LU2IN002 - Introduction à la programmation orientée-objet

Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ

Cours 7 – 28 octobre 2022

Plan du cours

Subsomption et surcharges (1)

- Attention ! Le mélange des 2 peut être (d)étonnant...

```java
public class MaClasseA {
  public void affiche(double x) {
    System.out.println("La classe A affiche un double : "+x);
  }
}
```

```java
public class MaClasseB extends MaClasseA{
  public void affiche(int n) {
    System.out.println("La classe B affiche un entier : "+n);
  }
}
```


Subsomption et surcharges (2)

```java
public class TestMesClassesAB {
  public static void main(String[] args) {
    MaClasseA a = new MaClasseA();
    a.affiche(11.38);

    MaClasseB b = new MaClasseB();
    b.affiche(42);
    b.affiche(11.38);

    MaClasseA ab = new MaClasseB();
    ab.affiche(11.38);
    ab.affiche(42);
  }
}
```

- Qu'obtient-on ?

Subsomption et surcharges (4)

```
// MaClasseA a = new MaClasseA();
Résultat pour a.affiche(11.38) :
La classe A affiche un double : 11.38
```

```
// MaClasseB b = new MaClasseB();
Résultat pour b.affiche(42) :
La classe B affiche un entier : 42
```

```
Résultat pour b.affiche(11.38) :
La classe A affiche un double : 11.38
```

```
// MaClasseA ab = new MaClasseB();
Résultat pour ab.affiche(11.38) :
La classe A affiche un double : 11.38
```

```
Résultat pour ab.affiche(42) :
La classe A affiche un double : 42.0
```

- Pourquoi ce résultat ?

Plan du cours

Nouveaux concepts

- **Classe abstraite**
  - Classe qui ne sera pas instanciable
  - Les classes filles pourront être instanciables
  - Exemple :
    - Animal (abstraite) : définit un comportement général
    - Mouton, Tigre : animaux avec comportements spécifiques
- **Méthode abstraite**
  - Seulement dans les classes abstraites
  - Elle contient une signature mais pas de code
  - Exemple :
    - Animal (abstraite) : `String regimeAlimentaire()`
    - Mouton, Tigre $\Longrightarrow$ `"herbivore"`, `"carnivore"`

Classe abstraite

Définition

- Représente une classe qui ne peut pas être instanciée
- 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 définir ce qui est déclaré dans la classe mère (signature de méthode abstraite)
  - tous les animaux ont un régime alimentaire...
    $\rightarrow$ la signature de la méthode regimeAlimentaire() est donnée dans la classe mère Animal
  - ... mais la nature du régime est propre au mouton, tigre...
    $\rightarrow$ le code de la méthode regimeAlimentaire() est défini dans chaque classe fille Mouton, Tigre,...

Classe abstraite : syntaxe

```java
public abstract class Animal {
  ...
  // signature seulement : PAS D'ACCOLADES!
  public abstract String regimeAlimentaire();
}
```

- Il est impossible de créer une instance de la classe `Animal`

```java
new Animal();  // --> ERREUR compilation: abstract class
```

- Des classes peuvent hériter de Animal, elles doivent alors
  - soit implémenter regimeAlimentaire()

```java
public class Mouton extends Animal {
  ...
  public String regimeAlimentaire() { // code méthode
    return "Herbivore";
  }
}
```

  - soit être elles-même abstraites

```java
public abstract class Poisson extends Animal {
  ...
}
```


Propriétés des classes abstraites

- Les classes abstraites sont des classes, elles peuvent avoir
  - des attributs
  - des constructeurs
  - des méthodes "normales"
- mais en plus, elles peuvent aussi avoir (ou pas)
  - des méthodes abstraites

```java
public abstract class Animal {
  private int age;
  public Animal(int age) {
    this.age = age;
  }
  public int getAge(){
    return age;
  }
  public abstract String regimeAlimentaire();
}
```

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

- ne pas modifier les classes existantes
- ajouter des classes filles

Idée :

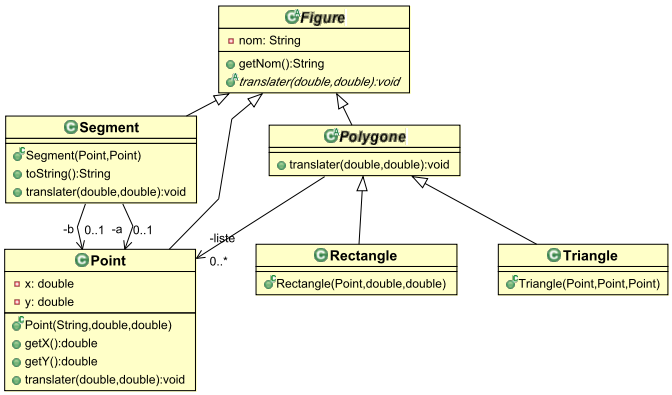
Structurer un projet avec des classes abstraites :

- les classes filles possèdent des fonctionnalités dès leur création
  - factorisation du code
- ajout de contraintes sur les classes filles
  - 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
  - ex. : tableau hétérogène $\Rightarrow$ + de possibilités

Classe abstraite : bilan

- une classe abstraite ne peut pas être instanciée
- si une classe contient une méthode abstraite, alors cette classe doit être abstraite
- si une classe hérite d'une méthode abstraite, elle doit
  - soit définir le code de cette méthode
  - soit être déclarée comme une classe abstraite
- une classe abstraite peut ne pas contenir de méthode abstraite
- pourquoi déclarer une classe abstraite ?
  - pour empêcher son instanciation
  - pour représenter un concept abstrait dans l'application
    $\rightarrow$ par exemple, un instrument de musique (cf. exo 41)

Exemple du logiciel de dessin



Récupérer le type d'une instance dynamiquement

Cas amusant :

le type des instances est parfois (souvent) inconnu du développeur

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 // Quel type d'instance contient f ?
```


Récupérer le type d'une instance dynamiquement

- Le compilateur vérifie (statiquement) le type des variables

1) comment connaître le type des instances lors de l'exécution ?
   - opérateur instanceof
   - méthode getClass()
2) connaître le type de l'instance pour accéder à ses méthodes
   - utiliser 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 // Quel type d'instance contient f ?
8 if(f instanceof Point)
9   System.out.println("C'est un Point");
10 else
11   System.out.println("C'est un Segment");
```


1) récupération des informations : instanceof

var instanceof *NomClasse*

⇒ retourne un boolean

→ rend true si l'objet référencé dans *var* est une instance de la classe *NomClasse*

→ rend false dans le cas contraire

```
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 if(f instanceof Point)
7   System.out.println("C'est un Point");
8 else
9   System.out.println("C'est un Segment");
```

- comprendre instanceof comme "EST UN ?"
- MAIS : souvent, il existe d'autres moyens de faire...
  Globalement, sauf exception :

instanceof = mauvaise programmation

instanceof : discussion

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 Segment");
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 ?

⇒ Le client doit modifier le code du fournisseur ! ! !

Moralité : instanceof existe, mais il faut souvent éviter de **l'utiliser** !

instanceof : attention à la hiérarchie

```
1  public static void main(String[] args) {
2      Point p1 = new Point("toto", 0, 2);
3      Point p2 = new Point( "toto 2", 3, 2);
4      Figure f = new Segment(p1, p2);
5
6      if(f instanceof Point)
7          System.out.println("f est un Point");
8      if(f instanceof Segment)
9          System.out.println("f est un Segment");
10     if(f instanceof Figure)
11         System.out.println("f est une Figure");
12 }
```

Le programme suivant retourne :

```
1 f est un Segment
2 f est une Figure
```

Le résultat est logique : comprendre instanceof comme "EST UN ?"

instanceof : comment l'éviter...

Code spécifique dans les classes :

- dans Figure :

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

- dans Point :

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

- dans Rectangle :

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

- Code générique (éventuellement en dehors des classes) :

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


Alternative à instanceof : getClass()

getClass() : Class

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

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

Usage classique pour comparer le type de deux instances :

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


Héritage et Cast

Cast = 2 modes de fonctionnement

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

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

- Conversion dans les hiérarchies de classes : la variable est modifiée, l'instance est inchangée

```
Figure f = new Segment(p1, p2);
Segment s = (Segment) f;
// Pour vérifier que vous avez compris:
// 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 :

```
Point p1 = new Point("toto", 0, 2);
Point p2 = new Point( "toto 2", 3, 2);
Figure f = new Segment(p1, p2);

Figure f2 = (Figure) p1; // OK mais inutile
Figure f3 = p1; // OK Subsomption classique
Segment s = (Segment) f; // compilation OK
Point p3 = (Point) f; // compilation OK (!)
```

Exécution :

Crash du programme avec le message suivant

```
Exception in thread "main" java.lang.ClassCastException:
Segment cannot be cast to Point
        at Test.main(Test.java:8)
```


Cast : avec plusieurs niveaux de hiérarchie

```
// subsomption
Figure f = new Segment(p1, p2);
Figure f2 = new Carre(p1, cote);
Figure f3 = new Triangle(p1, p2, p3);;
Polygone p = new Triangle(p4, p5, p6);

// cast OK
Triangle t = (Triangle) p; // OK (comme précédemment)

Polygone p2 = (Polygone) f2; // OK compil + exec:
                             // un Carre EST UN Polygone

Polygone p3 = (Triangle) f3; // OK:
      // f3 est un Triangle => conversion OK (JVM)
      // p3 peut référencer un Triangle OK (compilé)

// cast KO
Carre c = (Carre) f; //KO JVM
Cercle c = (Cercle) p; //KO Compil : opération impossible
```


Cast : sécurisation

Idée

Vérifier le type de l'instance avant la conversion

```
        Figure f = new Segment(p1, p2);
        Segment s;
        if(f instanceof Segment) // ici: bon usage de instanceof
            s = (Segment) f;
            s.methodeDeSegment();
```

⇒ vous utiliserez systématiquement cette sécurisation

Cast : usage dans la définition de equals()

Redéfinition de la fonction equals()

- méthode pour tester l'égalité entre 2 objets
- comportement inadéquat par défaut (héritée de Object)
- → nécessité de la redéfinir : utilisation d'un cast pour accéder aux attributs à comparer

Exemple pour la classe Point :

```java
public boolean equals(Object obj) { // version V1
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (getClass() != obj.getClass())
    return false;
  Point other = (Point) obj; // cast: on est sûr de la classe
  if (x != other.x)
    return false;
  if (y != other.y)
    return false;
  return true;
}
```


getClass *vs* instanceof

Imaginons la redéfinition suivante pour `equals` :

```java
    public boolean equals(Object obj) {// version V2
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (!(obj instanceof Point))  // Incorrect ici !
            return false;
        Point other = (Point) obj;
        if (x != other.x)
            return false;
        if (y != other.y)
            return false;
        return true;
    }
```

Quel est le défaut de l'implémentation V2 ?

Prise en défaut :

```java
Point p = new Point(1,2);
PointNomme p2 = new PointNomme("toto",1,2);
if (p.equals(p2))
  System.out.println("ils sont égaux !!! ");
if (p2.equals(p))
  System.out.println("et ici ???");
```

Avec :

```java
public class PointNomme extends Point{
  private String qqch;
  public PointNomme(String nom, double x, double y) {
    super( x, y);
     qqch = nom; // un attribut en plus
  }
}
```

V1 : pas d'égalité

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

Plan du cours

Mot clé 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
- *Rem :* constante indépendante des instances ⇒ static
- *Usage :* une constante est définie en majuscule

```java
public class MaClasse{
  public final static int MACONSTANTE = 10;
  ...
```

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...)

Autres usages (liés à l'héritage)

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

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

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

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

// Compilation impossible : classe "mère" final
public class PointNomme extends Point{
}
```