6. TP 4 : Le Polymorphisme

Objectifs

- Comprendre les deux types de polymorphisme en Java.
- Apprendre à utiliser l'opérateur `instanceof`.
- Manipuler le transtypage (upcasting et downcasting).

Partie 1 : Questions guidées avec réponses

6.0.1 Introduction au polymorphisme

Question 1 :

Qu'est-ce que le polymorphisme en Java et quels sont ses types ?

Réponse : Le polymorphisme est un concept de la programmation orientée objet (POO) qui permet à un même nom de méthode d'opérer sur des objets de classes différentes, tant qu'ils sont liés par une super-classe commune. Il existe deux types de polymorphisme :

- Polymorphisme de surcharge (Overloading).
- Polymorphisme d'héritage ou d'inclusion (Overriding).

Question 2 :

Qu'est-ce que le polymorphisme de surcharge ?

Réponse : Le polymorphisme de surcharge permet d'utiliser plusieurs méthodes ayant le même nom mais avec des signatures différentes (nombre ou type de paramètres). Il est décidé lors de la compilation (**early binding**).

```
    public class Calculatrice {

    // Surcharge de la méthode additionner() avec différents types de paramètres
    public int additionner(int a, int b) {
        return a + b;
    }

    public double additionner(double a, double b) {
        return a + b;
    }

    public int additionner(int a, int b, int c) {
        return a + b + c;
    }

    public static void main(String[] args) {
        Calculatrice calc = new Calculatrice();

        // Appel de la méthode additionner avec différents types et nombre de paramètres
        System.out.println("Somme de 2 entiers : " + calc.additionner(5, 10)); // Output: 15
        System.out.println("Somme de 2 doubles : " + calc.additionner(5.5, 10.5)); // Output: 16.0
        System.out.println("Somme de 3 entiers : " + calc.additionner(5, 10, 15)); // Output: 30
    }
}
```

Question 3 :

Qu'est-ce que le polymorphisme d'héritage ?

Réponse : Le polymorphisme d'héritage permet de redéfinir une méthode dans une classe fille tout en ayant la même signature que celle dans la classe mère. Le choix de la méthode à exécuter est fait à l'exécution (**late binding**), selon l'objet réel sur lequel elle est appelée.

```
    // Classe Mère
class Animal {
    // Méthode dans la classe mère
    public void faireDuBruit() {
        System.out.println("L'animal fait du bruit.");
    }
}

// Classe Fille 1
class Chien extends Animal {
    // Redéfinition (override) de la méthode faireDuBruit()
    @Override
```

```
    public void faireDuBruit() {
        System.out.println("Le chien aboie.");
    }
}

// Classe Fille 2
class Chat extends Animal {
    // Redéfinition (override) de la méthode faireDuBruit()
    @Override
    public void faireDuBruit() {
        System.out.println("Le chat miaule.");
    }
}

public class PolymorphismeHeritage {
    public static void main(String[] args) {
        Animal monAnimal; // Déclaration de référence de type
    Animal

        // Référence pointe un objet Chien
        monAnimal = new Chien();
        monAnimal.faireDuBruit(); // Output: Le chien aboie.

        // Référence pointe un objet Chat
        monAnimal = new Chat();
        monAnimal.faireDuBruit(); // Output: Le chat miaule.
    }
}
```

6.1 Utilisation du transtypage (casting)

Question 2 :

Qu'est-ce que le transtypage (casting) et quels sont ses deux types en Java ?

Réponse : Le transtypage permet de convertir un objet d'une classe à une autre. Il existe deux types de casting en Java :

- Upcasting : conversion d'une classe fille vers une classe mère (implicite).
- Downcasting : conversion d'une classe mère vers une classe fille (explicite).

Question 3 :

Montrez un exemple de sur-casting (upcasting) en Java.

Réponse :

```
Fruit f1 = new Pomme(60); // Upcasting implicite
f1.affiche(); // Appelle la méthode affiche() de Pomme.
```

Question 4 :

Comment effectuer un sous-casting (downcasting) en Java ?

Réponse : Le sous-casting doit être effectué explicitement. Par exemple :

```
Fruit f1 = new Pomme(60);
((Pomme)f1).affichePoids(); // Downcasting explicite
```

Question 5 :

Pourquoi utiliser le downcasting ?

Exemple :

```
Fruit f1 = new Pomme(60);
((Pomme)f1).affichePoids(); // Downcasting explicite
```

Explication :

Dans cet exemple, la variable `f1` est de type `Fruit`, mais elle fait référence à un objet de type `Pomme`. La classe `Fruit` ne possède pas la méthode `affichePoids()`, qui est propre à la classe `Pomme`. Si vous essayez d'appeler `f1.affichePoids()`, une erreur de compilation se produira car Java ne sait pas que `f1` est en réalité un objet `Pomme`.

Pour accéder à cette méthode spécifique, il est nécessaire d'effectuer un **downcasting** explicite :

```
((Pomme)f1).affichePoids();
```

Pourquoi le downcasting est-il nécessaire ?

Le downcasting est utilisé pour indiquer explicitement au compilateur que l'objet référencé par `f1`, bien que déclaré comme un `Fruit`, est en réalité une instance de la sous-classe `Pomme`. Cela permet d'accéder aux méthodes spécifiques à `Pomme`, comme `affichePoids()`.

Résumé :

Le **downcasting** est nécessaire pour accéder à des méthodes spécifiques à une sous-classe lorsque l'on manipule un objet via une référence de la classe mère. Cela évite des erreurs de compilation en informant Java du type réel de l'objet.

6.2 Utilisation de `instanceof` et du transtypage (casting)

Question 1 :

Comment l'opérateur `instanceof` est-il utilisé pour vérifier le type d'un objet ?

Réponse : L'opérateur `instanceof` permet de vérifier si un objet est une instance d'une classe donnée. Cela est utile pour éviter des erreurs lors du transtypage (casting). Par exemple :

```
Fruit f = new Pomme(60);
if (f instanceof Pomme) {
```

```
    System.out.println("C'est une pomme.");
}
```

Utilité de l'opérateur instanceof et du downcasting

L'opérateur instanceof est principalement utilisé dans les situations suivantes :

— **Vérification avant le casting** : Il permet de vérifier le type réel de l'objet avant d'effectuer un **downcasting**, afin d'éviter les erreurs telles que ClassCastException.

— **Traitement spécifique en fonction du type d'objet** : Il permet d'exécuter des actions spécifiques à certaines sous-classes, même si l'on manipule une référence de la classe mère.

Voici un exemple illustrant l'utilisation de instanceof pour vérifier le type d'un objet avant de faire un downcasting :

```
class Animal {
    public void faireDuBruit() {
        System.out.println("L'animal fait du bruit.");
    }
}

class Chien extends Animal {
    public void aboyer() {
        System.out.println("Le chien aboie.");
    }
}

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

public class InstanceofDemo {
    public static void main(String[] args) {
        Animal monAnimal1 = new Chien(); // Upcasting
    implicite
        Animal monAnimal2 = new Chat();  // Upcasting
    implicite

        // Utilisation de instanceof pour vérifier si c'est un
     Chien
        if (monAnimal1 instanceof Chien) {
            // Downcasting explicite pour accéder à la méthode
     aboyer()
            ((Chien) monAnimal1).aboyer();
        }

        // Utilisation de instanceof pour vérifier si c'est un
```

```
    Chat
31          if (monAnimal2 instanceof Chat) {
32              // Downcasting explicite pour accéder à la méthode
    miauler()
33              ((Chat) monAnimal2).miauler();
34          }
35      }
36  }
```

Explication du code :

- **Classe mère :** `Animal` définit une méthode générale `faireDuBruit()`.
- **Sous-classes :** `Chien` et `Chat` ajoutent des méthodes spécifiques (`aboyer()` pour `Chien`, `miauler()` pour `Chat`).
- **Upcasting implicite :** Les objets `Chien` et `Chat` sont assignés à une référence de type `Animal`, ce qui est un upcasting. Cela se fait automatiquement par Java.
- **Utilisation de `instanceof` :** Avant d'accéder aux méthodes spécifiques à `Chien` ou `Chat`, nous utilisons `instanceof` pour vérifier le type réel de l'objet.
- **Downcasting explicite :** Après vérification avec `instanceof`, nous effectuons un `downcasting` pour accéder aux méthodes spécifiques des sous-classes (`aboyer()` et `miauler()`).

Résultat :

```
// Le chien aboie.
// Le chat miaule.
```

6.2.1 Exemple pratique : Transtypage et polymorphisme

Question 1 :

Écrivez un programme qui crée un tableau de fruits (pommes et oranges), affiche leurs informations et utilise le casting pour accéder à des méthodes spécifiques.

Réponse :

```
1   public class Fruit {
2       public void affiche() {
3           System.out.println("C'est un fruit.");
4       }
5   }
6
7   public class Pomme extends Fruit {
8       int poids;
9       public Pomme(int poids) {
10          this.poids = poids;
11      }
12      public void affiche() {
```

```
        System.out.println("C'est une pomme.");
    }
    public void affichePoids() {
        System.out.println("Le poids de la pomme est : " +
   poids + " grammes.");
    }
}

public class Orange extends Fruit {
    int poids;
    public Orange(int poids) {
        this.poids = poids;
    }
    public void affiche() {
        System.out.println("C'est une orange.");
    }
    public void affichePoids() {
        System.out.println("Le poids de l'orange est : " +
   poids + " grammes.");
    }
}

public class PolymorphismeDemo {
    public static void main(String[] args) {
        Fruit[] fruits = {new Pomme(60), new Orange(100), new
   Pomme(55)};

        for (Fruit fruit : fruits) {
            fruit.affiche();
            if (fruit instanceof Pomme) {
                ((Pomme)fruit).affichePoids();
            } else if (fruit instanceof Orange) {
                ((Orange)fruit).affichePoids();
            }
        }
    }
}
```

Partie 2 : Exercice basique sans réponses

Exercice :

Créez une classe `Vehicule` avec une méthode `seDeplacer()`, puis créez deux sous-classes `Voiture` et `Velo` qui redéfinissent cette méthode. Testez le polymorphisme en appelant `seDeplacer()` sur des objets de type `Vehicule`.

Exercice final : Polymorphisme avancé et transtypage

1. Créez une classe `Employe` avec les attributs `nom` (de type `String`) et `salaire` (de type `int`). Ajoutez une méthode `afficherInfos()` qui affiche les infor-

mations de base de l'employé.

2. Créez une sous-classe `Developpeur` qui hérite de `Employe`. Ajoutez un attribut `langage` (de type `String`) pour le langage de programmation et redéfinissez la méthode `afficherInfos()` pour afficher le nom, le salaire, et le langage de programmation du développeur. Ajoutez une méthode spécifique `afficherLangage()` qui affiche le langage utilisé par le développeur.
3. Créez une sous-classe `Manager` qui hérite de `Employe`. Ajoutez un attribut `nombreDeSubordonnes` (de type `int`) pour le nombre de subordonnés et redéfinissez la méthode `afficherInfos()` pour afficher le nom, le salaire, et le nombre de subordonnés. Ajoutez une méthode spécifique `gererEquipe()` qui simule la gestion de l'équipe.
4. Utilisez le **polymorphisme** pour créer un tableau d'objets de type `Employe`, contenant des objets `Developpeur` et `Manager`. Parcourez ce tableau et appelez la méthode `afficherInfos()` sur chaque élément pour afficher les informations spécifiques à chaque type d'employé.
5. Expliquez et utilisez le **sur-casting (upcasting)** en créant un objet de type `Developpeur`, mais en l'assignant à une référence de type `Employe`. Affichez les informations de l'employé en appelant `afficherInfos()`.
6. Utilisez le **sous-casting (downcasting)** pour accéder à des méthodes spécifiques aux sous-classes. Créez un objet de type `Developpeur` ou `Manager`, assigné à une référence de type `Employe`. Puis, après avoir vérifié le type réel de l'objet avec `instanceof`, effectuez un sous-casting pour appeler une méthode spécifique (`afficherLangage()` pour `Developpeur` ou `gererEquipe()` pour `Manager`).