Paradigme de programation : 1/ POO

Programmation Orientée Objet (POO)

La POO repose sur quatre concepts clés : l'**encapsulation**, l'**héritage**, le **polymorphisme** et l'**abstraction**. Elle permet de structurer le code autour d'objets qui combinent état (données) et comportements (méthodes), et favorise la réutilisation et la modularité.

---

1. Encapsulation

L'encapsulation consiste à regrouper des données (attributs) et des méthodes (comportements) au sein d'une classe. Elle vise à protéger ces données en les rendant inaccessibles directement de l'extérieur, sauf via des méthodes spécifiques, comme les **getters** et **setters**. Ces méthodes permettent un accès contrôlé aux attributs privés d'une classe.

Getters et Setters

Les **getters** permettent de lire un attribut privé, tandis que les **setters** permettent de le modifier tout en contrôlant la validité des nouvelles valeurs. Ils favorisent une gestion sécurisée des données.

Exemple en Java :

```
class Person {
    private String name; // Attribut privé, accessible uniquement à l'intérieur de
la classe

    // Getter : Permet de lire l'attribut privé de manière contrôlée
    public String getName() {
        return name;
    }

    // Setter : Permet de modifier l'attribut privé tout en appliquant des règles
(ici vérification si name n'est pas null ou vide)
    public void setName(String name) {
        if (name != null && !name.isEmpty()) {
            this.name = name;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.setName("Alice"); // Utilisation du setter pour définir le nom
        System.out.println(person.getName()); // Utilisation du getter pour obtenir
le nom
    }
```

```
}
```

Concept : Ici, les données (attribut `name`) sont protégées, on ne peut y accéder qu'en passant par les méthodes publiques `getName()` et `setName()`. Cela permet de contrôler l'accès et la modification des données.

Exemple en JavaScript :

```
class Person {
    #name; // Attribut privé, inaccessible directement depuis l'extérieur

    constructor(name) {
        this.#name = name; // Initialisation via le constructeur
    }

    // Getter : Récupère la valeur de l'attribut privé
    get name() {
        return this.#name;
    }

    // Setter : Modifie la valeur de l'attribut privé avec un contrôle (ici,
vérification si le nouveau nom est valide)
    set name(newName) {
        if (newName) {
            this.#name = newName;
        }
    }
}

const person = new Person("Alice");
console.log(person.name); // Utilise le getter pour obtenir le nom
person.name = "Bob"; // Utilise le setter pour modifier le nom
console.log(person.name); // Le getter permet de vérifier la nouvelle valeur
```

Concept : Les getters et setters permettent d'accéder à des données privées en suivant certaines règles, illustrant ainsi le principe d'encapsulation.

---

2. Héritage

L'héritage permet à une classe dérivée de **réutiliser** les attributs et méthodes d'une classe de base. Cela favorise la modularité et la réutilisation du code.

Exemple en Java :

```java
class Animal {
    // Méthode que toutes les sous-classes peuvent utiliser
    public void speak() {
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    // La sous-classe Dog redéfinit (override) la méthode speak
    @Override
    public void speak() {
        System.out.println("Dog barks");
    }
}
```

Concept : La classe `Dog` hérite de la classe `Animal`. Elle peut utiliser la méthode `speak()` d'Animal, mais elle la redéfinit pour son propre comportement (polymorphisme).

Exemple en JavaScript :

```javascript
class Animal {
    // Méthode générique pour toutes les classes héritant d'Animal
    speak() {
        console.log("Animal speaks");
    }
}

class Dog extends Animal {
    // Redéfinition de la méthode speak pour la classe Dog
    speak() {
        console.log("Dog barks");
    }
}

const myDog = new Dog();
myDog.speak(); // Utilisation de la méthode redéfinie dans Dog
```

Concept : Le mécanisme d'héritage permet à `Dog` de réutiliser la méthode `speak` d'`Animal`, tout en la redéfinissant.

---

3. **Polymorphisme**

Le polymorphisme permet d'utiliser une même méthode de façon différente selon l'objet qui l'appelle. Cela rend le code plus flexible.

Exemple en Java :

```java
class Animal {
    // Méthode générique pour parler
    public void speak() {
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    // La classe Dog redéfinit speak pour son propre comportement
    @Override
    public void speak() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Polymorphisme : myDog est de type Animal, mais instance de Dog
        myDog.speak(); // Appelle la méthode speak de Dog grâce au polymorphisme
    }
}
```

Concept : Le polymorphisme permet à `myDog` d'utiliser la méthode `speak` de `Dog`, même si `myDog` est déclaré comme un `Animal`. C'est une manière flexible de travailler avec des objets dérivés.

Exemple en JavaScript :

```javascript
class Animal {
    // Méthode générique pour les animaux
    speak() {
        console.log("Animal speaks");
    }
}

class Dog extends Animal {
    // Redéfinition pour le chien
    speak() {
        console.log("Dog barks");
    }
}

const myDog = new Dog();
myDog.speak(); // Appelle la méthode speak de Dog
```

Concept : Comme en Java, le polymorphisme permet à `myDog`, même s'il hérite d'`Animal`, d'utiliser une version spécialisée de `speak`.

---

4. Abstraction

L'abstraction permet de définir des comportements sans forcément les implémenter tout de suite. Cela permet de cacher les détails d'implémentation et de forcer les sous-classes à fournir leurs propres implémentations.

Exemple en Java :

```java
// Classe abstraite : ne peut pas être instanciée directement
abstract class Animal {
    // Méthode abstraite : aucune implémentation ici
    public abstract void speak();
}

class Dog extends Animal {
    // Implémentation de la méthode abstraite dans la sous-classe
    @Override
    public void speak() {
        System.out.println("Dog barks");
    }
}
```

Concept : La classe `Animal` définit un comportement (`speak()`) que toutes les sous-classes doivent implémenter. Cela permet d'imposer une structure tout en cachant les détails d'implémentation dans la super-classe.

Exemple en JavaScript :

```javascript
class Animal {
    constructor() {
        if (this.constructor === Animal) {
            throw new Error("Cannot instantiate abstract class");
        }
    }

    // Méthode abstraite : doit être implémentée par les sous-classes
    speak() {
        throw new Error("Abstract method must be implemented");
    }
}

class Dog extends Animal {
    // Implémentation concrète de la méthode abstraite
    speak() {
        console.log("Dog barks");
    }
}

const myDog = new Dog();
myDog.speak(); // Appelle la méthode implémentée dans Dog
```

Concept : En JavaScript, on simule l'abstraction en empêchant l'instanciation directe de la classe `Animal`. Cela force les sous-classes à implémenter la méthode `speak()`.

Cette fiche met en lumière les quatre piliers de la Programmation Orientée Objet (POO), avec des exemples en **Java** et **JavaScript** commentés pour illustrer chaque concept.