

Encapsulation et égalité

Programmation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



UFR IEEA
Formations en
Informatique de
Lille 1

Contrôle d'accès

- ▶ Lors de la définition d'une classe il est possible de **restreindre** la visibilité des attributs ou méthodes des instances de cette classe.
- ▶ JAVA : **modificateurs** d'accès lors de la déclaration d'attributs ou méthodes :

private/public

private accessible uniquement depuis des instances de la classe

public accessible par tout le monde (ie. tous ceux qui possèdent une référence sur l'objet)

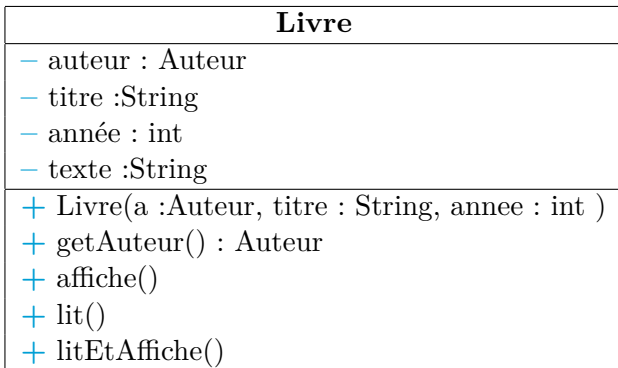
exemples :

```
private Auteur monAuteur;  
public void lit() { ... }
```

Les méthodes et constructeurs aussi peuvent être **private**.

UML

- = private et + = public



Intérêt ?

- ▶ masquer l'implémentation
↪ toute la décomposition du problème n'a besoin d'être connue du "programmeur client"
- ▶ évolutivité
↪ il est possible de modifier tout ce qui n'est pas public sans impact pour le programmeur client
- ▶ protéger
↪ ne pas permettre l'accès à tout dès que l'on a une référence de l'objet

Encapsulation

interface publique d'une classe

Règle

Règle

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

accesseur/modificateur \equiv getter/setter

attribut auteur \implies `getAuteur()` : accesseur
`setAuteur(...)` : modificateur

Pourquoi ?

- ▶ contrôler les accès en lecture et/ou écriture,
↪ préserver l'intégrité des objets.
- ▶ prendre en compte les “2 programmeurs”
↪ le “programmeur créateur” contrôle son interface par rapport au
“programmeur utilisateur”
le “programmeur créateur” est *responsable* de son code, càd qu'il a
une responsabilité de fiabilité vis-à-vis des autres classes qui
utilisent son code
- ▶ protéger le code contre des “usages abusifs”
- ▶ faciliter maintenance/évolution

Illustration

```

public class Additionneur {
    private int resultat = 0;
    public void calcule(int nb1, int nb2) {
        this.resultat = nb1 + nb2;
    }
    public int getResultat() {
        return this.resultat;
    }
}

Additionneur add = new Additionneur();
add.calcule(5,3);
System.out.println(add.getResultat());
add.resultat = -12;          !!! interdit!!!, évite corruption résultat
System.out.println(add.getResultat());

```

- ▶ **resultat** ne doit pas pouvoir être modifié directement, il doit correspondre au résultat de l'addition.
 “**Contrat**” de la classe.

- ▶ gestion de compte en banque, classe `Compte` avec attribut `solde` exprimant le solde

```
Compte compte = new Compte();
```

1. `solde` est “public”

dès que l'on a la référence de `compte` :

si un “programmeur client” programme un module d'affichage de `solde` :

```
System.out.println(compte.solde);
```

il peut alors tout aussi bien modifier les soldes!!!

```
compte.solde = 1000000; // ben tiens!
```

Si l'on décide dans un second temps d'encapsuler `compte` :

tous les programmes clients doivent être modifiés!

2. solde est “private” et encapsulé :

↪ utilisation dès le départ de `compte.getSolde()` et

`compte.setSolde(...)`

Le “programmeur créateur” peut modifier le comportement sans impact :

- ▶ contrôle lors de la modification
- ▶ suppression de l'attribut `solde` remplacé par `credit` et `debit`
- ▶ etc.

Java : Schéma standard

```
private Auteur monAuteur ;

public void setMonAuteur(Auteur auteur) {
    this.monAuteur = auteur ;
}

public Auteur getMonAuteur() {
    return this.monAuteur ;
}
```

```
public class Livre {
    // les attributs de la classe livre
    private Auteur auteur;
    private String titre;
    private int annee;
    private String texte;
    // constructeur
    public Livre(Auteur unAuteur, String titre, int annee, String texte) {
        this.auteur = unAuteur;
        this.titre = titre;
        this.annee = annee;
        this.texte = texte;
    }
    // les méthodes de la classe Livre
    public Auteur getAuteur() {
        return this.auteur;
    }
    public void setAuteur(Auteur auteur) {
        this.auteur = auteur;
    }
}
```

Exploitation

(en dehors de la class Livre)

```

Livre leLivre = new Livre("JRR Tolkien", "Le Seigneur des Anneaux", 1954);
leLivre.affiche();
System.out.println(leLivre.auteur);           !!! interdit!!!
System.out.println(leLivre.getAuteur());
leLivre.auteur = new Auteur("un autre");    !!! interdit!!!
leLivre.texte = "Quand M. Bilbon Sacquet, ..."; !!! interdit!!!
leLivre.setAuteur(new Auteur("un autre"));

```

Vive le public

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

Considérons une application dans laquelle on manipule des objets
 Disque.

L'important est ce que l'on peut faire avec ces disques :

```

surface() :float,           perimetre() :float,           getRayon() :float,
getCentre() :Point,       appartient(p :Point) :boolean,           etc.
  
```

Quel état pour ces objets ?

Version1

état = le centre et le rayon

Disque
<ul style="list-style-type: none"> - rayon : float - centre : Point
<ul style="list-style-type: none"> + Disque(centre : Point, rayon : float) + Disque(centre : Point, diametre : float) + Disque(point1 : Point, poin2 : Point, point3 : Point) + surface() : float + perimetre() : float + getRayon() : float + getDiametre() : float + getCentre() : Point + appartient(p : Point) : boolean

```
public class Disque {
    private Point centre;
    private float rayon;
    public Disque(Point centre, float diametre) {
        this.centre = centre;
        this.rayon = diametre/2;
    }
    ...
    public float perimetre() {
        return 2*(3.14159)* this.rayon;
    }
    public float getRayon() {
        return this.rayon;
    }
    public float getDiametre() {
        return 2* this.rayon;
    }
    ...
}
```

Version 2

état = le centre et le diamètre

Disque
- diamètre : float
- centre : Point
+ Disque(centre : Point, rayon : float)
+ Disque(centre : Point, diamètre : float)
+ Disque(point1 : Point, point2 : Point, point3 : Point)
+ surface() : float
+ perimetre() : float
+ getRayon() : float
+ getDiametre() : float
+ getCentre() : Point
+ appartient(p : Point) : boolean

```
public class Disque {
    private Point centre;
    private float diametre;
    public Disque(Point centre, float diametre) {
        this.centre = centre;
        this.diametre = diametre;
    }
    ...
    public float perimetre() {
        return (3.14159)* this.diametre;
    }
    public float getRayon() {
        return this.diametre/2;
    }
    public float getDiametre() {
        return this.diametre;
    }
    ...
}
```

Version 3

état = 3 points distincts situés sur le périmètre du disque

Disque
- p1 : Point
- p2 : Point
- p3 : Point
+ Disque(centre : Point, rayon : float)
+ Disque(centre : Point, diametre : float)
+ Disque(point1 : Point, poin2 : Point, point3 : Point)
+ surface() : float
+ perimetre() : float
+ getRayon() : float
+ getDiametre() : float
+ getCentre() : Point
+ appartient(p : Point) : boolean

```
public class Disque {
    private Point p1;
    private Point p2;
    private Point p3;
    public Disque(Point centre, float diametre) {
        ... traitement pour calculer 3 points distincts
        ... du périmètre du disque à partir de centre et diametre
        this.p1 = ...;
        this.p2 = ...;
        this.p3 = ...;
    }
    ...
    public float perimetre() {
        return 2*(3.14159*this.getRayon());
    }
    public float getRayon() {
        ... traitement pour calculer le rayon à partir des 3 points
        return ...;
    }
    public float getDiametre() {
        return 2* this.getRayon();
    }
    ...
}
```

- ▶ dans les 3 cas on arrive à écrire le traitement nécessaire
- ▶ même si ce traitement change, le 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 construire 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.
- ▶ l'accès aux variables est limité au bloc où elles sont déclarées : règle de **portée**

Attention DANGER !

```
Livre id1Livres = new Livre() ;  
Livre id2Livres = id1Livres ;
```

le contenu de la référence `id1Livres` est copiée dans `id2Livres`,

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 `id1Livres` ou par `id2Livres` revient au même

Passage des arguments par valeur

En java, les arguments sont transmis **pas valeur**.

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

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

trace d'exécution : java TestPassageParCopie

avant -> 3

dans méthode -> 5

après -> 3

```
private static class Disque {
    private int rayon;
    public Disque(int r) {
        this.rayon = r;
    }
    public void setRayon(int nouveauR) {
        this.rayon = nouveauR;
    }
    public String toString() {
        return ""+this.rayon;
    }
}
```

Mais attention

Les références étant copiées, il y a **partage de référence** entre le paramètre formel et le paramètre effectif!

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

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

Problème de l'égalité

- ▶ identificateur d'objet = information sur comment trouver l'objet référencé
- ▶ comparer 2 identificateurs = vérifier si cette information est la même
- ▶ **Rien à voir avec le contenu des objets référencés**
- ▶ identificateur = pointeur, donc on retrouve la problématique de l'égalité de pointeur.

```
String ch1 = new String("Le Seigneur des Anneaux");
String ch2 = new String("Le Seigneur des Anneaux");
```

- ▶ 2 références différentes sur 2 objets **différents**
- ▶ `ch1 == ch2` \implies `false`
- ▶ **pour comparer les états des instances** on utilise la méthode `equals`
`ch1.equals(ch2)` \implies `true`
- ▶ La méthode `equals`, doit être définie et adaptée pour chaque classe.
Par défaut, elle fait comme `==` !
- ▶ `String ch3 = ch1;`
 \hookrightarrow range dans `ch3` l'information stockée dans `ch1`

`ch1 == ch3` \implies `true`
`ch1.equals(ch3)` \implies `true`

En Java tout est objet ?

Oui... sauf les types primitifs

type primitif	ex	Size	minimum	maximum	classe "Wrapper"
boolean	true	-	-	-	Boolean
char	'x'	16 bits	Unicode 0 (\u0000)	Unicode $2^{16} - 1$ (\uFFFF)	Character
byte	12	8 bits	-128	+127	Byte
short	12	16 bits	-2^{15}	$+2^{15} - 1$	Short
int	12	32 bits	-2^{31}	$+2^{31} - 1$	Integer
long	12L	64 bits	-2^{63}	$+2^{63} - 1$	Long
float	12.0f	32 bits	$-10^{38} \dots - 10^{-38}$	$10^{-38} \dots + 10^{38}$	Float
double	12.0	64 bits	$-10^{308} \dots - 10^{-308}$	$10^{-308} \dots + 10^{308}$	Double
void	-	-	-	-	Void

justificatif? facilité d'utilisation et de manipulation

Variables primitives

déclaration de variable primitive : pas de new (pas d'objet !)

```
int i;                boolean fini = true;
```

variable primitive = espace mémoire réservé

taille fixe (objet pas de taille fixe (référence si!) y compris même classe)

- ▶ variable de **type primitif** contient la **valeur** de la **variable**
- ▶ variable **référence d'objet** contient l'information sur **comment trouver l'objet**

Retour sur égalité

Type primitif : variable contient valeur

donc

```
int i = 5;
```

```
int j = 5;
```

```
i == j  $\implies$  true
```

== ne regarde que le contenu des variables

nul n'est parfait...

```
int biggest = Integer.MAX_VALUE ;
int biggerThanBiggest = biggest+1;
System.out.println("biggest = "+biggest);
System.out.println("biggerThanBiggest = "+biggerThanBiggest);
```

```
+-----+
| biggest = 2147483647
| biggerThanBiggest = -2147483648
+-----+
```