Encapsulation et égalité

Programmation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1

Université Lille1 Sciences et Technologies | IEEA - FIL Informatique

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 String monAuteur;
public void lit() { ... }
```


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 ≡ getter/setter

attribut `auteur` ⟹ `getAuteur()` : accesseur
`setAuteur(...)` : modificateur

Pourquoi ?

- contrôler les accès en lecture et/ou écriture,
  ↪ préserver l'intégrité des objets.
- prise en compte des "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 int calcule(int nb1, int nb2) {
      return this.resultat = nb1 + nb2 ;
   }
   public void reset() {
      this.resultat = 0;
   }
   public int getResultat() {
      return this.resultat;
   }
}

Additionneur add = new Additionneur();
add.calcule(5,3);
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 String monAuteur;

public void setMonAuteur(String auteur) {
   this.monAuteur = auteur;
}
public String getMonAuteur() {
   return this.monAuteur;
}
```



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


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 = "un autre";                           !!!  interdit !!!
leLivre.texte = "Quand M. Bilbon Sacquet, ...";        !!!  interdit !!!
leLivre.setAuteur("un autre");
```


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 id1Livre = new Livre();
Livre id2Livre = id1Livre;
```

le contenu de la référence `id1Livre` est copiée dans `id2Livre`,

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 `id1Livre` ou par `id2Livre` revient au même

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` $\Longrightarrow$ `false`
- pour comparer les états des instances on utilise la méthode **`equals`**
  `ch1.equals(ch2)` $\Longrightarrow$ `true`
- La méthode `equals`, doit être définie et adaptée pour chaque classe. Par défaut, elle fait comme == !
- `String ch3 = ch1;`
  ↪ range dans `ch3` l'information stockée dans `ch1`

`ch1 == ch3` $\Longrightarrow$ `true`

`ch1.equals(ch3)` $\Longrightarrow$ `true`


```
public class C {
   private int val = 5;
   public void setVal(int val) { this.val = val; }
   public int getVal() { return val; }
}

public class C1 {
   private C o1;
   public void setO1(C i) {
      this.o1 = i;
   }
   public C getO1() { return this.o1; }
}

public class C2 {
   private C o2;
   public void setO2(C i) {
      this.o2 = i;
   }
   public C getO2() { return this.o2; }
}

C i = new C();
C1 i1 = new C1();
C2 i2 = new C2();
i1.setO1(i);                    // !!!  i1 et i2 partagent
i2.setO2(i);                    // une référence !!!

// toute manipulation de i1 sur o1                    +-trace---------
i1.getO1().setVal(18);                                |
// est nécessairement ''perçue'' au niveau de o2 ds i2 | > 18
System.out.println(" > "+i2.getO2().getVal());        |
// illustration :                                     | >> true
System.out.println(" >> "+i1.getO1() == i2.getO2() ); |
                                                      +---------------
```


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	$-10^{38}\ldots-10^{-38}$	$10^{-38}\ldots+10^{38}$	Float
double	12.0	64 bits	$-10^{308}\ldots-10^{-308}$	$10^{-38}\ldots+10^{38}$	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` $\Longrightarrow$ `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
+-----------------------------------
```