

## IFT 2255 - Genie Logiciel

# Paradigme orienté-objet

# Caractéristiques de l'orienté-objet

1. Encapsulation
2. Dissimulation d'implémentation et de l'information
3. Classes et objets
4. Héritage
5. Polymorphisme
6. Généricité

# Encapsulation

- *Définition:* regroupement de concepts reliés en une seule unité référencée par un seul nom
- Création d'abstractions qui permettent de conceptualiser le problème à un plus haut niveau
- Définir des types de données abstraits
  - Type de données avec des opérations effectuées sur leurs instances

```
class RationalClass
{
    public int      numerator;
    public int      denominator;

    public void sameDenominator (RationalClass r, RationalClass s)
    {
        // code to reduce r and s to the same denominator
    }

    public boolean equal (RationalClass t, RationalClass u)
    {
        RationalClass      v, w;
        v = t;
        w = u;
        sameDenominator (v, w);
        return (v.numerator == w.numerator);
    }

    // methods to add, subtract, multiply, and divide two rational numbers
}

// class RationalClass
```

# Raffinement par étapes

## 1. Concevoir le produit en fonction de **concept**s de haut niveau

- Peu importe de savoir comment ce sera implémenté
- Suppose l'existence du niveau plus bas

Différer les décisions sur les détails au plus tard possible

## 2. Concevoir les composants de **plus bas niveau**

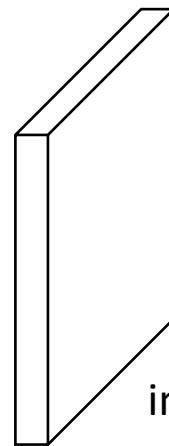
- Ignorer complètement l'existence du niveau supérieur
- Se concentrer sur l'implémentation du comportement
- Ce principe est **récursif**, il y a généralement **plusieurs niveaux d'abstraction**

# Dissimulation

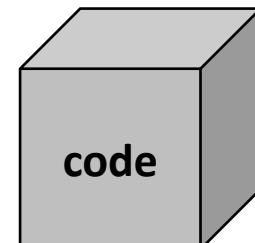
- *Définition:* utilisation de l'encapsulation pour **restreindre la perception depuis l'extérieur** sur les mécanismes **internes** au logiciel
- Deux points de vues d'une entité encapsulée
  - **Vue publique**
  - **Vue privée**



utilisateur



interface



code

# Dissimulation

## Information

- Restreindre la vue de l'utilisateur sur l'information
  - Variables, attributs, format des données, etc.
- L'utilisateur doit utiliser les méthodes publiques pour accéder à l'information

## Implémentation

- Restreindre la vue de l'utilisateur sur l'implémentation
  - Méthodes, algorithmes, etc.
- L'utilisateur peut utiliser une méthode sans savoir comment elle traite le contenu

# Dissimulation impl / info

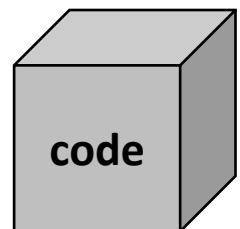
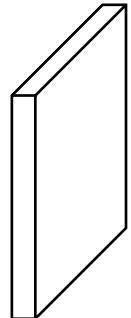
- Concevoir les modules de telle sorte que **ce qui va probablement changer reste caché**
  - Bonne encapsulation et dissimulation impl/info facilite à:
    - **Localiser** les décisions de conception
    - **Séparer** l'information de sa représentation
- Facilite la **réutilisation**

# Règle des getters / setters

- Ne jamais permettre à d'autres classes d'accéder directement aux attributs de ma classe
- Une fois rendu **privé**, un attribut ne peut plus être changer directement (sans garde fou)
- Rendre les attributs accessibles via les **méthodes get/set**
  - `this.ligne` *Dangereux*
  - `this.getLigne()` *Bon*

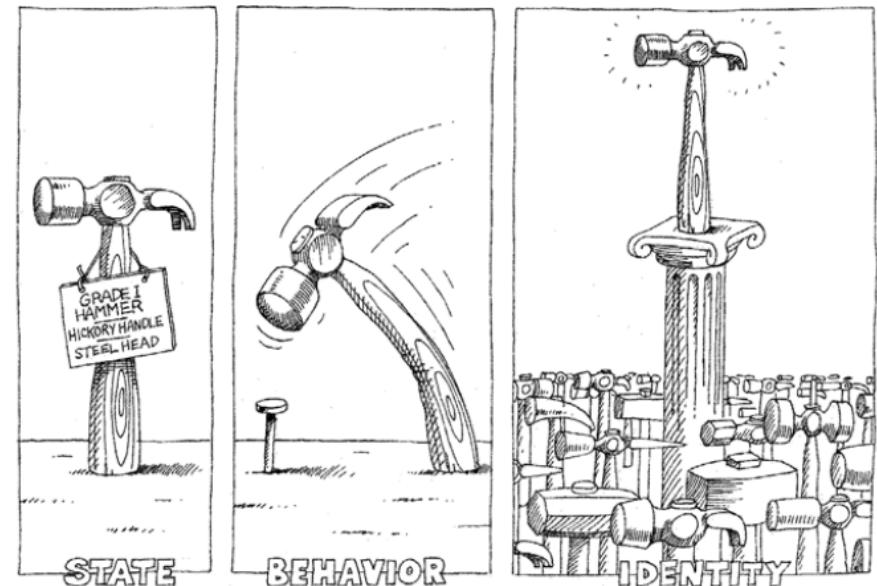
# Avantages

- Concepteur et utilisateur sont indépendants:
  - Ils doivent uniquement s'entendre sur l'**interface**
- L'**évolution** du logiciel est plus facile
  - Changements futurs anticipés/contrôlés
- **Abstraction** de l'utilisateur est élevée
  - Il n'a plus à s'occuper de comment ça fonctionne
- La **réutilisation** du code est plus facile
  - Design **modulaire**

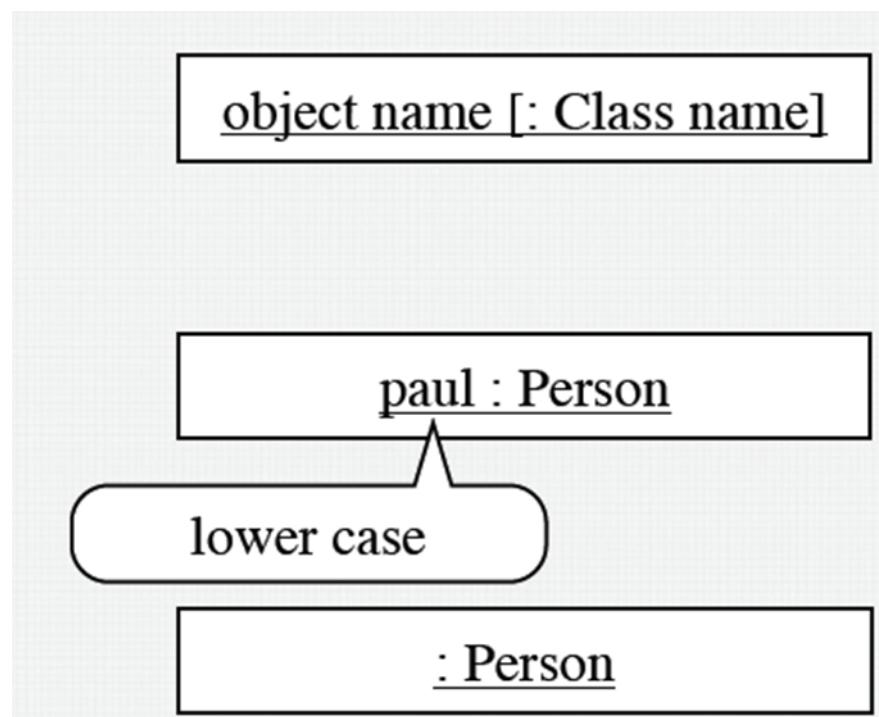


# Objet

- *Définition:* représente une **entité** réelle ou conceptuelle, **singulière** et **identifiable** avec un **rôle** bien défini dans le domaine du problème
- Peut jouer plusieurs rôles
  - Ex. Compte bancaire
- Propriétés d'un objet
  - Caractéristique inhérente ou distinctive

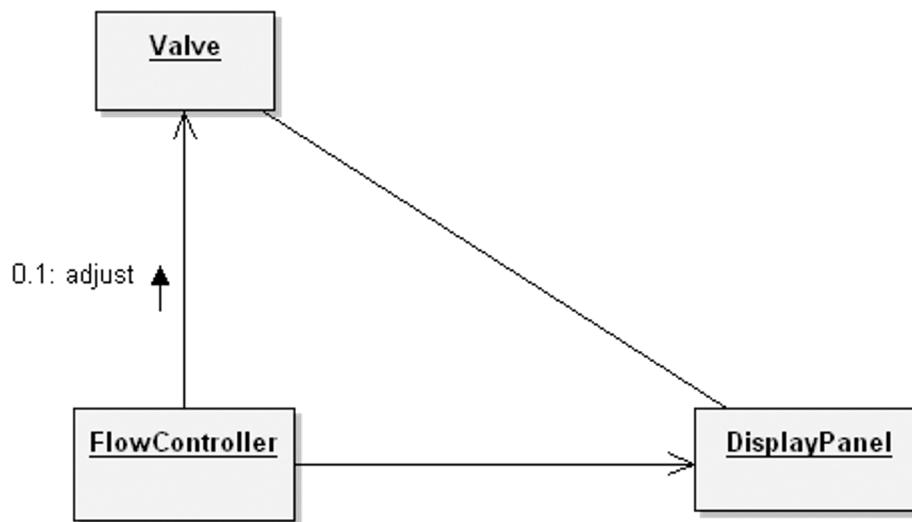


# Représentation d'un objet en UML



# Liens entre objets

- Connections physiques ou conceptuelles entre les objets
- Permet la collaboration entre les objets
  - Objet client applique les services d'un objet fournisseur
  - Objet peut naviguer vers un autre objet

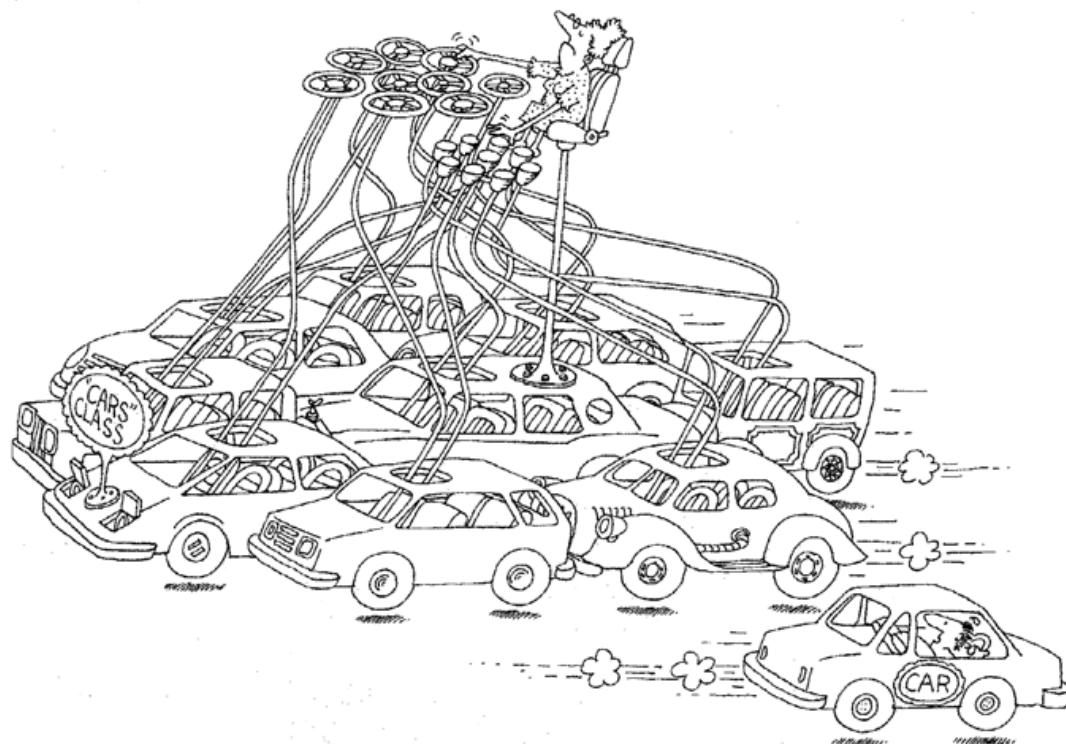


# Message

- *Définition:* véhicule par lequel un objet expéditeur **transmet une information** à un objet cible pour **appliquer une de ses opérations**
- Pour que le message de o1 exécute correctement une méthode de o2, o1 doit
  - Avoir une référence à o2
  - Connaitre la signature de la méthode de o2 à invoquer
  - Passer les bons arguments d'entrée
  - S'attendre à recevoir les bons arguments de sortie
- En Java on écrit (dans la classe o1): o2.methode(args)

# Classe

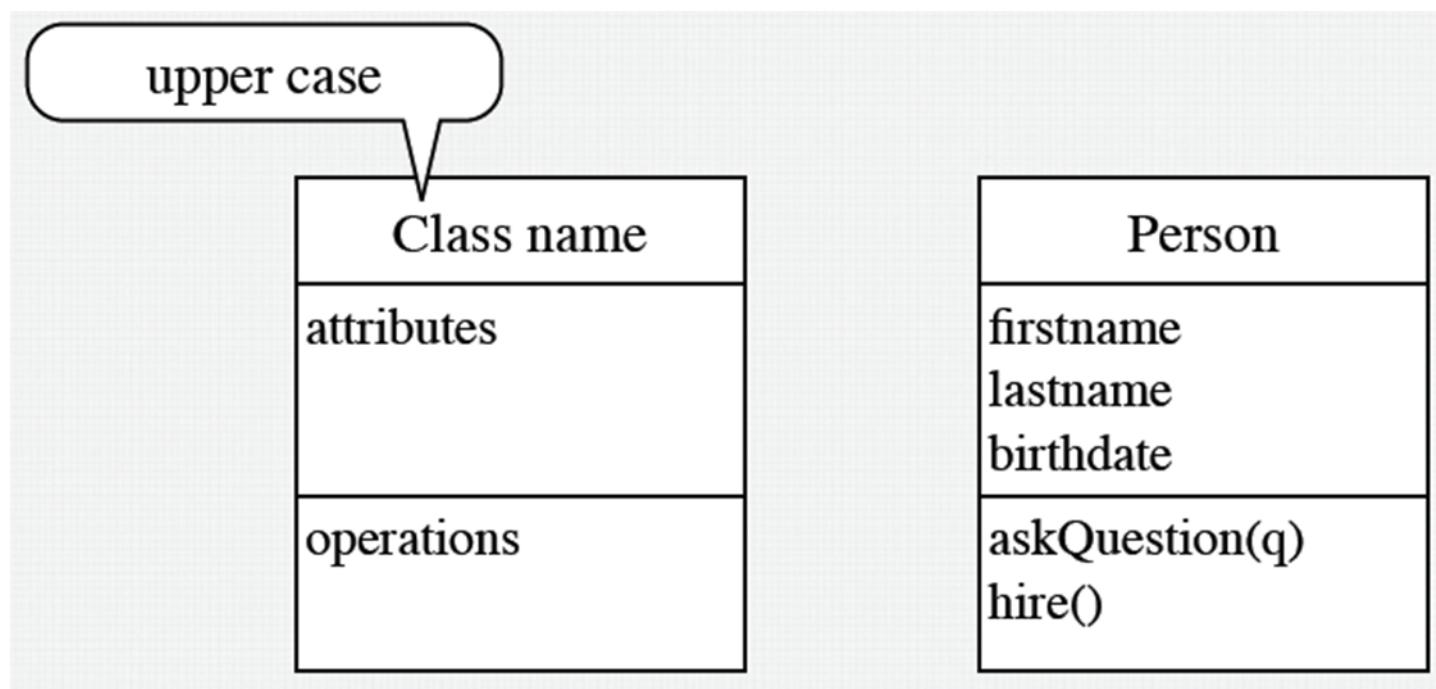
- *Définition:* représentation d'un **ensemble** d'objets qui partagent une **structure**, un **comportement** et une **sémantique en commun**
- « Moule » par lequel des objets sont créés (instanciés)



# Classe

- Tous les objets instanciés à partir de la même classe sont **structurellement identiques**
  - Attributs: nom + type
  - Méthodes: signature + code
- Ont-ils le même comportement?
- Si o appartient à l'ensemble des objets que définit C,  
alors **o est une instance de C**

# Représentation d'une classe en UML



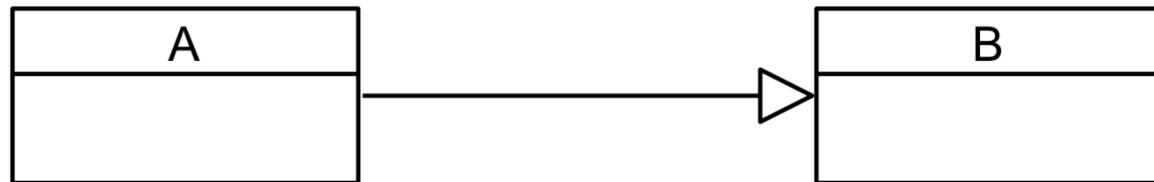
# Classe vs. objet

- Les classes sont **statiques** et évaluées lors de la compilation
- **Une seule copie** de la classe existe
- La mémoire pour stocker les méthodes est allouée **une seule fois**
- Les objets sont **dynamiques** et créés lors de l'exécution
- Une copie de l'objet est créée **à chaque fois** que la classe est instanciée
- La mémoire pour stocker les attributs est allouée **pour chaque objet instancié**

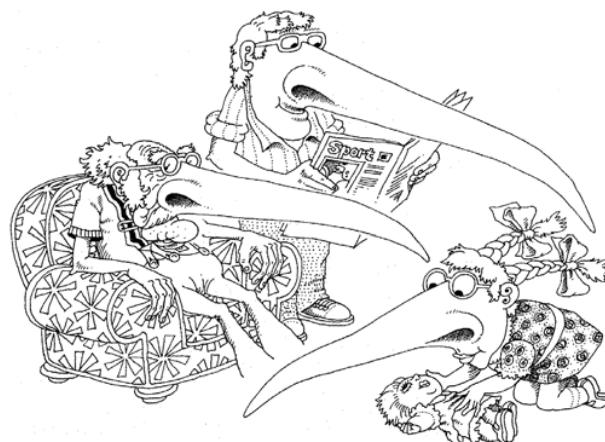
# Interface de classe

- Une classe est une sorte de contrat entre une abstraction et tout ses clients
  - Langages fortement typés peuvent détecter des violations de ce contrat lors de la compilation
- Classe est une encapsulation modulaire
- Modificateurs de visibilité
  - Publique : +
  - Privé : -
  - Protégé : #
  - Paquet : ~

# Héritage



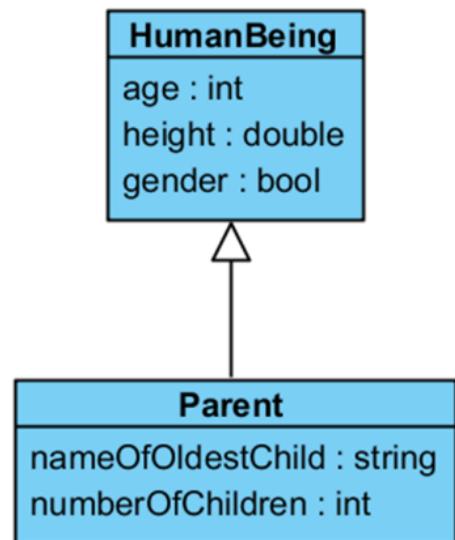
- Relation de **partage de la structure et du comportement**
- Tous les attributs, opérations, associations et contraintes de B sont définies implicitement dans A



# Pratiques de l'héritage

[Gamma1995]

- Permet de définir une nouvelle sorte de classe rapidement à partir d'une classe existante en **réutilisant** les fonctionnalités de la classe parent
- Permet de définir **par différence** plutôt qu'à partir de zéro
- Permet d'avoir de nouvelles implémentations sans effort en héritant ce qui est **commun** avec les classes ancestrales



# Généralisation / spécialisation

- Relation « **est un** »
  - Un parent *est un* humain
  - Une automobile *est un* véhicule motorisé (*est un* véhicule)
- Un humain est plus général qu'un parent
- Une automobile est plus spécifique qu'un véhicule motorisé

# Héritage en Java

```
public class HumanBeing
{
    protected int age;
    protected double height;
    protected bool gender;
}

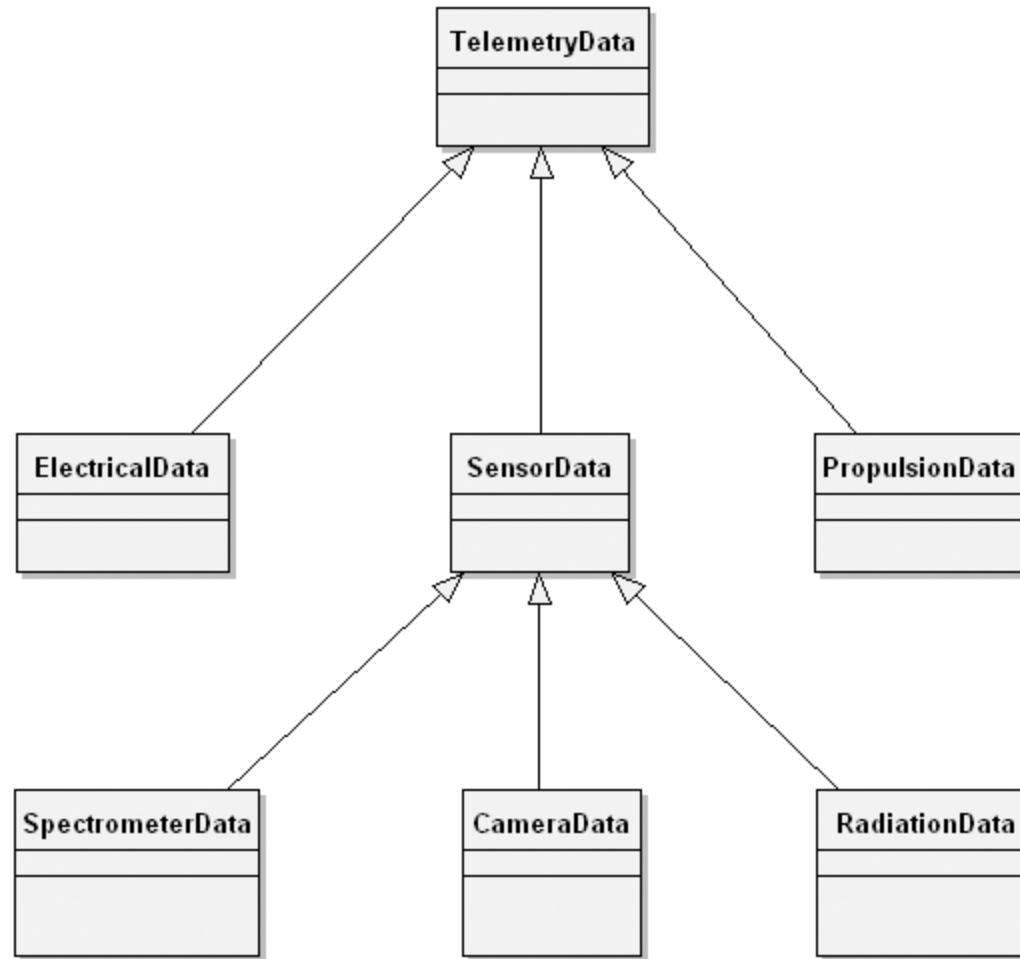
public class Parent extends HumanBeing
{
    private String nameOfOldestChild;
    private int numberOfChildren;
}
```

# QUESTION

*Quelle est la relation entre ces concepts?*

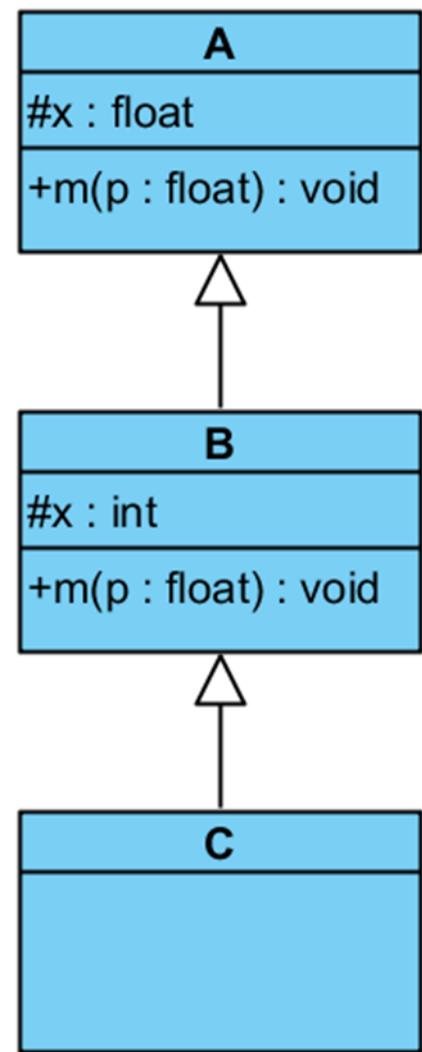
1. *Animal, Chien, Lassie*
2. *Syriani, Étudiant, Professeur, Tremblay, Personne*

# Hiérarchie des concepts

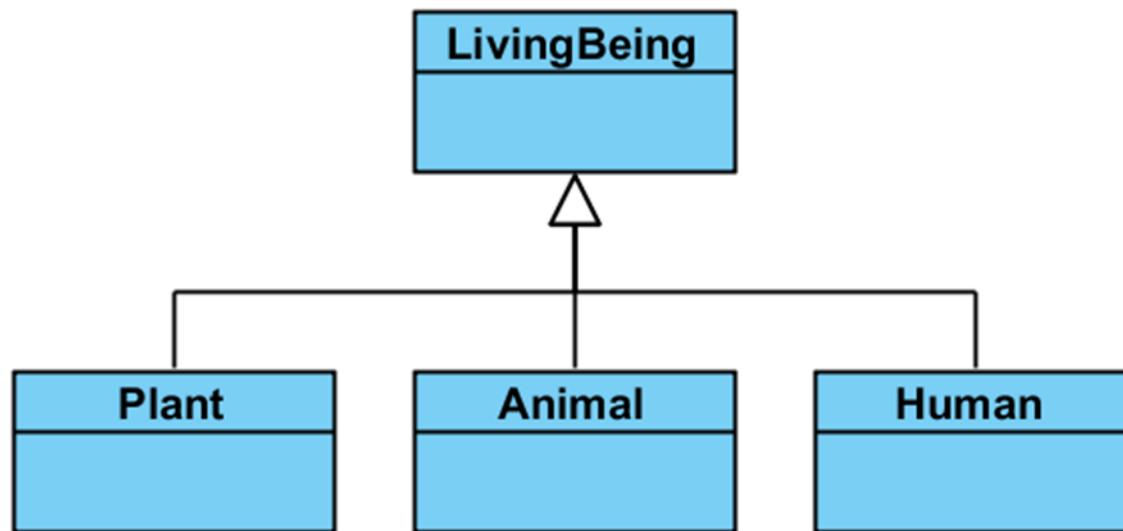


# Redéfinition (*Overriding*)

- Redéfinition d'une méthode/attribut de A dans une de ses sous-classes
- La version de `x` que C aura est celle de B
- L'implémentation de `m` que C aura est celle de B
- **Déclaration abstraite** : forcer la redéfinition

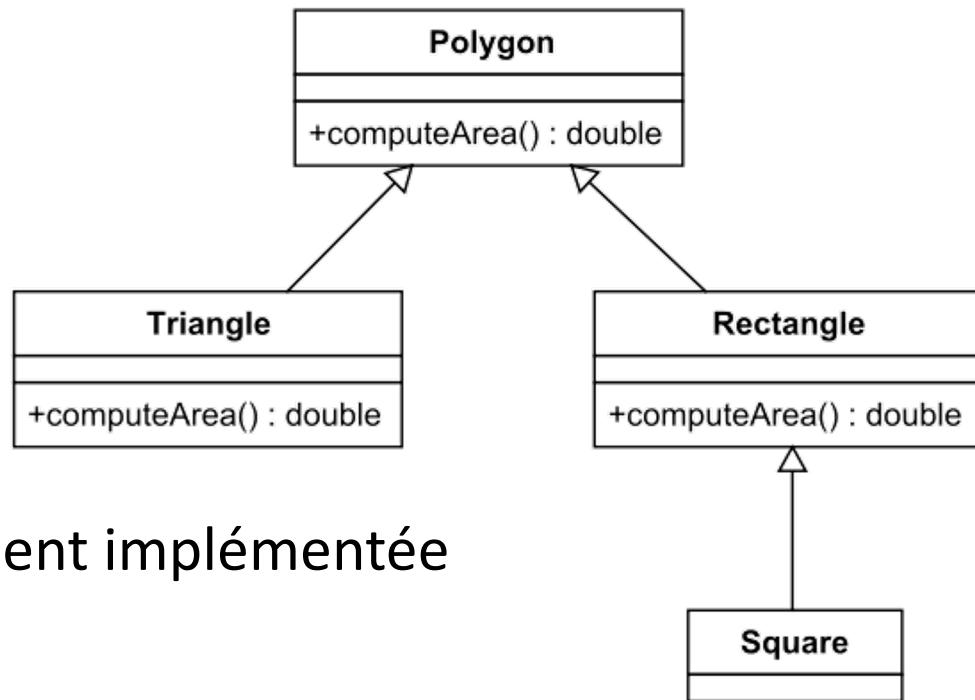


# Plusieurs classes peuvent hériter d'une même classe



# Polymorphisme

- *Définition:* Une opération définie dans plus d'une classe qui prend **différentes implémentations**
- Un polygone doit savoir comment calculer son aire lui-même
- La méthode doit être correctement implémentée pour chaque sous-classe
  - Utilisateur n'a pas à se soucier de l'implémentation selon le type d'objet
  - Bonne opportunité d'utiliser une méthode abstraite

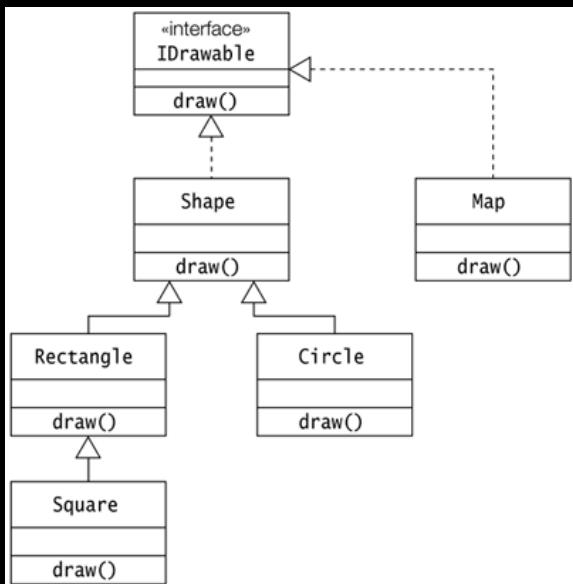


# Liaison dynamique

- Le code de l'utilisateur est indépendant du type concret reçu
  - S'attend à un polygone mais peut recevoir un triangle (qui *est un* polygone)
- Faire attention car source de problèmes durant le débogage et la maintenance
  - Différentes implémentations pour la même méthode
  - Imprimer la classe source de l'appel

# QUESTION

*Quel est le output de ce code ?*



```

class Shape implements IDrawable {
    public void draw() { System.out.println("Drawing a Shape."); }
}
class Circle extends Shape {
    public void draw() { System.out.println("Drawing a Circle."); }
}
class Rectangle extends Shape {
    public void draw() { System.out.println("Drawing a Rectangle."); }
}
class Square extends Rectangle {
    public void draw() { System.out.println("Drawing a Square."); }
}
class Map implements IDrawable {
    public void draw() { System.out.println("Drawing a Map."); }
}
public class PolymorphRefs {
    public static void main(String[] args) {
        Shape[] shapes = {new Circle(), new Rectangle(), new Square()}; // (1)
        IDrawable[] drawables = {new Shape(), new Rectangle(), new Map()}; // (2)
        System.out.println("Draw shapes:");
        for (int i = 0; i < shapes.length; i++) // (3)
            shapes[i].draw();
        System.out.println("Draw drawables:");
        for (int i = 0; i < drawables.length; i++) // (4)
            drawables[i].draw();
    }
}
  
```

Draw shapes:

Drawing a Circle.

Drawing a Rectangle.

Drawing a Square.

Draw drawables:

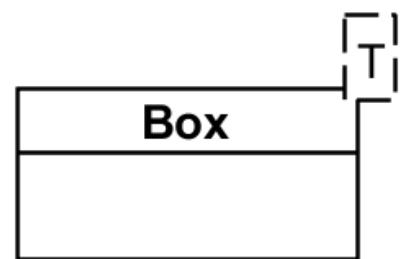
Drawing a Shape.

Drawing a Rectangle.

Drawing a Map.

# Généricité

- *Définition:* Mécanisme pour que les clients décident du **type** d'objets dans une classe à travers des **paramètres** passé lors de la déclaration et qui est évalué lors de la compilation



- Types paramétrés
  - Construction de classe où certains types (classes) qu'elle utilise à l'interne ne soit fournies que lors de l'exécution

```
public class Box<T> {  
    private T element;  
  
    public void set(T e) { this.element = e; }  
    public T get() { return element; }  
}
```

# Exemple de types génériques

```
public class GenericMethodTest
{
    // generic method printArray
    public static < E > void printArray( E[] inputArray )
    {
        // Display array elements
        for ( E element : inputArray ){
            System.out.printf( "%s ", element );
        }
        System.out.println();
    }

    public static void main( String args[] )
    {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray ); // pass an Integer array

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array

        System.out.println( "\nArray characterArray contains:" );
        printArray( charArray ); // pass a Character array
    }
}
```

```
Array integerArray contains:  
1 2 3 4 5 6  
  
Array doubleArray contains:  
1.1 2.2 3.3 4.4  
  
Array characterArray contains:  
H E L L O
```

# Critique du paradigme OO

Forces:

- Favorise la **réutilisation**
  - Classes, héritage, généricité, polymorphisme
- Favorise la **dissimulation** des détails et oblige à dépendre d'**interfaces publiques**
  - Modificateurs privés, classes abstraites, interfaces

Faiblesses:

- Prolifération de fichiers (un par classe, généralement )
- Pas idéal pour le développement d'interfaces graphiques
- Classe à la base de la hiérarchie est **fragile** aux modifications

# Paradigmes de programmation

- Un jour ce paradigme sera sûrement remplacé
- Des prétendants existent déjà
  - Orienté aspect
  - **Ingénierie dirigée par les modèles (MDE)**
  - Modélisation spécifique au domaine (DSM)
  - Programmation logique, basée sur les règles
  - Programmation fonctionnelle