

# Encapsulation et égalité

## Programmation Orientée Objet

Jean-Christophe Routier  
Licence mention Informatique  
Université Lille 1



classe Client

```
public void orderItem(Order order, String reference) {
    Catalogue cata = order.getCatalogue();
    Item item = cata.getItem(reference);
    item.price = 0;
    order.addItem(item);
}
```

comment « protéger » les attributs ?

## Contrôle d'accès

- **restreindre** la visibilité des attributs ou méthodes d'une classe.
- JAVA : **modificateurs** d'accès précisés lors de la définition d'attributs ou méthodes :

### private/public

**private** accessible uniquement pour les instances de la classe  
càd uniquement depuis le code des méthodes de la classe

**public** accessible pour tout le monde  
càd dans le code de n'importe quelle méthode

exemples :

```
private Author myAuthor;
public void read() { ... }
```

Méthodes et constructeurs aussi peuvent être **private**.

## la classe Item

```
public class Item {
    private float price;
    private String reference;
    public float getPrice() {
        return this.price;
    }
    public float getReference() {
        return this.reference;
    }
    public boolean moreExpensiveThan(Item otherItem) {
        return this.price > otherItem.price;
    }
    public Item(float p, String ref) {
        this.price = p;
        this.reference = ref;
    }
}
```

## getTotalPrice dans Order

```
public class Order {  
    ...  
    public float getTotalPrice() {  
        float total = 0;  
        // cumuler les prix de tous les articles  
        for(Item item : this.allItems()) {  
            total = total + item.price; // interdit private !  
        }  
        return total;  
    }  
}  
  
public class Item {  
    private float price;  
    public float getPrice() {  
        return this.price;  
    }  
    ...  
}
```

## Règle

### Règle

Rendre **privés** les attributs caractérisant l'état de l'objet et fournir **si besoin** des méthodes **publiques** permettant de modifier/accéder à l'attribut

accesseur/modificateur  $\equiv$  getter/setter

attribut author  $\Rightarrow$  getAuthor() : accesseur  
setAuthor(...) : modificateur

principe d'**encapsulation**

**interface publique** d'une classe

intérêts ?

### ■ masquer l'implémentation

$\hookrightarrow$  toute la décomposition du problème n'a pas besoin d'être connue du "programmeur utilisateur"

### ■ permettre l'évolutivité

$\hookrightarrow$  il est possible de modifier tout ce qui n'est pas public sans impact pour le "programmeur utilisateur"

### ■ protéger

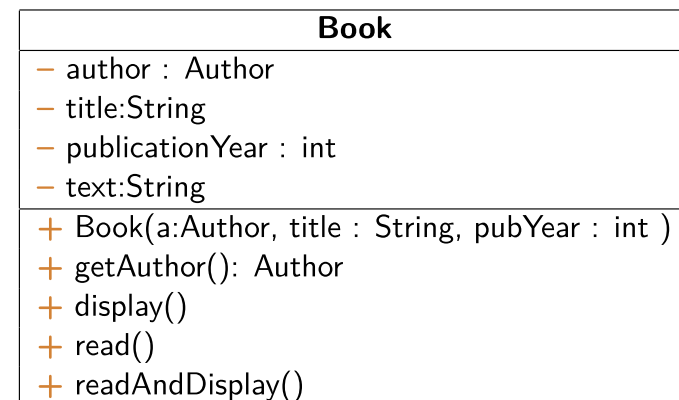
$\hookrightarrow$  ne pas permettre l'accès à tout dès que l'on a une référence de l'objet

$\hookrightarrow$  préserver l'intégrité des objets

$\hookrightarrow$  le "programmeur créateur" contrôle (et est responsable) de son interface par rapport au "programmeur utilisateur"

## UML

$-$  = private et  $+$  = public



Java : schéma standard

```

private Author myAuthor;

public Author getAuthor() {           // accès en lecture
    return this.myAuthor;
}

public void setAuthor(Author newAuthor) {    // accès en écriture
    this.myAuthor = author;
}

```

« contrat »

```

public class Adder {
    public Adder () { this.result = 0; }
    /** result of last computation */

    private int result = 0; ←

    public void compute(int nb1, int nb2) {
        this.result = nb1 + nb2 ;
    }
    /** @return result of last computation made by this adder, 0 if none */
    public int getResult() {
        return this.result;
    }
}

```

Le contrat de getResult() est précisé dans la documentation.

```

// == UTILISATION << ailleurs >>
Adder add = new Adder();
add.compute(5,3);
System.out.println(add.getResult());

add.result = -12;           /\ interdit par compilateur car private /\

System.out.println(add.getResult());           pas de rupture du contrat possible

```

Vive le public

exploitation

Ce qui comptent ce sont les fonctionnalités proposées par une classe,  
son interface publique

```

Disc disk = new Disc(5, new Point(0,0));
float radius = disk.radius();
float diameter = disk.diameter();
float perimeter = disk.perimeter();
float surface = disk.surface();
boolean b = disc.contains(new Point(3,2.32));

```

## Version 1

état = le centre et le diamètre

Disc
- diameter : float - center : Point
+ Disc(radius : float, center : Point) + Disc(center : Point, diameter : float) + surface() : float + perimeter() : float + radius() : float + diameter() : float + center() : Point + contains(p : Point) : boolean

```
public class Disc {  
    private Point center;  
    private float diameter;  
    public Disc(float radius, Point theCenter) {  
        this(theCenter, 2*radius);  
    }  
    public Disc(Point theCenter, float theDiameter) {  
        this.center = theCcenter;  
        this.diameter = theDiameter;  
    }  
    ...  
    public float perimeter() {  
        return (3.14159)* this.diameter;  
    }  
    public float radius() {  
        return this.diameter/2;  
    }  
    public float diameter() {  
        return this.diameter;  
    }  
    ...  
}
```

## Version 2

état = le centre et le rayon

Disc
- radius : float - center : Point
+ Disc(radius : float, center : Point) + Disc(center : Point, diameter : float) + surface() : float + perimeter() : float + radius() : float + diameter() : float + center() : Point + contains(p : Point) : boolean

```
public class Disc {  
    private Point center;  
    private float radius;  
    public Disc(float radius, Point center) {  
        this.center = center;  
        this.radius = radius;  
    }  
    public Disc(Point theCenter, float theDiameter) {  
        this(theDiameter/2, theCenter);  
    }  
    ...  
    public float perimeter() {  
        return 2*(3.14159)* this.radius;  
    }  
    public float radius() {  
        return this.radius;  
    }  
    public float diameter() {  
        return 2* this.radius;  
    }  
    ...  
}
```

## exploitation

```
Disc disk = new Disc(5, new Point(0,0));
float radius = disk.radius();
float diameter = disk.diameter();
float perimeter = disk.perimeter();
float surface = disk.surface();
boolean b = disk.contains(new Point(3,2.32));
```

- dans les 2 cas on arrive à écrire le traitement nécessaire
- même si ce traitement change, les service rendu est le même
- ce qui compte pour l'utilisateur de la classe ce sont les fonctionnalités proposées : les méthodes publiques
- peu importe quelle structure de l'état a été utilisée

Lors de l'analyse objet du problème :

- 1 identifier les méthodes (fonctionnalités) dont on a besoin
- 2 définir l'état en fonction de ce qui est nécessaire pour réaliser ces méthodes

## Attributs et variables

- les **attributs** caractérisent l'état des instances d'une classe. Ils participent à la modélisation du problème.
- les **variables** sont des mémoires locales à des méthodes. Elles sont là pour faciliter la gestion du traitement.
- la notion d'accessibilité (privé/public) n'a de sens **que pour** les attributs.
- la visibilité des variables est limitée au bloc où elles sont déclarées  
**règle de portée**

## les méthodes aussi

intérêt : décomposer les traitements (sans changer l'interface de la classe)

```
public class Thermometer {
    ...
    public String getMessage() {
        String msg = "il fait ";
        if (this.temperature < 10) {
            msg = msg + "froid";
        }
        else if (this.temperature < 22) {
            msg = msg + "moyen";
        }
        else {
            msg = msg + "chaud";
        }
        msg = msg + ":" + this.temperature;
        return msg;
    }
}
```

```
public class Thermometer {
    ...
    public String getMessage() {
        String msg = "il fait ";
        String msg = msg + this.tempToWord();
        return msg + ":" + this.temperature;
    }
    private String tempToWord() {
        if (this.temperature < 10) {
            return "froid";
        }
        else if (this.temperature < 22) {
            return "moyen";
        }
        else {
            return "chaud";
        }
    }
}
```

## Attention DANGER !

```
Book id1Book = new Book();
Book id2Book = id1Book;
```

le contenu de la référence id1Book est copiée dans id2Book,

mais l'objet référencé **n'est pas copié**  
**2 identifiants / 1 objet**

les deux références contiennent la même information sur comment trouver un objet

⇒ càd. le même objet

⇒ envoyer un message à l'objet désigné/référencé par id1Book ou par id2Book revient au même

```
Author tolkien = new Author("Tolkien", "JRR",1892) ;
```

```
// le même objet Author est utilisé pour créer les deux objets Book
Book lotr = new Book(tolkien, "The Lord of the Rings", 1954);
Book hobbit = new Book(tolkien, "The Hobbit", 1937);
```

```
hobbit.getAuthor().getDeathYear(); // ~> UnknownValueException
```

```
lotr.getAuthor().setDeathYear(1973);
lotr.getAuthor().getDeathYear(); // → 1973
```

```
hobbit.getAuthor().getDeathYear(); // → 1973
```

Université Lille 1 - Licence Informatique	Programmation Orientée Objet	21	Université Lille 1 - Licence Informatique	Programmation Orientée Objet	22
Encapsulation oooooooooooooooo	Partage de référence o	Passage par valeur ●oooo	Encapsulation oooooooooooooooo	Partage de référence o	Passage par valeur o●ooo

## Passage des arguments par valeur

en java, les arguments sont transmis **par copie de valeur**.

La **valeur** d'un **paramètre effectif** est **copiée** dans son **paramètre formel**.

Une méthode travaille donc avec une version **locale** des paramètres.

```
public class TestPassageParCopie {
    public void methodeAvecInt(int i) {
        i = 5;
        System.out.println("dans méthode ->"+i); ←
    }
    public static void main(String[] args) {
        TestPassageParCopie test = new TestPassageParCopie();
        int valeur = 3;
        System.out.println("avant -> "+valeur); ←
        test.methodeAvecInt(valeur); ←
        System.out.println("après -> "+valeur); ←
    }
}
```

```
trace d'exécution : java TestPassageParCopie
avant -> ?3
dans méthode -> ?5
après -> ?3
```

```
public class TestPassageParCopie {
    public void methodeAvecDisc(Disc disque) {
        disque = new Disc(5);
        System.out.println("dans méthode ->"+disque); <==
    }
    public static void main(String[] args) {
        TestPassageParCopie test = new TestPassageParCopie();
        Disc d = new Disc(3);
        System.out.println("avant -> "+d); <==
        test.methodeAvecDisc(d); <==
        System.out.println("après -> "+d); <==
    }
}
```

trace d'exécution : java TestPassageParCopie  
 avant -> rayon : ?3  
 dans méthode -> rayon : ?5  
 après -> rayon : ?3

```
public class Disc {
    private int radius;
    public Disc(int r) {
        this.radius = r;
    }
    public void setRadius(int nouveauR) {
        this.radius = nouveauR;
    }
    public String toString() {
        return "rayon : "+this.radius;
    }
}
```

```
public class TestPassageParCopie {
    public void changeDiscRadius(Disc disque) {
        disque.setRadius(5);
        System.out.println("dans méthode ->"+disque.toString()); <==
    }
    public static void main(String[] args) {
        TestPassageParCopie test = new TestPassageParCopie();
        Disc d = new Disc(3);
        System.out.println("avant -> "+d); <==
        test.changeDiscRadius(d); <==
        System.out.println("après -> "+d); <==
    }
}
```

trace d'exécution : java TestPassageParCopie  
 avant -> rayon : ?3  
 dans méthode -> rayon : ?5  
 après -> rayon : ?5

## Passage des arguments par valeur

en java, les arguments sont transmis **par copie de valeur**.

La **valeur** d'un **paramètre effectif** est **copiée** pour liaison au **paramètre formel**.

Une méthode travaille donc avec une version **locale** des paramètres.

mais,

si cette **valeur est une référence**, la référence est **copiée**  
 il y a alors **partage de référence** entre le paramètre formel et le paramètre effectif

## Problème de l'égalité

?

quand peut-on dire que 2 objets sont égaux ?

égalité d'objets ou de valeur ?

- égalité d'objet : les 2 références désignent le même objet  
**égalité testée par l'opérateur ==**
- égalité de valeurs : les objets des 2 références sont *équivalents*  
**égalité testée par la méthode equals**

```
String str1 = new String("Le Seigneur des Anneaux");  
String str2 = new String("Le Seigneur des Anneaux");
```

- 2 références différentes sur 2 objets **différents**

```
str1 == str2  $\implies$  false
```

- les deux objets référencés sont **équivalents**

```
str1.equals(str2)  $\implies$  true
```

la méthode `equals`, doit être définie et adaptée pour chaque classe.

Par défaut, elle se comporte comme `==`