure) p1; // OK mais inutile
6 Figure f3 = p1; // OK Subsomption classique
7 Segment s = (Segment) f;
```

Cast : limite

Un système peu sécurisé à la compilation :

```
1 Point p1 = new Point("toto", 0, 2);
2 Point p2 = new Point( "toto 2", 3, 2);
3 Figure f = new Segment(p1, p2);
4
5 Figure f2 = (Figure) p1; // OK mais inutile
6 Figure f3 = p1; // OK Subsomption classique
7 Segment s = (Segment) f; // compilation OK
8 Point p3 = (Point) f;
```

Cast : limite

Un système peu sécurisé à la compilation :

```
1 Point p1 = new Point("toto", 0, 2);
2 Point p2 = new Point( "toto␣2", 3, 2);
3 Figure f = new Segment(p1, p2);
4
5 Figure f2 = (Figure) p1; // OK mais inutile
6 Figure f3 = p1; // OK Subsomption classique
7 Segment s = (Segment) f; // compilation OK
8 Point p3 = (Point) f; // compilation OK (!)
```

Exécution :

Crash du programme avec le message suivant

```
1 Exception in thread "main" java.lang.ClassCastException:
2 cours2.Segment cannot be cast to cours2.Point
3         at cours2.Test.main(Test.java:26)
```

Cast : avec plusieurs niveaux de hiérarchie

```
1  // subsomption
2  Figure f = new Segment(p1, p2);
3  Figure f2 = new Carre(p1, cote);
4  Figure f3 = new Triangle(p1, p2, p3);;
5  Polygone p = new Triangle(p4, p5, p6);
6
7  // cast OK
8  Triangle t = (Triangle) p; // OK (comme précédemment)
9
10 Polygone p2 = (Polygone) f2; // OK compil + exec:
11                             // un Carre EST UN Polygone
12
13 Polygone p3 = (Triangle) f3; // OK:
14         // f3 est un Triangle => conversion OK (JVM)
15         // p3 peut référencer un Triangle OK (compilé)
16
```

Cast : avec plusieurs niveaux de hiérarchie

```
1  // subsomption
2  Figure f = new Segment(p1, p2);
3  Figure f2 = new Carre(p1, cote);
4  Figure f3 = new Triangle(p1, p2, p3);;
5  Polygone p = new Triangle(p4, p5, p6);
6
7  // cast OK
8  Triangle t = (Triangle) p; // OK (comme précédemment)
9
10 Polygone p2 = (Polygone) f2; // OK compil + exec:
11                             // un Carre EST UN Polygone
12
13 Polygone p3 = (Triangle) f3; // OK:
14         // f3 est un Triangle => conversion OK (JVM)
15         // p3 peut référencer un Triangle OK (compilé)
16
17 // cast KO
18 Carre c = (Carre) f; //KO JVM
19 Cercle c = (Cercle) p; //KO Compil : opération impossible
```

Cast : sécurisation

Idée

Vérifier le type de l'instance avant la conversion

```
1        Figure f = new Segment(p1, p2);
2        Segment s;
3        if(f instanceof Segment)
4            s = (Segment) f;
```

$\Rightarrow$ vous utiliserez systématiquement cette sécurisation

Cast : usage dans equals

fonction equals

Il y a toujours un cast (sécurisé) dans equals pour pouvoir accéder aux attributs à comparer

Exemple sur la classe Point :

```
1   public boolean equals(Object obj) { // V1
2       if (this == obj)
3           return true;
4       if (obj == null)
5           return false;
6       if (getClass() != obj.getClass())
7           return false;
8       Point other = (Point) obj;
9       if (x != other.x)
10          return false;
11      if (y != other.y)
12          return false;
13      return true;
14  }
```

getClass *vs* instanceof

Imaginons la redéfinition suivante pour `equals` :

```
1     public boolean equals(Object obj) {// V2
2         if (this == obj)
3             return true;
4         if (obj == null)
5             return false;
6         // if (getClass() != obj.getClass()) en V1
7         if (!(obj instanceof Point))
8             return false;
9         Point other = (Point) obj;
10        if (x != other.x)
11            return false;
12        if (y != other.y)
13            return false;
14        return true;
15    }
```

Quel est le défaut de l'implémentation v2 ?

Prise en défaut :

```
1 Point p = new Point(1,2);
2 PointTrafique p2 = new PointTrafique("toto",1,2);
3 if(p.equals(p2))
4                 System.out.println("ils␣sont␣égaux␣!!!␣");
5 if(p2.equals(p))
6                 System.out.println("et␣ici␣???");
```

Avec :

```
1 public class PointTrafique extends Point{
2     private String qqch;
3     public PointTrafique(String nom, double x, double y) {
4         super( x, y);
5         qqch = nom; // un attribut en plus
6     }
7 }
```

V1 : pas d'égalité

V2 : égalité détectée... Est ce légitime ?
Pb : quid de la symétrie ?

2i002 - Gestion de projets, package, compilation, IDE

Vincent Guigue

Introduction

Bonne architecture = beaucoup de petites classes...
... chacune étant ciblée, lisible, ré-utilisable
⇒ Le repertoire de projet devient rapidement illisible !

Solution = arborescence de répertoires

- Sous-repertoires associés aux concepts de bas niveaux,
- Sous-sous-répertoires de test

Exemple

Gestion d'une course de voiture autonomes

1. Réfléchir à un découpage de bas niveau :
   - **Circuit**
   - **Voiture**
   - Autonome ⇒ gestion de l'**IA** / **stratégies**
2. Ajouter les outils (transverses)
   - Gestion de la **géométrie**
   - Gestion des fichiers (sauvegardes/chargements)
   - Interface graphique (IHM)

Exemple

Gestion d'une course de voiture autonomes

1. Réfléchir à un découpage de bas niveau :
   - **Circuit**
   - **Voiture**
   - Autonome ⇒ gestion de l'**IA** / **stratégies**
2. Ajouter les outils (transverses)
   - Gestion de la **géométrie**
   - Gestion des fichiers (sauvegardes/chargements)
   - Interface graphique (IHM)
3. Package de test :

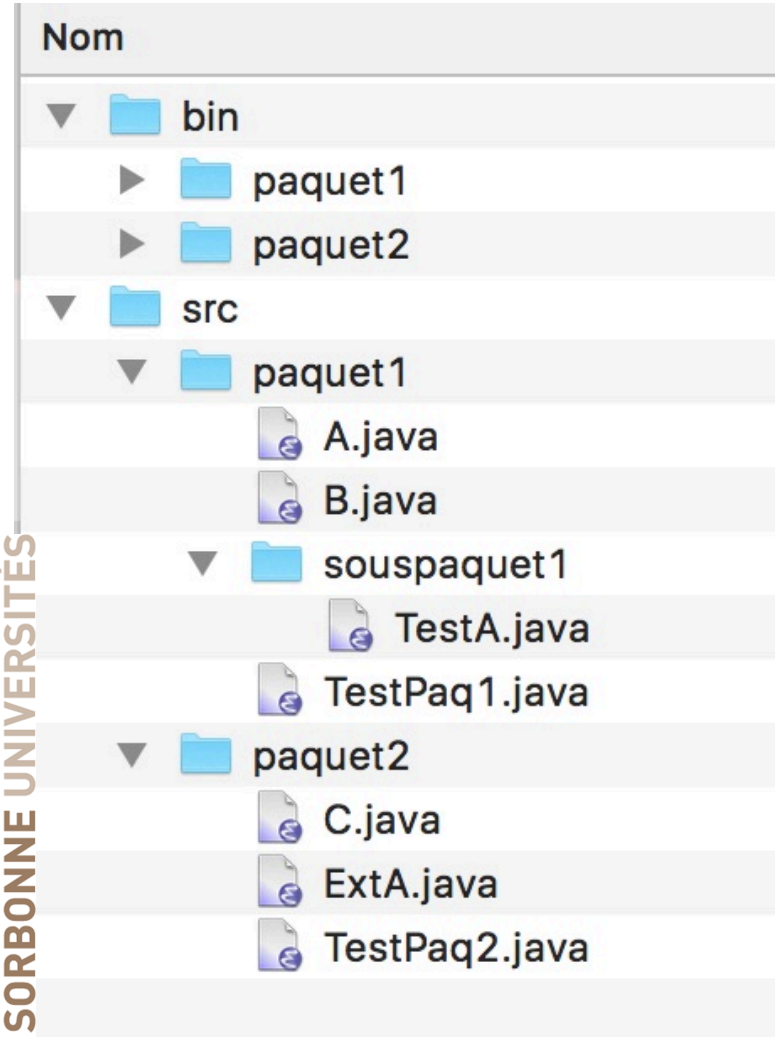
Idée :

valider le fonctionnement de chaque objet indépendamment du reste du projet (dans la mesure du possible).

⇒ **sous-répertoire de test** dans chaque package principal

Déclarations obligatoires

Arborescence :



1 Déclaration de paquet

```
1 // Fichier A.java
2 package paquet1;
3 public class A {
4 ...
```

2 Déclaration d'import (pour les classes de paquets différents)

```
1 package paquet2;
2 import paquet1.A;
3 public class ExtA extends A{
4     public ExtA() {
5         super();
6     }
```

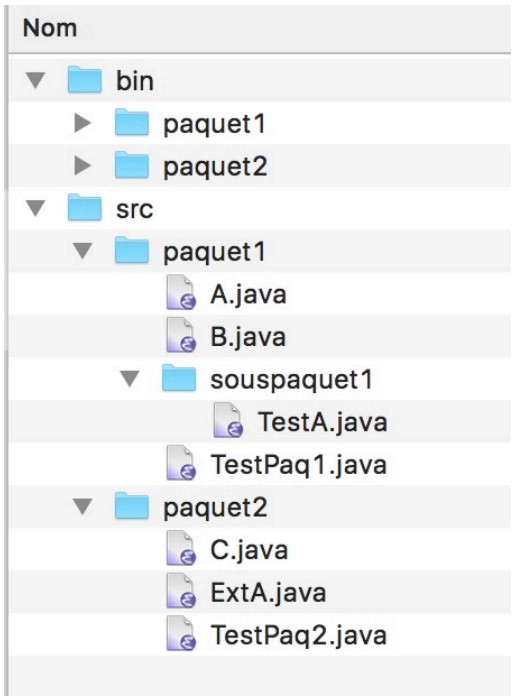
3 Sous-package

```
1 package paquet1.souspaquet1;
2 public class TestA {
3     public static void main(String[] args) {
4         // tests spécifiques à A
```

4 Classe JDK

```
1 import java.util.ArrayList;
```

Compilation / exécution du code



- **Compilation** (position = racine)
  - Spécification d'un répertoire cible : `-d`
  - Spécification du répertoire de gestion des sources : `-cp`

```
» javac -cp src -d bin src/paquet1/TestPaq1.java
```

⇒Compile l'exécutable + toutes les dépendances

- **Exécution**
  - Instruction pour se positionner dans le répertoire d'exécution : `-cp`
  - Chemin avec des `.` (pas des `/`)

```
» java -cp bin paquet1.TestPaq1
```

ou

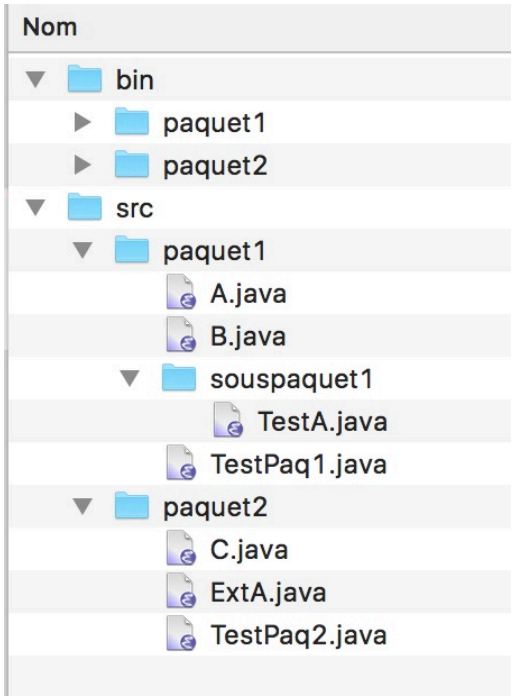
```
» cd bin
» java paquet1.TestPaq1
```

Niveaux de visibilité

introduction des packages = subtilités sur la visibilité

```java
package paquet1;
public class A {
  public int i;      // public
  protected int j;   // protected
  private int k;     // private
  int n;             // package (nouveau)

  public A(){
      i=1; j=2; k=3; n=4;
  }
}
```



Visibilités des attributs de `A` depuis :

		i	j	k	n
Même répertoire	`B, TestPaq1`	√	√	×	√
Classe fille	`ExtA`	√	√	×	×
Autres cas	`C,TestPaq2,TestA`	√	×	×	×

2i002 - Final

Vincent Guigue

`final` ⇒ ATTRIBUTS STATIC = CONSTANTES

Idée

Pour sécuriser le code, interdisons les modifications de certaines valeurs (notamment les constantes).

Exemple : `Math.PI`, sécurisation = impossibilité de modifier

Note : une constante est indépendante des instances ⇒ `static`

Usage : une constante est définie en majuscule

final ⇒ ATTRIBUTS STATIC = CONSTANTES

Idée

Pour sécuriser le code, interdisons les modifications de certaines valeurs (notamment les constantes).
Exemple : `Math.PI`, sécurisation = impossibilité de modifier
Note : une constante est indépendante des instances ⇒ `static`
Usage : une constante est définie en majuscule

```
1 public class MaClasse{
2   public final static int MACONSTANTE = 10;
3   ...
```

Usage :

- Constantes *universelles* (`Color.RED`, `Color.YELLOW`, `Math.PI`, `Double.POSITIVE_INFINITY`...)
- Typologie (type de codage d'un pixel, organisation du BorderLayout)...
- Bornes algorithmiques (`NB_ITER_MAX`, `TAILLE_MAX`...)

final POUR LES ATTRIBUTS

Idée : protéger ses objets... Et ses programmes

Initialiser les valeurs des attributs sans pouvoir les modifier ensuite

Exemple : `String`

```
1 public class Point{
2   public final double x,y;
3   public Point(double x, double y){
4     this.x = x; this.y = y;
5   }
6   // interdiction de modifier x, y dans la suite
7   // (et chez le client)
8 }
```

- Interdiction de modifier `x` et `y` dans les méthodes (pas de setter, pas de translation...)
- Modification d'un `Point` = création d'une nouvelle instance
- Possibilité de laisser les attributs `public`... Puisque non modifiable
- Sécurité lorsqu'un objet est passé en argument de méthode

Autres usages (liés à l'héritage)

- Méthode final : qui ne peut pas être redéfinie dans les classes filles
- Classe final : qui ne peut pas être étendue (String, Integer, Double...)

```
1 public class Point{
2   public final double getX(){ ... }
3 }
4
5 public class PointNomme extends Point{
6 ...
7 // Compilation impossible
8   public double getX(){ ... }
9 }
```

Autres usages (liés à l'héritage)

- Méthode final : qui ne peut pas être redéfinie dans les classes filles
- Classe final : qui ne peut pas être étendue (String, Integer, Double...)

```java
public class Point{
  public final double getX(){ ... }
}

public class PointNomme extends Point{
...
// Compilation impossible
  public double getX(){ ... }
}
```

```java
public final class Point{
  ...
}

// Compilation impossible
public class PointNomme extends Point{
}
```