IFT 2255

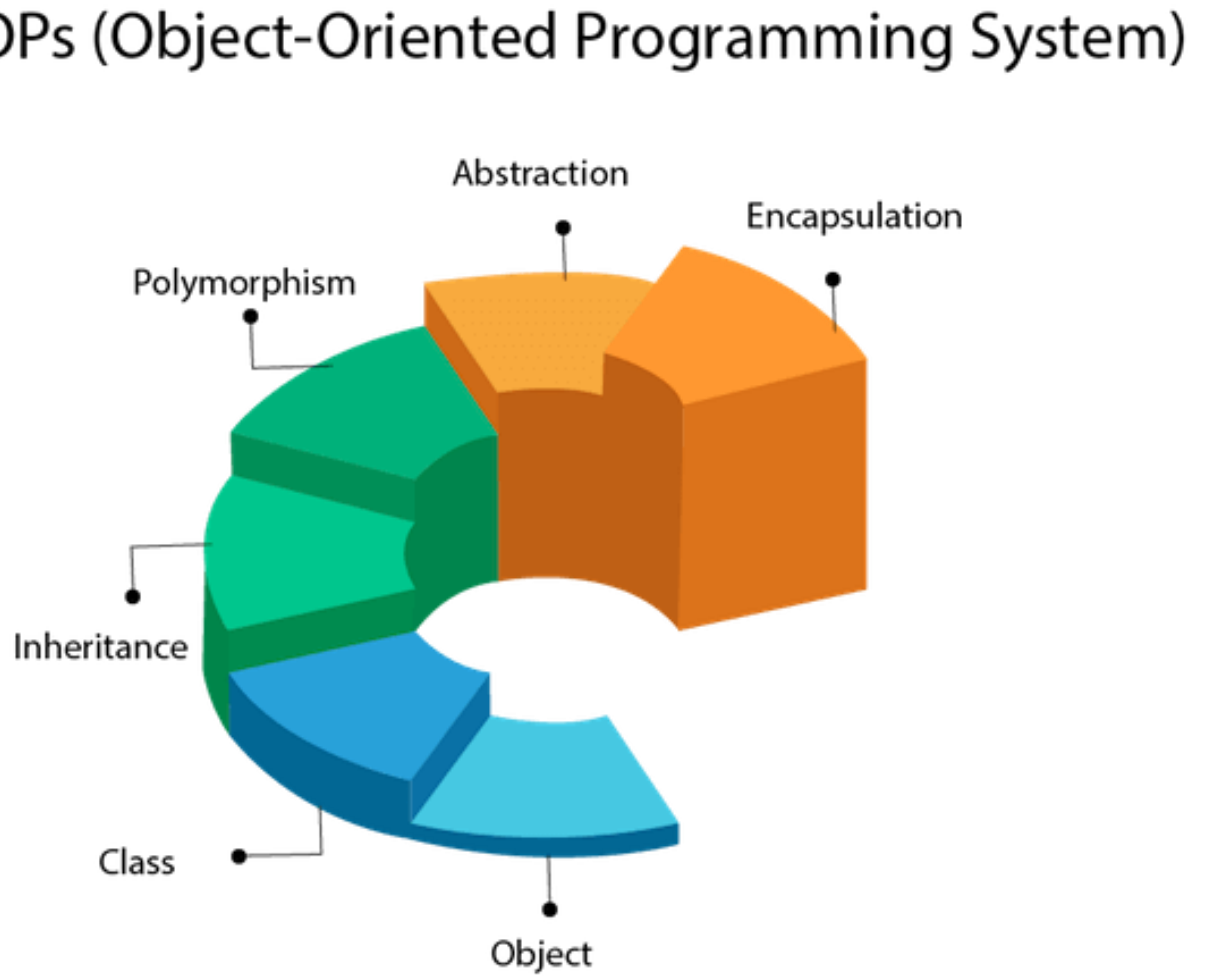
Paradigme orienté-objet

Louis-Edouard LAFONTANT

Caractéristiques

- Classes et objets
- Héritage
- Polymorphisme
- Abstraction
- Encapsulation

- Généricité



Classes et Objets

Définition « Objet »

Représente une **entité** réelle ou conceptuelle, **singulière** et **identifiable** avec un **rôle** bien défini dans le domaine (contexte) du problème

Définition « Classe »

Représentation d'un **ensemble** d'objets qui partagent une **structure**, un **comportement** et une **sémantique en commun**

Objet

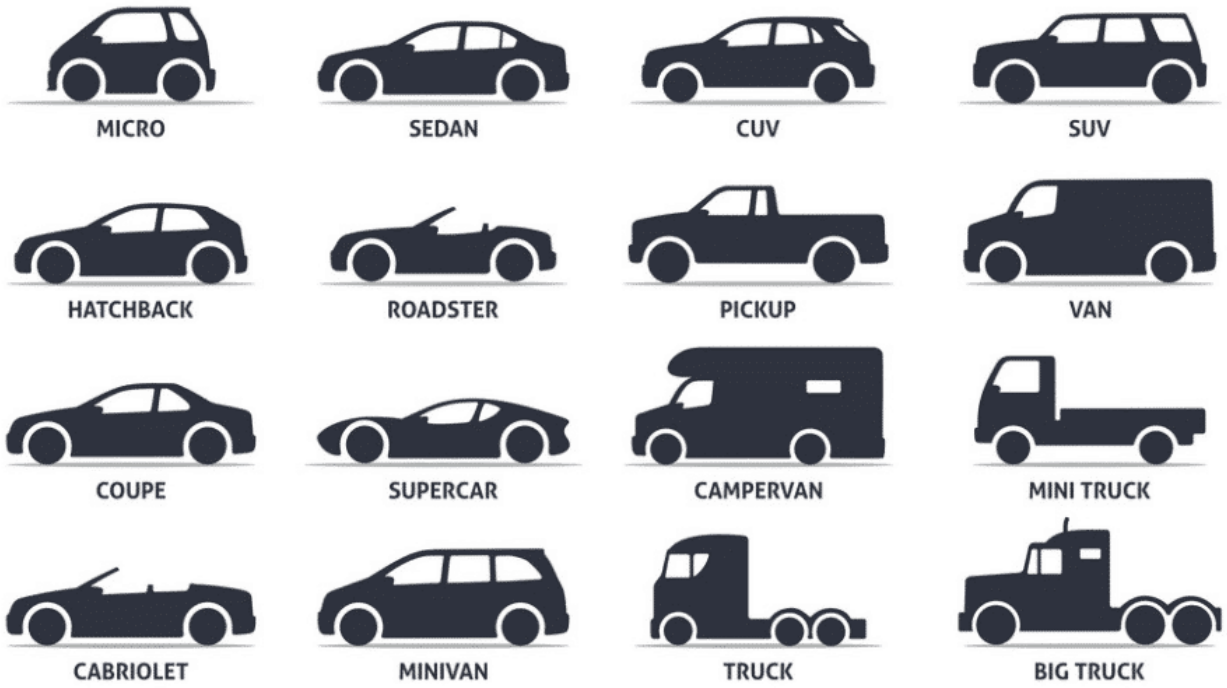
Peut jouer plusieurs **rôles**
Ex: Compte bancaire, Véhicule

Propriétés d'un objet

- Caractéristique inhérente ou distinctive

Liens entre objets

- Connections physiques ou conceptuelles entre les objets
- Permet la collaboration entre les objets



Classe

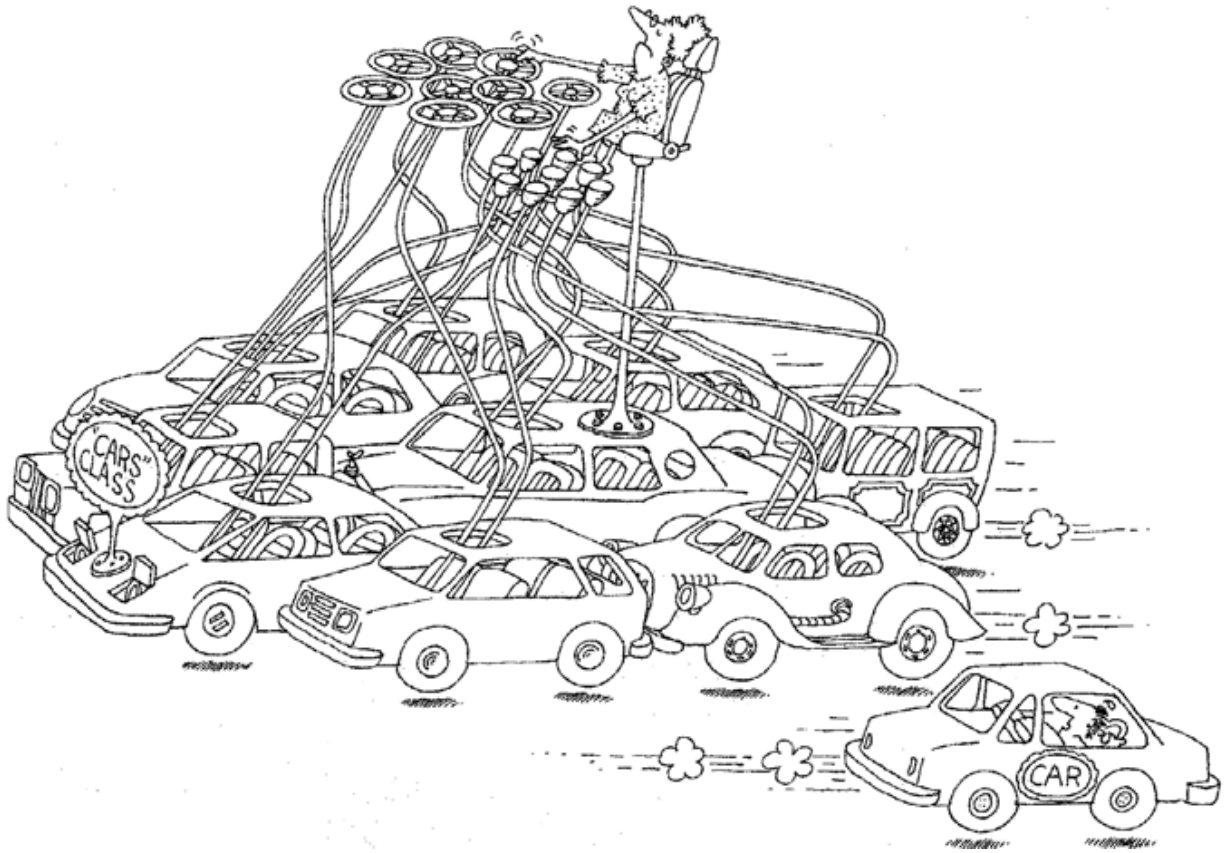
Tous les objets instanciés à partir de la même classe sont **structurellement identiques**

- Attributs: nom + type
- Méthodes: signature + code

Ont-ils le même comportement?

- Dépend du contexte et de l'état de l'objet (valeurs des attributs)

Si o appartient à l'ensemble des objets que définit C, alors **o est une instance de C**



Classe vs. Objet

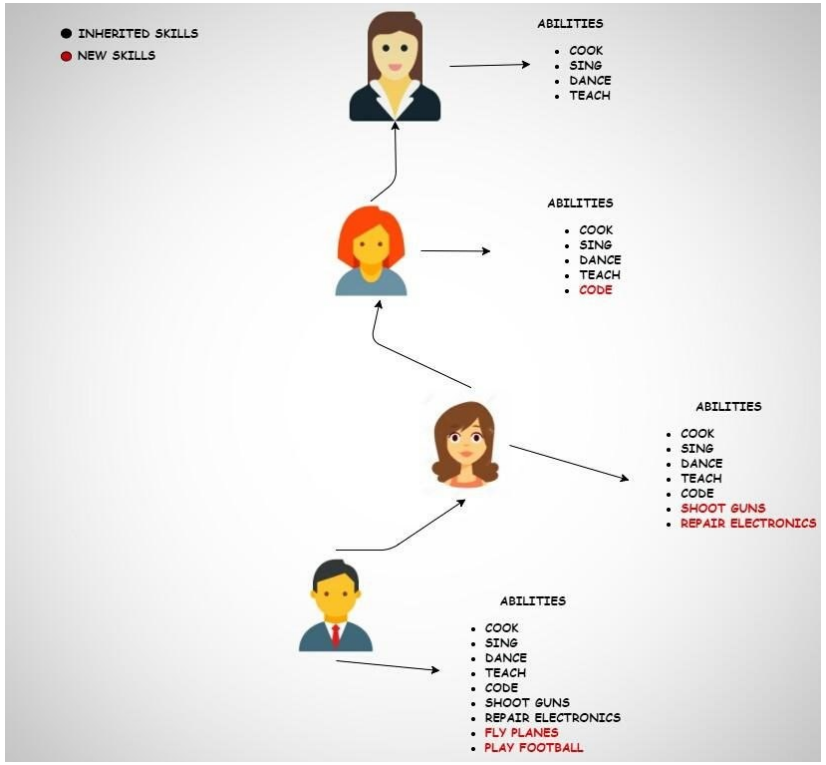
- Les classes sont **statiques** et évaluées lors de la compilation
- **Une seule copie** de la classe existe
- La mémoire pour stocker les méthodes est allouée **une seule fois**

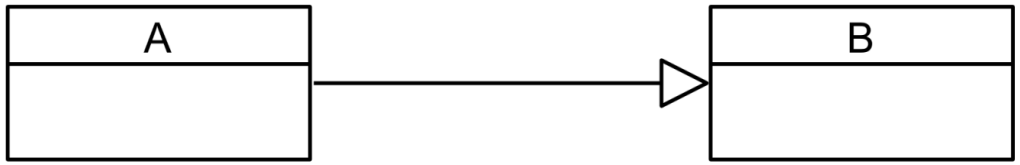
- Les objets sont **dynamiques** et créés lors de l'exécution
- Une copie de l'objet est créée **à chaque fois** que la classe est instanciée
- La mémoire pour stocker les attributs est allouée **pour chaque objet instancié**

Héritage

Relation de **partage de la structure et du comportement**

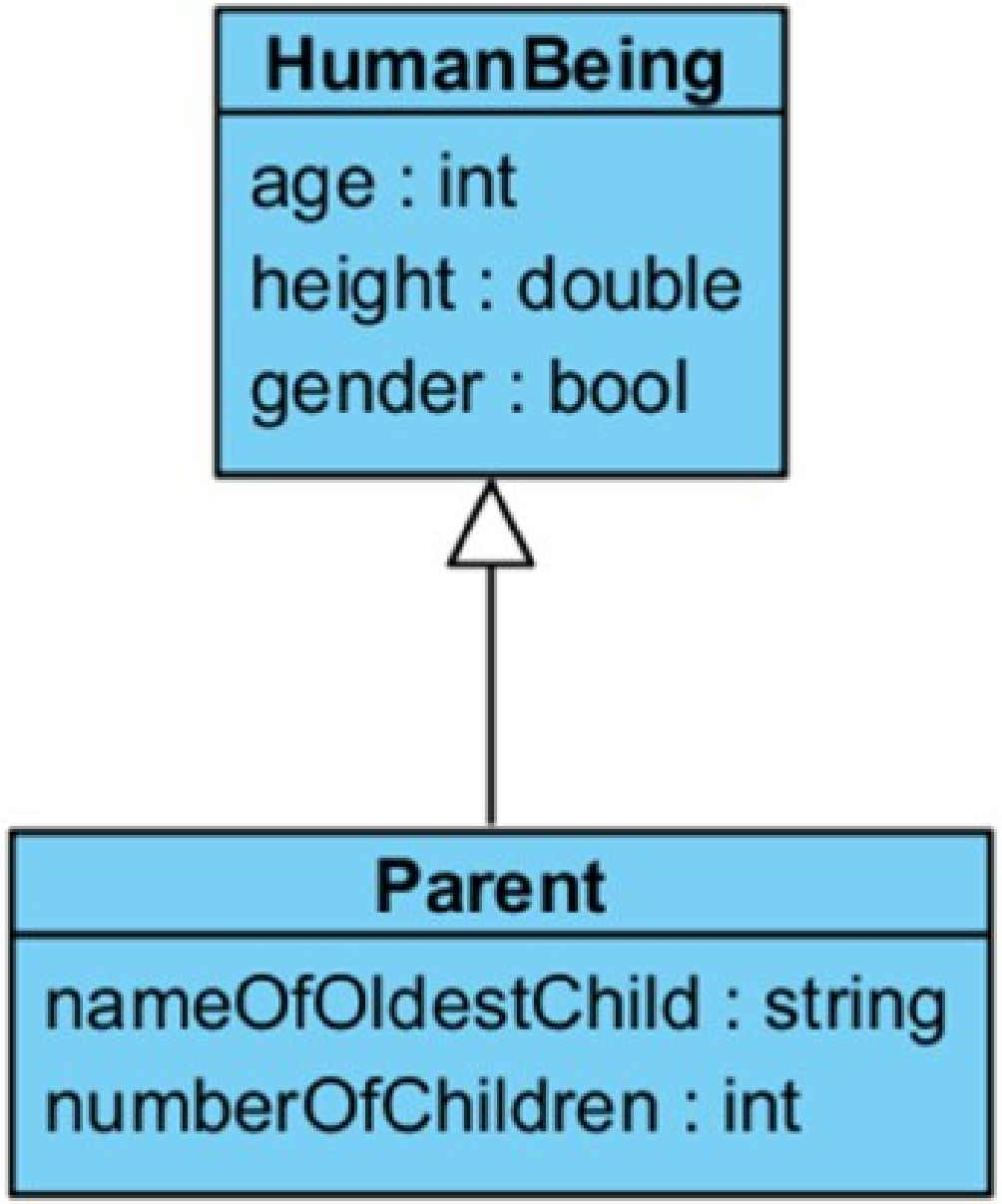
Tous les attributs, opérations, associations et contraintes de B sont définies implicitement dans A





Pourquoi l'héritage ?

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

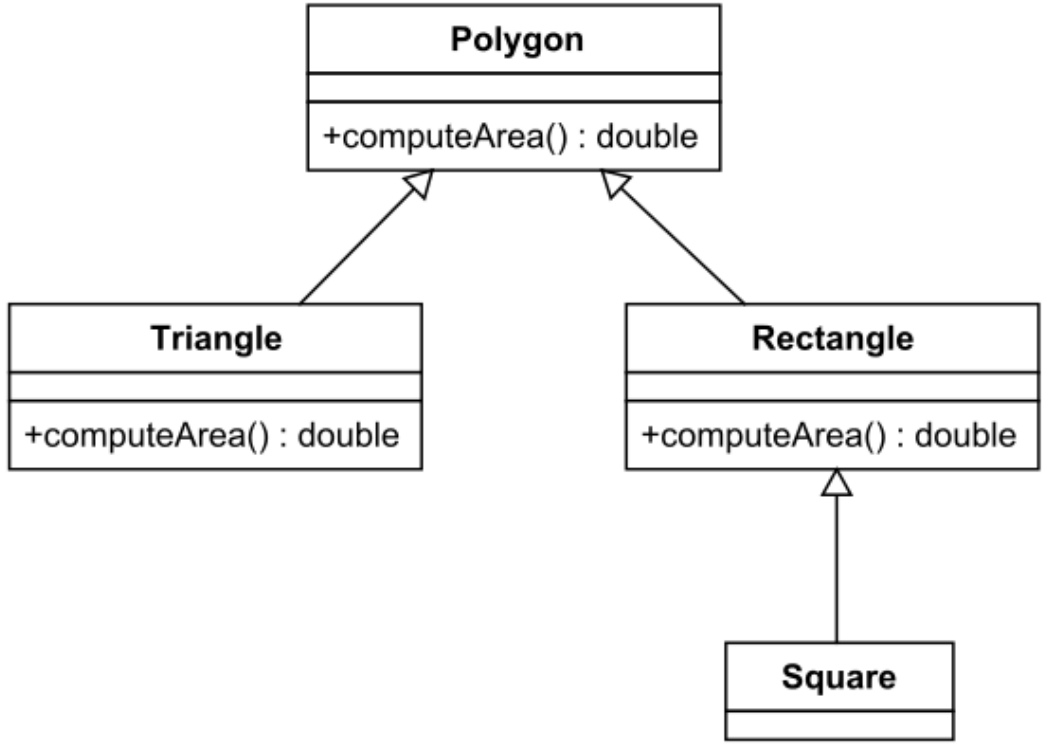
```
public class HumanBeing
{
    protected int age;
    protected double height;
    protected bool gender;
}

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

Polymorphisme

Définition: Une opération définie dans plus d'une classe qui prend **différentes implémentations**

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

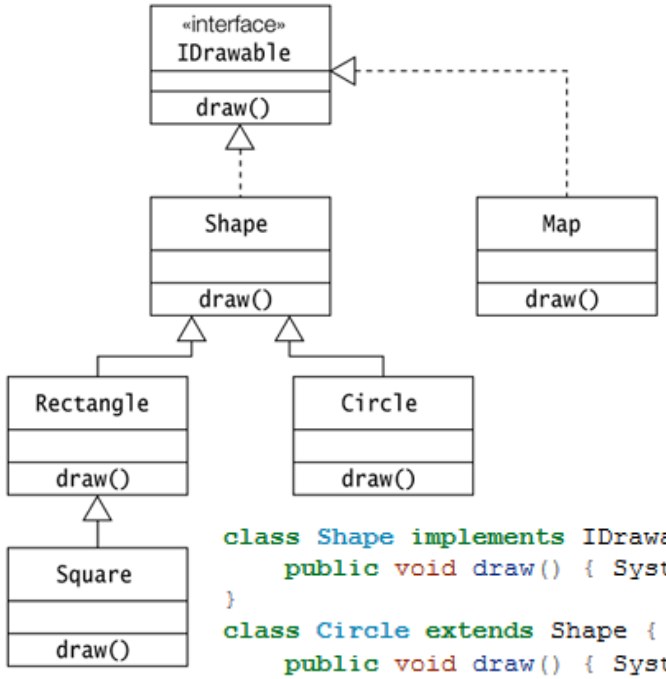


Liaison dynamique

Le code de l'utilisateur est indépendant du type concret reçu

*Exemple: S'attend à un polygone mais peut recevoir un triangle (qui **est un** polygone)*

⚠ **Attention:** Source de problèmes durant le débogage
Différentes implémentations pour la même méthode




```
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();
    }
}
```

Encapsulation

Définition: **regroupement** de **concepts reliés** en une seule **unité** référée par un seul **nom**

- Création d'**abstractions** qui permettent de conceptualiser le problème à un plus haut niveau
- Définir des **types de données abstraits** 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 **concepts 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

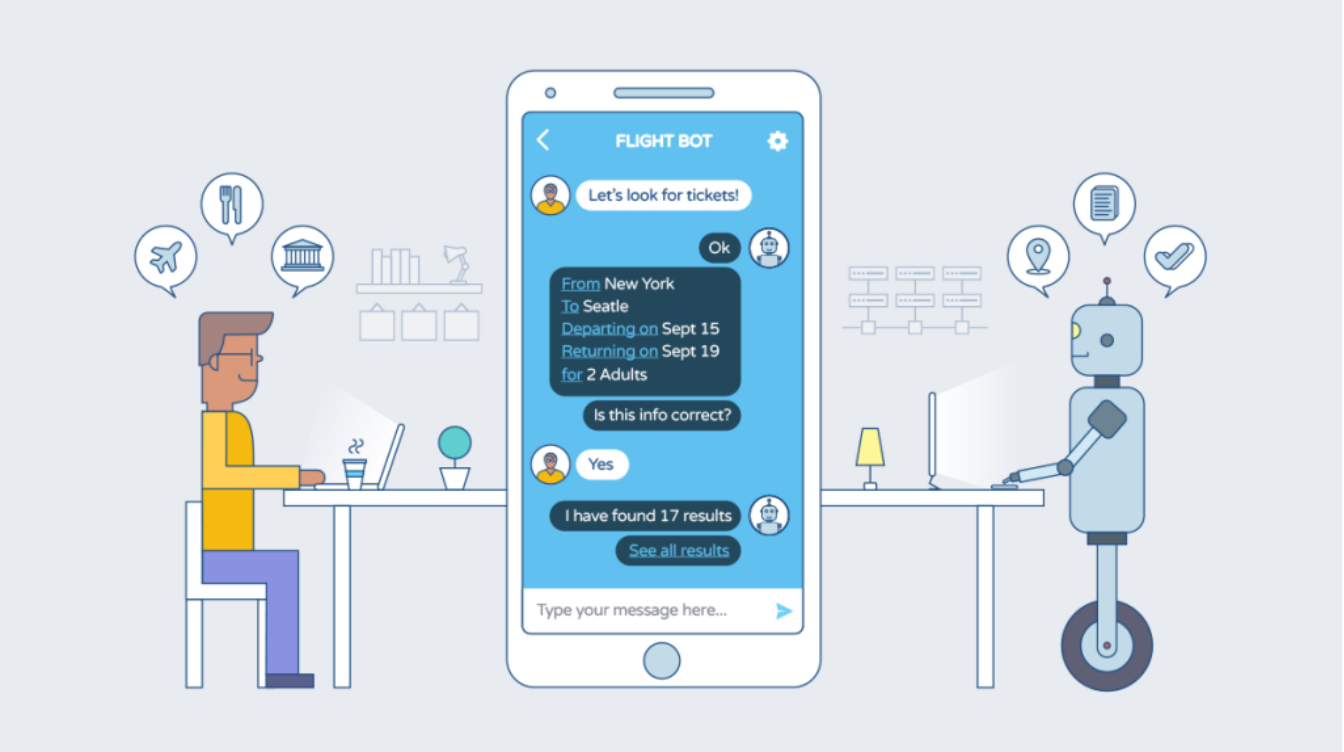
Ce principe est **récursif**, il y a généralement **plusieurs niveaux d'abstraction**

Dissimulation

Définition: utilisation de l'encapsulation pour **restreindre** la perception depuis **l'extérieur** sur les mécanismes **internes.**

POV d'une entité encapsulée

- **Vue publique**
- **Vue privée**



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 facilite à:
  - ✓**Localiser** les décisions de conception
  - ✓**Séparer** l'information de sa représentation

Facilite la **réutilisation**

Règle des getters/setters

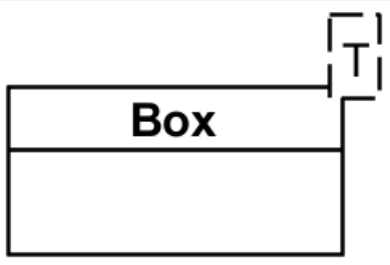
- Ne jamais permettre à d'autres classes d'accéder directement aux attributs de ma classe
- Une fois rendu **privé** , un attribut ne peut plus être changer directement (sans garde fou)
- Rendre les attributs accessibles via les **méthodes** `get/set`

  `obj.ligne` *Dangereux* 👎

  `obj.getLigne()` *Sécuritaire* 👍

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; }
}
```

Résumé

Avantages et inconvénients

Avantages

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

Inconvénients

- Prolifération de fichiers (un par classe)
- Pas idéal pour le développement d’interfaces graphiques
- Classe à la base de la hiérarchie est fragile aux modifications
- Contraint la conception à une manipulation d’objets

Alternatives

Autres paradigmes de programmation

- Programmation fonctionnelle
- Programmation logique, basée sur les règles

Autres approches de développement

- **Ingénierie dirigée par les modèles (MDE)**
- Modélisation spécifique au domaine (DSM)