LU2IN002 - Introduction à la programmation objet

Christophe Marsala

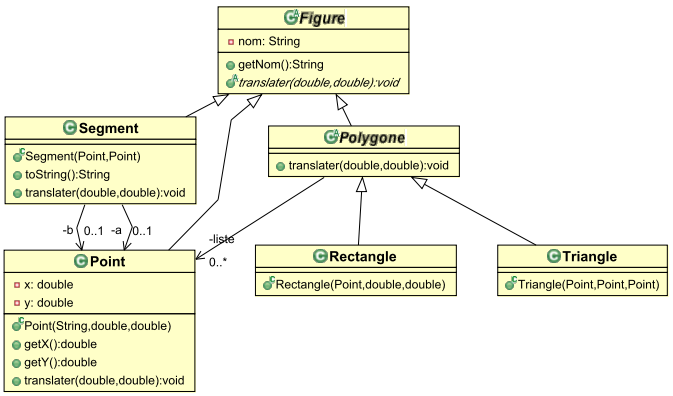
SCIENCES SORBONNE UNIVERSITÉ

Cours 8 – 25 octobre 2024

Programme du jour

Plan du cours

Exemple du logiciel de dessin



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 : COMMENT L'ÉVITER...

Bonne stratégie : déléguer le 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   System.out.println("C'est un "+f.getTypeFigure());
3 }
```


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 ?"

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

```
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 Segment
```

Usage classique pour comparer le type de deux instances :

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


HÉRITAGE ET CAST

Cast = 2 modes de fonctionnement

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

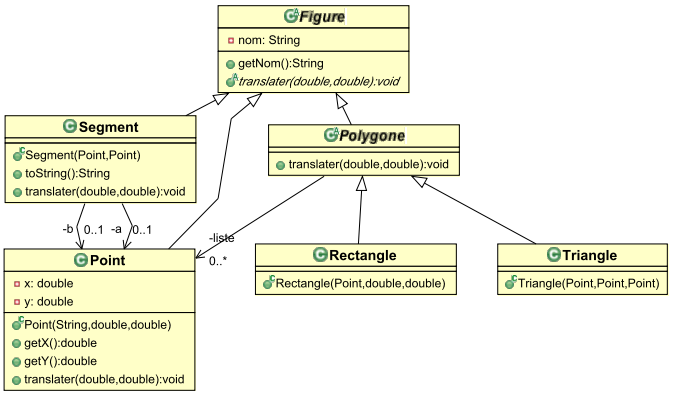
```
1 double d = 1.4;
2 int i = (int)d;  // i=1
```

- 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...

EXEMPLE DU LOGICIEL DE DESSIN



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 (même hiérarchie)
```

Problème à l'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: Cercle et Carre ne sont pas sur la même branche
Carre c = (Carre)f; //KO JVM
Cercle c = (Cercle)p; //KO Compil : opération impossible
```


Cast : sécurisation par instanceof

Bon usage de instanceof : sécuriser un cast

L'utiliser pour 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();
```

⇒ utiliser **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

Exemple pour la classe Point :

```
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 :

```
    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 :

```
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 :

```
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

Attribut final ⇒ CONSTANTE

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

```
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...)

Autres usages : méthode et classe

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

Impossible de redéfinir une méthode finale dans une classe fille :

```
1 public class Point{
2   public final double getX(){ ... }
3 }
```

```
1 public class PointNomme extends Point{
2 ...
3 // Compilation impossible : méthode existante final
4   public double getX(){ ... }
5 }
```

Impossible d'hériter d'une classe finale :

```
1 public final class Point{
2   ...
3 }
```

```
1 // Compilation impossible : classe "mère" final
2 public class PointNomme extends Point{
3 }
```


Plan du cours

Certaines situations posent problèmes

- Ecrire des classes
  - avec la possibilité de sauvegarder leur résultat
  - avec la possibilité d'afficher le résultat sous différents format (écran, page web,...)

  ⟹ type de comportements partagés par des classes qui ne sont pas sur la même branche de la hiérarchie

⇒ MAIS limite de l'héritage : pas d'héritage multiple en JAVA

- une classe ne peut hériter que d'une seule classe

⇒ un autre outil existe : les interfaces

Interface : définition et usage

Usage

Une interface définit un comportement :

- un cahier des charges (e.g. Circuit...)
- une propriété (e.g. Serializable, Clonable, Pilotable, ...)

Elle donne la signature des méthodes à implémenter.

Ce que contient une interface

- signatures de méthodes (comme des méthodes abstraites)
- Mais (<java 1.8) :
  - pas de code
  - pas d'attribut

⟹ Une interface n'est pas une classe !

Interface : exemples & syntaxe

1. Vu de l'extérieur de l'objet...
   exemple : qu'est-ce qui caractérise un véhicule pilotable ?

Interface : exemples & syntaxe

Qu'est-ce qui caractérise un véhicule Pilotable ?
→ caractéristiques vues de l'extérieur de l'objet

- accélérer, freiner, tourner
- son état actuel (position, direction, vitesse, dérapage)
- observation des propriétés (capacités de braquage, vmax...)

```java
public interface Pilotable { // pour être considéré pilotable
   // pour le pilotage
   public void accelerer(double d);
   public void freiner(double d);
   public void tourner(double d);

   // pour connaître son état actuel
   public double getVitesse();
   public Vecteur getPosition();
   public Vecteur getDirection();
}
```

$\Longrightarrow$ Une classe sera Pilotable si elle implémente toutes les méthodes déclarées dans l'interface Pilotable

Interface : exemples & syntaxe (2)

Les interfaces pour énoncer des propriétés pour des objets

Par exemple : qu'est ce qu'un objet qui serait sauvegardable ?

Interface : exemples & syntaxe (2)

Les interfaces pour énoncer des propriétés pour des objets
Qu'est ce qu'un objet qui serait sauvegardable ?

Réponse :

- c'est un objet qui peut être sauvegardé sur disque
- c'est un objet capable de répondre à la méthode suivante public void save(String filename)

- Définissons une interface précisant ce comportement :

```java
public interface Sauvegardable {
    public void save(String filename);
}
```

$\Longrightarrow$ Une classe dont les instances seront sauvegardables doit implémenter la méthode déclarée dans cette interface

Interface : exemples & syntaxe (2)

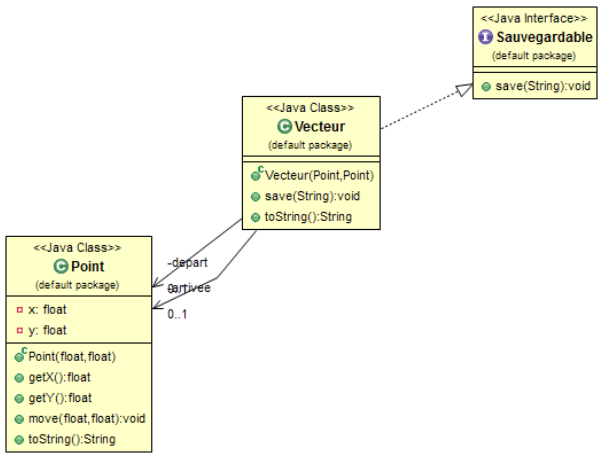
Définir un objet sauvegardable : exemple avec la classe Vecteur.

- un objet sauvegardable implémente l'interface Sauvegardable.
- la classe Vecteur doit contenir l'implémentation de la méthode : public void save(String filename)
  $\Longrightarrow$ respect du contrat "sauvegardable"
- une instance de Vecteur pourra donc se sauvegarder : respect d'un cahier des charges

```java
public class Vecteur implements Sauvegardable{
   ...
   public void save(String filename){
      ... // instructions à réaliser
   }

}
```


Interface & héritage : représentation UML



Interface & héritage : propriétés

- Une classe ne peut hériter que d'**une seule classe mère**
- mais une classe peut implémenter **plusieurs interfaces**

- Exemple pour un logiciel de dessin
- interfaces : Sauvegardable (méthode save), Deplacable (méthode move)...

```
public class Polygone extends Figure
        implements Sauvegardable, Deplacable{
(...)
  // Hérite des méthodes de Figure
  // doit implémenter les méthodes save() et move()
}
```

Rem : si une classe implémente une interface, toute sa descendance (classe fille, etc.) hérite des méthodes correspondantes
$\Longrightarrow$ toutes les classes descendantes implémentent donc aussi l'interface par héritage

Interfaces et variables

- Il est possible de déclarer une variable d'un type interface
  - principe de subsomption
    - par exemple : tableau à partir d'une interface
  - mais on n'instancie pas une interface
- Seules les méthodes de l'interface sont accessibles
- Exemple : soit les classes qui implémentent Sauvegardable :
  - classes Vecteur, Point, Figure,...
  - classes Personne, Menagerie,...

```
Sauvegardable[] tab = new Sauvegardable[42];
tab[0] = new Vecteur();
tab[3] = new Point();
tab[7] = new Menagerie(12);
...
for (int i=0; i<tab.length;i++)
    if (tab[i] != null)
        tab[i].save("fichier "+i);
```

- Très pratique pour appliquer un traitement identique (ici save) à un ensemble de classes non liées par héritage

Interface et héritage

- Une interface peut hériter d'une autre interface
  - c'est un héritage pas une implémentation...

```
public interface Positionnable{
  public Vecteur getPosition();
}

public interface Deplacable extends Positionnable{
  public void move(Vecteur v);
}
```

- Une classe qui implémente Deplacable doit fournir
  - une définition pour la fonction move
  - une définition pour la fonction getPosition

Les interfaces : bilan

- Une interface permettent de définir un comportement
  - définition d'un cahier des charges à respecter
  - énoncé des propriétés requises pour un objet
  - elle déclare un ensemble de méthodes
  - une classe choisit de respecter ce comportement : implements
- Une interface n'est pas une classe
  - mais elle peut hériter d'une autre interface : extends
- $\Longrightarrow$ deux types de hiérarchies en Java
  - hiérarchie des classes : classe mère Object
  - hiérarchie des interfaces

Plan du cours

Introduction

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

Solution = arborescence de répertoires

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

Création de Packages de classes

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. Prévoir des classes de test :

Idée :

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

⇒ **sous-répertoire de test** dans le projet principal

Créer un package

Package java

- Un Package est un est ensemble de classes mises dans un même répertoire.
- ⟹ les classes d'un même package forment "une famille" : nouveau type de visibilité

- Définition d'un package

```
1 package nomdupackage ; // en début de fichier de classe
```

- Importer une classe d'un package

```
1 import nomdupackage.LaClasseVoulue ;
```

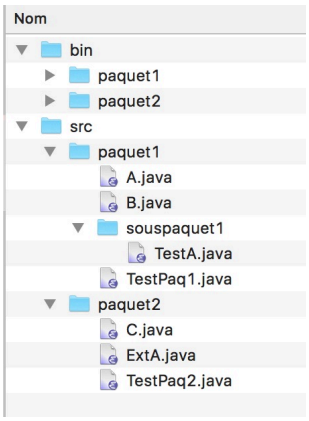
- Importer toutes les classes d'un package

```
1 import nomdupackage.* ;
```

⟹ Convention : le nom d'un package est en minuscules

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     }
```

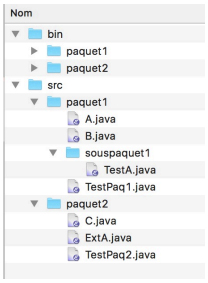
3. Sous-package

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

4. Classe JDK

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


Compilation / exécution du code



- On se place à la racine des répertoires
  - src et bin : s/s répertoires
- **Compilation**
  - où trouver les classes : -cp
  - répertoire cible : -d

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

⇒ compile l'exécutable + toutes les dépendances

- **Exécution**
  - répertoire des .class : -cp
  - chemin avec des . (pas des /)

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

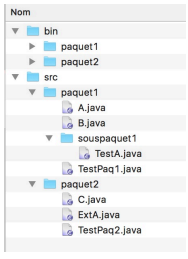
ou

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


Niveaux de visibilité

Introduction des packages = subtilités sur la visibilité

```
1 package paquet1;
2 public class A {
3   public int i;     // public
4   protected int j;  // protected
5   private int k;    // private
6   int n;            // package (nouveau)
7
8   public A(){
9       i=1; j=2; k=3; n=4;
10  }
11 }
```



Visibilités des attributs de `A` depuis :

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