Introduction à la programmation déclarative

En quoi la programmation déclarative facilite la compréhension et la testabilité du code ?

Définition :

La **programmation déclarative** est un paradigme de programmation qui consiste à créer des applications sur la base de composants logiciels indépendants du contexte et ne comportant aucun état interne.

Paradigmes de la Programmation Déclarative

Programmation Fonctionnelle

La programmation fonctionnelle repose sur les fonctions pures, l'immuabilité, et l'absence d'effets de bord, mettant l'accent sur le calcul des valeurs.

- **Exemples** : Haskell, Lisp, et certains aspects de JavaScript (fonctions `map`, `filter`, `reduce`)
- **Caractéristiques** : Favorise la lisibilité et la réutilisabilité. Utilisée dans les calculs mathématiques et les transformations de données.
- **Avantages** : Fiabilité accrue, testabilité, et facilité de parallélisme.

Depuis quelques années, la programmation fonctionnelle a gagné en popularité, notamment grâce à des langages hybrides comme F# et Scala.

De plus, les langages et les frameworks modernes intègrent de plus en plus de concepts fonctionnels.

Programmation Logique

La programmation logique spécifie des règles et des relations logiques, permettant au moteur d'inférence de dériver des conclusions.

- **Exemples** : Prolog, Datalog
- **Caractéristiques** : Utile pour les systèmes experts, l'IA, et les applications nécessitant un raisonnement logique ou des requêtes complexes.

Programmation de Requêtes (SQL)

SQL est un langage déclaratif pour manipuler et interroger des bases de données relationnelles.

- **Exemples** : Requêtes SQL dans MySQL, PostgreSQL, Oracle
- **Caractéristiques** : Permet de spécifier les données souhaitées sans détailler comment elles doivent être récupérées.

Programmation Réactive

La programmation réactive repose sur des flux de données et la propagation automatique des changements, gérant des données dynamiques et asynchrones.

- **Exemples** : RxJS (JavaScript), ReactiveX, React
- **Caractéristiques** : Simplifie la gestion des changements d'état complexes dans les applications en temps réel.

Programmation Basée sur les Règles

Ce paradigme définit des règles qui déclenchent des actions lorsque certaines conditions sont remplies.

- **Exemples** : Drools (Java), CLIPS
- **Caractéristiques** : Utilisée pour l’automatisation, la gestion d’entreprise et les systèmes de recommandation.

Programmation Basée sur les Contraintes

La programmation par contraintes permet de définir des contraintes logiques ou mathématiques, le système trouvant une solution satisfaisant toutes les contraintes.

- **Exemples** : Oz, IBM CPLEX, Google OR-Tools
- **Caractéristiques** : Utilisée pour la planification, l'optimisation et les problèmes de satisfaction de contraintes.

Les différences entre la programmation impérative et déclarative

Programmation Impérative

On ordonne à l'ordinateur **comment** effectuer une tâche, en lui donnant une série d'instructions.

Chaque **instruction** est exécutée dans un ordre précis, modifiant l'état du programme.

Les boucles, les conditions et les variables sont des éléments clés de la programmation impérative.

Les langages impératifs les plus connus sont C, C++, C#, Java, Python, etc.

Qu’est-ce qu’une instruction ?

C’est une commande qui indique à l’ordinateur **comment** effectuer une tâche.

Par exemple:

- Affecter une valeur à une variable
- Exécuter une opération arithmétique
- Modifier l’état d’une structure de données

Qu’est-ce qu’une expression ?

C’est une combinaison de valeurs, de variables, d’opérateurs et de fonctions qui produit un résultat.

Par exemple:

- 5 + 3 : retourne la valeur 8.
- x * 10 : retourne le produit de x multiplié par 10, selon la valeur de x.
- myFunction(y) : retourne la valeur produite par myFunction appliquée à y.

Exemples :

On cherche à filtrer une liste d'étudiants pour ne garder que ceux qui ont plus de 18 ans, et en extraire leur nom.

En utilisant une approche **impérative** (C#), on doit écrire un code qui spécifie **comment** effectuer cette tâche.

En utilisant une approche **déclarative** (SQL ou F#), on spécifie **quoi** doit être fait, sans se soucier de la manière dont cela est réalisé.

Impérative (C#):

```
var students = new List<Student> {
    new Student { Name = "Alice", Age = 20 },
    new Student { Name = "Bob", Age = 17 }
};
var result = new List<string>();
foreach (var student in students) {
    if (student.Age > 18) {
        result.Add(student.Name);
    }
}
```

Déclarative (SQL):

```
SELECT name FROM students WHERE age > 18;
```

Déclarative (F#):

```
students
|> List.filter (fun s -> s.Age > 18)
|> List.map (fun s -> s.Name)
```

La somme des carrés d'une liste de nombres en utilisant une approche impérative et déclarative

Le code est beaucoup plus concis et lisible en utilisant une approche déclarative.

En C# :

```
public class MathUtils {
    public int SumOfSquares(List<int> numbers) {
        int sum = 0;
        foreach (int number in numbers) {
            sum += number * number;
        }
        return sum;
    }
}
```

En F# :

```
let sumOfSquares numbers = numbers
                           |> List.map (fun x -> x * x)
                           |> List.sum
```

En C# avec une approche déclarative avec LINQ

```
public class MathUtils {
    public int SumOfSquares(List<int> numbers) {
        return numbers.Select(x => x * x).Sum();
    }
}
```

Les avantages de la programmation fonctionnelle

1. Fiabilité accrue du code

Un des principaux avantages de la programmation fonctionnelle est son potentiel à améliorer la fiabilité du code.

En utilisant des fonctions pures et l'immutabilité, les développeurs peuvent minimiser les effets de bord. Cela conduit à moins de bogues et rend le code plus facile à tester.

Fonctions Pures

Toute fonction qui, pour un même ensemble d'entrées, produit toujours le même résultat et n'a aucun effet de bord.

On parle de « referential transparency », c'est-à-dire qu'on peut remplacer par la valeur qu'elle renvoie sans modifier l'état du système.

Ceci veut dire qu'elles n'ont aucun effet de bord.

- **Avantages** : Prévisibilité, testabilité et optimisation facilitée.
- **Exemples** :

F#:

```
let add x y = x + y
```

C# :

```
int Ajouter(int x, int y) {
    return x + y;
}
```

Exemples de fonctions impures

Une fonction impure est une fonction qui a des effets de bord, c'est-à-dire qu'elle modifie l'état du programme ou interagit avec l'environnement extérieur.

Les fonctions impures sont plus difficiles à tester et peuvent entraîner des comportements inattendus.

Il est plus difficile de prédire le résultat d'une fonction impure, car elle peut dépendre de l'état global du programme.

Elle peut également entraîner des effets secondaires indésirables, tels que des modifications de variables globales ou des appels à des fonctions externes.

```
public int CalculerAge(DateTime dateNaissance)
{
    return (int)((DateTime.Now - dateNaissance).TotalDays / 365.25);
}
```

```
public void EffectuerPaiement(IBAN iban, decimal montant)
{
    // effectue un paiement
}
```

```
public void AfficherMessage(string message)
{
    Console.WriteLine(message);
}
```

```
public void Sauvegarder(List<string> lignes)
{
    // sauvegarde les lignes dans un fichier
}
```

2. Débogage et test plus faciles

Puisque la programmation fonctionnelle met l'accent sur les fonctions pures et l'immuabilité, le débogage devient une tâche plus simple.

Vous pouvez isoler les fonctions et les tester indépendamment sans vous soucier de l'état de l'ensemble de l'application.

Cette modularité permet des tests unitaires plus efficaces.

Ils sont aussi plus rapides à écrire car c'est plus facile de générer des données d'entrée qu'un contexte tout entier.

3. Lisibilité et maintenabilité améliorées

La programmation fonctionnelle conduit souvent à un code plus propre et plus lisible.

L'utilisation de noms de fonctions descriptifs et l'évitement des effets secondaires facilitent la compréhension de ce que fait un morceau de code d'un coup d'œil.

Cette clarté peut significativement réduire le temps consacré à la maintenance.

F#:

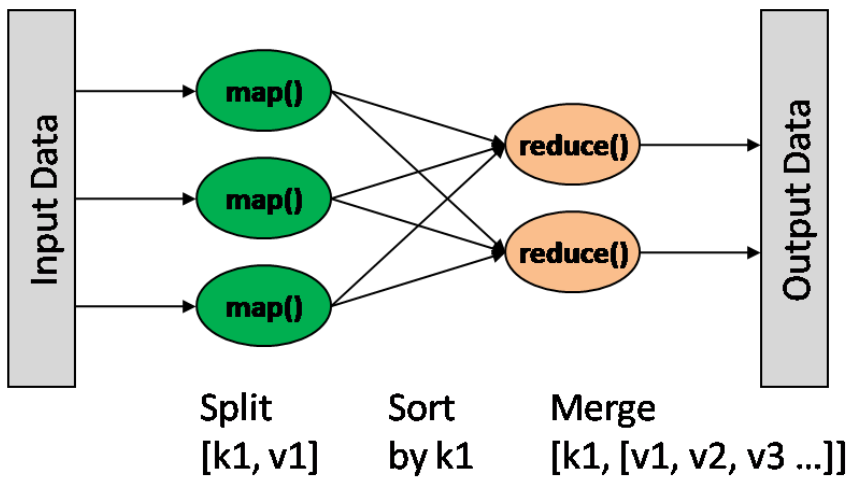
```
let array2 = [| "hello"; "world"; "and"; "hello"; "world"; "again" |]

let sumOfLengthsOfWords array =
        array
        |> Array.filter (fun x -> x.StartsWith "h")
        |> Array.sumBy (fun x -> x.Length)
```

4. Concurrence et parallélisme

La programmation fonctionnelle est intrinsèquement plus adaptée à l'exécution concurrente et parallèle.

Puisque les fonctions ne partagent pas l'état, elles peuvent être exécutées simultanément sans risque de corruption des données.

C'est particulièrement utile dans les processeurs multicœurs d'aujourd'hui, où les performances peuvent être considérablement améliorées en exploitant le parallélisme.



5. Réutilisabilité du code

De petites fonctions pour une grande modularité

Les fonctions en programmation fonctionnelle sont conçues pour être réutilisables.

En créant de petites fonctions ciblées, les développeurs peuvent facilement les combiner pour créer des opérations plus complexes.

Cette modularité non seulement fait gagner du temps mais favorise aussi la réutilisation du code dans différents projets.

Exemple en F# d'une fonction qui prend une liste de nombres et retourne la somme des carrés des nombres pairs.

```
let sumOfSquaresOfEvenNumbers numbers =
    numbers
    |> List.filter (fun x -> x % 2 = 0)
    |> List.map (fun x -> x * x)
    |> List.sum
```

Les fonctions `filter`, `map` et `sum` peuvent être réutilisées pour d'autres opérations.

Problèmes Résolus par la Programmation Fonctionnelle

Pourquoi utiliser la programmation fonctionnelle ?

Quelques problèmes courants faciles à résoudre grâce à un style fonctionnel :

- Flux asynchrones complexes
- Manipulation de gros volumes de données immuables
- Concurrence sur plusieurs cœurs de processeur
- Différences de comportement entre tests unitaires et exécution réelle

Comment changer de façon de penser pour adopter un style fonctionnel ?

Le Concept Clé de la programmation fonctionnelle :

Le code est écrit sous forme d’expressions, et non de séquences d’instructions.

Les principaux concepts de la programmation fonctionnelle

Quels sont les outils de la programmation fonctionnelle ?

Le principe de base est de **traiter les fonctions comme des valeurs** et de **composer des fonctions** pour créer des transformations de données complexes.

Pour pouvoir adopter un style fonctionnel, il est important de comprendre les concepts suivants :

- Fonctions d'ordre supérieur
- Fonctions pures
- Immuabilité
- Composition de fonctions
- Récursivité

Immuabilité

L’immutabilité signifie que les données ne peuvent pas être modifiées après leur création.

Avantages :

- Réduit les effets de bord, facilite le débogage, rend le code plus prévisible.
- Permet le partage de données sans risque de corruption.
- Facilite la programmation concurrente et parallèle.
- Evite les effets secondaires indésirables ou imprévus.

Fonctions d’ordre supérieur

Les fonctions d’ordre supérieur sont des fonctions qui acceptent une ou plusieurs fonctions comme arguments et/ou renvoient une fonction.

Les fonctions d’ordre supérieur sont utiles dans la programmation fonctionnelle car elles permettent :

- de combiner plusieurs fonctions pour en créer de nouvelles,
- d’utiliser des callbacks, d’abstraire des modèles courants en fonctions réutilisables
- d’écrire un code plus concis et plus expressif.

Composition de fonctions

La composition de fonctions consiste à combiner plusieurs fonctions pour créer une nouvelle fonction.

C'est un concept clé de la programmation fonctionnelle, car elle permet de créer des transformations de données complexes en combinant des fonctions simples.

La composition de fonctions permet de réduire la complexité du code et de le rendre plus lisible et réutilisable.

Au lieu d'avoir une longue séquence d'instructions, vous pouvez composer des fonctions pour créer des pipelines de traitement de données.

Cela permet de séparer les préoccupations et de créer des fonctions plus petites et plus spécialisées.

Exemple en F# :

```
let ajouter1 x = x + 1
let doubler x = x * 2
let ajouterEtDoubler = doubler << ajouter1 // composition de fonctions
```

Récursivité

Une fonction récursive est une fonction qui s'appelle elle-même jusqu'à une condition d'arrêt précise.

La récursivité est un concept clé de la programmation fonctionnelle, car elle permet de résoudre des problèmes de manière élégante et concise.

La récursivité est souvent utilisée pour parcourir des structures de données imbriquées, comme les arbres ou les listes.

Exemple en F# :

```
let rec factorielle n =
    if n ⩽ 1 then 1 // cas de base
    else n * factorielle (n - 1) // appel récursif
```

Conclusion