Classes et Objets

Rezak AZIZ

CNAM

Objectifs

- Comprendre la relation entre **classe** et **objet**.
- Comprendre les principes de la POO
  - Abstraction
  - Encapsulation
- Comprendre la notion de package.

Structure d'une classe en Java

```
// Déclaration d'une classe
public class NomDeClasse {

    // 1. Attributs
    type nomAttribut1;
    type nomAttribut2;

    // 2. Méthode
    retour nomMethode(type param) {
        // corps de la méthode
        return ...;
    }
}
```

```
public class Livre {
    String titre;
    String auteur;

    // Méthode
    void afficher() {
        System.out.println(
          titre + " - " + auteur
        );
    }
}
```

Classe de test

```
public class Main {
    public static void main(String[] args) {
        // Création d'un objet Livre
        Livre l1 = new Livre();

        // Appel d'une méthode
        l1.afficher();
    }
}
```

Explications :

- Le mot-clé `new` sert à **créer un objet** à partir d'une classe.
- Ici : `Livre l1 = new Livre();` crée un objet de type `Livre`.
- Une fois l'objet créé, on peut **appeler ses méthodes**. Exemple : `l1.afficher();`

Exercice

1. Créer un projet Java sous le nom de TP_Point
2. Créer une classe `Point` :
   - Attributs : `x`, `y` (entiers).
   - Méthode : `afficher()` qui affiche les coordonnées.
3. Créer une classe `Main` pour tester

Objectif : créer des objets `Point` et afficher leurs coordonnées.

Vers l'encapsulation

Soit un compte bancaire où on peut déposer ou retirer de l'argent.

Problèmes :

- Solde négatif imposé arbitrairement.
- La vérification métier est contourné.
- Données incohérentes dans le système.

Solution: l'encapsulation

Code

```
class CompteBancaire {
    double solde; // accessible directement
    void deposer(double montant) {
        solde += montant;
    }
    void retirer(double montant) {
        if (solde > 0)
            solde -= montant;
    }
}

CompteBancaire c = new CompteBancaire();
c.solde = -5000; //  Violation !
```

Encapsulation (1)

Sans encapsulation, n'importe quel code peut modifier le solde directement. Cela peut créer des incohérences ou contourner les règles métiers.

Encapsulation (2)

- Regrouper les données et les traitements dans une même classe et contrôler l'accès aux données.
- Cacher l'état interne d'un objet et ne révéler qu'une interface publique.

- Pour réaliser l'**encapsulation**, on utilise les modificateurs d'accès.
- Les deux plus utilisés :
  - `private` : l'attribut ou la méthode est accessible uniquement à l'intérieur de la classe.
  - `public` : l'attribut ou la méthode est accessible depuis l'extérieur de la classe.
- Bonne pratique :
  - déclarer les **attributs en** `private`,
  - fournir des méthodes **getters/setters en** `public` pour y accéder.

Exemples

Sans Encapsulation

```
class CompteBancaire {
    double solde;

    void deposer(double montant) {
       if(montant > 0)
            solde += montant;
    }
    void retirer(double montant) {
      if (solde > 0 && montant > 0)
             solde -= montant;
    }
}
```

Avec Encapsulation

```
public class CompteBancaire {
    private double solde; // attribut protégé

    public void deposer(double montant) {
        if(montant > 0)
             solde += montant;
    }
    public void retirer(double montant) {
        if(montant < solde)
             solde -= montant;
    }
}
```

Comment faire pour connaitre le solde maintenant qu'il est privé ?

Getters et Setters

- Quand un attribut est `private`, il n'est pas accessible directement depuis l'extérieur.
- On utilise alors :
  - un **getter** : lire la valeur,
  - un **setter** : modifier la valeur.
- Les getters et setters servent à contrôler l'accès (rajouter des validations,...).

Le mot clé `this`

`this` = l'objet courant.
Utile aussi pour différencier l'attribut d'une classe du paramètre d'une méthode.

```
public class Livre {
    private String titre;

    // Getter
    public String getTitre() {
        return this.titre;
    }

    // Setter
    public void setTitre(String titre) {
        this.titre = titre;
    }
}
...
Livre l = new Livre();
l.setTitre("Java Facile");
System.out.println(l.getTitre());
```

Exercice

1. Rendre les attributs `x` et `y` **privés**.
2. Ajouter les méthodes `getX()`, `getY()`.
3. Ajouter les méthodes `setX(int a)`, `setY(int b)`.
4. Dans la classe Main créez un point avec les coordonnées (3,5) puis afficher les coordonnées.

Objectif : appliquer le principe d'encapsulation.

Constructeurs

- Méthode spéciale appelée automatiquement lors de la création d'un objet avec `new`.
- Même nom que la classe, pas de type de retour (même pas `void`).
- Sert à initialiser les attributs de l'objet.

```
public class Livre {
    private String titre;

    // Constructeur
    public Livre(String t) {
        this.titre = t;
    }
}

// Utilisation
Livre l1 = new Livre("Java Facile");
System.out.println(l1.getTitre());
// Affiche : Java Facile
```

Question et remarques

Questions :

- Une classe doit-elle disposer d'un constructeur ?
- Peut on créer un objet d'une classe n'ayant pas de constructeur ?

Question et remarques

Questions :

- Une classe doit-elle disposer d'un constructeur ?
- Peut on créer un objet d'une classe n'ayant pas de constructeur ?

Remarques

- S'il n'y a **aucun** constructeur défini, Java fournit un **constructeur par défaut** sans arguments.
- Dès qu'un constructeur est défini, le constructeur par défaut **disparaît** (à moins de le définir explicitement).
- Une classe peut disposer de plusieurs constructeurs (Qu'est ce cela vous rappelle ?)

Exercice

Modifier la classe `Point` :

1. Ajouter un constructeur `Point(int a, int b)` qui initialise `x` et `y`.
2. Remplacer l'objet crée dans main par `new Point(3,5)` et afficher l'objet.
3. Modifier les coordonnées par les Setters pour (4,2) puis afficher de nouveau.

Objectif : comprendre l'initialisation automatique avec un constructeur.

Bonnes pratiques de conception orientée objet

1. Respecter l'encapsulation

- Déclarer tous les attributs en `private`.
- Fournir des méthodes publiques (`getters` / `setters`) pour y accéder.
- Limiter les accès directs aux données internes.

2.. Séparer interface et implémentation

Élément	**Rôle**	**Exemple**
Interface	Ce que la classe propose	`public void deposer(double montant)`
Implémentation	Comment la classe fonctionne	`solde += montant;`

3. Appliquer le principe du contrat

Une classe définit un **contrat** par son interface et son implémentation reste privée.

Structure d'une classe en Java

```
// Déclaration d'une classe
public class NomDeClasse {

    // 1. Attributs
    private type attribut1;
    private type attribut2;

    // 2. Constructeur
    public NomDeClasse(type a1) {
        this.attribut1 = a1;
    }

    // 3. Getter/Setter
    public type getAttribut1() {
        return attribut1;
    }
    public void setAttribut1(type a1) {
        this.attribut1 = a1;
    }

    // 4. Méthode
    public void afficher() {
        System.out.println(attribut1
            + " - " + attribut2);
    }
}
```

Attributs de classe

Constat

Chaque objet possède ses propres attributs. Mais certaines informations doivent être **partagées par toutes les instances**.

- Exemple : compter le nombre d'objets créés.
- Si on utilise un attribut normal, chaque objet garde son propre compteur.
- On ne peut donc pas connaître le total global.

Attributs de classe

Constat

Chaque objet possède ses propres attributs. Mais certaines informations doivent être **partagées par toutes les instances**.

- Exemple : compter le nombre d'objets créés.
- Si on utilise un attribut normal, chaque objet garde son propre compteur.
- On ne peut donc pas connaître le total global.

Solution

Déclarer un **attribut de classe** (`static`) :

- il appartient à la classe, non aux objets ;
- il est partagé et commun à toutes les instances ;
- accessible sans créer d'objet : `Point.nbPoints()`.

Exemple : compteur d'objets `Point`

Sans `static` (problème)

```
class Point {
  private int compteur = 0;
  public Point() {
    compteur++; // Propre à chaque objet
  }
  public int getCompteur() {
    return compteur;
  }
}

public class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point();
    Point p2 = new Point();
    System.out.println(p2.getCompteur());
    // affiche 1
  }
}
```

Avec `static` (solution)

```
class Point {
  private static int compteur = 0;
  public Point() {
    compteur++; // commun à tous
  }
  public static int getCompteur() {
    return compteur;
  }
}

public class TestPoint {
  public static void main(String[] args) {
    new Point(1, 2);
    new Point(3, 4);
    System.out.println(Point.getCompteur());
    // affiche 2
  }
}
```

Surdéfiniton de méthodes

- On parle de surdéfinition (ou surcharge) lorsqu'un symbole possède plusieurs significations entre lesquelles on choisit en fonction du contexte.
- En java, il est possible de surdéfinir les méthodes d'une classe y compris celles qui sont statiques. Plusieurs méthodes peuvent porter le même nom à condition que le nombre et le type de leurs arguments permettent au compilateur d'effectuer son choix.

```
class Point {
  private int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  // Déplacer selon les deux coordonnées
  public void deplacer(int dx, int dy) {
    x += dx; y += dy;
  }
  // Déplacer uniquement sur l'axe des X
  public void deplacer(int dx) {
    x += dx;
  }
  // Déplacer avec un facteur de déplacement
  public void deplacer(double facteur) {
    x += (int)(dx * facteur);
    y += (int)(dy * facteur);
  }
}
```

Conclusion

Les quatre piliers à retenir

- **Abstraction** : isoler les caractéristiques essentielles d'un objet réel.
- **Encapsulation** : protéger les données en contrôlant leur accès.
- **Héritage** : réutiliser et spécialiser du code existant.
- **Polymorphisme** : traiter des objets différents de manière uniforme.

Ce qu'il faut retenir

- Une **classe** décrit le **comportement et l'état** des objets.
- Un **objet** est une entité autonome : il possède ses données et ses actions.
- Concevoir en POO, c'est **modéliser le monde réel** sous forme d'objets qui coopèrent.

Cela sera développé dans le TP 11