Paradigmes de Programmation : 2/ Programmation fonctionnelle

Programmation Fonctionnelle

---

1. Fonctions Pures

Une fonction pure est une fonction dont le résultat est uniquement déterminé par ses entrées, sans effets secondaires. Cela signifie qu'elle ne modifie aucune variable externe et ne dépend pas de l'état du programme.

Exemple en Java :

```java
// Fonction pure : le résultat dépend uniquement des paramètres et aucune variable externe n'est modifiée.
public class MathUtils {
    public static int add(int a, int b) {
        return a + b; // Retourne toujours la même somme pour les mêmes entrées
    }

    public static void main(String[] args) {
        System.out.println(add(3, 5)); // Résultat = 8
    }
}
```

Concept : `add` est une fonction pure car elle retourne toujours la même somme pour les mêmes paramètres et n'a aucun effet secondaire.

Exemple en JavaScript :

```javascript
// Fonction pure : dépend uniquement des arguments et pas d'état externe
function add(a, b) {
    return a + b; // Retourne toujours le même résultat pour les mêmes arguments
}

console.log(add(3, 5)); // Résultat = 8
```

Concept : Comme en Java, la fonction `add` en JavaScript est pure, car elle ne modifie aucune variable globale ou externe.

---

2. Immuabilité

L'immutabilité consiste à ne jamais modifier directement un objet ou une variable après sa création. En programmation fonctionnelle, on préfère créer des **nouveaux objets** au lieu de modifier ceux existants.

Exemple en Java :

```java
import java.util.Collections;
import java.util.List;

public class ListUtils {
    // Cette méthode retourne une nouvelle liste avec un élément ajouté, sans
modifier la liste d'origine
    public static List<String> addElement(List<String> list, String element) {
        List<String> newList = new java.util.ArrayList<>(list); // Crée une nouvelle
liste basée sur l'ancienne
        newList.add(element); // Ajoute l'élément à la nouvelle liste
        return Collections.unmodifiableList(newList); // Retourne une version
immuable de la liste
    }

    public static void main(String[] args) {
        List<String> originalList = List.of("Apple", "Banana");
        List<String> newList = addElement(originalList, "Orange");

        System.out.println(originalList); // Original reste inchangé : [Apple,
Banana]
        System.out.println(newList); // Nouvelle liste : [Apple, Banana, Orange]
    }
}
```

Concept : En créant une nouvelle liste au lieu de modifier l'ancienne, on respecte le principe d'immutabilité.

Exemple en JavaScript :

```javascript
const originalArray = ["Apple", "Banana"];

// Fonction qui retourne un nouveau tableau sans modifier l'original
function addElement(arr, element) {
    return [...arr, element]; // Crée un nouveau tableau en utilisant la
décomposition (spread operator)
}

const newArray = addElement(originalArray, "Orange");

console.log(originalArray); // Original reste inchangé : ["Apple", "Banana"]
console.log(newArray); // Nouveau tableau : ["Apple", "Banana", "Orange"]
```

Concept : Le tableau `originalArray` n'est pas modifié, on retourne une nouvelle version avec l'élément ajouté, ce qui respecte l'immutabilité.

---

3. **Récursivité**

En programmation fonctionnelle, la **récursivité** est souvent utilisée pour remplacer les boucles. Une fonction récursive s'appelle elle-même jusqu'à ce qu'une condition de sortie soit atteinte.

Exemple en Java :

```java
public class Factorial {
    // Fonction récursive pour calculer la factorielle d'un nombre
    public static int factorial(int n) {
        if (n == 0) {
            return 1; // Cas de base
        } else {
            return n * factorial(n - 1); // Appel récursif
        }
    }

    public static void main(String[] args) {
        System.out.println(factorial(5)); // Résultat = 120
    }
}
```

Concept : La fonction `factorial` s'appelle elle-même jusqu'à ce que `n` atteigne 0 (le cas de base), illustrant le principe de récursivité.

Exemple en JavaScript :

```javascript
// Fonction récursive pour calculer la factorielle
function factorial(n) {
    if (n === 0) {
        return 1; // Cas de base
    } else {
        return n * factorial(n - 1); // Appel récursif
    }
}

console.log(factorial(5)); // Résultat = 120
```

Concept : Comme en Java, la fonction `factorial` en JavaScript s'appelle récursivement pour calculer le résultat, en se basant sur une condition de sortie (`n === 0`).

---

4. Fonctions d'ordre supérieur

Les fonctions d'ordre supérieur sont des fonctions qui prennent d'autres **fonctions en argument** ou qui **retournent des fonctions**. Cela permet de créer des comportements plus flexibles et réutilisables.

Exemple en Java :

```java
import java.util.function.Function;

public class HigherOrderFunctionExample {
    // Fonction d'ordre supérieur : prend une fonction en argument
    public static int applyFunction(int x, Function<Integer, Integer> func) {
        return func.apply(x); // Applique la fonction passée en argument
    }

    public static void main(String[] args) {
        // Passe une fonction lambda comme argument (double l'entrée)
        int result = applyFunction(5, y -> y * 2);
        System.out.println(result); // Résultat = 10
    }
}
```

Concept : `applyFunction` est une fonction d'ordre supérieur car elle prend une autre fonction (lambda) en paramètre et l'applique à son argument.

Exemple en JavaScript :

```javascript
// Fonction d'ordre supérieur : prend une fonction en argument
function applyFunction(x, func) {
    return func(x); // Applique la fonction passée en paramètre
}

// Utilisation avec une fonction fléchée (double l'entrée)
const result = applyFunction(5, y => y * 2);
console.log(result); // Résultat = 10
```

Concept : En JavaScript aussi, `applyFunction` prend une fonction fléchée en argument et l'applique à une valeur, illustrant le concept des fonctions d'ordre supérieur.

Cette fiche couvre les principaux concepts de la **programmation fonctionnelle** (fonctions pures, immuabilité, récursivité, et fonctions d'ordre supérieur) avec des exemples commentés en **Java** et **JavaScript**.