

Programmation Orientée Objets avec Java

Prof. Yassin Rekik et Prof. Stéphane Malandain – 2023 - 2024

(Basé sur le cours de Joël Cavat)



Chapitre 2

Programmation Orientée Objets

Concepts traités

2

- Classes et Objets
- Attributs
- Méthodes
- Visibilité et encapsulation
- Accesseurs
- Surcharge des méthodes
- Constructeurs
- Méthodes et attributs statiques

- Un objet est une variable complexe
 - La définition des attributs : rappelle la notion de STRUCTURE en C
 - La notion des opérations ou méthodes : rappelle les types Abstraits
- Les objets représentent les abstractions qui vont être manipulées lors de l'exécution d'un programme Java
 - Gestion de compte bancaire : Compte , Client , Agence , Banque , ...

Exemple

- Compte bancaire
 - ID = 6648939032873478
 - client = Michel Blanc
 - Balance = 5537873 CHF
 -
- Client
 - Nom = Martin
 - Prénom = Jacques
 - Date de naissance = 16.04.1984
 - Adresse = rue des lilas, 8
 - ID Client = XDA88746648CVFG

Les méthodes sur les objets

5

- Ce qui est important dans la POO :
 - ce n'est pas seulement la modélisation des champs/attributs des objets, mais c'est surtout la modélisation des opérations associées : comportement
- Exemple des objets Comptes
 - Ce qu'on peut / veut / souhaite pouvoir faire comme traitement sur un compte
 - Créer un compte
 - Faire un versement sur un compte
 - Faire un retrait d'un compte
 - Modifier le statut d'un compte
 - Afficher la situation d'un compte
 - ...

- La classe est la collection des objets de même structure
- La classe est le «MODÈLE» pour un ensemble d'objets similaires

Exemple

7

- Modélisation des attributs

```
1  public class Account {  
2      public String owner;  
3      public double balance;  
4  }
```

- Le fichier doit s'appeler `Account.java` (même nom que la classe)

Exemple

- Modélisation des opérations : Les méthodes

```
1  public class Account {  
2      public String owner;  
3      public double balance;  
4  
5      /* getters */  
6      public String getOwner() { return this.owner; }  
7      public double getBalance() { return this.balance; }  
8  
9      /* setters */  
10     public void setOwner(String owner) {  
11         |   this.owner = owner;  
12     }  
13     public void setBalance(double balance) {  
14         |   this.balance = balance;  
15     }  
16 }
```


Classe = nouveau type

9

- Chaque classe définit un nouveau type complexe en Java
 - Classes offertes par Java : String - Float – Integer - ...
 - Classes définies par les développeurs : Personne – Client – Cours – Compte – Forme
- Comme pour les variables de type primitif, chaque objet créé en Java doit être déclaré avec son type complexe associé : la classe

```
Personne p1 ;  
Account myAccount ;
```

- Les attributs d'une classe représentent les valeurs qui peuvent caractériser chaque objet de cette classe

```
1  public class Account {  
2      public String owner;  
3      public double balance;  
4  }
```

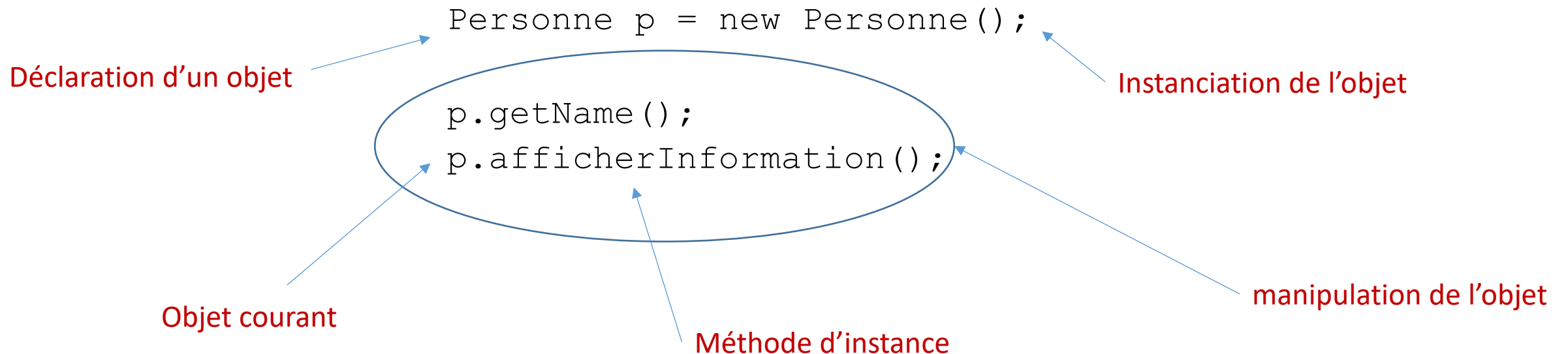
- La définition d'un attribut intègre
 - La visibilité de l'attribut : public , private
 - Le type de l'attribut : int , float , String ,
 - Le nom de l'attribut
 - Éventuellement une initialisation de la valeur de l'attribut

Attributs complexes

11

- Certains attributs d'une classe peuvent eux aussi être de type complexe
 - Tableaux
 - Chaînes de caractères
 - Autres classes
- Exemples
 - Un des attributs d'un **Compte Bancaire** est le **Client** détenteur du compte
 - Un des attributs d'un **Etudiant** est le **Groupe** auquel il appartient

- Les Méthodes d'une classe représentent les opérations possibles sur n'importe quel objet de cette classe
- Par défaut : une méthode est une «méthode d'instance» , appelée encore «méthode d'objet» : elle doit toujours s'exécuter sur un objet spécifique



Example

13

```
1  public class Account {
2      public String owner;
3      public double balance;
4
5      /* getters */
6      public String getOwner() { return this.owner; }
7      public double getBalance() { return this.balance; }
8
9      /* setters */
10     public void setOwner(String owner) {
11         |   this.owner = owner;
12     }
13     public void setBalance(double balance) {
14         |   this.balance = balance;
15     }
16 }
```

Utilisation des objets

14

- Définition de la classe

```
1 public class Account {  
2     public String owner;  
3     public double balance;  
4 }
```

- Création d'une variable de type Account
 - Dans la même classe, ou
 - Dans une autre classe

Création d'un objet

```
Account myAccount = new Account();
```

- Déclaration d'une référence : Account myAccount
- Instanciation : new Account()
- Initialisation : Account()



Utilisation des objets

15

```
1  Account myAccount = new Account();
2  myAccount.owner = "Marc";
3  myAccount.balance = 15000.0;
4
5  System.out.println(myAccount.owner + " possède: " + myAccount.balance);
6
7  // Affichage :   Marc possède: 15000.0
8
9  // Modification des champs possible
10 myAccount.balance *= 2.0;
11
```

Encapsulation

16

- Principe de base dans la programmation orientée objet
- Cacher l'implémentation interne d'une classe : limiter la visibilité
- Une classe doit uniquement exposer une interface permettant de la manipuler
 - Certains attributs publics : à limiter au maximum, sauf nécessité absolue
 - Certaines méthodes publiques : suffisantes pour manipuler les objets

Retour sur l'exemple

17

```
1  Account myAccount = new Account();
2  myAccount.owner = "Marc";
3  myAccount.balance = 15000.0;
4
5  System.out.println(myAccount.owner + " possède: " + myAccount.balance);
6
7  // Affichage :   Marc possède: 15000.0
8
9  // Modification des champs possible
10 myAccount.balance *= 2.0;
11
```

Retour sur l'exemple

18

- Remarques
 - Encapsulation violée
 - Modification possible des champs : une mauvaise pratique
- Améliorations possibles
 - Rendre les champs privés
 - Accès et modifications des attributs à l'aide de méthodes
 - Accès via des accesseurs (getters) généralement `getXXX`
 - Modification via des modificateurs (setters) généralement `setXXX`

Ou XXX est le nom d'un champ

- Quatre visibilités possibles en Java
 - public : visible par toutes les autres classes
 - default : visibilité par défaut , visible au sein du même Package
 - protected : visible au sein de la classe et de ces sous-classes (héritage à voir)
 - private : visible uniquement au sein de la même classe

Limiter la visibilité

20

```
1 public class Account {  
2     private String owner;  
3     private double balance;  
4 }
```

On ne peut plus accéder aux champs !

```
1 Account myAccount = new Account();  
2 myAccount.owner = "Marc";  
3 myAccount.balance = 15000.0;  
4  
5 System.out.println(myAccount.owner + " possède: " + myAccount.balance);  
6
```

Accesseurs

21

```
1  public class Account {
2      private String owner;
3      private double balance;
4
5      /* Ajout des Méthodes */
6
7      /* getters */
8      public String getOwner() { return this.owner; }
9      public double getBalance() { return this.balance; }
10
11     /* setters */
12     public void setOwner(String owner) {
13         | this.owner = owner;
14     }
15     public void setBalance(double balance) {
16         | this.balance = balance;
17     }
18 }
```

Utilisation avec les accesseurs

22

```
1  /* Utilisation */  
2  
3  Account myAccount = new Account();  
4  myAccount.setOwner("Marc");  
5  myAccount.setBalance(15000.0);  
6  
7  System.out.println(myAccount.getOwner() + " possède: " + myAccount.getBalance());  
8
```

Encapsulation : faisons mieux

23

- Dans certains cas, la modification des attributs n'est pas justifiée, ou elle nécessite des actions spécifiques
- Dans l'exemple des `Account`
 - Le changement du détenteur est impossible. On crée alors juste un getteur , pas de setteur
 - Le changement de la Balance ne peut se faire que via un dépôt ou un retrait

Encapsulation : faisons mieux

24

```
1  public class Account {
2      private final String owner;
3      private double balance;
4
5      public Account(String owner, double balance) {
6          this.owner = owner;
7          this.balance = balance;
8      }
9
10     public Account(String owner) { this(owner, 0.0); }
11
12     /* getters */
13     public String getOwner() { return this.owner; }
14     public double getBalance() { return this.balance; }
15
16     public void deposit(double amount) { this.balance += amount; }
17     public void withdraw(double amount) { this.deposit(-amount); }
18 }
```


- Modéliser une classe : Livre
- Donner des exemples de code d'utilisation

Le passage de paramètre en Java

26

- Le passage de paramètre en Java dépend du type de paramètre
 - Si le paramètre est d'un type PRIMITIF
 - Passage par valeur
 - Si le paramètre est d'un type COMPLEXE
 - Passage par valeur, **mais la valeur est une référence !!!**

Le passage de paramètre en Java

27

```
1  public static void test(maClasse b) {
2      b.nom = "Paul"; // nouveau pointeur vers l'emplacement en mémoire
3      // est créé puis l'attribut est modifié.
4  }
5
6  public static void main(String[] args) {
7
8      maClasse a = new maClasse();
9      a.nom = "Jean";
10
11     test(a);
12     System.out.println(a.nom); // affiche "Paul"
13 }
```

Le passage de paramètre en Java

28

```
1  public static void test(maClasse b) {
2      b = new maClasse(); // La variable b contient un pointeur
3                          // vers un nouvel emplacement en mémoire.
4
5      b.nom = "Paul"; // La valeur de l'attribut stocké dans le
6                      // nouvel emplacement est modifié.
7  }
8
9  public static void main(String[] args) {
10
11      maClasse a = new maClasse();
12      a.nom = "Jean";
13
14      test(a);
15      System.out.println(a.nom); // affiche "Jean"
16                                // car la valeur n'a pas été modifiée
17  }
```

Constructeur

29

- Revenons à notre exemple sur les `Account`
- Aucune valeur par défaut pour le nom

```
2  Account myAccount = new Account();  
3  
4  System.out.println(myAccount.getOwner().toUpperCase()); // Runtime Error  
5
```

- Déclarer et créer des objets sans initialisation est dangereux
- Risque d'avoir des attributs complexes `Null`

- Amélioration : Initialiser les objets créés : Définir des **CONSTRUCTEURS**

```
public class Account {  
    private String owner;  
    private double balance;  
  
    public Account(String owner, double balance) {  
        this.owner = owner;  
        this.balance = balance;  
    }  
  
    public Account(String owner) {  
        this(owner, 0.0);} // Appel du constructeur à un arg.
```

- Le constructeur est une méthode ayant toujours le même nom que la classe
- La méthode `constructeur` est appelé à chaque fois qu'on appelle **new**
- On peut (préférable) avoir plusieurs constructeurs pour la même classe
- La définition de plusieurs constructeurs est basée sur la notion de **SURCHARGE**

Constructeur par défaut

32

- Il y a toujours un constructeur par défaut VIDE pour chaque classe
- Le constructeur n'est plus valable dès qu'on définit un autre constructeur

- Remarques

- il n'est plus possible d'utiliser un constructeur sans argument à moins d'en créer un (`public Account() { ... }`)
- On oblige l'utilisateur à initialiser les valeurs importantes

```
public class Account {  
    private String owner;  
    private double balance;  
  
    public Account(String owner, double balance) {  
        this.owner = owner;  
        this.balance = balance;  
    }  
  
    public Account(String owner) {  
        this(owner, 0.0);} // Appel du constructeur à un arg.
```


Surcharge des constructeurs

33

- La surcharge est un mécanisme en Java permettant d'utiliser le même nom de méthode plusieurs fois pour définir des méthodes ayant le même nom mais avec des signatures différentes

- Type des paramètres différent
- Nombre de paramètres différent
- Les deux

```
public class Account {  
    private String owner;  
    private double balance;  
  
    public Account(String owner, double balance) {  
        this.owner = owner;  
        this.balance = balance;  
    }  
  
    public Account(String owner) {  
        this(owner, 0.0); } // Appel du constructeur à un arg.
```

Nom commun

Paramètres différents

- Reprendre la classe Livre
 - Gérer l'initialisation
 - Re-tester votre classe

- Jusqu'à maintenant, nous avons définie une classe uniquement avec
 - Des attributs d'instance
 - Des méthodes d'instance
- Un attribut d'instance est toujours lié à un objet particulier
- Une méthode d'instance est toujours appliquée sur un objet particulier
- Que faire si un attribut est commun à toute la classe, donc à tous les objets de la classe ?
 - Exemple : la classe Personne et l'attribut AgeMajorité
- Que faire si la méthode proposée par une classe est générique et ne s'applique pas à un objet particulier ?
 - Exemple Classe Math, méthode calculerPuissance

Exemple

36

```
1  public class Point{
2      private static int origine; // abscisse absolue de l'origine
3      private int x;              // abscisse du point
4
5      public Point(int x) { this.x = x;}
6      public void affiche() {
7          System.out.print("abscisse = "+(this.x-origine));
8          System.out.println(" relative à l'origine =" +origine);
9      }
10     public static void setOrigine(int o) {origine=o;}
11     public static int getOrigine() { return origine;}
12 }
13
14 public class TstOrig {
15     public static void main(String[] args) {
16         Point a = new Point(10);
17         a.affiche();
18         Point.setOrigine(3);
19         a.affiche();
20     }
21 }
```

Suite de l'exemple

37

- La méthode `Max` retourne le point le plus éloigné de l'origine avec une méthode `static`:

```
public static Point max(Point a, Point b) {  
    if (Math.abs(a.x - origine) > Math.abs(b.x - origine))  
        return a;  
    else  
        return b;  
}
```

- Les appels

```
Point p1 = new Point(4);  
Point p2 = new Point(9);  
Point.setOrigine(5);  
Point p = Point.max(p1,p2);  
p.affiche();
```

Suite de l'exemple

38

- Même exemple, mais avec une méthode usuelle:

```
public Point max(Point a) {  
    if (Math.abs(this.x - origine) > Math.abs(a.x - origine))  
        return this;  
    else  
        return a;  
}
```

- Les appels

```
Point p1 = new Point(4);  
Point p2 = new Point(9);  
Point.setOrigine(5);  
Point p = p1.max(p2);  
p.affiche();
```

Le mot clef this

39

- Le mot-clé `this` fait référence à l'objet courant.
- Il est obligatoire si un champ est caché par un argument :

```
public void setOwner(String owner) {  
    this.owner = owner;  
}
```

- Il permet d'appeler un autre constructeur

```
public Account(String owner) {  
    this(owner, balance: 0.0) // Appel du constructeur à un arg.
```

Égalité des objets

40

- Attention :
 - l'égalité des objets en Java n'est pas une égalité des contenus, mais plutôt une égalité des références
- Le résultat du test avec deux objets est **true** uniquement si les deux variables pointent le même objet en mémoire

Redéfinition de equals()

41

- Toutes les classes héritent de la super classe `Object`
- Elles admettent toutes une méthode

```
public boolean equals(Object obj)
```

- Pour tester l'égalité des contenus entre objets, vous devez donc redéfinir la méthode **`equals()`**

Exemple

42

```
1  class Identite {
2      private String nom;
3      private String prenom;
4
5      Identite(String nom, String prenom) {
6          this.nom = nom;
7          this.prenom = prenom;
8      }
9
10     String getNom() {
11         return nom;
12     }
13
14     String getPrenom() {
15         return prenom;
16     }
17
18     public boolean equals(Object obj) {
19         return (obj instanceof Identite) &&
20             ((Identite)obj).getNom().equals(nom) &&
21             ((Identite)obj).getPrenom().equals(prenom);
22     }
23 }
```

```
1  class Info {
2      private int val;
3      private Identite identite ;
4
5      Info(int val, String nom, String prenom) {
6          this.val = val;
7          identite = new Identite(nom, prenom);
8      }
9
10     int getVal() {
11         return val;
12     }
13
14     Identite getIdentite() {
15         return identite;
16     }
17
18     public boolean equals(Object obj) {
19         return (obj instanceof Info) &&
20             (((Info)obj).getVal() == this.val) &&
21             ((Info)obj).getIdentite().equals(identite);
22     }
23 }
```

Suite exemple

44

```
1  class EssaiInfo {
2      public static void main(String arg[]) {
3          Info info1 = new Info(15, "Charon", "Irene");
4          Info info2 = new Info(15, "Charon", "Irene");
5          Info info3 = new Info(15, "Charon", "Lou");
6          Info info4 = new Info(12, "Charon", "Irene");
7
8          System.out.println("avec ==, info1 et info2 sont egaux : " + (info1 == info2));
9          System.out.println("avec equals, info1 et info2 sont egaux : " + info1.equals(info2));
10         System.out.println("avec equals, info1 et info3 sont egaux : " + info1.equals(info3));
11         System.out.println("avec equals, info1 et info4 sont egaux : " + info1.equals(info4));
12     }
13 }
```

On obtient :

```
avec ==, info1 et info2 sont egaux : false
avec equals, info1 et info2 sont egaux : true
avec equals, info1 et info3 sont egaux : false
avec equals, info1 et info4 sont egaux : false
```

Copie d'objets

45

```
Object ob1 = new Object();  
Object ob2 = ob1;
```

- `ob1` et `ob2` pointent vers le même objet maintenant car il ne s'agit que d'une référence aux données d'origine ; aucune copie ne se produit ici.

Copie superficielle

46

- En Java, `java.lang.Object` fournit la méthode `clone()`. Cette méthode est largement utilisée pour créer une copie de l'objet.
 - L'implémentation par défaut `Object.clone()`
 - La méthode renvoie une copie exacte de l'objet d'origine.
 - Il le fait en affectant champ par champ des types primitifs, mutables et immuables.
- Autrement dit, `Object.clone()` crée un nouvel objet du même type d'exécution que l'objet d'origine, et pour chaque champ primitif, mutable et immuable, il exécute `newObj.field = obj.field` opération, où `newObj` et `obj` sont respectivement le nouvel objet et l'objet d'origine.
- Attention quand les attributs sont eux aussi des références !!!

Bonnes pratiques, pour finir

47

- La cohésion est le fait qu'un objet regroupe les attributs et les méthodes qui le concerne, et ne s'occupe pas d'autres aspects en dehors de son périmètre
 - Une classe `Etudiant` doit gérer les attributs d'un étudiant
 - Elle ne doit jamais s'occuper de la classe ou du groupe de l'étudiant
 - Elle ne doit rester Cohérente
- Les bonnes pratiques en POO
 - Modularité
 - Encapsulation
 - Cohérence et limitation de responsabilité
- Programmer toujours pour les autres, qui sont par définition des «incompétents»