Programmation orientée objets

Leçon 4 : Interfaces et Classes abstraites

Prof. Rolf Ingold

Département d'informatique

UNI
FR

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Contenu de la leçon

- Classe abstraite
- Méthode abstraite
- Concept d'interface
- Définition d'une interface
- Implémentation d'une interface
- Interface vue comme un type
- Héritage simple vs. héritage multiple

- Illustration avec des objets graphiques

Objectifs de la leçon

- Comprendre les notions de méthodes et de classes abstraites
- Comprendre la notion d'interface
- Etre capable de déclarer et d'implémenter une interface
- Savoir utiliser les interfaces commes des types

Classes abstraites

- En java, il est possible de définir des classes incomplètes
  - qui ne peuvent pas être instanciées
  - mais pouvant servir de superclasse

- On les appelle des **classes abstraites** et elles sont définies avec le modificateur `abstract`
  - par opposition, on parle de **classes concrètes** pour les autres classes

- Une classe abstraite peut contenir les mêmes définitions qu'une classe concrète
  - y compris des constructeurs, qui peuvent être invoqués dans les sous-classes à l'aide `super(...)`
  - surtout , elle peut contenir une ou plusieurs **méthodes abstraites**

UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Méthodes abstraites

- Dans une classe abstraite, on peut déclarer des méthodes incomplètes
  - qui ont un en-tête, mais pas de corps
  - qui doivent être redéfinies dans les sous-classes

- On les appelles des **méthodes abstraites** et elles sont définies avec le modificateur `abstract` et leur corps est remplacé par `;`
  - par opposition, on parle aussi de **méthodes concrètes** pour désigner des méthodes avec un corps

- Les méthodes abstraites ne peuvent être déclarées que dans des classes abstraites

Exemple : classe abstraite avec deux sous-classes

```
public abstract class GraphicObject {
    protected int x, y;
    public abstract void draw(Canvas canvas);
    public void translate(int dx, int dy) {
        x = x + dx; y = y + dy;
    }
}
public class Rectangle extends GraphicObject {
    protected int width, height;
    public Rectangle(int xMin, int xMax, int yMin, int yMax) {
        x = xMin; y = yMin;
        width = xMax - xMin; height = yMax - yMin;
    }
    public void draw(Canvas canvas) {
        canvas.drawLine(x, y, x + width, y);
        canvas.drawLine(x + width, y + height);
        ...
    }
}
public class Circle extends GraphicObject {
    ...
}
```


UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Introduction à la notion d'interface

- Pour l'organisation de logiciels complexes, il est important de pouvoir établir des règles d'interaction entre les objets
- La notion d'interface permet d'établir un contrat entre le contenu d'une classe et ses classes utilisatrices ou classes-clientes

- Concrètement, une **interface** déclare essentiellement des méthodes sans corps (comme les méthodes abstraites)
  - elle ne définit pas de méthodes concrètes
  - mais elle peut contenir
    - des champs statiques
    - des méthodes statiques
    - un initialiseur statique
  - enfin, elle peut définir des **méthodes par défaut** (nouveau, depuis la version 8, 2014)

Caractéristiques d'une interface

- Une interface a des points communs avec les classes abstraites
  - elle définit un type
  - elle n'est pas instantiable
  - elle définit de manière abstraite les méthodes associées à ce type

- Elle se différencie des classes abstraites de la manière suivante
  - elle est déclarée avec le mot clé `interface` (sans `abstract`)
  - une classe "cliente" peut **implémenter** une ou plusieurs interfaces en utilisant le mot-clé `implements`
  - elle doit alors définir toutes les méthodes qui y sont déclarées (sauf si elle est elle-même abstraite)

- Si une classe implémente une interface, ses objets sont compatibles avec le type associé à cette interface

Définition d'une interface

- La forme générale d'une interface est

```
[modifiers] interface InterfaceName [extends other, ...]  {
    // static fields
    [modifiers] static type name;
    ...
    // method declarations without body
    [modifiers] type name(arguments);
    ...
}
```

  - tous les champs statiques sont implicitement publics et constants
  - toutes les méthodes sont implicitement publiques
  - une **méthode par défaut** peut être définie; elle est introduite avec le modificateur `default` (leur utilisation est toutefois controversée)
  - une interface peut éventuellement avoir des méthodes statiques et un bloc d'initialisation statique
- Une interface est en principe définie dans un fichier ayant le même nom et l'extension `.java` (comme une classe)
  - il est possible de déclarer des interfaces internes à une classe

Exemple d'interfaces

```
public interface Translatable {
    public void translate(double dx, double dy);
    public void translateHorizontally(double dx);
    public void translateVertically(double dy);
}

public interface Rotatable {
    public void rotate(double x0, double y0, double angle);
}

public interface Scalable {
    public void scale(double scale);
}

public interface Transformable extends Translatable, Rotatable,
                                       Scalable {
}
```

Implémentation d'une interface

- Toute classe Java peut implémenter une ou plusieurs interfaces
- Pour cela il faut compléter la déclaration de classe avec une clause d'importation de la forme

```
modifiers class MyClass ... implements Interf1,... {
    ...
}
```

- La classe `MyClass` doit implémenter toutes les méthodes déclarées dans `Interf1, ...` (sauf si elle est abstraite)

Exemple de classe implémentant des interfaces

```
public class Rectangle implements Translatable, Scalable {
    public void translate(double dx, double dy) {
        x += dx;
        y += dy;
    }
    public void translateHorizontally(double dx){
        x += dx;
    }
    public void translateVertically(double dy) {
        y += dy;
    }
    public void scale(double scale) {
        x += (1 - scale) * width / 2.0;
        y += (1 - scale) * height / 2.0;
        width *= scale;
        height *= scale;
    }
    ...
}
```

Utilisation d'une interface comme type

- Le nom d'une interface peut être utilisé comme un type
  - dans une déclaration de variable
  - comme paramètre formel d'une méthode

- On peut alors lui associer tout objet d'une classe qui
  - implémente cette interface
  - ou qui est sous-classe d'une telle classe

- Par exemple, les instance de toutes les classes (et toutes leurs sous-classes) qui implémentent l'interface **`Scalable`** sont de type **`Scalable`**
  - elles peuvent alors invoquer la méthode **`scale(value);`**

Héritage multiple de types

- En Java une classe peut implémenter plusieurs interfaces ce qui permet un **héritage mutiple de types**

- Par contre, Java (contrairement à d'autres langages) n'a **pas d'héritage multiple de classes**
  - pour éviter l'héritage multiple de variables d'instance qui peut conduire à des inconsistances

l'argument ne convinct pas complètement puisque, avec Java 8, il peut y avoir des méthodes par défaut qui engendrent potentiellement les mêmes problèmes !

L'interface Cloneable

- Rappel du cours précédent : la méthode **`clone()`** définie dans la classe **`Object`** est applicable seulement si l'objet appartient à une classe qui implémente **`Cloneable`**
- Comme l'interface **`Cloneable`** est vide
  - elle ne déclare pas la méthode **`clone()`**
  - celle-ci est définie dans la classe **`Object`**
- En conséquence, pour clôner un objet, il suffit de spécifier
  - que sa classe implémente **`Cloneable`**

- Le comportement par défaut de **`clone()`** est de copier l'ensemble des champs
  - s'il s'agit d'objets ce sont les références qui sont copiés
  - idem pour les tableaux
- Si ce comportement ne convient pas, on peut redéfinir la méthode **`clone()`**

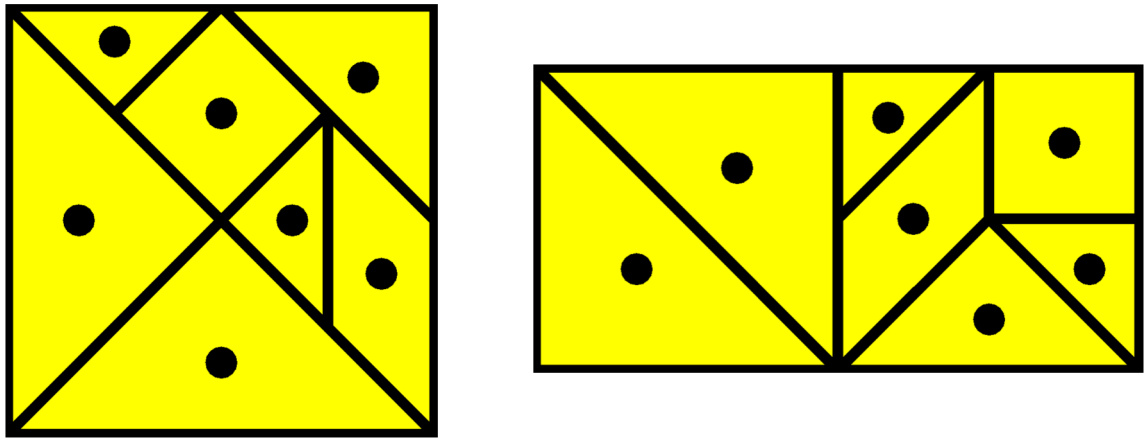
Etude de cas : objets graphiques

- Pour illustrer les différents concepts nous allons considérer une bibliothèque permettant de travailler avec des objets graphiques

- Les **objets graphiques** sont des éléments géométriques que l'on peut afficher dans une fenêtre puis déplacer
- Parmi les formes communes on considère les cercles, les carrés, les rectangles, les triangles et d'autres polygones
- Ces objets sont donc caractérisés par
  - le type (cercle, carré, etc.)
  - les dimensions (longueurs et angles)
  - la couleur
  - une position dans la fenêtre (par rapport à un système de coordonnées)
- Ils peuvent subir les transformations euclidiennes (translation, rotation, changement d'échelle)

Exemple d'utilisation

- De nombreuses applications utilisent des objets graphiques
  - les animations (type Flash)
  - les jeux, par exemple le **tangram**

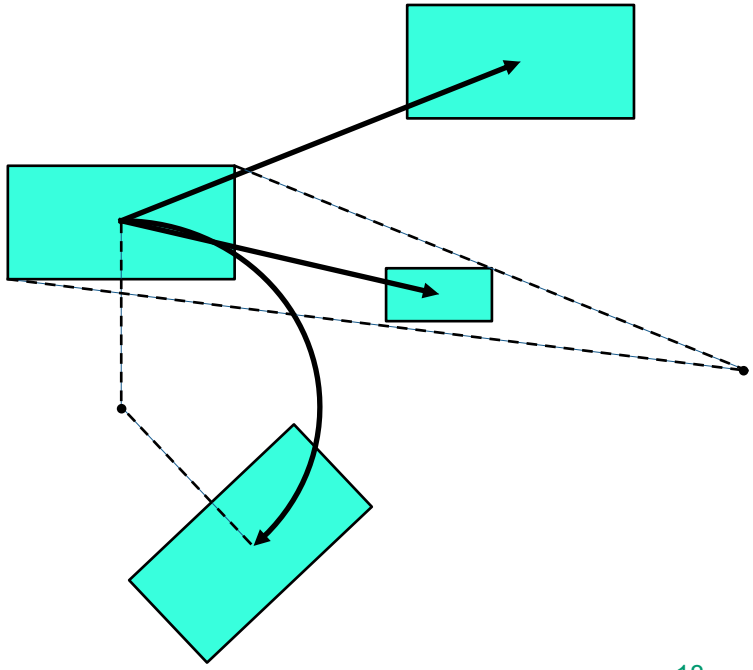


  - les programmes de simulation
  - etc.

Analyse conceptuelle

- Lors de la création les objets graphiques sont définies par
  - les caractéristiques géométriques (forme et dimensions)
  - la couleur
  - la position, définie pour le centre de l'objet

- Comme opérations, on considère les **transformations euclidiennes**, soit
  - la **translation** selon un vecteur
  - la **rotation** autour d'un centre de rotation
  - le **changement d'échelle**, par rapport à un centre de référence



UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Point d'ancrage

- Afin d'augmenter la flexibilité, chaque objet graphique est muni d'un **point d'ancrage**
  - ce point d'ancrage sert notamment de point de référence par défaut pour les rotations et les changement d'échelle

- Lors de l'initialisation le point d'ancrage est placé au **centre** de l'objet, défini de la manière suivante
  - pour le cercle le centre
  - pour un polygone, le barycentre des sommets (à ne pas confondre avec le centre de gravité de la surface)

- Le point d'ancrage peut être déplacé

Gestion de l'affichage

- L'affichage des objets est réalisé par la librairie **`oop.lib`**

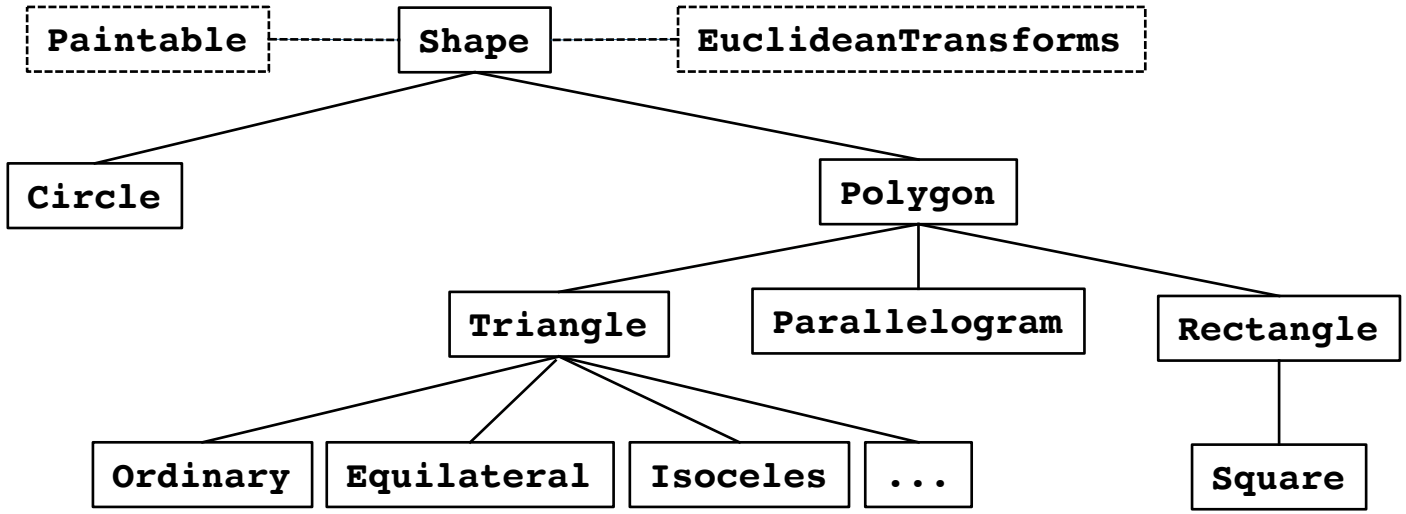
- Rappel : la classe **`Animation`** définit une fenêtre dans laquelle on ajoute les objets graphiques

- Ces objets graphique doivent implémenter l'interface **`Paintable`**, c'est à dire définir une méthode **`paint(painting)`**
  - où **`painting`** est un objet qui donne accès aux méthodes permettant de dessiner, soit

```
painting.setColor(Color color)
painting.fillCircle(double[] center, double radius)
painting.fillPolygon(double[][] vertices)
```

Hiérarchie naturelle des objets graphiques

- Il s'agit d'organiser les types d'objets graphiques selon une hiérarchie qui permet de tirer profit de l'héritage
  - au sommet de la hiérarchie on a une classe abstraite **`Shape`** qui déclare les interfaces **`Paintable`** et **`EuclideanTransforms`**



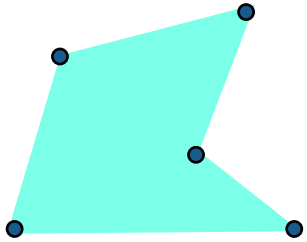
- Remarque : bien que mathématiquement un rectangle soit un cas particulier de parallélogramme, cette hiérarchie considère que les deux classes sont dérivées directement de la classe polygone

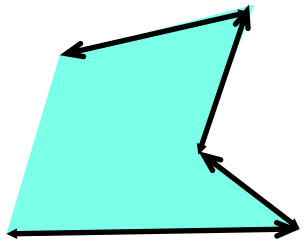
Classe `Shape`

- La classe Shape est une classe abstraite (non instanciable)
  - elle déclare implémenter les interfaces **`Paintable`** et **`EuclideanTransforms`**
  - elle implémente les fonctionnalités communes de tous les objets graphiques
    - **`setColor(color)`** pour définir la couleur
    - **`getAnchor()`** pour accéder au point d'ancrage
  - elle définit des méthodes abstraites additionnelles
    - **`center()`** qui définit le centre de l'objet
    - **`contains(point)`** qui détermine si un point se trouve à l'intérieur de l'objet ou pas
  - elle définit également un comportement par défaut de **`rotate(angle)`** et **`scale(factor)`**, avec un seul argument
  - elle contient un créateur protégé qui initialise la couleur et le point d'ancrage

Classes `Circle` et `Polygon`

- Les classes **`Circle`** et **`Polygon`** sont les descendants directs de **`Shape`**
  - pour la création d'un **cercle** il faut spécifier son diamètre, sa couleur et sa position (du centre)
  - pour la création d'un **polygone**, il faut spécifier la forme (soit la liste des sommets, soit un contour), la couleur et la position (du centre)





  - attention la liste des sommets est surdéterminée (elle définit le polygone à un translation près !)
- Les classes implémentent **`Painting`** et **`EuclideanTransforms`** ainsi que les méthodes abstraites de **`Shape`**

Classes `Parallelogram`, `Rectangle` et `Square`

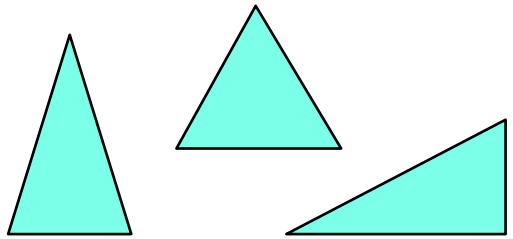
- Les classes **`Prallelogram`**, **`Rectangle`** et **`Square`** représentent des objets qui sont des cas particuliers de polygones
  - elles peuvent hériter de toutes leurs fonctionnalités
  - seuls les créateurs doivent être définis

- La classe Square est même définie comme une classe dérivée de Rectangle et s'écrit simplement

```
public class Square extends Rectangle {
    public Square(double size, Color clr, Point pt) {
        super(size, size, clr, pt);
    }
}
```

Classes `Triangle`

- La classe **`Triangle`** permet de travailler avec différents types de triangles aux propriétés remarquables
  - les triangle équilatéraux
  - les triangles isocèles
  - les triangles rectangles



  en plus des triangles ordinaires (sans contraintes)
- La classe **`Triangle`** est une classe abstraite avec un créateur privé et plusieurs classes internes
  - **`Triangle.Ordinary`**
  - **`Triangle.Equilateral`**
  - **`Triangle.Isoceles`**
  - **`Triangle.Right`**
  - **`Triangle.RightIsoceles`**
- toutes déclarées comme sous-classes de **`Triangle`**

Challenge

- La classe **`Polygon`** permet d'accéder aux points et en les déplaçant, de modifier la forme
  - pour un polygone, il n'y a rien de gênant; c'est une fonctionnalité souhaitée
  - pour un parallélogramme, un rectangle ou un carré, cela pose problème puisqu'en déplaçant un seul point on détruit sa supposée propriété !

- Comment peut-on palier ce problème ?