7. TP 5 : Classes Abstraites et Interfaces

7.1 Introduction

Ce cours a pour objectif de vous faire découvrir les classes abstraites et les interfaces en Java. Ces deux concepts permettent de structurer et d'organiser le code en programmant selon des contrats et des comportements communs. Nous verrons leur utilité et quand les utiliser dans le cadre de la programmation orientée objet.

7.2 Rappel de l'héritage

L'héritage est un des concepts fondamentaux de la programmation orientée objet. Il permet à une classe dérivée (fille) d'hériter des caractéristiques et comportements d'une classe de base (mère).

7.2.1 Exemple : Héritage simple

Voici un exemple simple d'héritage :

```
1  class Animal {
2      int poids;
3      String couleur;
4
5      public void manger() {
6          System.out.println("Je mange.");
7      }
8      public void crier() {
9          System.out.println("Je crie.");
```

```
    }
}

class Chien extends Animal {
    @Override
    public void crier() {
        System.out.println("J'aboie.");
    }
}
```

Dans cet exemple, la classe ‘Chien‘ hérite de la classe ‘Animal‘. Elle utilise les méthodes communes comme ‘manger()‘ et redéfinit la méthode ‘crier()‘.

7.3 Les classes abstraites

Une classe abstraite est une classe qui ne peut pas être instanciée. Elle est souvent utilisée pour définir un modèle générique ou un comportement commun que d'autres classes spécialisées vont implémenter.

7.3.1 Définition et syntaxe

Une classe abstraite est déclarée avec le mot-clé `abstract` :

```
public abstract class Animal {
    abstract void communiquer();
    public void dormir() {
        System.out.println("L'animal dort.");
    }
}
```

Ici, la méthode `communiquer()` est abstraite, c'est-à-dire qu'elle n'a pas d'implémentation. Les sous-classes doivent obligatoirement la redéfinir. Cependant, la méthode `dormir()` a déjà une implémentation concrète.

7.3.2 Exemple complet avec classes abstraites

Voyons maintenant un exemple complet d'utilisation des classes abstraites.

```
abstract class Animal {
    abstract void communiquer();

    public void dormir() {
        System.out.println("L'animal dort.");
    }
}

class Chien extends Animal {
    @Override
    public void communiquer() {
        System.out.println("Le chien aboie.");
```

```
    }
}

class Chat extends Animal {
    @Override
    public void communiquer() {
        System.out.println("Le chat miaule.");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal chien = new Chien();
        chien.communiquer();  // Affiche "Le chien aboie."
        chien.dormir();       // Affiche "L'animal dort."

        Animal chat = new Chat();
        chat.communiquer();   // Affiche "Le chat miaule."
    }
}
```

7.3.3 Quand utiliser une classe abstraite ?

Utilisez une classe abstraite lorsque :

— Vous voulez partager du code commun entre plusieurs classes.

— Vous avez des méthodes que toutes les sous-classes doivent implémenter, mais aussi des méthodes déjà implémentées que les sous-classes peuvent utiliser.

7.4 Les interfaces

Les interfaces sont similaires aux classes abstraites, mais elles ne définissent que des méthodes abstraites. Elles servent à établir un contrat que les classes qui les implémentent doivent respecter.

7.4.1 Définition et syntaxe

Une interface est déclarée avec le mot-clé `interface` :

```
public interface Communicant {
    void communiquer();
}
```

Toutes les méthodes d'une interface sont abstraites par défaut. Une classe qui implémente une interface doit redéfinir toutes les méthodes de cette interface.

7.4.2 Exemple complet avec interface

Voici un exemple d'implémentation d'une interface :

```java
interface Communicant {
    void communiquer();
}

class Chien implements Communicant {
    @Override
    public void communiquer() {
        System.out.println("Le chien aboie.");
    }
}

class Chat implements Communicant {
    @Override
    public void communiquer() {
        System.out.println("Le chat miaule.");
    }
}
```

7.5 Comparaison : Classes abstraites vs Interfaces

7.5.1 Différences fondamentales

Voici quelques différences importantes entre les classes abstraites et les interfaces, accompagnées d'exemples :

— **Méthodes concrètes et attributs d'instance :** Une classe abstraite peut avoir des méthodes concrètes et des attributs d'instance, tandis qu'une interface ne peut définir que des méthodes abstraites et des constantes.

```java
// Classe abstraite avec méthode concrète et attributs d'instance
abstract class Vehicule {
    int vitesse; // Attribut d'instance

    public Vehicule(int vitesse) {
        this.vitesse = vitesse;
    }

    public void demarrer() {  // Méthode concrète
        System.out.println("Le véhicule démarre à une vitesse de " + vitesse + " km/h.");
    }

    abstract void accelerer();  // Méthode abstraite
}

// Interface avec des méthodes abstraites et constantes
interface Volant {
    int ALTITUDE_MAX = 10000; // Constante

```

```
20      void voler();  // Méthode abstraite
21  }
22
```

Dans cet exemple, la classe abstraite `Vehicule` contient une méthode concrète `demarrer()` et un attribut d'instance `vitesse`. En revanche, l'interface `Volant` ne peut définir qu'une constante et des méthodes abstraites sans implémentation.

— **Héritage et implémentation :** Une classe ne peut hériter que d'une seule classe abstraite, mais elle peut implémenter plusieurs interfaces.

```
1   // Classe abstraite
2   abstract class Animal {
3       abstract void communiquer();
4   }
5
6   // Interface
7   interface Nageant {
8       void nager();
9   }
10
11  // Interface
12  interface Volant {
13      void voler();
14  }
15
16  // Classe qui hérite d'une classe abstraite et implémente
        deux interfaces
17  class Canard extends Animal implements Nageant, Volant {
18      @Override
19      public void communiquer() {
20          System.out.println("Le canard cancane.");
21      }
22
23      @Override
24      public void nager() {
25          System.out.println("Le canard nage.");
26      }
27
28      @Override
29      public void voler() {
30          System.out.println("Le canard vole.");
31      }
32  }
33
```

Ici, la classe `Canard` hérite de la classe abstraite `Animal` et implémente deux interfaces, `Nageant` et `Volant`. Cela montre qu'une classe peut implémenter plusieurs interfaces tout en ne pouvant hériter que d'une seule classe (ici abstraite).

— **Constructeur :** Les classes abstraites peuvent avoir un constructeur, contrairement aux interfaces qui ne peuvent pas en définir.

```java
// Classe abstraite avec constructeur
abstract class Animal {
    String nom;

    public Animal(String nom) {
        this.nom = nom;
    }

    public void identite() {
        System.out.println("Je suis un " + nom);
    }

    abstract void communiquer();
}

// Interface ne pouvant pas avoir de constructeur
interface Volant {
    void voler();
}

// Classe qui hérite d'une classe abstraite
class Oiseau extends Animal implements Volant {
    public Oiseau(String nom) {
        super(nom);  // Appel au constructeur de la classe abstraite
    }

    @Override
    public void communiquer() {
        System.out.println("L'oiseau chante.");
    }

    @Override
    public void voler() {
        System.out.println("L'oiseau vole.");
    }
}

```

Ici, la classe abstraite `Animal` a un constructeur qui initialise l'attribut `nom`. La classe `Oiseau` appelle ce constructeur via le mot-clé `super()`. En revanche, une interface comme `Volant` ne peut pas définir de constructeur.

7.5.2 Quand utiliser une classe abstraite au lieu d'une interface ?

Il existe plusieurs situations où l'utilisation d'une classe abstraite est préférable à celle d'une interface. Voici deux cas clés, accompagnés d'exemples.

— **Si la relation sous-classe – super-classe est véritablement une relation de**

type “est une”
Lorsque la relation entre la classe abstraite et ses sous-classes est une relation de type "est une" (a.k.a "is-a" en anglais), une classe abstraite est plus appropriée. Cela signifie que la sous-classe est une version spécialisée de la classe abstraite.

```
// Classe abstraite Animal avec relation "est une"
abstract class Animal {
    String nom;

    public Animal(String nom) {
        this.nom = nom;
    }

    public void dormir() {
        System.out.println(nom + " dort.");
    }

    // Méthode abstraite
    abstract void communiquer();
}

// Classe Chien qui est un Animal
class Chien extends Animal {
    public Chien(String nom) {
        super(nom);
    }

    @Override
    public void communiquer() {
        System.out.println(nom + " aboie.");
    }
}

```

Dans cet exemple, la classe `Chien` est un type spécifique d’`Animal`. Il s’agit d’une relation de type "est une", car un chien **est un** animal. Ici, la classe abstraite `Animal` permet de définir un comportement commun (comme `dormir()`), et la méthode `communiquer()` est redéfinie dans chaque sous-classe.

— **Si la classe abstraite peut fournir une implémentation au niveau approprié d’abstraction**
Une classe abstraite peut fournir une implémentation concrète partielle pour certaines méthodes, ce qui permet de partager du code commun entre les sous-classes. C’est utile lorsque plusieurs sous-classes partagent un comportement similaire, mais ont besoin de spécialiser certaines parties de ce comportement.

```
// Classe abstraite Forme avec méthode concrète et méthode
    abstraite
```

```java
abstract class Forme {
    // Méthode concrète
    public void dessiner() {
        System.out.println("Je dessine une forme.");
    }

    // Méthode abstraite pour calculer l'aire
    abstract double calculerAire();
}

// Sous-classe Rectangle
class Rectangle extends Forme {
    double longueur, largeur;

    public Rectangle(double longueur, double largeur) {
        this.longueur = longueur;
        this.largeur = largeur;
    }

    @Override
    public double calculerAire() {
        return longueur * largeur;
    }
}

// Sous-classe Cercle
class Cercle extends Forme {
    double rayon;

    public Cercle(double rayon) {
        this.rayon = rayon;
    }

    @Override
    public double calculerAire() {
        return Math.PI * rayon * rayon;
    }
}

```

Ici, la classe abstraite `Forme` fournit une méthode concrète `dessiner()` que toutes les sous-classes peuvent utiliser, car le processus de dessin d'une forme est commun. Cependant, chaque sous-classe (comme `Rectangle` et `Cercle`) doit fournir sa propre implémentation de la méthode abstraite `calculerAire()`, car le calcul de l'aire dépend de la forme spécifique.

7.5.3 Quand utiliser une interface au lieu d'une classe abstraite ?

Il existe plusieurs situations où l'utilisation d'une interface est préférable à celle d'une classe abstraite. Voici quelques cas clés, accompagnés d'exemples.

— **Lorsque les méthodes définies représentent une petite partie d'une classe**
Si les méthodes que vous souhaitez imposer ne représentent qu'une petite partie du comportement d'une classe, il est préférable d'utiliser une interface. Cela permet à plusieurs classes différentes d'implémenter l'interface sans hériter d'une hiérarchie complexe.

```
// Interface Volant : représente une petite partie du comportement
interface Volant {
    void voler();
}

// Classe Avion
class Avion implements Volant {
    @Override
    public void voler() {
        System.out.println("L'avion vole dans le ciel.");
    }
}

// Classe Oiseau
class Oiseau implements Volant {
    @Override
    public void voler() {
        System.out.println("L'oiseau vole avec ses ailes.");
    }
}
```

Dans cet exemple, l'interface `Volant` ne représente qu'une petite partie du comportement d'un `Avion` ou d'un `Oiseau`, qui peuvent avoir d'autres comportements indépendants de l'interface.

— **Lorsque la sous-classe doit hériter d'une autre classe**
En Java, une classe ne peut hériter que d'une seule classe, mais elle peut implémenter plusieurs interfaces. Si une classe doit hériter d'une autre classe tout en implémentant des comportements supplémentaires, il est préférable d'utiliser une interface.

```
// Classe abstraite Vehicule
abstract class Vehicule {
    String marque;

    public Vehicule(String marque) {
        this.marque = marque;
    }

    abstract void demarrer();
}
```

```
12 // Interface Electrique
13 interface Electrique {
14     void recharger();
15 }
16
17 // Classe VoitureElectrique hérite de Vehicule et implé
   mente Electrique
18 class VoitureElectrique extends Vehicule implements
   Electrique {
19     public VoitureElectrique(String marque) {
20         super(marque);
21     }
22
23     @Override
24     public void demarrer() {
25         System.out.println("La voiture électrique démarre.
   ");
26     }
27
28     @Override
29     public void recharger() {
30         System.out.println("La voiture est en charge.");
31     }
32 }
33
```

Ici, `VoitureElectrique` hérite de la classe abstraite `Vehicule` et implémente l'interface `Electrique` pour ajouter des fonctionnalités spécifiques à la voiture électrique.

— **Lorsque vous ne pouvez pas raisonnablement implémenter l'une des méthodes**

Si certaines méthodes ne peuvent pas être implémentées de manière raisonnable dans une classe abstraite (parce que leur implémentation dépend fortement de la classe qui les utilise), il est préférable d'utiliser une interface pour laisser chaque classe implémenter ces méthodes à sa manière.

```
1  // Interface Payable : on ne peut pas implémenter une mé
   thode de paiement commune
2  interface Payable {
3      double calculerSalaire();
4  }
5
6  // Classe Employe
7  class Employe implements Payable {
8      private double salaireMensuel;
9
10     public Employe(double salaireMensuel) {
11         this.salaireMensuel = salaireMensuel;
12     }
13
```

```
14      @Override
15      public double calculerSalaire() {
16          return salaireMensuel;
17      }
18  }
19
20  // Classe Freelance
21  class Freelance implements Payable {
22      private double tauxHoraire;
23      private int heuresTravaillees;
24
25      public Freelance(double tauxHoraire, int
        heuresTravaillees) {
26          this.tauxHoraire = tauxHoraire;
27          this.heuresTravaillees = heuresTravaillees;
28      }
29
30      @Override
31      public double calculerSalaire() {
32          return tauxHoraire * heuresTravaillees;
33      }
34  }
35
```

Dans cet exemple, l'interface `Payable` ne peut pas fournir une implémentation commune de `calculerSalaire()`, car chaque classe (ici `Employe` et `Freelance`) a sa propre logique de calcul.

— **Lorsqu'il y a besoin de séparer complètement spécification et implémentation**

Les interfaces sont idéales lorsqu'il est nécessaire de séparer complètement la spécification (le contrat) du comportement concret (l'implémentation). L'interface définit simplement ce que la classe doit faire, sans se soucier de la manière dont elle le fait.

```
1   // Interface Comparable
2   interface Comparable<T> {
3       int compareTo(T autre);
4   }
5
6   // Classe Personne implémentant Comparable pour comparer l
        ' ge
7   class Personne implements Comparable<Personne> {
8       private String nom;
9       private int age;
10
11      public Personne(String nom, int age) {
12          this.nom = nom;
13          this.age = age;
14      }
15
```

```
16      @Override
17      public int compareTo(Personne autre) {
18          return Integer.compare(this.age, autre.age);
19      }
20  }
21
```

Ici, l'interface `Comparable` définit un contrat de comparaison, et chaque classe qui l'implémente doit fournir sa propre logique de comparaison. La séparation entre la spécification (comparaison d'objets) et l'implémentation (comment comparer deux personnes) est totale.

7.6 Exercices pratiques

7.6.1 Exercice 1 : Les formes géométriques

Créez une classe abstraite `Forme` avec des méthodes abstraites pour calculer la surface et le périmètre. Implémentez les classes `Cercle`, `Rectangle`, et `Triangle` qui héritent de `Forme`.

7.6.2 Exercice 2 : Gestion de stock

Créez une interface `Produit` avec des méthodes pour calculer la valeur du stock. Implémentez des classes `ProduitAlimentaire` et `ProduitElectronique` qui implémentent l'interface `Produit`.

Exercice final : Utilisation des classes abstraites et des interfaces

1. Créez une classe abstraite `Vehicule` avec les attributs `nom` (de type `String`) et `typeCarburant` (de type `String`). Ajoutez des méthodes abstraites `demarrer()` et `arreter()`, qui seront implémentées par les classes filles. Ajoutez également une méthode concrète `afficherInfos()` qui affiche le nom du véhicule et son type de carburant.
2. Créez une sous-classe `Voiture` qui hérite de `Vehicule`. Implémentez les méthodes `demarrer()` et `arreter()` pour afficher un message correspondant au démarrage et à l'arrêt d'une voiture. Ajoutez une méthode spécifique `rouler()` qui affiche un message indiquant que la voiture roule.
3. Créez une sous-classe `Bateau` qui hérite de `Vehicule`. Implémentez les méthodes `demarrer()` et `arreter()` pour afficher un message correspondant au démarrage et à l'arrêt d'un bateau. Ajoutez une méthode spécifique `flotter()` qui affiche un message indiquant que le bateau flotte.
4. Créez une interface `Roulant` avec une méthode `rouler()`. Ensuite, implémentez cette interface dans la classe `Voiture`.
5. Créez une interface `Flottant` avec une méthode `flotter()`. Ensuite, implémentez cette interface dans la classe `Bateau`.
6. Créez une nouvelle classe `Hydravion`, qui hérite de `Vehicule` et implémente à la fois les interfaces `Volant` et `Flottant`. Ajoutez une méthode

`voler()` qui affiche un message indiquant que l'hydravion vole et une méthode `flotter()` qui affiche qu'il flotte.

7. Utilisez le **polymorphisme** pour créer un tableau d'objets de type `Vehicule`, contenant des objets de type `Voiture`, `Bateau` et `Hydravion`. Parcourez ce tableau et appelez les méthodes `demarrer()` et `arreter()` sur chaque véhicule.
8. Utilisez le polymorphisme d'interface pour appeler les méthodes spécifiques `rouler()` et `flotter()` sur les objets appropriés. Utilisez le mot-clé `instanceof` pour vérifier si l'objet implémente l'interface `Roulant` ou `Flottant`, puis effectuez un transtypage (`casting`) pour appeler les méthodes spécifiques.