2I002
HÉRITAGE:
LES CLASSES ABSTRAITES

Vincent Guigue

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

SCIENCES SORBONNE UNIVERSITÉ

LIP6

MLIA Machine Learning & Deep Learning for Information Access

- **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()`

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 methodes 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

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

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.

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.

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.

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.

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

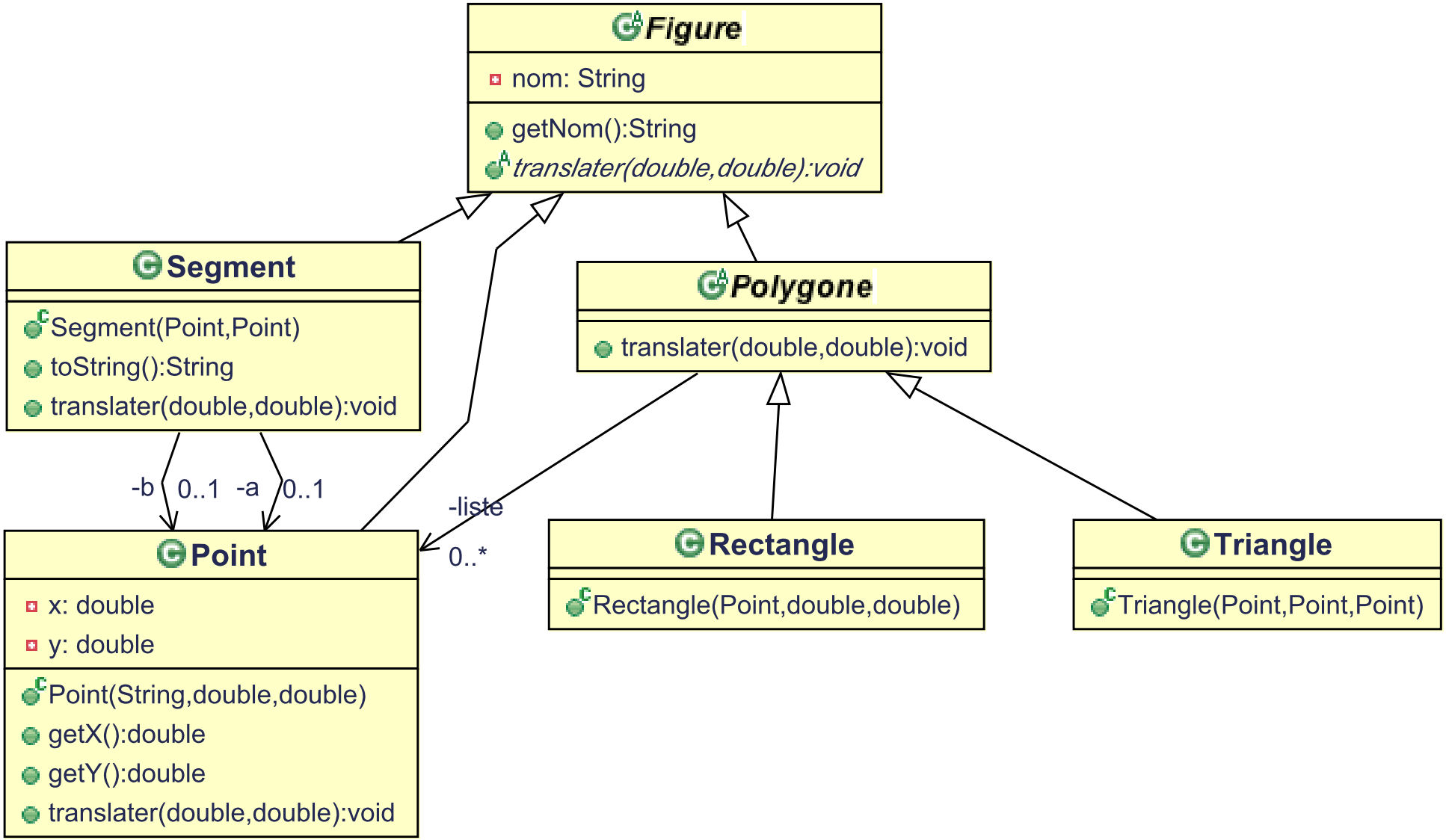
- Figures simples: Point, Segment, Droite
- Polygones: Triangle, Carré, Losange, Rectangle, Parallélépipède
- Courbes: Béziers, Lignes brisées, Cercles, Ellipses

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 protege (pas d'instances)
   protected Figure(String nom) {
     this.nom = nom;
   }
   public String getNom() { // (3) methode concrete
     return x;
   }
   public String toString() {
     return nom+": ";
   }
   //  (4) methodes abstraites
   // (implementation 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();
```

- 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);
```

- 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));
```

- 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) !!
```