TP2 : Héritage, Polymorphisme et Encapsulation en Java

Objectifs :

1. Comprendre et mettre en pratique le concept d'héritage en Java.
2. Appliquer le polymorphisme pour créer des programmes plus flexibles et réutilisables.
3. Approfondir la manipulation des objets à travers des classes dérivées.

Prérequis :

- Avoir complété le TP1 et maîtriser la création de classes et d'objets en Java.
- Notions de base en programmation orientée objet.

Partie 1 : Héritage en Java

Création d'une classe de base

1. **Création d'une classe `Animal` :**
   Créer un fichier nommé `Animal.java`.
   Implémentez une classe `Animal` qui représente un animal générique avec les attributs `name` et `age`, et une méthode `makeSound()`.

```java
public class Animal {
    String name;
    int age;

    // Constructeur
    public Animal(String name, int age) {
        this.name = name;
```

```
        this.age = age;
    }

    // Méthode pour émettre un son (à surcharger)
    public void makeSound() {
        System.out.println("L'animal fait un bruit.");
    }

    // Méthode pour afficher les informations de l'animal
    public void displayInfo() {
        System.out.println(name + " est un animal de " + age + " ans.");
    }
}
```

2. **Création d'une classe `Dog` (Héritage) :**
   Créer un fichier nommé `Dog.java` qui contient une classe `Dog` héritant de la classe `Animal`.
   Cette classe doit redéfinir la méthode `makeSound()` pour afficher un son spécifique au chien.

```java
public class Dog extends Animal {

    // Constructeur
    public Dog(String name, int age) {
        super(name, age); // Appel du constructeur de la classe mère
    }

    // Redéfinition de la méthode makeSound
    @Override
    public void makeSound() {
        System.out.println(name + " aboie: Woof! Woof!");
    }
}
```

3. **Création d'une classe `Cat` (Héritage) :**
   Créer un fichier nommé `Cat.java` avec une classe `Cat` qui hérite de la classe `Animal`.
   Cette classe doit également redéfinir la méthode `makeSound()` pour afficher un son spécifique au chat.

```java
public class Cat extends Animal {

    // Constructeur
    public Cat(String name, int age) {
        super(name, age);
    }

    // Redéfinition de la méthode makeSound
    @Override
    public void makeSound() {
        System.out.println(name + " miaule: Meow! Meow!");
    }
```

```
}
```

Tester l'héritage

1. **Création du fichier `Main.java` :**
   Créer un fichier `Main.java` où vous allez instancier des objets `Dog` et `Cat`, puis tester les méthodes héritées et redéfinies.

```
public class Main {
    public static void main(String[] args) {
        // Création d'objets Dog et Cat
        Dog dog = new Dog("Rex", 5);
        Cat cat = new Cat("Mia", 3);

        // Appel des méthodes
        dog.displayInfo();
        dog.makeSound();

        cat.displayInfo();
        cat.makeSound();
    }
}
```

2. **Compilation et exécution :**
   Compilez tous les fichiers et exécutez le programme principal.

   Vous devriez voir les informations et les sons spécifiques aux animaux créés.

Partie 2 : Polymorphisme en Java

Utilisation du polymorphisme

1. **Déclaration de références de type `Animal` :**
   Modifiez le fichier `Main.java` pour utiliser le polymorphisme. Déclarez des références de type `Animal` et affectez-leur des objets `Dog` et `Cat`.

```
public class Main {
    public static void main(String[] args) {
        // Utilisation de polymorphisme
        Animal myDog = new Dog("Rex", 5);
        Animal myCat = new Cat("Mia", 3);

        // Appel des méthodes polymorphiques
        myDog.displayInfo();
        myDog.makeSound();

        myCat.displayInfo();
        myCat.makeSound();
    }
}
```

2. **Explication :**
   Le polymorphisme permet d'utiliser une même référence de type `Animal` pour appeler des méthodes spécifiques aux objets `Dog` et `Cat`. La méthode appropriée est automatiquement appelée en fonction du type réel de l'objet.

Array d'objets polymorphiques

1. **Création d'un tableau d'animaux :**
   Modifiez `Main.java` pour créer un tableau d'objets `Animal`, contenant à la fois des `Dog` et des `Cat`. Parcourir ce tableau et appelez les méthodes `displayInfo()` et `makeSound()`.

```
public class Main {
    public static void main(String[] args) {
```

```
        Animal[] animals = new Animal[2];
        animals[0] = new Dog("Rex", 5);
        animals[1] = new Cat("Mia", 3);

        // Parcourir le tableau et appeler les méthodes
        for (Animal animal : animals) {
            animal.displayInfo();
            animal.makeSound();
        }
    }
}
```

2. **Compilation et exécution :**
   Compilez et exécutez à nouveau le programme. Observez comment les méthodes sont appelées de manière polymorphique sur les objets dans le tableau.

Partie 3 : Encapsulation

Dans ce TP, vous allez approfondir votre compréhension des concepts d'héritage et de polymorphisme en Java, tout en découvrant l'encapsulation, un principe fondamental de la programmation orientée objet (POO). Vous manipulerez les modificateurs d'accès `private`, `protected`, et `public`, ainsi que les mots-clés `static` et `final`.

Encapsulation en Java

L'encapsulation est un principe de la POO qui consiste à restreindre l'accès direct aux attributs d'une classe et à fournir des méthodes pour les manipuler. Ce principe permet de protéger l'intégrité des données en empêchant leur modification non contrôlée.

Modificateurs d'Accès

- **`private`** : Limite l'accès aux membres (attributs ou méthodes) d'une classe à l'intérieur de cette classe uniquement.
- **`protected`** : Permet l'accès aux membres d'une classe à l'intérieur du même package ou par des classes héritées.

- **`public`** : Permet l'accès aux membres d'une classe depuis n'importe quelle autre classe.

Mots-clés `static` et `final`

- **`static`** : Indique qu'un membre (attribut ou méthode) appartient à la classe plutôt qu'aux instances de la classe. Un membre `static` est partagé par toutes les instances de la classe.
- **`final`** : Indique qu'un attribut ne peut être modifié après son initialisation, qu'une méthode ne peut être redéfinie, ou qu'une classe ne peut être héritée.

Exercice 1 : Encapsulation avec des Modificateurs d'Accès

1. Créer une classe `BankAccount` avec les attributs suivants :
    - `accountNumber` (numéro de compte) : `private`
    - `balance` (solde) : `private`
    - `ownerName` (nom du propriétaire) : `protected`
    - Une méthode publique `deposit(double amount)` pour déposer de l'argent.
    - Une méthode publique `withdraw(double amount)` pour retirer de l'argent.
    - Une méthode publique `getBalance()` pour obtenir le solde actuel.
2. Implémentez les méthodes de la classe en respectant les règles de l'encapsulation.
3. Créer une classe `SavingsAccount` qui hérite de `BankAccount`. Ajouter un attribut `interestRate` (taux d'intérêt) `private` et une méthode publique `applyInterest()` qui applique les intérêts au solde.
4. Testez votre code en créant des instances de `BankAccount` et de `SavingsAccount` et en manipulant les comptes via les méthodes publiques.

Exercice 2 : Utilisation des Mots-clés `static` et `final`

1. **Membres `static`** :

- Ajouter un attribut `static` à la classe `BankAccount` pour suivre le nombre total de comptes bancaires créés.
- Ajouter une méthode `static` publique `getTotalAccounts()` pour obtenir le nombre total de comptes.

2. **Membres `final`** :
   - Déclarez l'attribut `accountNumber` de la classe `BankAccount` comme `final` pour qu'il ne puisse pas être modifié après l'initialisation.
   - Créer une méthode `final` dans la classe `BankAccount` appelée `displayAccountInfo()` qui affiche les informations du compte. Essayez de redéfinir cette méthode dans la classe `SavingsAccount` pour observer ce qu'il se passe.
3. **Testez votre code** en instanciant plusieurs objets `BankAccount` et `SavingsAccount`, en vérifiant le nombre total de comptes créés et en essayant de redéfinir la méthode `displayAccountInfo()`.

Partie 4 : Les types génériques en java

Les types génériques en Java sont un mécanisme qui permet de créer des classes, des interfaces et des méthodes qui peuvent fonctionner avec n'importe quel type d'objet, tout en garantissant la sécurité des types au moment de la compilation.

Exemple simple

Imaginons que vous vouliez créer une classe `Boîte` pour stocker un objet. Sans les types génériques, vous pourriez écrire la classe comme ceci :

```
public class Boîte {
    private Object contenu;

    public Boîte(Object contenu) {
        this.contenu = contenu;
    }

    public Object getContenu() {
```

```
        return contenu;
    }

    public void setContenu(Object contenu) {
        this.contenu = contenu;
    }
}
```

Le problème ici est que la méthode `getContenu()` retourne un `Object`, et vous devez faire un "cast" pour convertir cet objet en son type d'origine lorsque vous l'utilisez, ce qui peut conduire à des erreurs.

Avec les types génériques

Avec les types génériques, vous pouvez définir la classe `Boîte` de manière à ce qu'elle puisse fonctionner avec n'importe quel type d'objet tout en conservant la sécurité des types :

```java
public class MaClasseGenerique<T> {
    private T contenu;

    public MaClasseGenerique(T contenu) {
        this.contenu = contenu;
    }

    public T getContenu() {
        return contenu;
    }

    public void setContenu(T contenu) {
        this.contenu = contenu;
    }
}
```

Ici, `T` est un type générique. Lorsque vous créez une `Boîte`, vous spécifiez le type d'objet qu'elle doit contenir :

```java
MaClasseGenerique<String> boîteDeChocolats = new MaClasseGenerique<>("Chocolats");
MaClasseGenerique<Integer> boîteDeNombres = new MaClasseGenerique<>(123);
```

- `Boîte<String>` signifie que cette boîte ne peut contenir que des objets de type `String`.
- `Boîte<Integer>` signifie que cette boîte ne peut contenir que des objets de type `Integer`.

Avantages des types génériques

1. **Sécurité des types :** Les erreurs de type sont détectées au moment de la compilation, et non au moment de l'exécution.
2. **Réutilisabilité :** Vous pouvez écrire du code qui fonctionne avec n'importe quel type, sans avoir à dupliquer le code pour chaque type.

3. **Clarté :** Le code est plus clair et plus facile à comprendre, car le type des objets est explicite.

En résumé, les types génériques rendent votre code Java plus flexible, sûr et réutilisable.

Exercice 1 : Introduction aux Classes Génériques

Objectif : Créer une classe générique simple pour comprendre les bases de la généricité.

1. **Création d'une classe générique :**
    - Créer une classe générique `Boîte<T>` qui contient une liste `List` d'éléments de type `T`.
    - La classe `Boîte` doit avoir un attribut privé `contenu` de type `List<T>`.
    - Ajouter un constructeur permettant d'initialiser le contenu de la boîte.
    - Ajouter des méthodes `getContenu()` et `setContenu(List<T> contenu)` pour accéder et modifier le contenu de la boîte.
    - Ajouter de nouvelles méthodes pour manipuler ses attributs comme une méthode `add` pour ajouter, `remove` pour retirer, etc. Libre à vous de décider des méthodes pertinentes à ajouter !
2. **Test de la classe générique :**
    - Dans la méthode `main`, créez une instance de `Boîte` pour stocker un `String`.
    - Créer une autre instance de `Boîte` pour stocker un `Integer`.
    - Affichez les contenus des deux boîtes.

Exercice 2 : Méthodes Génériques

Objectif : Comprendre comment définir et utiliser des méthodes génériques.

1. **Création d'une méthode générique :**
    - Ajouter une méthode générique `merge` dans la classe `Boite` qui prend deux listes d'éléments de type `T` et retourne une liste contenant les éléments fusionnés.

- La méthode `merge` doit être statique et accepter deux paramètres du même type générique `T`.

Exemple de code à adapter :

```
public static <T> T[] swap(T premier, T second) {
    T[] tableau = (T[]) new Object[2];
    tableau[0] = second;
    tableau[1] = premier;
    return tableau;
}
```

2. **Test de la méthode générique :**

   - Appeler la méthode `merge` avec deux listes de `String` et afficher les résultats.

Exemple de code qui appelerait la méthode `swap` précédente :

```
public static void main(String[] args) {
    String[] resultString = swap("Premier", "Second");
    System.out.println("Résultat du swap : " + resultString[0] + ", " + resultString[1]);

    Integer[] resultInt = swap(1, 2);
    System.out.println("Résultat du swap : " + resultInt[0] + ", " + resultInt[1]);
}
```

Exercice 3 : Collections Génériques

Objectif : Apprendre à utiliser des collections génériques avec des classes génériques.

1. **Création d'une collection générique :**

- Créer une liste `List<Boîte<String>>` pour stocker plusieurs instances de `Boîte<String>`.
- Ajouter quelques boîtes à la liste avec des contenus différents.

2. **Manipulation de la collection :**
   - Parcourir la liste et affichez le contenu de chaque boîte.
   - Ajouter une méthode `afficheBoîtes` dans la classe `Main` qui accepte une collection de `Boîte<T>` et affiche le contenu de chaque boîte.

3. **Test de la méthode avec différentes collections :**
   - Créer une liste de `Boîte<Integer>` et appelez `afficheBoîtes`.
   - Créer une liste de `Boîte<String>` et appelez `afficheBoîtes`.

Exercice 4 : Gestion d'une Bibliothèque

Objectif : Implémenter un système de gestion d'une bibliothèque en utilisant l'héritage, le polymorphisme et les types génériques.

1. **Création des Classes de Base :**
   - Créer une classe `Document` avec les attributs suivants : `titre`, `auteur`, `annéeDePublication`.
   - Ajouter une méthode `afficherInfos()` dans la classe `Document`, qui sera implémentée dans les sous-classes pour afficher les informations du document.

2. **Sous-classes :**
   - Créer trois sous-classes : `Livre`, `Magazine`, et `DVD`.
   - Chaque sous-classe doit ajouter des attributs spécifiques (par exemple, `nombreDePages` pour `Livre`, `numéroÉdition` pour `Magazine`, et `durée` pour `DVD`).
   - Implémentez la méthode `afficherInfos()` dans chaque sous-classe pour afficher les informations spécifiques à chaque type de document.

3. **Utilisation du Polymorphisme :**

    - Créer une classe `Bibliothèque` qui contient une liste de `Boite<Document<` (oui, c'est en quelque sorte, une liste de liste!). Chaque boîte représente une armoire dans lequelles différents documents séraient rangés, des livres avec les livres, magazines avec des magazines, etc.
    - Implémentez une méthode `afficherTousLesDocuments()` qui parcourt la liste de Boîtes et appelle la méthode `afficherInfos()` sur chaque élément, en utilisant le polymorphisme.

4. **Test :**

    - Dans une classe principale, Créer des instances de `Livre`, `Magazine`, et `DVD`, Ajoutez-les à la bibliothèque, et testez l'affichage des informations.

Conclusion :

À la fin de ce TP, vous devriez comprendre comment utiliser l'héritage pour créer des classes dérivées en Java, ainsi que le polymorphisme pour écrire du code flexible et extensible. Vous avez également pratiqué l'implémentation et l'utilisation de méthodes redéfinies dans des classes dérivées. Dans les prochaines séances, nous approfondirons d'autres concepts avancés de la POO, tels que les interfaces et les classes abstraites.