TD3

Membres statiques

1. Expliquez la différence entre un attribut statique et un attribut d'instance en Java.

   Un attribut statique appartient à la classe elle-même et est partagé par toutes les instances de la classe. Un attribut d'instance appartient à une instance spécifique de la classe et chaque instance a sa propre copie de cet attribut.

2. Donnez un exemple de méthode statique et expliquez dans quel cas il est approprié d'utiliser une méthode statique.

```java
public class MathUtils {
    public static int additionner(int a, int b) {
        return a + b;
    }
}
```

   Il est approprié d'utiliser une méthode statique lorsque la méthode ne dépend pas des attributs d'instance et peut être appelée sans créer une instance de la classe.

3. Pourquoi les méthodes statiques ne peuvent-elles pas accéder aux membres d'instance directement ?

   Les méthodes statiques ne peuvent pas accéder aux membres d'instance directement car elles n'ont pas de référence à une instance spécifique de la classe. Elles appartiennent à la classe elle-même et non à une instance.

Polymorphisme

4. Expliquez le concept de polymorphisme en POO et donnez un exemple (en Java, sous forme de diagramme ou sous forme de texte qui explique bien)

   Le polymorphisme permet à des objets de classes différentes d'être traités comme des objets du même type. Cela permet d'utiliser une interface commune pour manipuler des objets de classes différentes.

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

class Chien extends Animal {
```

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

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

5. Quelle est la différence entre le polymorphisme à la compilation et le polymorphisme à l'exécution ? Donnez un exemple pour chacun.

   Le polymorphisme à la compilation (surcharge de méthodes) est déterminé au moment de la compilation, tandis que le polymorphisme à l'exécution (redéfinition de méthodes) est déterminé au moment de l'exécution.

```
// Polymorphisme à la compilation
class MathUtils {
    int additionner(int a, int b) {
        return a + b;
    }

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

// Polymorphisme à l'exécution
class Animal {
    void faireDuBruit() {
        System.out.println("L'animal fait du bruit");
    }
}

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

6. Comment le polymorphisme améliore-t-il la flexibilité et la maintenabilité du code ?

   Le polymorphisme améliore la flexibilité et la maintenabilité du code en permettant de traiter des objets de différentes classes de manière uniforme. Cela permet de créer des systèmes plus extensibles et de réduire le couplage entre les classes.

Expressions Lambda

7. Qu'est-ce qu'une expression lambda en Java et pourquoi est-elle utile ?

Une expression lambda est une fonction anonyme qui permet de simplifier le code en évitant de créer des classes ou des méthodes supplémentaires. Elle est utile pour écrire du code plus concis et lisible, notamment pour les interfaces fonctionnelles.

8. Expliquez le concept d'interface fonctionnelle et donnez un exemple en Java.

   Une interface fonctionnelle est une interface qui ne contient qu'une seule méthode abstraite. Elle peut être utilisée avec des expressions lambda pour fournir une implémentation de cette méthode.

```java
@FunctionalInterface
interface Operation {
    int calculer(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        Operation addition = (a, b) -> a + b;
        System.out.println(addition.calculer(5, 3)); // Affiche 8
    }
}
```

Généricité

9. Qu'est-ce que la généricité en Java et quels sont ses avantages ?

   La généricité permet de créer des classes, des interfaces et des méthodes avec des types paramétrés. Elle permet de réutiliser du code avec différents types sans le dupliquer, d'améliorer la sécurité du type et de réduire les erreurs de type à l'exécution.

10. Expliquez comment les contraintes de type peuvent être appliquées aux types génériques. Donnez un exemple.

    Les contraintes de type permettent de restreindre les types qui peuvent être utilisés avec des classes ou des méthodes génériques en utilisant le mot-clé `extends`.

```java
class Cage<T extends Animal> {
    private T contenu;

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

    public T getContenu() {
        return contenu;
    }
}

class Animal {}
class Chien extends Animal {}

public class Main {
```

```
    public static void main(String[] args) {
        Cage<Chien> cageDeChien = new Cage<>();
        cageDeChien.setContenu(new Chien());
        System.out.println(cageDeChien.getContenu());
    }
}
```

Collections

11. Quelles sont les principales interfaces de collections en Java et quelles sont leurs différences ?

    Les principales interfaces de collections en Java sont `List`, `Set`, et `Queue`. `List` est une collection ordonnée qui permet les éléments en double. `Set` est une collection qui ne permet pas les éléments en double. `Queue` est une collection qui suit le principe FIFO (First In, First Out).

12. Expliquez la différence entre une `HashMap` et une `TreeMap`. Donnez un exemple d'utilisation pour chacune.

    `HashMap` est une implémentation de la carte qui ne garantit pas l'ordre des éléments. `TreeMap` est une implémentation de la carte qui maintient les éléments dans un ordre trié.

```
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> hashMap = new HashMap<>();
        hashMap.put("Pierre", 30);
        hashMap.put("Paul", 25);
        hashMap.put("Jacques", 35);
        System.out.println(hashMap);

        Map<String, Integer> treeMap = new TreeMap<>();
        treeMap.put("Pierre", 30);
        treeMap.put("Paul", 25);
        treeMap.put("Jacques", 35);
        System.out.println(treeMap);
    }
}
```

13. Comment la bibliothèque `Stream` de Java peut-elle être utilisée pour manipuler des collections ? Donnez un exemple.

    La bibliothèque `Stream` permet de manipuler des collections de manière fonctionnelle en utilisant des opérations telles que `filter`, `map`, `reduce`, etc.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
```

```java
public class Main {
    public static void main(String[] args) {
        List<Integer> nombres = Arrays.asList(1, 2, 3, 4, 5, 6);
        List<Integer> nombresPairs = nombres.stream()
                                            .filter(n -> n % 2 == 0)
                                            .collect(Collectors.toList());
        System.out.println(nombresPairs); // Affiche [2, 4, 6]
    }
}
```