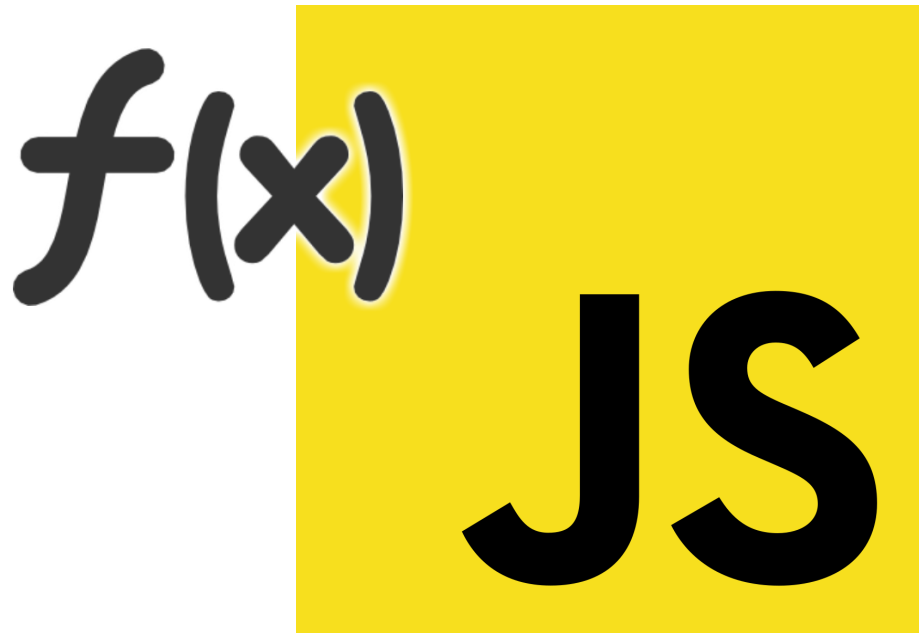


La programmation fonctionnelle en javascript



Romain Pierucci - Sami Zerrai - Sylvain Boudacher - Coraline Esedji

Table des Matières

Table des Matières

1 - Introduction

1.1 - Présentation du cours

1.2 - Tirer le meilleur parti de ce cours : à qui s'adresse-t-il ?

2 - Paradigmes

2.1 - Qu'est-ce qu'un paradigme

2.2 - Les différents paradigmes de programmation

2.3 - Programmation impérative

2.3.1 - Programmation Procédurale

2.3.2 - Programmation Événementiel

2.3.3 - POO : Programmation Orienté Objet

2.4 - Programmation déclarative

2.4.1 - Programmation Logique

2.5 - Programmation fonctionnelle, l'intersection des deux paradigmes

2.6 - QCM de la partie (10 questions)

2.7 - Correction du QCM (video)

3 - Historique de la programmation fonctionnelle

3.1 - Prologue

3.2 - Le lambda-calcul

3.3 - Le LISP

3.4 - Le ML (1978)

3.5 - De 1990 à aujourd'hui : Haskell

3.6 - Maintenant

3.3 - QCM de la partie (10 questions)

3.4 - Correction du QCM (video)

4 - Javascript

4.1 - Pourquoi le javascript pour la programmation fonctionnelle ?

4.2 - Rapide rappel de base

4.2.1 - Les variables Immuable

4.2.2 - Les fonctions

4.2.3 - Les fonctions de callback

4.2.4 - Les objets

4.3 - Notions requises pour la PF

4.3.1 - Les différents déclaration de variable

4.3.2 - Méthodes permettant l'immuabilité

4.4 - QCM de la partie (10 questions)

4.5 - Correction du QCM (video)

5 - La programmation fonctionnelle en détails

5.1 - Comprendre ce qu'elle est et à quels besoins elle répond

5.2 - Les piliers de la programmation fonctionnelle

5.2.1 - Fonction pure

5.2.2 - Immutabilité

5.2.3 - Transparence référentielle et effets de bords

5.2.4 - Les fonctions d'ordre supérieur (HOF)

Cas pratique : Recoder la fonction Array.map() (vidéo)

5.2.5 - La composition de fonction

5.2.6 - La Curryfication (ou Currying)

Cas pratique : Application du currying (vidéo)

5.2.7 - Le « Point Free style » (ou Tacit Programming)

5.2.8 - Les monades

A - Loi de l'Identité à gauche

B - Loi de l'Identité à droite

C - Loi de l'Associativité

D - Identity monad

E - D'autres monades...

5.2.9 - En résumé

Conclusion

Glossaire :

Références

1 - Introduction

1.1 - Présentation du cours

La programmation fonctionnelle est un paradigme de programmation qui se concentre sur l'utilisation de fonctions pour résoudre des problèmes informatiques. Contrairement à d'autres paradigmes de programmation, comme la programmation impérative ou orientée objet, qui mettent l'accent sur la modification de l'état de la programmation à travers des variables et des structures de données mutables, la programmation fonctionnelle utilise des fonctions pures et des données immuables pour retourner des résultats.

En javascript, la programmation fonctionnelle est devenue de plus en plus populaire ces dernières années, grâce à sa simplicité et sa capacité à faciliter la création de programmes efficaces. Avec l'ajout de fonctionnalités telles que les fonctions fléchées et les itérateurs comme "Array.map()", javascript est devenu un langage de programmation de premier plan pour la programmation fonctionnelle dans le cadre d'un projet web.

Dans ce cours, nous allons apprendre les concepts de base de la programmation fonctionnelle en javascript comme, la composition de fonctions et le currying. Nous allons également explorer les avantages de l'utilisation de la programmation fonctionnelle, comme la réduction de la complexité du code et l'amélioration de la lisibilité.

En utilisant ces techniques, il sera possible de créer des programmes plus concis et plus faciles à maintenir, qui permettront de résoudre efficacement des problèmes complexes.

Ce cours offrira les outils nécessaires pour utiliser la programmation fonctionnelle en javascript de manière efficace vos projets web.

1.2 - Tirer le meilleur parti de ce cours : à qui s'adresse-t-il ?

Tout d'abord, ce cours est-il fait pour vous ? Il est important de savoir si les connaissances et techniques contenues dans ce cours valent la peine d'investir du temps vis-à-vis de votre profil.

Ce cours s'adresse notamment :

- Aux développeurs ayant comme seules bases des connaissances en javascript
- Aux développeurs expérimentés voulant étoffer leur techniques en appliquant les notions de la programmation fonctionnelle dans leur projet
- Aux personnes en soif de connaissances voulant simplement en apprendre plus sur les paradigmes et l'histoire entourant la programmation fonctionnelle sans pour autant l'appliquer

Le cours a été structuré de façon à vous permettre d'apprendre ce qui vous intéresse. Ainsi, en fonction de votre bagage technique, vous serez naturellement attiré par un domaine plus qu'un autre (par un manque d'intérêt par exemple).

Voici les grandes catégories de ce cours :

- **1ère partie** - Etude des différents paradigmes : une vue d'ensemble sur les différents paradigmes existants, comment s'appliquent-ils et à quels besoins répondent-ils ?
- **2ème partie** - Création, application à travers l'histoire : une rétrospective de la création de la programmation fonctionnelle, ses applications à travers ses différents langages.
- **3ème partie** - Rappel javascript : les techniques enseignées seront appliquées en javascript. Cette partie fera office de rappel des connaissances de bases et celles qui sont nécessaire pour continuer le cours
- **4ème partie** - la plus conséquente de toutes puisqu'elle est la partie centrale du cours. C'est dans cette section que seront expliquées une-à-une les notions de la programmation fonctionnelle, le tout accompagné d'exemples et de cas pratiques à visionner au fur et à mesure de votre progression.

Maintenant que les différentes parties vous ont été présentées, voici les parcours que vous pouvez emprunter pour suivre ce cours :

- Vous êtes un développeur expérimenté. Vous n'êtes pas intéressé par l'histoire ni aux raisons de la création de ce paradigme mais bel et bien à son application concrète dans le quotidien d'un développeur : Rendez-vous directement à la **partie 4**.
- Vous êtes un développeur mais n'utilisez pas javascript comme langage principal au quotidien. Vous êtes uniquement intéressé par la partie technique : commencez par un rapide rappel des bases en **3ème partie**, puis poursuivez avec la **partie 4**.
- Vous n'êtes pas forcément développeur mais travaillez dans un environnement technique où une bonne connaissances des technologies est nécessaire dans votre travail : Vous voyez ce cours comme de la veille technique, apprenez alors les connaissances qui se trouvent en **partie 1** et **partie 2**.
- Enfin, peu importe qui vous êtes et ce que vous faites, vous voulez découvrir ce cours dans son intégralité : dans ce cas lisez **l'ensemble des parties**.

2 - Paradigmes

2.1 - Qu'est-ce qu'un paradigme

Provenant du grec ancien *παράδειγμα* (paradeigma) signifiant “modèle” ou “exemple”, le mot paradigme possède plusieurs connotations. Au sens large, il désigne un modèle de pensée, une manière de voir et de construire le monde. Il s'applique dans plusieurs domaines comme la science, la psychologie, les mathématiques, l'informatique... Cette notion est malheureusement souvent utilisée à tort en raison d'une mauvaise compréhension.

En informatique, dans sa définition la plus simple, le paradigme de programmation est une manière de programmer. Elle apporte un certain conformisme sur la façon d'approcher et de structurer le code ; selon l'approche choisie, elle fournira des outils, des techniques ainsi qu'un ensemble de méthodes et de principes à suivre pour respecter le cadre.

Les paradigmes confèrent une facilité de lecture, d'apprentissage ainsi qu'un modèle sur lequel se baser permettant de résoudre et d'ordonner des problématiques complexes.

Un paradigme de programmation offre et définit la vue qu'a le développeur sur l'exécution de son programme.

Par exemple, dans la programmation fonctionnelle, un programme peut être vu comme une série d'évaluations de fonctions sans états, alors que dans la programmation orientée objet, les développeurs peuvent voir le programme comme une collection d'objets interconnectés.

Idéalement, les règles du paradigme doivent être respectées pour ne jamais causer de problèmes que le paradigme lui-même résout. Ce qu'il faut comprendre c'est qu'un paradigme offre un process à suivre, un patron, un cadre pour s'assurer que tout le monde puisse parler la même langue. Si ces préceptes ne sont pas respectés, ils n'engendrent pas systématiquement une erreur ou un mauvais résultat lors de l'exécution du programme mais causent une anomalie dans le modèle.

A retenir : Un paradigme de programmation n'est pas meilleur qu'un autre, chacun a sa place et son utilité au vue des besoins du projet.

Prenons pour exemple une métaphore basée sur la construction d'un château de sable. Il existe plusieurs façons de le construire, qui offrent toutes des résultats différents: sans outil, à la main, avec une pelle ou encore avec un moule.

A la main je serais précis et le château sera modélisable à mes souhaits , à la pelle je serais plus rapide mais moins précis , tandis que je serai très rapide et fiable avec un moule mais sans aucune personnalisation.

Chacune de ces méthodes représentent un paradigme différent , elles ont toutes des avantages et des inconvénients. Aucun paradigme n'est meilleur qu'un autre.



Il est par contre plus approprié de choisir un paradigme en fonction des problèmes que l'on cherche à résoudre.

2.2 - Les différents paradigmes de programmation

Au fil des années, les paradigmes de programmation se sont de plus en plus développés, renouvelés, diversifiés pour répondre aux différentes problématiques nouvelles ou existantes.

Par ailleurs, les langages phares ont contribué à la popularisation de certains paradigmes, souvent les plus utiles.

En voici une liste :

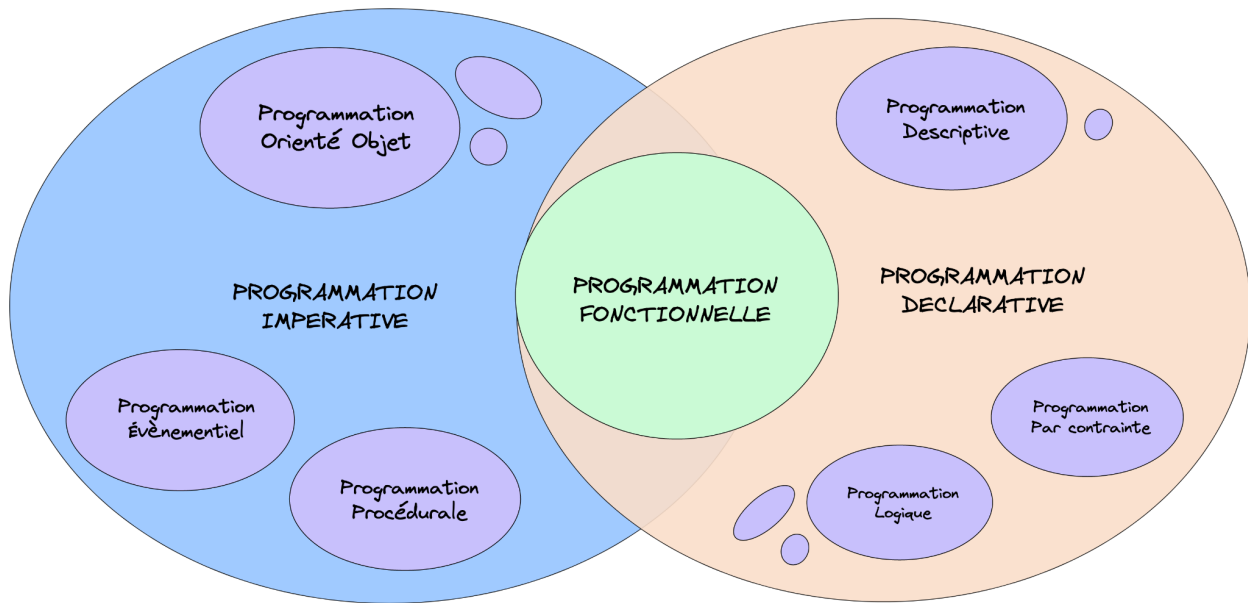
- Premièrement, de la Seconde Guerre mondiale jusqu'à aujourd'hui, l'architecture des processeurs est fondamentalement impérative.
Plus simple à comprendre par la majorité des développeurs, elle a été adoptée sous la forme de "**l'impératif structuré**" par les premiers langages comme FORTRAN en 1954, ALGOL en 1958 ou encore COBOL en 1959.
- Plus tard, le **paradigme fonctionnel** émerge suite aux travaux de Church sur le λ -calcul et les travaux sur la calculabilité. LISP est le premier langage à l'intégrer en 1958. Puis vient le Haskell en 1990 ou bien Scala en 2003.
- Par ailleurs, au début des années 1960, les norvégiens Ole-Johan Dahl et Kristen Nygaard ont créé le **paradigme de la programmation orienté objet** (ou POO). Son développement a été poursuivi par les travaux de l'Américain Alan Kay durant les années 1970.
Simula 67 ou encore Smalltalk 71 sont des exemples de langages qui ont mis ce paradigme en avant. De nos jours, la majorité des langues supportent ce paradigme.
- C'est ensuite dans le cadre des recherches sur la démonstration automatique des théorèmes menées de 1930 à 1965 que le paradigme de la logique a vu le jour. Il a été popularisé par le langage de programmation PROLOG, qu'Alain Colmerauer  et Philippe Roussel  ont développé à Luminy vers 1972.

Aujourd'hui, Python, Java, C# et C++ supportent le paradigme impératif objet, structuré et fonctionnel. Ce sont des exemples de langage multi-paradigmes. Aujourd'hui, il est très peu probable qu'un langage n'utilise qu'un seul paradigme. Cette souplesse permet la versatilité d'adapter la structure de son code selon le type de problème posé.

Voici des cas d'exemple :

- La POO est le choix idéal pour de la programmation d'interface graphique évoluée.
- La programmation fonctionnelle est le seul moyen d'effectuer un parcours d'arbre.
- L'impératif structuré permet de gérer plus facilement un calcul matriciel.

Pour finir, voici un diagramme de la décomposition des différentes catégories principales.



Si au premier abord ce diagramme peut paraître déroutant il sera expliqué terme par terme ci-dessous afin de faciliter sa lecture.

2.3 - Programmation impérative

Nous commencerons par nous intéresser aux explications des deux grands groupes de paradigmes. Notamment par le plus simple à mettre en œuvre : la programmation impérative.

En effet, il suffit de définir étape par étape les instructions à exécuter afin d'obtenir le résultat souhaité.

Reconnu pour être le plus répandu et le plus ancien, il représente une suite d'instructions qui change l'état courant de l'ordinateur. Changer l'état de l'ordinateur signifie que les données et les variables stockées dans la mémoire sont modifiées par les instructions du programme.

A savoir : Nos processeurs, qui équipent nos ordinateurs, fonctionnent sous le mode impératif.

La programmation impérative est très utile pour résoudre des problèmes de bas niveau tels que la manipulation de données, la gestion des entrées / sorties, la gestion des erreurs, et l'optimisation des performances. C'est un bon choix pour les tâches qui nécessitent un contrôle précis sur les données et les instructions de l'ordinateur.

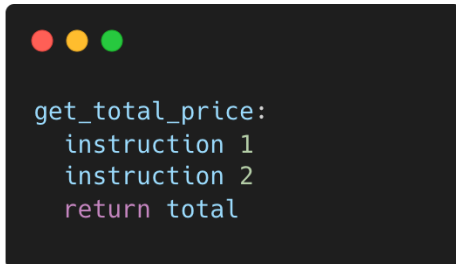
Dans son ensemble, elle comporte plusieurs type d'instructions :

- Séquence d'instructions
- Affectation
- Conditions
- Boucles
- Appel/saut [fonctions, procédure]

2.3.1 - Programmation Procédurale

Une procédure est l'ancien terme utilisé pour définir une fonction. Elle représentait initialement une suite d'instructions regroupées pour exécuter une tâche précise, enregistrer un texte par exemple, qu'on peut à tout moment appeler dans le code. Une procédure est donc caractérisée par un nom d'appel et un regroupement de lignes de code.

Plus tard, l'arrivée des fonctions reprennent le même concept en y ajoutant un retour de résultat. Par exemple, une fonction qui calcule le prix final d'un panier de course en ligne :



```
get_total_price:  
  instruction 1  
  instruction 2  
  return total
```

Ce découpage apporte de la modularité au code : elle favorise la factorisation qui à terme permettra d'obtenir un programme moins lourd, rendra la maintenabilité du code plus simple à appréhender et améliorera sa lisibilité.


Une fonction peut être appelée à différents endroits du code :

- Dans une suite d'instructions
- Dans une autre fonction
- S'appelle elle-même (notion de récursivité)

2.3.2 - Programmation Événementiel

Très simplement, la programmation événementielle correspond au déclenchement d'une exécution d'actions automatiques suite à l'apparition d'un événement, qu'elle provienne d'un changement d'état ou bien d'une intervention extérieure du système (par l'utilisateur ou le système lui-même).

Pour l'illustrer, javascript est un bon exemple car il inclut cette notion d'interaction avec l'utilisateur sur une page web.



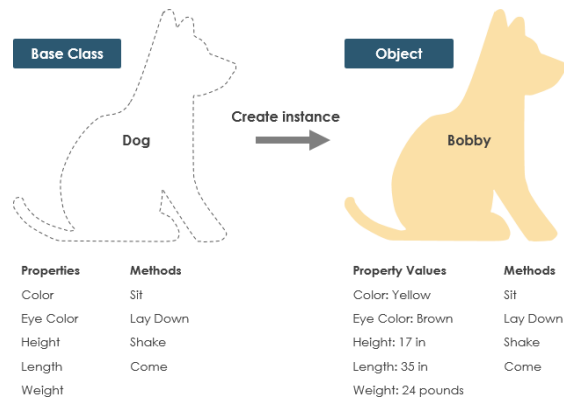
```
function eventHandler(event) {  
    console.log('Button has been clicked!')  
}  
  
const button = document.getElementById('button')  
button.addEventListener('click', eventHandler)
```

Ce programme affiche un message dans la console Web lors d'un clic sur un bouton.

2.3.3 - POO : Programmation Orienté Objet

Dans ce paradigme, tout le projet s'articule autour du concept d'objet qui rassemble le comportement d'un programme accompagné de son état.

Ces objets sont définis et interagissent entre eux.



Un objet contient :

- des données appelées **attribut** ou **propriété**
- des fonctions appelées **méthode**

Pour donner comme exemple, la classe Dog définit les caractéristiques d'un chien comme sa race, sa couleur ou encore son âge et ses différentes fonctions qui expriment ses activités : courir, manger, dormir...

```

class Dog {
  constructor(name, size, age, race, color) {
    this.name = name;
    this.size = size;
    this.age = age;
    this.race = race;
    this.color = color;
  }

  eat() {
    console.log(`${this.name} mange ses croquettes !`);
  }
  sleep() {
    console.log(`${this.name} dort paisiblement !`);
  }
  sit() {
    console.log(`${this.name} s'assoit !`);
  }
  run() {
    console.log(`${this.name} s'amuse dans le jardin !`);
  }
}

const myDog = new Dog('Buddy', 5, 2, 'Bulldog', 'Brun');
myDog.run(); // affiche "Buddy s'amuse dans le jardin !"

```

Ce code ci-dessus inclut la définition d'une classe et l'instanciation d'un objet.

La définition d'une classe est tout simplement la structure de l'objet avec ses attributs et ses méthodes. Elle définit les caractéristiques et les comportements d'un groupe d'objets. Les méthodes peuvent modifier accéder, modifier, créer ou supprimer ses attributs.

L'instanciation d'un objet consiste à créer une nouvelle instance d'une classe en utilisant le mot-clé `new` suivi du nom de la classe. Cette nouvelle instance est appelée objet.

Dès l'instanciation de l'objet, la méthode "constructor" est appelée afin d'initialiser ses caractéristiques.

La programmation orienté objet vient avec plusieurs concepts fondamentaux :

- encapsulation
- abstraction
- polymorphisme
- interface
-

En général, ce paradigme est plus couramment utilisé avec des langages tels que le PHP ou le Java.

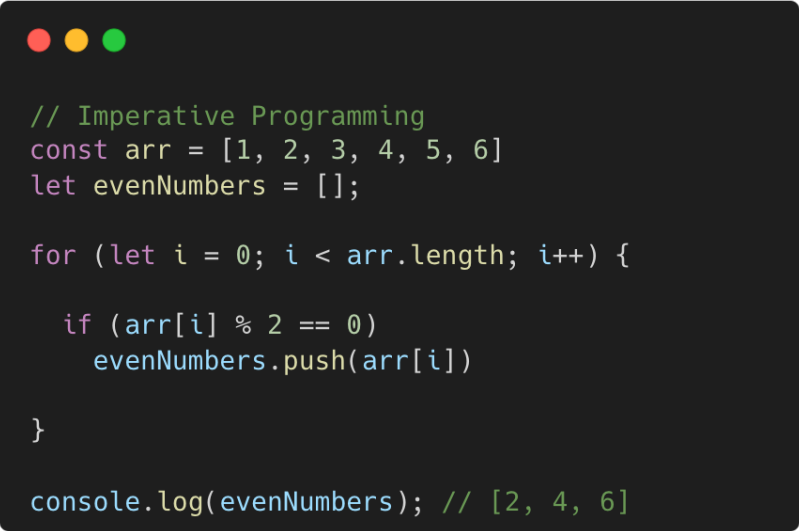
2.4 - Programmation déclarative

Le paradigme impératif se concentre sur les étapes spécifiques pour atteindre un résultat, tandis que le paradigme déclaratif se concentre sur la définition du résultat souhaité sans spécifier comment il doit être atteint.

La meilleure façon de mettre en évidence cette explication, c'est par le code.

Voici deux bouts de codes, impératif et déclaratif, donnant le même résultat.

On cherche à récupérer les nombres pairs d'un tableau.



```
// Imperative Programming
const arr = [1, 2, 3, 4, 5, 6]
let evenNumbers = []

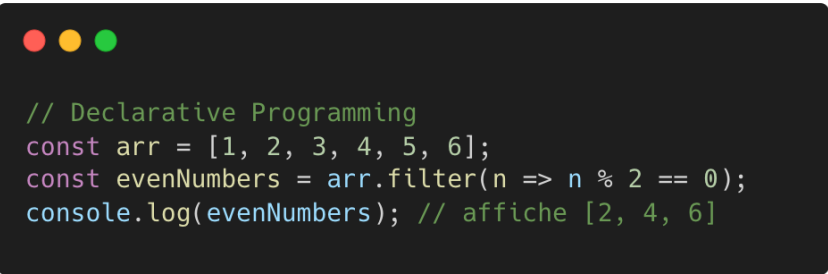
for (let i = 0; i < arr.length; i++) {

  if (arr[i] % 2 == 0)
    evenNumbers.push(arr[i])

}

console.log(evenNumbers); // [2, 4, 6]
```

Dans ce code, une boucle *for* est utilisée de manière impérative pour parcourir chaque élément du tableau et vérifier s'il répond à la condition de parité.



```
// Declarative Programming
const arr = [1, 2, 3, 4, 5, 6];
const evenNumbers = arr.filter(n => n % 2 == 0);
console.log(evenNumbers); // affiche [2, 4, 6]
```

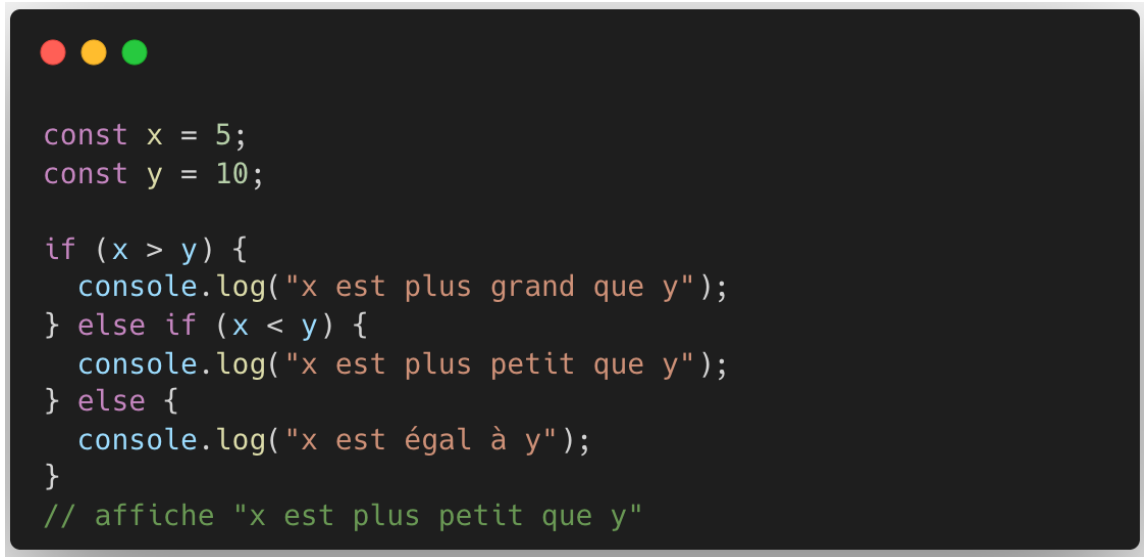
Dans ce code, la fonction *filter* est utilisée de manière déclarative pour sélectionner les éléments du tableau qui répondent à une condition (ici, être divisibles par 2).

2.4.1 - Programmation Logique

La programmation logique est une forme de programmation qui utilise des formules logiques pour représenter et résoudre des problèmes. Elle est souvent utilisée dans les domaines de l'intelligence artificielle et de la recherche opérationnelle, car elle permet de modéliser de manière précise des situations complexes avec des contraintes et des incertitudes.

En javascript, il est possible d'utiliser la programmation logique en combinant les opérateurs logiques (&&, ||, !) et les comparaisons (==, >, <, etc.) pour écrire des expressions qui peuvent être évaluées comme vraies ou fausses. Ces expressions peuvent être utilisées dans des structures de contrôle de flux, comme les if et les while, pour exécuter différentes parties de code en fonction de leur résultat.

Voici un exemple de code javascript qui utilise la programmation logique pour résoudre un problème simple :



```
const x = 5;
const y = 10;

if (x > y) {
  console.log("x est plus grand que y");
} else if (x < y) {
  console.log("x est plus petit que y");
} else {
  console.log("x est égal à y");
}
// affiche "x est plus petit que y"
```

Dans ce code, l'expression `x > y` est évaluée comme *false*, car `x` n'est pas plus grand que `y`. L'expression `x < y` est quant à elle évaluée comme *true*, ce qui permet d'exécuter le bloc de code correspondant.

2.5 - Programmation fonctionnelle, l'intersection des deux paradigmes

Pour terminer, la programmation fonctionnelle est une approche de programmation qui met l'accent sur l'utilisation de fonctions pour résoudre des problèmes comme les calculs mathématiques, la manipulation de données, la gestion de flux de données, la résolution de problèmes de logique et bien d'autres !

Elle comporte l'union de la programmation déclarative et impérative.

Néanmoins, son utilisation est principalement basée sur une approche déclarative, tandis que la programmation impérative utilise une approche plus directe et plus précise pour résoudre ces problèmes.

3 - Historique de la programmation fonctionnelle

3.1 - Prologue

Quelques décennies avant l'invention des premiers ordinateurs programmables , beaucoup de mathématiciens tentaient de résoudre le "Problème de l'arrêt".

Une machine basée sur des instructions mathématiques peut-elle s'arrêter un jour ou va-t-elle continuer éternellement à calculer ? Et pouvons-nous, ou non, anticiper son arrêt ?

En 1936, les mathématiciens Alonzo Church et Alan Turing trouvent tous deux une réponse identique à cette question. Leurs manuscrits sont publiés avec seulement un mois d'écart et indiquent, d'une façon tout à fait différente qu'il n'est mathématiquement impossible de prouver qu'une instruction s'arrêtera dans un temps fini ou non. Turing utilise le concept d'une machine abstraite plus connue sous le nom de la "machine de Turing" quand Church lui, bâtit sans le savoir l'origine de la programmation fonctionnelle.

3.2 - Le lambda-calcul

Church crée un langage de programmation appelé le lambda-calcul (ou λ -calcul) , ce langage est théorique et n'est donc pas prévu pour être exécuté sur un ordinateur (rappelons que le premier ordinateur électronique est créé en 1945). Il édifie à travers ces travaux le concept de fonction et d'application et est surtout le premier à intégrer, définir et caractériser le principe de fonctions récursives.

Afin de répondre à des problèmes fondamentaux de logique mathématique , le λ -calcul permettra à d'autres mathématiciens de pouvoir construire et formaliser plusieurs grands concepts mathématiques comme celui de la théorie de la calculabilité ainsi que du modèle de Herbrand-Gödel.

3.3 - Le LISP

Le LISP est le premier langage de programmation fonctionnel et deuxième plus ancien langage de programmation encore en usage (après le FORTRAN).

Créé en 1958 suite à la publication de *Recursive Functions of Symbolic Expressions and Their Computation by Machine* par le mathématicien et informaticien John McCarthy lors d'un projet pour le MIT (*Massachusetts Institute of Technology*), le LISP se distingue très rapidement grâce à son typage dynamique des données et sa gestion automatique de la mémoire. Il est surtout le premier langage à lancer le domaine de "l'apprentissage automatique", que l'on nomme aujourd'hui "L'intelligence artificielle".

Très en avance sur son temps, le LISP propulse la programmation fonctionnelle et donne naissance à d'autres sous langages de programmation. Ils permettent à tous ses utilisateurs de créer des programmes très souples et beaucoup moins gourmands en ressources pour le processeur. Ils consomment cependant plus de mémoire. Etant donné que la mémoire coûte à cette époque environ un dollar par bit et n'est pas aussi facilement ajustable, la programmation fonctionnelle ne décollera pas plus avant la fin des années 80 et du début de l'abondance de mémoire bon marché.

3.4 - Le ML (1978)

Bien que très avancé sur le plan mathématique, le LISP ne convient pas à tout le monde. En 1973, Robin Milner et ses étudiants de l'université d'Edimbourg rencontrent des difficultés avec le système de typage du LISP qui, dans le cadre de leurs recherches, permettait de prouver des assertions mathématiques totalement fausses. Pour pallier ces problèmes et pouvoir démontrer formellement la "*Logic for Computable Functions*", Milner crée le *Meta Language* (ML).

Le *ML* est le premier langage à inclure l'inférence de type polymorphique, ainsi qu'un mécanisme de gestion des exceptions intégrée. Pour plus de clarté, il est le premier langage à disposer du typage automatique; il est donc le premier à distribuer la charge de travail au compilateur et non plus à l'utilisateur.

Cependant, le *Meta Language* est un langage de programmation fonctionnel dit "impur" car il est possible de programmer en impératif dessus. Les fonctions sont donc sujettes à des effets néfastes et non désirés comme l'effet de bord ou d'autres problématiques...

3.5 - De 1990 à aujourd'hui : Haskell

Nommé d'après le logicien Haskell B. Curry, le Haskell créé en 1990 est le fruit des travaux d'un comité de chercheurs en théorie des langages qui s'était intéressé aux langages fonctionnels et au principe d' "évaluation paresseuse" (technique d'implémentation des programmes récursif).

Le Haskell dépoussière et modernise le paradigme des langages fonctionnels en ajoutant diverses fonctionnalités qui servent toujours aujourd'hui de base comme par exemple les fonctions d'ordre supérieur ou le currying.

Très prisé par les mathématiciens et la communauté de l'intelligence artificielle, le Haskell tombe rapidement dans les mains de développeurs cherchant à créer et maintenir de grands projets. Le langage fait ses preuves grâce à ses nombreux et incontestables avantages comme le gain de productivité via un code court, clair et facile à entretenir pour l'époque mais surtout par le grand gain de fiabilité que le langage apporte.

3.6 - Maintenant

Après Haskell, de nombreux autres langages de programmation fonctionnels ont été développés, tels que OCaml, F#, et Elixir:

OCaml est un langage de programmation fonctionnel qui a été développé dans les années 1990 à l'Université de Paris et est largement utilisé dans l'industrie et la recherche. Il est connu pour être efficace et pour avoir un type de données record très puissant.

F# est un langage de programmation fonctionnel et impératif développé par Microsoft. Il a été conçu pour être utilisé dans le développement de logiciels commerciaux et est souvent utilisé dans les domaines de la finance et de la gestion de données.

Elixir est un langage de programmation fonctionnel créé en 2011 qui a été conçu pour être utilisé dans la création de systèmes distribués de haute performance. Il est basé sur le langage de programmation fonctionnel Erlang et a gagné en popularité grâce à sa facilité d'utilisation et à sa capacité à gérer efficacement les processus en parallèle.

En plus de ces langages, il y a également eu une augmentation de l'intérêt pour les concepts de la programmation fonctionnelle dans d'autres langages de programmation, tels que Python et JavaScript. De nombreux développeurs utilisent ces langages de manière fonctionnelle pour résoudre certains problèmes de manière plus efficace et plus concise.

Cela peut être aussi utilisé pour améliorer la performance d'un programme en permettant une

parallélisation plus efficace des tâches. Comme les fonctions purement fonctionnelles ne dépendent pas de l'état externe du programme, elles peuvent être exécutées de manière indépendante les unes des autres, ce qui permet une exécution plus rapide sur les processeurs multi-coeurs.

En résumé, l'utilisation de concepts de programmation fonctionnelle peut améliorer la performance, la lisibilité et la facilité de maintenance du code, ce qui explique pourquoi de nombreux développeurs utilisent des langages de programmation qui ne sont pas de base fonctionnels de manière fonctionnelle.

4 - Javascript

La programmation fonctionnelle gagne de plus en plus de popularité parmi les développeurs depuis ces dernières années. Dès lors plusieurs langages de programmation sont spécifiquement conçus pour fonctionner avec ce paradigme, tels que Haskell, Elixir, Clojure, etc.

Le JavaScript est un langage de programmation créé par Brendan Eich, sorti en 1995. Le but de ce langage est de rendre les pages WEB plus interactives et dynamiques. Grâce à l'arrivée du WEB 2.0, le JavaScript est devenu incontournable dans la conception d'applications WEB. L'arrivée de Node.js en 2009 permet l'implémentation du langage côté serveur.

Nous avons donc choisi le JavaScript pour approfondir ce paradigme. Le JS possède certaines bibliothèques dans l'écosystème qui nous aident à travailler avec un style de programmation tourné vers le fonctionnel dans nos applications, comme Ramda et Lodash. Mais l'utilisation de ces bibliothèques n'est pas forcément nécessaire, nous avons la possibilité d'utiliser uniquement les principaux concepts de JavaScript pour obtenir un code de programmation plus fonctionnel.

4.1 - Pourquoi le javascript pour la programmation fonctionnelle ?

JavaScript est un langage de programmation polyvalent qui peut être utilisé non seulement pour la programmation fonctionnelle, mais aussi pour la programmation orientée objet et impérative. Il est souvent utilisé pour la programmation fonctionnelle car il contient des fonctionnalités qui facilitent l'utilisation de cette technique de programmation.

Parmi ces caractéristiques figurent :

- Fonctions anonymes et possibilité de transmettre des fonctions en tant que paramètres à d'autres fonctions. Cela vous permet d'abstraire des opérations complexes et de créer des fonctions de haut niveau que vous pouvez utiliser pour rendre votre code plus lisible et maintenable.
- Des fonctions fléchées qui vous permettent de définir vos fonctions de manière plus précise et lisible. Ils sont particulièrement utiles pour créer des fonctions anonymes et de les transmettre en tant que paramètres à d'autres fonctions sans utiliser la syntaxe "fonction".
- Possibilité de définir les propriétés et les méthodes d'un objet à l'aide de mots-clés tels que "get" et "set". Cela vous permet de créer des objets de "capacité" qui peuvent être utilisés pour implémenter des fonctionnalités avancées telles que la gestion d'état et l'encapsulation.

En résumé, JavaScript est un langage de programmation polyvalent qui offre de nombreuses fonctionnalités utiles pour la programmation fonctionnelle, ce qui en fait un choix populaire pour ceux qui souhaitent utiliser cette approche de programmation.

4.2 - Rapide rappel de base

Pour commencer la programmation fonctionnelle, un petit rappel des bases de certaines notions de JavaScript, telles que l'immutabilité des variables ou encore les différents types de fonctions.

4.2.1 - Les variables Immuable

L'immutabilité est un concept très utilisé dans les langages de programmation fonctionnels où la valeur d'une variable ne change pas après l'affectation, autrement dit constante.

Cela a l'avantage de simplifier la gestion de l'état de l'application en réduisant le nombre de variables partagées.

Comme nous le verrons, les objets immuables rendent les frameworks MVC comme React et AngularJS plus efficaces. C'est l'une des raisons pour lesquelles le terme d'immutabilité est entendu de plus en plus fréquemment dans la communauté JavaScript.

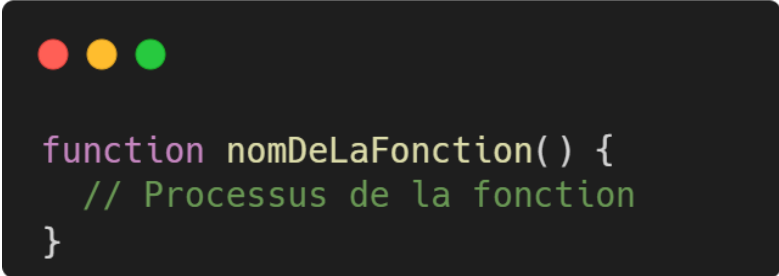
JavaScript a des types primitifs et des types de référence.

Les types primitifs incluent les nombres, les chaînes, les booléens, les valeurs nulles et indéfinies.

Les types de référence incluent les objets, les tableaux et les fonctions.

4.2.2 - Les fonctions

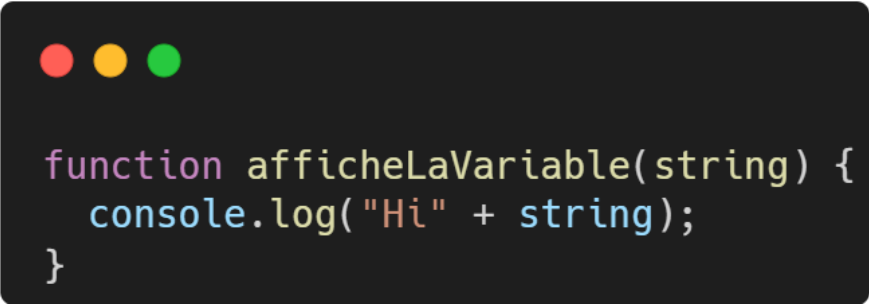
Une fonction est une suite de schéma d'actions à accomplir qui détermine un résultat final et le rend disponible. Elle est une des premières organisations du code. Nous avons la possibilité de pouvoir réutiliser nos fonctions créées. On peut déclarer une fonction de cette manière :



```
function nomDeLaFonction() {  
    // Processus de la fonction  
}
```

Les fonctions peuvent accepter des arguments, c'est-à-dire des valeurs qui sont passées à la fonction. La fonction peut ensuite utiliser ces valeurs à l'intérieur de sa logique. Si on veut ajouter des arguments à la fonction, il faut les insérer dans les parenthèses et les séparer par des virgules (arg1, arg2, arg3).

Par exemple la fonction suivante affiche à la console la valeur de "string" passée en argument :




```
function afficheLaVariable(string) {  
    console.log("Hi" + string);  
}
```

4.2.3 - Les fonctions de callback


Les fonctions de rappel (ou "callback" en anglais) en JavaScript sont des fonctions qui sont passées en tant qu'argument à une autre fonction. Elles sont utilisées pour déclencher une action ou une série d'actions lorsque certains événements se produisent ou certains processus se terminent. Les fonctions de rappel permettent de rendre le code plus modulaire et de faciliter la gestion des erreurs.

Dans le cas de ce programme ci-dessous, la fonction de rappel est déclarée en utilisant une fonction fléchée (`() =>`) qui prend un paramètre (`color`) et utilise `console.log(color)` pour afficher ce paramètre dans la console.



```
const colors = ['red', 'green', 'blue'];  
colors.map((color) => console.log(color));
```

Les fonctions de rappel anonymes sont souvent utilisées avec des fonctions d'ordre supérieur, comme *map*, *filter*, *forEach*, etc. pour itérer sur des tableaux ou des objets.



```
const colors = ['red', 'green', 'blue'];  
  
const myCustomCallback = (color) =>  
  console.log(color);  
  
colors.map(myCustomCallback);
```

Il est important de noter que les fonctions de rappel peuvent causer des problèmes de portée variable si elles font référence à des variables définies en dehors de leur contexte. Pour résoudre ce problème, il est courant d'utiliser la technique de fermeture (*closure*) pour conserver une copie de la variable dans la portée de la fonction de rappel.

Avant de passer aux notions requises pour commencer la programmation fonctionnelle, nous ferons un court rappel sur les objets en JavaScript.

4.2.4 - Les objets


JavaScript n'est pas un langage de programmation orienté objet complet comme Java, mais il est basé sur un modèle simple d'objets.

Les objets sont des structures avec des propriétés qui contiennent des variables ou d'autres objets. Les objets ont également des fonctions associées, appelées méthodes de l'objet. Outre les objets JavaScript de base (tels que les tableaux et les objets mathématiques), vous pouvez définir vos propres objets.

Les objets peuvent avoir des propriétés et des méthodes. Les propriétés sont les valeurs mutables qui composent un objet, et les méthodes représentent les tâches associées à l'objet. Il existe deux notations pour créer et manipuler des objets : littéral et l'utilisation des constructeurs.

Déclarer un objet :

Vous pouvez déclarer un objet directement dans une variable avec des propriétés et des méthodes :



```
1 var citation = {}  
2 citation.titre="KISS";  
3 citation.description="keep it simple, stupid";  
4 citation.auteur = "Internet";
```

Constructeur:

Lorsque vous créez un objet par l'intermédiaire d'un constructeur, une déclaration de fonction est utilisée.

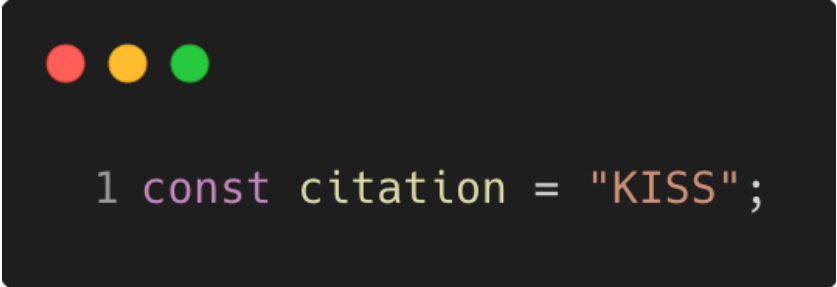
Par conséquent, vous pourrez créer plusieurs fois le même objet dans des variables différentes.

```
1 var citation = {}
2 citation.titre = "KISS";
3 citation.description = "keep it simple, stupid";
4 citation.auteur = "Internet";
5
6 function Citation(titre, description, auteur) {
7     this.titre = titre;
8     this.description = description;
9     this.auteur = auteur;
10 }
11
12 let _Kiss = new Citation("KISS",
13     "keep it simple, stupid",
14     "Internet");
```

4.3 - Notions requises pour la PF

4.3.1 - Les différents déclaration de variable

Depuis ES6 nous avons la possibilité de définir les variables de notre code en constantes, grâce au mot-clé “const”. La déclaration via “const” rend la valeur immuable.




```
1 const citation = "KISS";
```

Le mot-clé const crée uniquement une référence en lecture seule à la valeur. En d'autres termes, la valeur ne peut pas être réaffectée. La référence MDN indique :

“La déclaration *const* crée une référence en lecture seule à une valeur. Cela ne signifie pas que la valeur qu'elle contient est immuable, mais simplement que l'identificateur de variable ne peut pas être affecté.”

Donc, si nous voulons changer la valeur d'une constante, cela ne fonctionnera pas.




```
const citation = "KISS";  
  
citation = "Keep it simple stupid";  
// l'identifiant 'citation' a déjà été déclaré
```

Une nouvelle façon de déclarer des variables qui peuvent être considérées comme l'opposé du mot-clé “const”. Le mot-clé “let” vous permet de créer des variables mutables comme des constantes, mais grâce à ce mot-clé il est possible de leur attribuer de nouvelles valeurs.

4.3.2 - Méthodes permettant l'immuabilité

JavaScript est un langage orienté prototype et orienté objet et n'a pas été conçu pour être immuable. *Object.freeze* peut être utilisé pour garantir partiellement (premier niveau uniquement) l'immuabilité de l'objet, mais ce n'est pas une solution ultime.

Ainsi, certaines méthodes favorisent l'immuabilité afin de renvoyer de nouvelles valeurs, tandis que de nombreuses méthodes modifient directement les variables.



```
1 const merged = array.concat(arr1, arr2);  
2 const filtered = array.filter(fn);  
3 const mapped = array.map(fn);
```

Dans les langages fonctionnels, vous ne modifiez jamais les variables sur lesquelles les méthodes et les fonctions opèrent. Pour éviter cela et respecter l'immuabilité, créez une nouvelle variable en JavaScript en copiant tous les éléments de la variable précédente.

5 - La programmation fonctionnelle en détails

5.1 - Comprendre ce qu'elle est et à quels besoins elle répond

Les inventeurs de ce type de programmation se sont inspirés des mathématiques - en particulier des fonctions - afin de créer ce nouveau paradigme.

Les fonctions ont été inspirées des mathématiques dans lesquelles, lorsque l'on prend une fonction donnée et qu'on lui passe un paramètre, le paramètre de sortie en sera toujours identique. Qu'il s'agisse de la fonction la plus simple comme une fonction affine enseignée au collège, à la plus complexe.

Pour un certain nombre de paramètres et s'ils sont inchangés, le résultat sera le même, et ce peu importe l'environnement et les conditions dans lesquelles la fonction a été exécutée.

C'est ainsi que le monde du développement s'est inspiré de cette règle pour créer un nouveau paradigme : la programmation fonctionnelle.

Dans le monde réel - et en fonction du contexte des missions et de l'entreprise - le projet est amené de sorte à ce que la dette technique s'accumule de plus en plus, et ce de plus en plus vite.

Cela est dû à divers facteurs. Néanmoins une idée qui revient majoritairement est que les fonctions que l'on écrit se basent majoritairement sur d'autres paramètres et structures de données qui, s'ils sont amenés à changer (ou non), vont modifier le résultat de retour; ce qui est contraire à la logique mathématique auxquelles les sciences informatiques (considérées comme une branche spécialisée des mathématiques) ont emprunté la logique de fonctionnement.

On se retrouve avec le postulat suivant : Une fonction mathématique retournera toujours la même valeur.

Une fonction écrite dans le cadre d'un programme et / ou projet informatique ne retournera pas toujours la même valeur.

On peut grossièrement résumer que le but de la programmation fonctionnelle est d'essayer de reproduire le fonctionnement d'une fonction mathématique abstraite appliquée à l'informatique.

Pour se faire, il faut regarder les éléments qui posent problèmes actuellement dans les projets informatiques modernes :

- Les données peuvent changer dans une fonction. Si l'on se base sur un tableau donné, et que l'un des éléments de celui-ci est changé, alors le résultat le sera lui aussi. Il faut donc trouver un moyen de ne pas le modifier.
- Les effets de bords
- Des dépendances externes mais utilisées dans une fonction
- Calculs asynchrones
- Processus aléatoires

Cette nouvelle façon de coder a donc pour objectif de se rapprocher au plus près d'une fonction mathématique et ce par différents aspects et techniques :

- L'immutabilité
- Fonctions pures
- Transparence référentielle
- Fonctions de premiers ordres
- Compositions de fonctions
- Le *currying* (ou fonctions dites "curried")
- Le *point free* (ou "tacit programming")
- Les monades

Nous aborderons dans cette section les différentes notions de la programmation fonctionnelle avec des exemples en javascript.

5.2 - Les piliers de la programmation fonctionnelle

Le concept de la programmation fonctionnelle est désormais moins abstrait et plus facile à appréhender. Cependant qu'en est-il concrètement ?

Cette section portera sur des explications des notions importantes appliquées en javascript

5.2.1 - Fonction pure

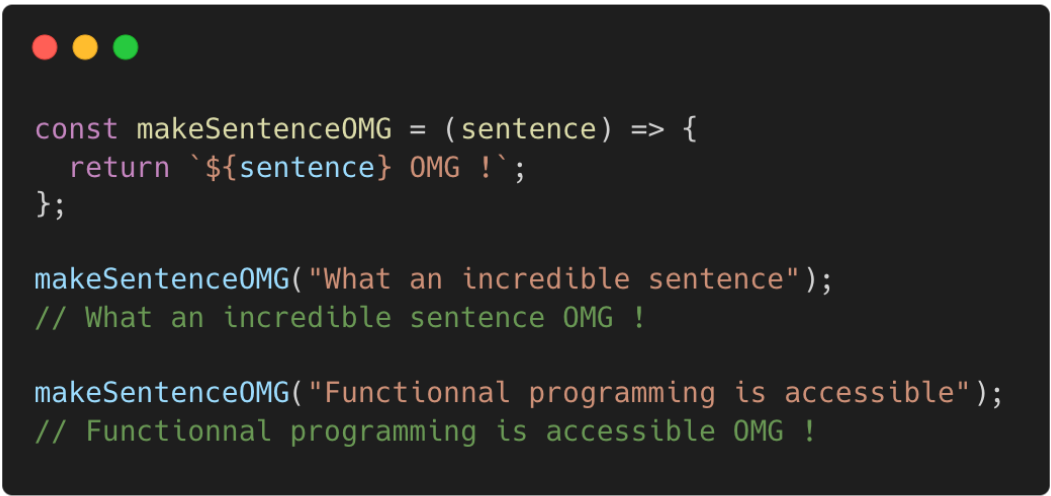
Comment écrire correctement une fonction ?

Cela peut paraître futile au vu de la partie précédente sur Javascript, néanmoins il est important dans ce contexte de programmation fonctionnelle d'établir une base solide sur ce qu'est une fonction.

Globalement, une fonction permet de découper un problème complexe en un ensemble de petits problèmes plus abordables grâce à l'utilisation d'ensembles de codes réutilisables qui évitent de conserver un grand ensemble monolithe plus complexe à comprendre.

Nous pouvons maintenant nous poser la question suivante : qu'est-ce qu'une bonne fonction en programmation fonctionnelle ? Pour simplifier : c'est un ensemble de lignes de code que l'on a regroupé dans le seul et unique objectif de répondre au besoin pour lequel elles ont été conçues.

Par exemple :



```
const makeSentenceOMG = (sentence) => {  
  return `${sentence} OMG !`;  
};  
  
makeSentenceOMG("What an incredible sentence");  
// What an incredible sentence OMG !  
  
makeSentenceOMG("Functionnal programming is accessible");  
// Functionnal programming is accessible OMG !
```

Cette fonction accepte un paramètre `sentence`, une chaîne de caractère. Elle a pour seul objectif de retourner la variable `sentence` suivi de “OMG !” à la fin de celle-ci.

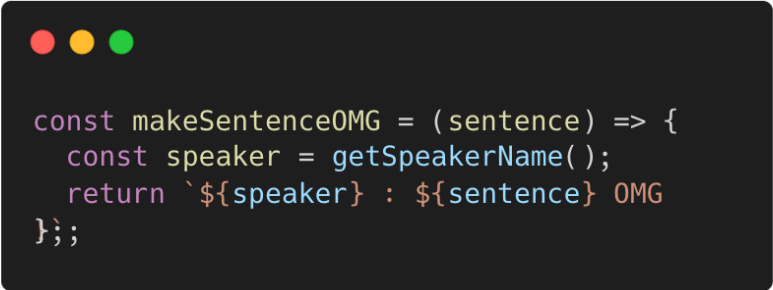
Avec la fonction précédemment écrite, si celle-ci vient à être appelée plusieurs fois de suite avec le même paramètre, on peut alors affirmer que la valeur de retour sera toujours identique.

Cette fonction est dite fonction pure, et constitue un des concepts les plus importants de la programmation fonctionnelle.

Est dite “fonction pure” une fonction respectant strictement les deux règles suivantes :

- La valeur renvoyée par la fonction doit toujours être identique pour les mêmes arguments passés en paramètre (donc aucune utilisation de variable en dehors du scope de celle-ci, comme des variables globales)
- L'entièreté de la fonction ne doit contenir aucun effets de bords (cf la section “transparence référentielle et effets de bords” pour plus de détails)

Utilisons l'exemple d'une fonction impure pour contraster :



```
const makeSentenceOMG = (sentence) => {  
  const speaker = getSpeakerName();  
  return `${speaker} : ${sentence} OMG  
};;
```

La fonction ci-dessus a été améliorée dans le but d'inclure l'auteur de la phrase utilisée en argument de la fonction, en dehors de ce détail, le but de la fonction reste inchangé.

Cependant un problème subsiste avec la façon dont on récupère l'auteur : Par l'intermédiaire de *getSpeakerName()*, on autorise la fonction à utiliser une ressource qui est extérieure au scope de la fonction.

Le résultat de *makeSentenceOMG* ne peut donc pas être garanti pour chaque appel avec les mêmes paramètres car absolument rien ne garantit que *getSpeakerName* retournera avec le même nom pour la même phrase. Rien ne garantit non plus que le nom retourné restera le même au cours de l'exécution du programme.

Et c'est parce que la fonction peut produire un résultat inattendu que l'on peut dire qu'elle possède un effet de bord.

Utiliser une fonction avec des effets de bords la rend impure, ce qui causera inévitablement des problèmes lors du processus de développement et de la vie du programme.

Utiliser au maximum les fonctions pures, que ce soit dans le contexte de la programmation fonctionnelle ou non, est une bonne pratique à respecter. C'est un gage de qualité pour le programme.

5.2.2 - Immutabilité

Il est important de poser les bases en ce qui concerne l'immutabilité. Ce point fait partie des fondements de la programmation fonctionnelle. De plus, lorsque l'on applique ce paradigme en javascript, il peut être facile de se permettre quelques facilités à l'instar du langage en lui-même.

En effet, Javascript est un langage à typage dynamique et reposant quant à lui sur le paradigme de l'orienté objet.

Des solutions simples existent pour développer dans un environnement favorable à l'immutabilité :

- Ne pas utiliser la déclaration *var*
- Ne pas utiliser la déclaration *let*

Il suffit d'utiliser les règles ci-dessus pour s'assurer d'une bonne base.

Viennent ensuite des règles légèrement plus complexes à mettre en place:

- Ne pas utiliser la mutation d'objets ou de tableaux

Il est vrai qu'il existe la déclaration *const* en javascript; elle ne permet cependant pas de créer une variable constante à proprement parler, seulement d'imposer une constante sur la référence de la variable créée.

Cela ne pose pas de problèmes pour les types primaires comme un nombre ou une chaîne de caractère, mais en pose bel et bien lors de l'utilisation d'objets ou de tableaux.

Un rapide exemple ci-dessous :

```
/** VARIABLES PRIMAIRES **/  
  
var pata = "pouet";  
// Incorrect : Car on peut toujours réaffecter la variable ultérieurement  
  
let pata = "pouet";  
// Incorrect : Pour les mêmes raisons, mais avec un scope différent  
  
const pata = "pouet";  
// Correct : La valeur est primaire. Elle ne peut être modifiée. On considère alors que cette valeur  
est une constante et par conséquent immuable  
  
/** VARIABLES OBJETS **/  
  
const pata = ["p", "o", "u", "e", "t"]; // Correcte mais...  
pata[0] = "c"; // Ne produit pas d'erreur. La référence de la constante pata est la même (le tableau)  
mais son contenu peut changer  
  
const pata = { pouet: true }; // Correct mais...  
pata.crepe = true; // Ne produit pas d'erreur non plus, pour les mêmes raison que le tableau ci-dessus  
  
Object.freeze(pata); // Seule façon de véritablement rendre immuable une variable comme les objet
```

D'après l'exemple ci-dessus, on constate que seules les variables de type Object posent problèmes et restent mutables même lorsque qu'on les déclare avec *const*.

Il existe cependant une solution pour pallier ce problème : *Object.freeze()*, une méthode de la classe *Object* permettant de geler l'objet passé en paramètre.

Geler un objet signifie que celui-ci ne sera pas modifiable. Il deviendra donc impossible de rajouter, supprimer ou encore modifier une propriété de l'objet.

Pour rappel, un tableau est aussi un objet en javascript; cette technique est donc aussi applicable pour les tableaux.

L'inconvénient résulte dans le fait qu'il faut écrire une instruction supplémentaire pour rendre une variable véritablement immuable en javascript.

—

Une solution serait d'automatiser cette instruction supplémentaire. Heureusement, dans la plupart des cas, la communauté javascript a déjà rencontré la majorité de ces problèmes - dont l'immutabilité - et une library open source hébergé sur github répond à ce besoin : *Immutable.js* (<https://immutable-js.com/>).

Cette dernière introduit de nouveaux types d'objets, chacun ayant été conçu pour répondre à une problématique précise, par exemple List, Map ou encore Set.

Des termes qui rappellent des types de structures en Python, qui possèdent eux aussi plusieurs manières de répondre à des problématiques de structure de données, par exemple de tableaux (sans forcément parler d'immutabilité).

—

Nous pouvons faire un aparté sur Typescript; un superset de Javascript qui permet - entre autres - de rendre Javascript plus sûr en mettant en place du typage de variable, valeur de retour.

Ce typage peut inclure un type *"readonly"* qui permet par exemple de rendre une propriété en lecture seule.

De cette façon on évite la mutation de la propriété et adopte donc un comportement immuable. Ce cours étant fait seulement en Javascript, Typescript ne sera pas développé ici mais évoqué afin de créer un parallèle ou bien d'envisager une autre possibilité, ou ouverture, quant à l'application de la programmation fonctionnelle dans l'univers Javascript.

—

Dans tous les cas, que ce soit par l'utilisation de bibliothèques externes ou d'autres méthodes, en respectant les bonnes pratiques de la programmation fonctionnelle; il ne sera plus nécessaire de faire des affectations de variables mais uniquement d'utiliser la composition de fonctions (voir le chapitre associé pour plus de détails).

Cette règle est accentuée par certains langages de programmation spécialisés en programmation fonctionnelle qui vont quant à eux jusqu'à interdire toute affectation.

Cette façon de réfléchir est diamétralement opposée à celle de la programmation impérative ou en POO, de sorte que, si correctement effectuée, il n'y a plus besoin de prêter attention à l'immutabilité d'une variable.

Avant de nous pencher sur ces notions avancées, il est nécessaire de s'intéresser à un dernier pilier; la transparence référentielle.

5.2.3 - Transparence référentielle et effets de bords

Bien que le terme “transparence référentielle” semble complexe, son explication est simple à comprendre.

La raison qui nous a poussés à aborder la fonction pure en première section de chapitre est que la transparence référentielle ne peut exister sans fonctions pures, car leur absence conduit inéluctablement à l’apparition d’effets de bords.

Un effet de bord est une instruction qui a pour but de modifier un état en dehors de l’environnement dans lequel elle a été écrite.

En Javascript, cela pourrait revenir à modifier l’état d’une variable en dehors du scope de la fonction (scope = périmètre d’actions, à partir de ES6, le scope d’une fonction est déterminé par les accolades de celle-ci “{ ... }”). Il s’agirait aussi de modifier une valeur en dehors du scope sans pour autant retourner de variable.

En d’autres termes, la fonction ne serait pas seulement basée sur des paramètres contrôlés, mais sur des éléments pouvant produire un résultat inattendu.

La fonction a pour obligation de ne faire que ce pour quoi elle a été conçue.

Prenons en exemple la fonction ci-dessous, remplie d’effets de bords :

```
const destroyTheEnemy = (enemy) => {  
  const target = identifyTheTarget(enemy);  
  const oblivion = new Machine({  
    type: "battleship",  
  });  
  oblivion.lunchTheNuclearMissile(target);  
  enemy.health = 0;  
  return enemy;  
};
```

Voici ce qu'on peut en déduire :

- Pour rappel, tout ce qui n'est pas explicitement contrôlé dans la fonction peut être considéré comme un effet de bord.

Ici, la fonction *identifyTheEnemy* n'est pas passée en paramètre. Par conséquent : *destroyTheEnemy* se base sur une autre fonction externe à son scope qui peut produire des effets de bords comme l'objet *target* retourné.

Celui-ci pourrait être amené à changer de propriété, ce qui pourrait par la suite poser problème lorsque l'on passera ce même objet dans les fonctions suivantes.

- Même problème ici avec la classe Machine. Elle se situe en dehors du scope de base, ce qui peut mener aux mêmes types de conséquences, comme la propriété "type" de l'objet passé en paramètre qui pourrait être changé.
- Encore une fois *launchTheNuclearMissile* est une méthode de l'objet créée par l'intermédiaire de la classe Machine, celle-ci peut être amenée à changer, voire ne pas exister : un autre effet de bord peut poser problème dans le futur de l'évolution du projet.
- Pour finir (et ce problème concerne l'entièreté de la fonction) la totalité de la fonction répond à plusieurs problématiques. *DestroyTheEnemy* englobe en effet :
 - Viser un ennemi à partir de la cible donnée, ce qui renvoie un objet *target*.
 - A partir de la *target*, on instancie une nouvelle machine ayant pour objectif de détruire la target donnée, ce qui retourne ladite machine.
 - Enfin, la machine utilise l'arme qui lui est propre pour détruire l'ennemi.
 - Ce qui finalement affecte 0 (personne ne peut survivre à un missile nucléaire) à la vie de l'ennemi, et le détruit, puis termine la fonction en renvoyant l'ennemi avec sa vie à 0.

Finalement, il aurait fallu diviser la fonction en plusieurs sous-objectifs, retournant pour chacun des valeurs simples, et ne pas mélanger l'ensemble dans une seule fonction avec des appels externes, menant in fine à des effets de bords.

Nous savons désormais comment éviter au maximum les effets de bords.

Il était essentiel de comprendre cette notion pour comprendre la suivante : la transparence référentielle.

Il est dit, qu'une expression est référentiellement transparente lorsqu'il est possible de remplacer ladite expression par une valeur sans que cela ne change le sens et la bonne exécution du programme.

NB : Lorsqu'elle ne l'est pas, on dit que l'expression est référentiellement opaque.

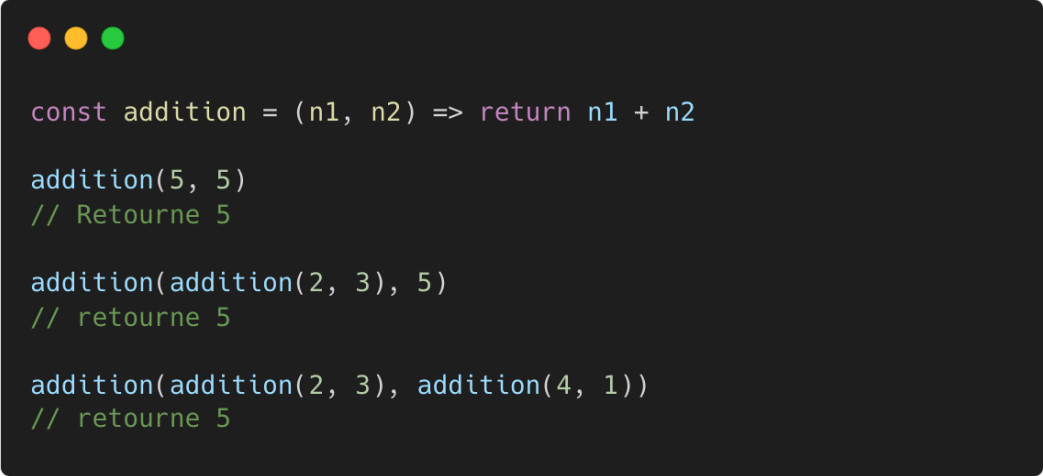
D'où l'intérêt des notions précédentes comme la définition de la pureté d'une fonction ou bien les conséquences des effets de bords décrits ci-dessus.

Car pour rendre une expression référentiellement transparente, il est nécessaire que celle-ci ne contienne pas d'effets de bords et doit impérativement être pure.

Car :

- Une fonction pure garantit la véracité de la valeur de retour. Dans un contexte comme la transparence référentielle, garantir que le résultat sera identique pour chaque appel d'un même argument donné est essentiel. Cela veut dire qu'il est possible de remplacer lesdites expressions par des valeurs. Un critère essentiel pour respecter la transparence référentielle.
- Éviter à tout prix d'inclure des effets de bords permet aussi de faciliter le remplacement des expressions par des valeurs. Ne pas inclure des processus externes au scope duquel il a été produit est un gage de qualité en ce qui concerne la transparence référentielle.

Voici un exemple d'expressions référentiellement transparentes:



```
const addition = (n1, n2) => return n1 + n2

addition(5, 5)
// Retourne 5

addition(addition(2, 3), 5)
// retourne 5

addition(addition(2, 3), addition(4, 1))
// retourne 5
```

On le remarque bien ici, avec cet exemple d'une fonction `addition`, prenant deux nombres en arguments, et retourne l'addition de ces derniers. On utilise uniquement des valeurs directes comme des nombres, ou bien une valeur et une expression, ou bien encore seulement des expressions, la fonction retournera exactement le même résultat.

On dit alors que cette fonction est référentiellement transparente.

—

La transparence référentielle présente bien des avantages. Comme par exemple le fait de pouvoir remplacer la totalité des instructions par des équations.

Elle suit logiquement la philosophie initiale de la programmation fonctionnelle, qui est de se rapprocher au plus près du même comportement des fonctions abstraites provenant des mathématiques.

Il devient alors possible de substituer ces instructions par des équations, et augmenter la lisibilité du code pour finalement faciliter sa compréhension.

Ce concept s'appelle "*equational reasoning*" et est très encouragé dans le langage Haskell.

—

Un autre avantage concerne l'optimisation, plus particulièrement la mémoïsation.

La mémoïsation consiste à mettre en cache les valeurs de retour des fonctions vis-à-vis des valeurs données en arguments. L'objectif étant de réduire le temps d'exécution du programme par la mémorisation des valeurs dans le cache.

La transparence référentielle facilite l'application de cette technique d'optimisation en sachant que toutes les expressions sont remplaçables par des valeurs. Cela veut dire que ces valeurs, si elles sont obtenues plus d'une fois peuvent être stockées dans le cache, et ainsi permettre d'économiser du temps de calcul alloué à ce processus.

Cette notion est tout autant importante que l'immutabilité ou les fonctions pures, puisqu'elle est la pierre angulaire du paradigme de la programmation fonctionnelle.

5.2.4 - Les fonctions d'ordre supérieur (HOF)

Maintenant que les concepts de bases ont été présentés, abordons la suite du cours avec des notions de plus en plus avancées.

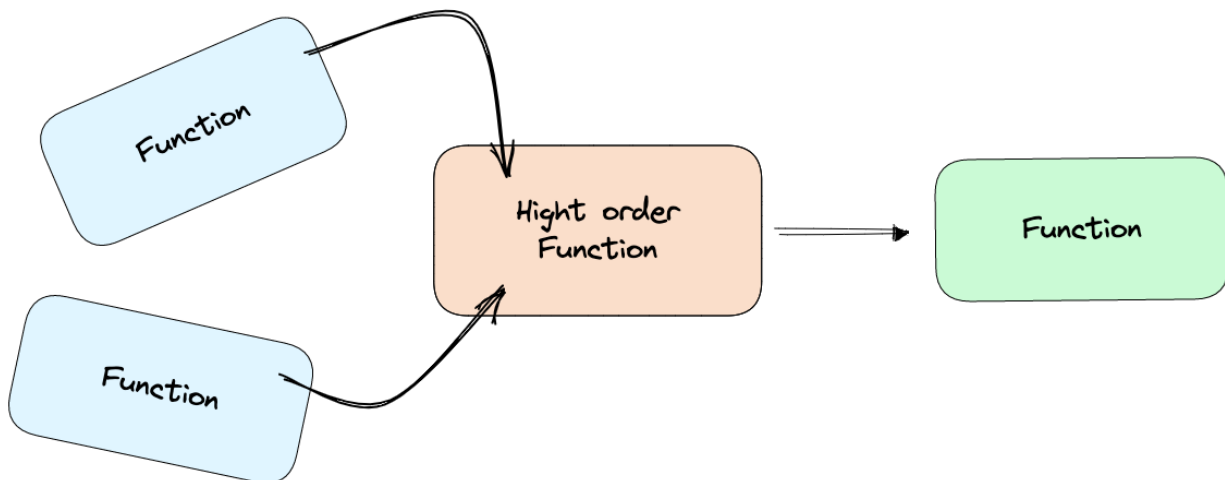
Dans certains langages de programmation, il est possible de retourner des structures de données ou fonctions directement à l'intérieur de fonctions. En langage C par exemple, ceci est impossible. En javascript, les objets sont dits "des objets de première classe"; c'est-à-dire que les objets peuvent être utilisés sans restrictions.

Ceci permet de créer des fonctions de premier ordre.

Les fonctions de premiers ordres sont définies par les caractéristiques suivantes:

- Elles doivent prendre en arguments une ou plusieurs fonctions
- Et / Ou elles doivent retourner elle même une autre fonction (non pas une expression de calcul, mais bel et bien une autre déclaration de fonction)

L'utilisation de cette technique permet de généraliser un processus, avant de l'appliquer à un processus en particulier et dans un contexte donné.



Voyons un exemple:

NB : A été choisie pour cet exemple la syntaxe classique pour déclarer des fonctions d'avant ES6, non pas avec les fonctions fléchées mais par des déclarations avec le mot clef `function`. Pour que le code soit plus lisible et pour des raisons pédagogiques.

```

function addProp(something) {
  return function (prop, value) {
    return {
      ...something,
      [prop]: value,
    };
  };
}

const sword = {
  name: "Master Sword",
  type: "One handed sword",
};

const upgradeWeapon = addProp(sword);

upgradedSword1 = upgradeWeapon("pommel", "copper");
// { name: 'Master Sword', type: 'One handed sword', pommel: 'copper' }

upgradedSword2 = upgradeWeapon("grip", "leather");
// { name: 'Master Sword', type: 'One handed sword', grip: 'leather' }

```

L'objectif ici est d'améliorer une arme.

Rappelons qu'en programmation fonctionnelle, il nous est impossible de modifier une variable vis-à-vis du concept d'immutabilité vu précédemment.

Nous pourrions très bien créer une simple fonction à la place qui se contenterait de rajouter ladite propriété et la retournerai.

Ce serait correct, mais redondant.

Utiliser une fonction de premier ordre dans ce contexte améliorerait la lisibilité du code

A la place, nous pouvons tout d'abord écrire une fonction générale (*addProp*) qui aura pour unique but d'ajouter une propriété à n'importe quel objet (*something*) passé en argument. Puis lorsqu'on appellera *addProp*, on donnera l'arme (*sword*) à la nouvelle fonction créée

(*upgradeWeapon*).

De cette façon, *upgrade weapon* aura forcément gardé notre épée en mémoire grâce à la mémorisation. Il suffit alors d'appeler cette dernière pour améliorer l'épée, qui va la retourner. Il ne reste plus qu'à faire quelque chose avec la valeur retournée.

Voici ce que donnerait la fonction avec la norme ES2015 / ES6 :

```
const addProp = (something) => (prop, value) => ({
  ...something,
  [prop]: value,
});

const sword = {
  name: "Master Sword",
  type: "One handed sword",
};

const upgradeWeapon = addProp(sword);

upgradedSword1 = upgradeWeapon("pommel", "copper");
// { name: 'Master Sword', type: 'One handed sword', pommel: 'copper' }

upgradedSword2 = upgradeWeapon("grip", "leather");
// { name: 'Master Sword', type: 'One handed sword', grip: 'leather' }
```

Grâce aux fonctions de premiers ordres, il est donc possible de créer des fonctions génériques et réutilisables.

Pour reprendre l'exemple ci-dessus, il aurait été possible de créer n'importe quelle fonction basée sur n'importe quel objet donné en paramètre. Que ce soit une arme, une voiture ou autre chose, il est possible de créer une fonction spécialisée, qui prendra en compte les paramètres qu'on lui donne, et produira une fonction qui répondra à un besoin très spécifique.

C'est aussi grâce à ce procédé qu'il est possible de diviser le code base d'un projet :

Créer des fonctions de premier ordre facilite autant la lisibilité du code que sa compréhension grâce à sa capacité à diviser intelligemment le code.

NB : Nous verrons une version d'une HOF prenant en paramètre une fonction plus tard dans le

cours

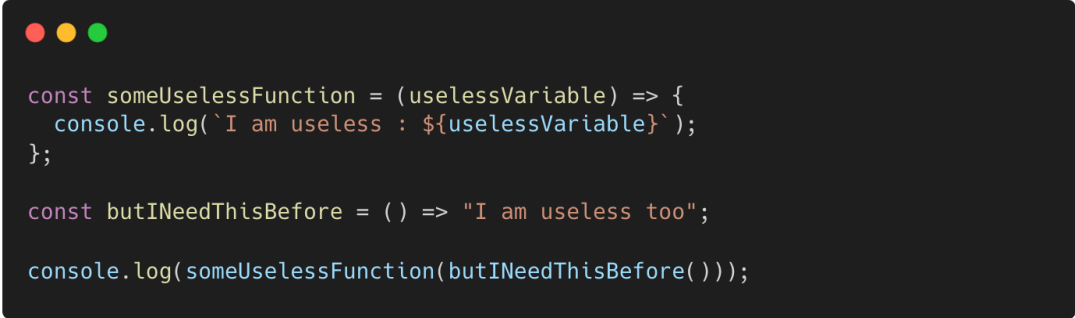
Cas pratique : Recoder la fonction `Array.map()` (vidéo)

Recoder la fonction `Array.map()` pour montrer que l'on utilise des procédés issus du fonctionnel sans s'en rendre compte

5.2.5 - La composition de fonction

Faisons un premier pas dans la notion avancée de la programmation fonctionnelle.

Nous utilisons déjà la composition de fonction dans d'autres styles de programmation, ne serait-ce que pour des fonctions simples comme les fonctions de débuggages :



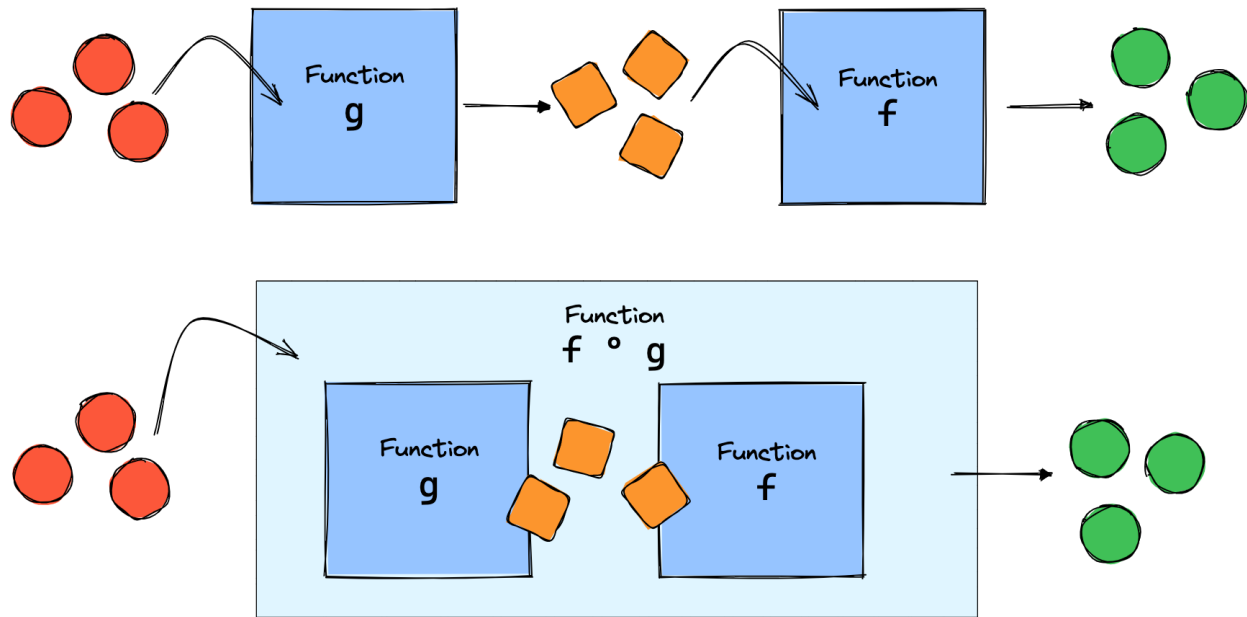
```
const someUselessFunction = (uselessVariable) => {  
  console.log(`I am useless : ${uselessVariable}`);  
};  
  
const butINeedThisBefore = () => "I am useless too";  
  
console.log(someUselessFunction(butINeedThisBefore()));
```

Ici, il est nécessaire d'obtenir le résultat de *“butINeedThisBefore”* pour être utilisé dans *“someUselessFunction”* afin d'afficher la valeur de retour dans *console.log* pour faire apparaître le tout dans la console de sortie.

C'est la même mécanique d'enchaînement de fonctions qui est utilisée dans la composition de fonctions. Sans s'en rendre compte, la composition de fonction fait déjà partie de notre quotidien de développeur.

Elle s'inspire directement de la composition de fonction des mathématiques :

Le résultat d'une fonction est utilisé en tant que paramètre d'une autre fonction. C'est donc un processus qui, à partir de deux fonctions, en construit une troisième.



NB : le symbole en forme de cercle "°" sur l'image est la notation mathématique pour définir une composition de fonction. Cette dernière se lit "f rond g"

En programmation, le principe reste identique, bien que le vocabulaire employé diffère.

—

Comment la programmation fonctionnelle fait-elle alors pour faire évoluer cette notion à un autre niveau ?

Voici un exemple plus parlant :

```
const removeFinalPoint = (sentence) => sentence.substring(0,
sentence.length - 1);
const makeSentenceOMG = (sentence) => `${sentence} OMG`;
const withAnger = (sentence) => `${sentence} !`;

const sentence = "Functional programming is awesome.";

withAnger(makeSentenceOMG(removeFinalPoint(sentence)));
// Retourne "Functional programming is awesome OMG !"
```

Cela revient à appliquer le processus de deux fonctions l'une après l'autre, mais sur une même donnée. Ce procédé peut s'appliquer avec plusieurs fonctions et pas nécessairement deux comme indiqué ci-dessus.

On en revient à créer une fonction composée de plusieurs fonctions.

Voici un dernier exemple expliquant le principe :

```
const crayons = ["rouge", "bleu", "vert"];

function choisirCrayon(numeroCrayon) {
  return crayons[numeroCrayon - 1];
}

function dessinerAvecCrayon(couleurCrayon) {
  console.log(`Dessine avec un crayon de couleur ${couleurCrayon}`);
}

const choisirEtDessiner = (numeroCrayon) => {
  const couleur = choisirCrayon(numeroCrayon);
  dessinerAvecCrayon(couleur);
};

console.log(choisirEtDessiner(1));
console.log(choisirEtDessiner(3));
```

Imaginons que nous avons une boîte de crayons de couleur. Chaque crayon a une couleur différente, comme rouge, bleu ou vert.

La première fonction, appelée "choisirCrayon", prend en entrée le numéro d'un crayon (par exemple, 1 pour le rouge, 2 pour le bleu, etc.) et renvoie la couleur de ce crayon.

La deuxième fonction, appelée "dessinerAvecCrayon", prend en entrée une couleur de crayon (par exemple, "rouge") et dessine quelque chose de cette couleur.

La fonction composée "choisirEtDessiner" est créée en combinant les deux fonctions précédentes. Elle prend en entrée le numéro d'un crayon, choisit la couleur correspondante à l'aide de la fonction "choisirCrayon", puis utilise cette couleur pour dessiner quelque chose avec la fonction "dessinerAvecCrayon".

C'est comme utiliser une boîte de crayon pour choisir une couleur et dessiner avec.

NB : En l'occurrence, on constate des effets de bords dans la fonction “choisirEtDessiner”, les fonctions sur lesquelles elle se base sont situées en dehors de son scope. La composition de fonction est une technique comme une autre, cela ne veut pas dire pour autant qu'elle ne peut pas contenir d'effets de bords ou tout autre effet néfaste lié à la programmation fonctionnelle.

5.2.6 - La Curryfication (ou Currying)

On préférera utiliser directement l'anglicisme *currying*, *curry*, *curried* dans ce cours pour des raisons pratiques.

On commence ici à entrer dans la partie la plus complexe. Les 3 notions suivantes - incluant celle-ci - sont parmi les plus difficiles à appliquer dans un contexte comme la programmation fonctionnelle.

Pourtant, fondamentalement, le *currying* ne paraît pas être aussi complexe que son terme peut le laisser penser.

Le *currying* consiste à transformer un nombre x de paramètres de fonctions en un enchaînement de fonctions traitant un seul paramètre à la fois.

Il transforme une fonction à plusieurs arguments en une série de fonctions à un seul argument. Elles permettent entre autres de créer des fonctions plus flexibles et réutilisables.

Par exemple, on peut voir leur utilité dans le cas où il serait utile d'appliquer partiellement le processus d'une fonction : l'exécuter partiellement et non totalement.

Imaginons que l'on doit passer un certain nombre de paramètres d'une fonction.

Admettons que pour des besoins quelconque, nous décidons de ne pas passer strictement tous les paramètres de ladite fonction.

La fonction *curried* va donc renvoyer une fonction partielle, qui pourra être exécutée plus tard avec le ou les paramètres manquants et produira ainsi l'effet final.

Très utile quand on ne connaît pas à un moment T tous les paramètres à passer à l'avance dans une fonction.

Voici comment on pourrait théoriquement l'utiliser :

```
// Fonction de base, retourne la somme de 3 nombres
const sum3 = (num1, num2, num3) => num1 + num2 + num3;

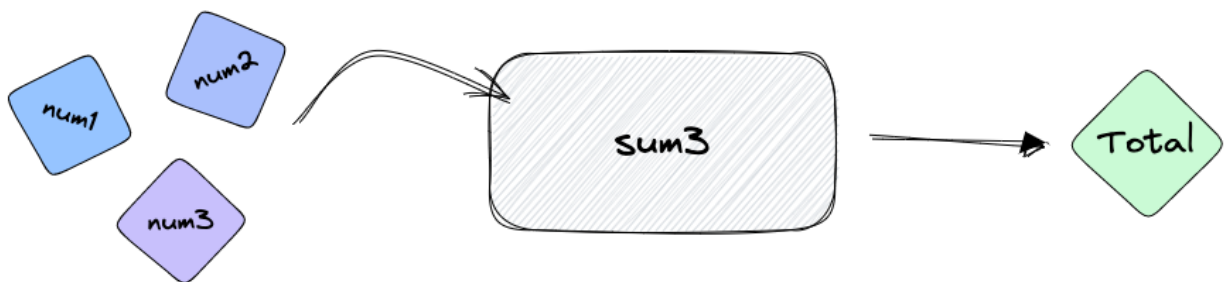
// Avec la currying, on transforme sum3 comme suit :
// En imaginant que "curry" est une fonction qui prend en paramètre une
// fonction et qui retourne une fonction curried
curry(sum3)(num1)(num2)(num3);
```

Voici ce qu'il se passe :

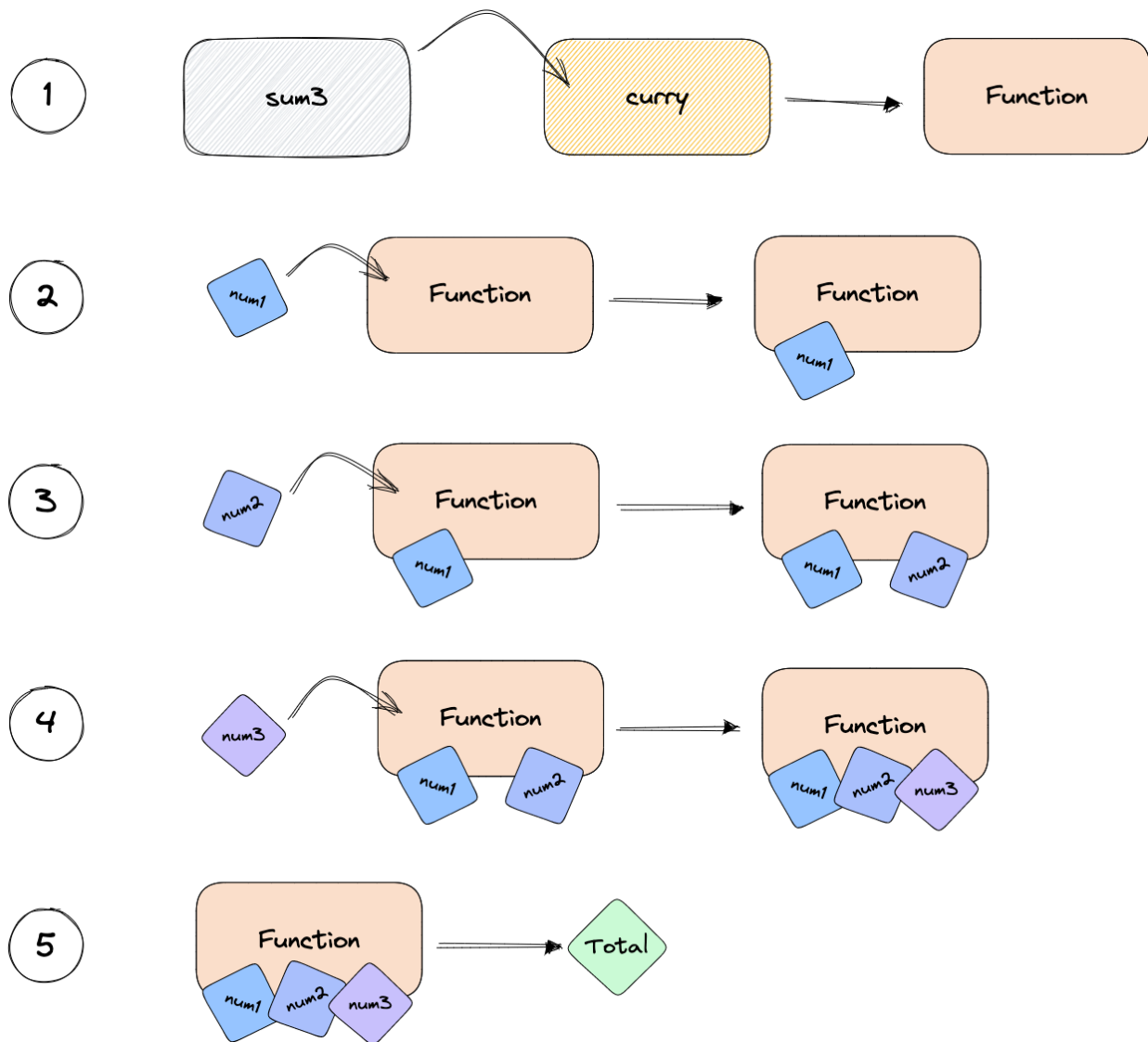
1. "Curry(add)" renvoie une fonction de façon à pouvoir être exécutée avec le paramètre "num1" tel que "Curry(add)(num1)"
2. "Curry(add)(num1)" renvoie une fonction de façon à pouvoir être exécutée avec le paramètre "num2" tel que "Curry(add)(num1)(num2)"
3. "Curry(add)(num1)(num2)" renvoie une fonction de façon à pouvoir être exécutée avec le paramètre "num3" tel que "Curry(add)(num1)(num2)(num3)"

Il pourrait y avoir autant de points que de paramètres.

Autrement dit, avec une façons plus visuel on passe d'un appel de fonction classique :



A une version plus segmentée, et permettant plus de flexibilité pour un même résultat :



On peut d'ores et déjà le remarquer avec ce schéma : la fonction "curry" accepte en paramètre une fonction et en résulte une autre. La fonction curry est donc une HOF

Voici un exemple simplifié de comment serait codé "curry" pour cet exemple précis :

```

const sum3 = (num1, num2, num3) => num1 + num2 + num3;

// Fonction curryfiée (simplifié, seulement 3 paramètres)
const curry = (fn) => {
  return function (num1) {
    return function (num2) {
      return function (num3) {
        return fn(num1, num2, num3);
      };
    };
  };
};

// Fonction "sum3" curryfiée
const curriedSum3 = curry(sum3);

console.log("curriedSum3", curriedSum3(1)(2)(3));
// Sortie : curriedAddVersion 6

```

On remarque bien ici les retours de fonctions successifs qui permettent la curryfication. La fonction *curried* "fn" a ainsi accès aux variables num1, num2 et num3 dans son scope.

Après avoir abordé le terme de "flexibilité" plus haut, on peut se permettre de creuser plus loin.

Dans une application partielle où beaucoup d'abstractions sont requises, une situation où l'on doit récupérer un paramètre après un certain nombre d'étapes préliminaires peut arriver.

Le currying permet de mâcher le travail et effectuant les tâches qu'elle est capable d'accomplir avant d'accomplir son ultime tâche (en l'occurrence ici, la somme des 3 nombres) :

Voici un autre exemple dans la même situation, avec la démonstration de la flexibilité que permet le currying :

```

const sum3 = (num1, num2, num3) => num1 + num2 + num3;

const curry = (fn) => {
  return function (num1) {
    return function (num2) {
      return function (num3) {
        return fn(num1, num2, num3);
      };
    };
  };
};

const curriedSum3 = curry(sum3);

const notTheResultButAlmost = curriedSum3(1)(2);
console.log("notTheResultButAlmost", notTheResultButAlmost);

// Make some process... and get the final parameter anyway
const finalparam = 3;

const result = notTheResultButAlmost(finalparam);
console.log("result", result);
// Sortie : curriedAddVersion 6

```

Ici “*notTheResultButAlmost*” ne retourne pas encore 6, car il manque un paramètre pour que la fonction de base soit exécutée.

Cependant elle retourne une fonction à laquelle ajouter un ultime paramètre afin d’accomplir la tâche initiale de la fonction.

L’écrire avec du code pré ES6 est plus simple à comprendre, mais il est évidemment plus facile d’écrire ce genre de fonction en une ligne avec une fonction fléchée :

```

// Fonction curryfiée (simplifié, seulement 3 paramètres)
const curryES6 = (fn) => (num1) => (num2) => (num3) => fn(num1, num2, num3);

```

Voyons maintenant comment coder la vraie fonction curry dans un cas pratique en vidéo

Cas pratique : Application du currying (vidéo)

La vraie fonction curry (recursive) et un cas pratique

5.2.7 - Le « Point Free style » (ou Tacit Programming)

Notion simple à comprendre, qui une fois poussée à l'extrême avec de multiples refactorisations de code s'avère être un réel avantage lorsque la lisibilité du code est une priorité dans un projet.

Voici un exemple en React.js. Un composant avec un formulaire et un bouton qui, une fois cliqué, envoie le formulaire. Un exemple commun de la vie de tous les jours dans le développement frontend :

```
1 const Component = () => {
2   const handleSubmit = (e) => {
3     e.preventDefault();
4     console.log("You clicked submit.");
5   };
6
7   return (
8     <form onSubmit={handleSubmit}>
9       <button type="submit">Submit</button>
10    </form>
11  );
12 };
13
14 export default Component;
```

Regardons de plus près la fonction “handleSubmit” déclarée ligne 2 et la façon dont elle est appelée ligne 8.

La fonction “handleSubmit” attend de recevoir un paramètre “e”, soit l'événement réceptionné après le clic. Or, quand on regarde comment est appelée la fonction, aucun paramètre n'est passé lors de l'appel de fonction.


Certes, React.js s'occupe de toute la partie concernant l'événement. En dehors de ça, aucun argument n'a été donné. Et pourtant, “e” est bel et bien reçu et exploité dans la fonction. C'est grâce au Tacit Programming, où plus communément appelé Point-Free style.

Le point freestyle est une technique en programmation fonctionnelle. Elle consiste à créer une fonction, la déclarer, sans pour autant définir les arguments de celle-ci de façon explicite.

Elle a pour avantage de réduire le nombre de lignes de codes et améliore en plus l'expérience de

développement en proposant une syntaxe plus lisible, concise et fiable.

Un premier exemple qu'on utilise au quotidien est lorsqu'on implémente des boucles, que ce soit avec "Array.forEach()" ou bien "Array.map()"



```
1 const sentence = ["creative", "development", "is", "awesome"];
2
3 const pronouncedAsAsmathic = (word) => `${word}. `;
4
5 const asmathicSentence = sentence.map(pronouncedAsAsmathic);
6
7 console.log(asmathicSentence);
8 // Sortie : [ 'creative. ', 'development. ', 'is. ', 'awesome. ' ]
```

Sur l'exemple ci-dessus, on remarque le même procédé que l'exemple en react

Au lieu de directement passer une callback dans la méthode ".map()", on extrait le processus qui sera effectué à chaque boucle en une fonction externe.

Cette fonction s'attend bel et bien à recevoir un paramètre "word" mais n'est jamais passé en tant qu'argument de fonction lors de son appel dans chaque itération de "map()".

5.2.8 - Les monades

Dernière notion abordée dans ce cours :

Commençons par une petite citation de Douglas Crockford, créateur du format JSON et développeur d'ESLint :

“Les monades sont étonnantes. Ce sont des choses simples, presque triviales à mettre en œuvre, avec un énorme pouvoir de gestion de la complexité.

Mais il est étonnamment difficile de les comprendre et la plupart des gens, une fois qu'ils ont eu ce moment 'ah-ha', semblent perdre leur capacité à les expliquer aux autres.”

Cette dernière annonce bien la couleur. Elle donne un avant goût de la complexité de la tâche qui s'annonce.

Commençons :

Qu'est-ce qu'une monade ? Comment l'expliquer simplement afin d'imager ce qu'elle est et d'expliquer les besoins auxquels elle répond ?

On peut voir une monade (une structure monadique) comme un nouveau type d'objet, au même titre que d'autres plus connues comme les tableaux, les objets.

Elle va aussi venir englober un ensemble de fonctionnalités obéissant à des règles précises relatives à son fonctionnement. En ce sens, elle peut aussi être considérée comme une variante d'un design pattern.

Chaque type de données répond à un ou plusieurs besoins et possède sa propre spécificité.

Par exemple, lorsque l'on souhaite classer des données dans une structure non ordonnée : Il est fort probable que l'on s'oriente vers l'utilisation d'un tableau. Si l'on souhaite en revanche créer une structure ayant comme objectif de représenter une entité avec des propriétés bien définies, il sera plus judicieux de se diriger vers un objet

NB : Objet Instancié à partir d'une classe ou non. En javascript, il est courant de déclarer un objet dit “on the fly” signifiant qu'il ne se base pas sur une classe existante. Javascript n'a pas de classe à proprement parler mais se repose en revanche sur un système d'enchaînement prototype. Ces derniers sont utilisés à l'aide de l'instruction “class” qui est en réalité un sucre syntaxique permettant d'avoir une structure ressemblant aux classes en provenance du

paradigme orienté objet.

En ce qui concerne les monades, c'est le même processus : chaque type de monade se raccroche à un but auquel il doit répondre. Et tout comme les design pattern, une monade peut répondre à un problème commun à l'aide d'une solution commune

Les structures monadiques sont classées par type, par rapport aux besoins auxquels elles répondent. La plus connue étant la monade Identity (Identity Functor) de part sa "simplicité" vis-à-vis de ses autres congénères plus complexes.

Une monade englobe une multitude de méthodes / fonctions, chacune obéissant à une règle spécifique.

Ces règles sont établies sur la base des lois monadiques.

Ces axiomes (règles) sont le principal frein à la compréhension des monades. Ainsi chaque loi sera traitée de façon à pouvoir comprendre le but des lois monadiques. S'en suivra après son application avec la création d'une monade en Javascript.

A - Loi de l'Identité à gauche

Identité à Gauche (Left Identity), appelé aussi loi de l'unité.

Cette loi stipule que, lorsqu'on ajoute une valeur à une structure de données monadique en utilisant la fonction "unit", le résultat doit être équivalent à la structure de données monadique initiale.

En notation pseudo mathématique : $\text{unit}(x).\text{flatMap}(f) = f(x)$ pour toute fonction f .

Autrement dit, cette loi prend la forme d'une fonction permettant d'attribuer une autre fonction à une valeur et de retourner le résultat de cette autre fonction. Souvent appelé "fold" même si aucune nomenclature n'est imposé en terme de nommage des fonctions dans les structures monadiques.

B - Loi de l'Identité à droite

La loi de l'Identité à Droite (Right Identity) est similaire à la loi de l'identité à gauche.

Cette loi décrit la façon dont les opérations liées à la structure de données monadique sont liées entre elles. Elle stipule que la sortie de la première opération doit être utilisée comme entrée pour la suivante.

En notation pseudo mathématique $m.\text{flatMap}(f).\text{flatMap}(g) = m.\text{flatMap}(x \Rightarrow f(x).\text{flatMap}(g))$ pour toutes les fonctions f et g .

Autrement dit, cette loi prend elle aussi la forme d'une fonction permettant d'attribuer une autre fonction à une valeur et de retourner le résultat de cette autre fonction.

A ceci près que ce dernier (le résultat) sera encapsulé dans son conteneur (la monade)

Cette loi garantit que les traitements peuvent s'enchaîner à la suite, et ce de façon linéaire. Cette méthode d'enchaînement de processus est appelée "chaining". Par abus de langage / anglicisme, on dit que l'on peut "chainer" les traitements.

Tout comme la loi précédente, la fonction liée à la loi de l'identité à droite est souvent appelée "map".

C - Loi de l'Associativité

La loi d'Associativité (Associativity).

Cette loi décrit la façon dont les opérations liées à la structure de données monadique sont combinées. Elle stipule que le résultat ne dépend pas de la façon dont les opérations sont agencées.

Cette loi est intimement liée à la composition de fonctions vu précédemment.

En grossissant le trait : le postulat est identique à celui des règles opératoires en mathématiques.

En effet, les additions de même que les multiplications peuvent s'effectuer dans n'importe quel ordre, chacun des éléments ou facteurs est interchangeable, le résultat sera toujours identique.

Soient x_1 , x_2 , x_3 des nombres quelconques, et $rAdd$ la somme des ces nombres et $rMul$ le produits de ces nombres :

$$x_1 + x_2 + x_3 = rAdd$$

$$\Leftrightarrow x_2 + x_1 + x_3 = rAdd$$

$$\Leftrightarrow x_3 + x_1 + x_2 = rAdd$$

$$\Leftrightarrow (x_3 + x_1) + x_2 = rAdd$$

$$\Leftrightarrow x_3 + (x_1 + x_2) = rAdd$$

$$x_1 * x_2 * x_3 = rMul$$

$$\Leftrightarrow x_2 * x_1 * x_3 = rMul$$

$$\Leftrightarrow x_3 * x_1 * x_2 = rMul$$

$$\Leftrightarrow (x_3 * x_1) * x_2 = rMul$$

$$\Leftrightarrow x_3 * (x_1 * x_2) = rMul$$

En appliquant cette loi non pas à des variables, mais à des fonctions on obtient l'équation suivante :

Soient f , g , et h , des fonctions :

$$(f \circ g) \circ h = f \circ (g \circ h)$$

Cette syntaxe est celle de la composition de fonction.

En reprenant les exemples précédents (additions) combinés aux fonctions, où chaque fonction f, g et h serait construite de la façon suivante :

```
const addX = (num) => num + {N'importe quel nombre X};
```

On pourrait alors dire que

NB : Pour une question de lisibilité et de praticité, on considère que la fonction “compose()” est issue de Ramda - une bibliothèque d'utilitaire pour la programmation fonctionnelle - car elle permet de gérer dynamiquement l'arité de la fonction.

```
import { compose } from "ramda";

// addX étant toujours une fonction retournant la somme d'un nombre avec X
// Hypothèse : compose(addX, addY, addZ) === compose(addZ, addX, addY)

// Exemple concret

const add10 = (num) => num + 10;
const add7 = (num) => num + 7;
const add42 = (num) => num + 42;

const addComp1 = compose(add10, add7, add42);
const addComp2 = compose(add7, add42, add10);
const addComp3 = compose(add42, add10, add7);

// Les additions respectent la loi d'associativité, donc :

console.log(addComp1(10), addComp2(10), addComp3(10)); // 69 69 69
console.log(addComp1(10) === addComp2(10), addComp3(10) === addComp2(10), addComp1(10) ===
addComp3(10)); // true
```

En notation pseudo mathématique : “m.flatMap(f).flatMap(g) = m.flatMap(x => f(x).flatMap(g))” pour toutes les fonctions f et g.

—

Récapitulons : Une monade...

- Peut être vu comme un nouveau type de données
- Peut se comporter comme une variante d'un design pattern
- Doit respecter les 3 lois monadiques

Mis à part répondre à un problème de lisibilité en rendant le code produit beaucoup plus clair et concis, il n'a pas encore été traité dans quel but les monades sont employés, et pourquoi.

Avant toute chose, il est nécessaire de pratiquer un minimum et de voir concrètement comment est fabriquée une monade.

D - Identity monad

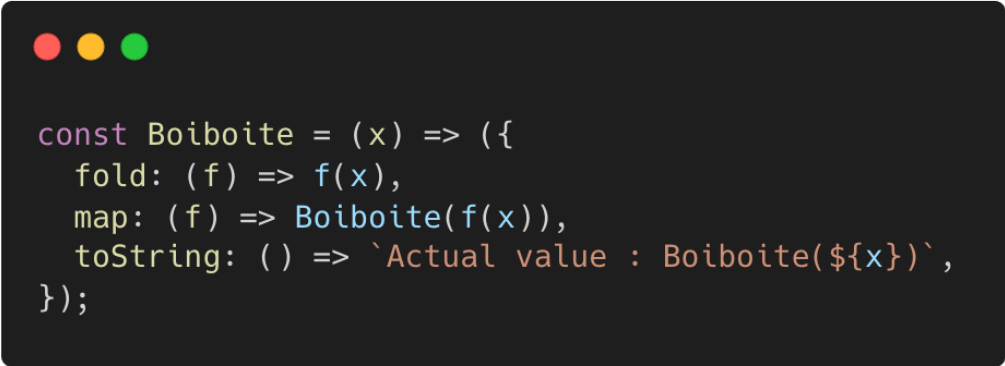
La monade Identité sert de cas d'école puisqu'elle est la plus simple à implémenter.

La particularité de celle-ci est qu'elle va servir de conteneur pour une valeur sans pour autant effectuer aucune opération sur cette dernière.

Elle est utilisée dans les situations où il est nécessaire d'ajouter une logique supplémentaire à une pipeline de transformation sans changer la valeur en elle-même. Par exemple suivre l'historique de transformation d'une valeur.

C'est l'exemple sur lequel sera construite la prochaine monade. Elle se basera sur les démonstrations ci-dessus, notamment le chaining des fonctions `add`.

La voici :



```
const Boiboite = (x) => ({
  fold: (f) => f(x),
  map: (f) => Boiboite(f(x)),
  toString: () => `Actual value : Boiboite(${x})`,
});
```

Analysons maintenant cette monade Identité répondant au doux nom de “Boiboite”.

Il était question de conteneur lorsqu'il s'agissait de définir une monade. C'est aussi vrai dans cet exemple et il apparaît bel et bien ici.

“Boiboite” agit comme un conteneur et possède tout un attirail lui permettant d'interagir avec son unique valeur `x` (qui ici sera toujours implémenté et pensé pour être instancié avec un nombre)

C'est à travers les propriétés du conteneur qu'il sera possible d'interférer avec x. Ces propriétés sont définies par le prisme des fonctions.

L'une d'entre-elles est en réalité un foncteur (functor), une notion issue de la théorie des catégories en mathématiques : la fonction "map"

"map" - à l'instar d'un foncteur - à pour objectif de recevoir une fonction en argument, de transformer la valeur contenue dans la monade avant de retourner une nouvelle monade avec un contexte différent puisque la valeur contenue dans la monade ne sera plus la même.. mais sera toujours une monade

NB : On peut voir le contexte comme un constructeur de type abstrait (comme une monade) qui peut encapsuler une certaine logique et des données pour fournir un moyen de manipuler les valeurs contenues à l'aide de fonctions.

En outre et en reprenant le paraphrase ci-dessus, lorsqu'il s'agit de "retourner une nouvelle monade avec un contexte différent" le contexte fait allusion à la donnée qui a changé et qui ne sera plus la même qu'à l'instanciation de la monade. Il sera plus clair de comprendre le contexte lorsque l'on se servira de cette monades dans les exemples ci-dessous.

Alors que "fold" s'attend à recevoir une fonction en argument et retourne la transformation de la valeur. La différence avec "map" est que "fold" retourne uniquement une valeur et perd son contexte dans le processus (donc la monade "Boiboite" dans cet exemple).

C'est pour cette raison que "fold" est parfois appelé "extract" vis-à-vis de sa fonction d'extraction qui parle d'elle-même. Elle est donc voué à être utilisée en tant qu'ultime instruction pour sortir la valeur de son contexte.

Enfin, puisque la fonction "map" retourne une autre "Boiboite", cela signifie qu'il est à nouveau possible d'utiliser les fonctions disponibles dans "Boiboite". Il devient alors possible d'appliquer le chaining de fonctions évoqué précédemment.

Ainsi, tant qu'il subsistera un contexte, il sera toujours possible d'appliquer une des fonctions contenues dans la monade. Le tout en respectant la loi d'associativité.

Regardons les cas d'application pour démontrer le chaining de "Boiboite"

NB : "toString" sert ici de fonction de log pour plus de praticité est n'est pas nécessaire à implémenter dans le cadre d'une monade.

```

1 // Même exemples que tout à l'heure
2
3 const add10 = (num) => num + 10;
4 const add7 = (num) => num + 7;
5 const add42 = (num) => num + 42;
6
7 const Boiboite = (x) => ({
8   fold: (f) => f(x),
9   map: (f) => Boiboite(f(x)),
10  toString: () => `Actual value : Boiboite(${x})`,
11 });
12
13 // Exemples d'applications
14
15 const example1 = Boiboite(10).map(add10).map(add7).map(add42);
16 console.log(example1);
17 console.log(example1.toString());
18
19 const example2 = Boiboite(10).map(add10).map(add7).fold(add42);
20 console.log(example2);
21 console.log(example2.map(add42)); // Error

```

Ici, nous reprenons les mêmes fonctions “add10”, “add7” et “add42”

Voici le résultat affiché ligne 16:

```

➤ FYC-ressources git:(main) x node "/Users/rpierucci/Projects
{
  fold: [Function: fold],
  map: [Function: map],
  toString: [Function: toString]
}

```

Ici on peut constater que la valeur retournée est une nouvelle monade. C’est une autre “Boiboite” avec cette fois un contexte différent. On ne peut pas le voir sur cet affichage car la valeur n’est pas définie dans la structure de la monade mais en paramètre de fonction.

Ce qu’il est possible de voir et de constater à la ligne suivante, ligne 17 :

```

Actual value : Boiboite(69)

```

On peut alors en déduire que la monade “example1” à la fin de son traitement renvoie toujours une monade avec un contexte différent. C’est avec ce type de structure monadique qu’il est possible de retracer l’historique et l’évolution d’une valeur. Que ce soit avec les appels successifs ou bien via une autre variable contenue dans la “Boiboite” qui aurait pu servir de fonction d’historique.

En revanche, lorsque l’on constate “example2”, on obtient une erreur, et c’est bien normal.

En effet, la ligne 19 ressemble à ceci près à “example1” sauf que le dernier processus est réalisé à partir de la fonction d’extraction “fold”. Il en résulte non pas une monade mais sa valeur extraite de son contexte. Il est donc impossible d’appliquer une autre fonction issue de “Boiboite”.

Sur les logs ci-dessous, on constate alors qu’une fonction “map” n’existe sur “example2” car la valeur x évaluée à 69 a été extraite de son contexte.

```
➤ FYC-ressources git:(main) x node "/Users/rpierucci/Projects/ESGI/FYC-ressources/ressources/ch5-fonctionnal-programming-in-details/8-monads/identityMonad.js"
69
file:///Users/rpierucci/Projects/ESGI/FYC-ressources/ressources/ch5-fonctionnal-programming-in-details/8-monads/identityMonad.js:21
console.log(example2.map(add42)); // Error
                      ^
TypeError: example2.map is not a function
    at file:///Users/rpierucci/Projects/ESGI/FYC-ressources/ressources/ch5-fonctionnal-programming-in-details/8-monads/identityMonad.js:21:22
    at ModuleJob.run (node:internal/modules/esm/module_job:194:25)
```

Ce dernier cas clôture cette première monade.

E - D'autres monades...

Bien évidemment cette dernière n'est pas un cas vraiment utilisable dans un projet au quotidien. Ce n'était pas le but. Cette démonstration avait pour but d'expliquer les monades et comment elles peuvent être appliquées au quotidien.

Les structures monadiques de par leur nature mathématique très prononcée peuvent faire l'objet d'un cours à part entière.

Ce cours a pour but d'introduire les notions de la programmation fonctionnelle en javascript. Une multitude d'ouvrages existent sur le sujet et sont rédigés sur des centaines de pages pour expliquer quelques concepts en détails.

Il existe cependant de nombreux types de monades nettement plus compliqués que "Boiboite". Comme évoqué au début de cette section. Elles peuvent prendre diverses formes, pour d'autres buts et peuvent être appliquées de différentes façons. Il n'existe pas de règles taillées dans le marbre en ce qui concerne la syntaxe des monades. Du moment qu'elles respectent la théorie des catégories ainsi que les lois monadiques. Alors ladite monade est correcte.

En voici quelques exemples :

Maybe Monad :

```
const Just = (value) => ({
  map: (fn) => Maybe(fn(value)),
  chain: (fn) => fn(value),
  of: () => value,
  getOr: () => value,
  filter: (fn) => (fn(value) ? Just(value) : Nothing()),
  type: "just",
});

const Nothing = () => ({
  map: (fn) => Nothing(),
  chain: (fn) => fn(),
  of: () => Nothing(),
  getOr: (substitute) => substitute,
  filter: () => Nothing(),
  type: "nothing",
});

const Maybe = (value) => (value === null || value === undefined || value.type === "nothing" ? Nothing() : Just(value));
```

Cette monade sert à gérer les valeurs nulles ou indéfinies en les enveloppant dans une structure.

La structure "Maybe" peut être soit "Just", qui contient une valeur valide, ou "Nothing", qui représente une valeur nulle ou indéfinie. Les méthodes telles que map, chain, filter, of, et getOr permettent de travailler avec les valeurs "Maybe" de manière fonctionnelle sans avoir à gérer explicitement les valeurs nulles.

Par exemple, au lieu de vérifier si une valeur est nulle avant de la manipuler, il est possible d'utiliser "map" pour manipuler la valeur, sachant que si elle est nulle, la fonction ne sera pas exécutée.

Et voici le genre de monad qu'il est possible d'obtenir lorsque l'on pousse le concept un peu plus loin

Task Monad :

```
const Task = (fork) => ({
  fork,
  ap: (other) => Task((rej, res) => fork(rej, (f) => other.fork(rej, (x) => res(f(x))))),
  map: (f) => Task((rej, res) => fork(rej, (x) => res(f(x)))),
  chain: (f) => Task((rej, res) => fork(rej, (x) => f(x).fork(rej, res))),
  concat: (other) =>
    Task((rej, res) =>
      fork(rej, (x) =>
        other.fork(rej, (y) => {
          console.log("X", x, "Y", y);
          res(x.concat(y));
        })
      )
    ),
  fold: (f, g) =>
    Task((rej, res) =>
      fork(
        (x) => f(x).fork(rej, res),
        (x) => g(x).fork(rej, res)
      )
    ),
});
```

L'explication de ce code nécessite un peu plus d'effort puisqu'il faut un grand niveau d'abstraction pour comprendre son utilisation :

Cette monade sert à gérer des opérations asynchrones comme les requêtes réseaux ou bien encore la lecture / écriture de fichiers en les enveloppant dans une structure.

Les méthodes "ap", "map", "chain", "concat", et "fold" permettent de combiner et de transformer les tâches asynchrones de différentes manières.

Par exemple, "chain" peut être utilisée pour enchaîner des tâches, tandis que "map" peut être utilisée pour transformer les résultats. "concat" peut être utilisée pour combiner plusieurs résultats.

—

Généralement, pour chaque problème récurrent existe une solution applicable à l'aide de monade. Celle-ci peut s'avérer compliquée à comprendre et concevoir dans un premier temps. Mais à l'instar de la programmation fonctionnelle, lorsque celle-ci est maîtrisée, le code produit devient alors plus court, précis et lisible.

5.2.9 - En résumé

La programmation fonctionnelle peut paraître plus difficile à concevoir de part sa quasi-obligation de connaître certaines notions mathématiques.

C'est le cas avec les monades. Elles se basent sur la théorie des catégories.

La théorie des catégories est un domaine mathématique qui étudie les structures algébriques abstraites telles que les objets et les transformations (appelées morphismes).

Ce sont des ensembles d'objets et de flèches qui représentent les transformations entre objets.

Les foncteurs qui en découlent sont des structures mathématiques qui représentent une transformation entre deux catégories en associant les objets et morphismes d'une catégorie à ceux d'une autre catégorie.

La théorie des catégories est utilisée dans divers domaines mathématiques et en programmation pour étudier les propriétés des objets et des transformations dans un cadre formel cohérent.

Les monades en programmation fonctionnelle sont similaires aux catégories et permettent de représenter des opérations à effet de bord de manière pure et contrôlée.

Acquérir ces notions à la fois pratiques (en javascript) et théorique (mathématique, pseudo code, concepts) feront de vous de meilleurs développeurs en ce qui concerne la qualité du code produit.

Conclusion

En conclusion, le paradigme de programmation fonctionnelle est un sujet fascinant qui a considérablement évolué au fil des années. Il offre une approche différente et efficace pour résoudre les problèmes de programmation en utilisant des fonctions pures et en évitant les effets de bord.

De plus en plus de développeurs se tournent vers ce paradigme en raison de ses avantages en termes de facilité de lecture, de compréhension et de maintenance du code.

Cependant, il est important de noter que le paradigme fonctionnel n'est qu'un outil parmi d'autres et que le choix du paradigme dépend du contexte et des besoins spécifiques du projet. Il y a toujours place pour une utilisation judicieuse et efficace de la programmation fonctionnelle en JavaScript et nous pouvons certainement nous attendre à voir de nouveaux développements dans ce domaine dans un avenir proche.

Il est fortement conseillé aux développeurs souhaitant aller plus loin et approfondir leur connaissance dans le domaine de la programmation fonctionnelle d'apprendre le Haskell.

Il est considéré comme un des langages les plus importants à apprendre pour la programmation fonctionnelle en raison de sa conformité aux principes fondamentaux de la programmation fonctionnelle, de sa communauté de développeurs active, de sa bibliothèque standard complète, de sa performance ainsi que de sa fiabilité.

Ce cours avait pour but de vous introduire au monde de la programmation fonctionnelle en javascript. Libre à vous de continuer à explorer les différents pans qui le composent.

Plusieurs pistes s'offrent à vous : des notions plus avancées comme les functors, pousser le point-free style à l'extrême, créer vos propres monades. Mais aussi d'autres langages ou superset de Javascript comme Typescript. Ou encore approfondir vos connaissances en mathématiques avec la théorie des catégories.

Tout est possible. Nous vous avons donné un kit de démarrage et quoi vous en sortir dans cette jungle de connaissances et de jargons compliqués : il ne tient qu'à vous de tracer votre propre chemin désormais.

Glossaire :

| | |
|---------------------------------------|--|
| Arité | Désigne le nombre d'arguments ou de paramètres que la fonction peut prendre. |
| Programmation fonctionnelle | Paradigme de programmation qui met l'accent sur l'utilisation de fonctions pures, immuables et sans effets de bord pour la résolution de problèmes. Cela permet une programmation plus claire, plus concise et plus facile à tester et à maintenir. |
| Fonction pure / pureté d'une fonction | Fonction qui retourne toujours le même résultat pour les mêmes entrées, sans produire d'effets de bords et sans dépendre d'états externes. |
| Scope (en javascript) | Désigne la portée d'un identificateur (comme une variable) au sein d'un programme. Elle définit une zone dans le code où une variable est accessible et peut être utilisée. Peut être globale ou locale. |
| Immutabilité | Désigne l'incapacité de changer une valeur une fois celle-ci assignée. |
| Etat / state | Désigne les données qui représentent un état spécifique d'un programme ou d'une application. |
| Transparence référentielle | Désigne la propriété d'une fonction qui retourne toujours le même résultat pour les mêmes entrées, et ce peu importe ce qui se produit dans le programme. |
| Effets de bords | Désignent des modifications non désirées à l'état d'un programme qui se produisent en dehors du contrôle direct de la fonction qui les a causés. |
| Equational reasoning | Désigne une méthode de déduction mathématique qui consiste à utiliser des équations pour démontrer la vérité ou la fausseté d'une affirmation. Elle se base sur l'idée que si deux équations représentent la même relation mathématique, alors toutes les conséquences logiques de cette relation sont vraies pour les deux équations. |
| Mémoïsation | Technique consistant à stocker les résultats de fonctions coûteuses en termes de temps pour éviter de les recalculer à chaque appel |
| Paradigme | Modèle de programmation qui définit la façon de concevoir |

| | |
|--------------------------------------|--|
| | et de construire des programmes. |
| Fonction d'Ordre Supérieur (HOF) | Fonction qui accepte une ou plusieurs fonctions en entrée ou qui renvoie une fonction en tant que résultat. |
| Composition de fonctions | Technique de programmation qui consiste à combiner plusieurs fonctions simples pour produire une fonction plus complexe. |
| Currying | Technique consistant à transformer une fonction à plusieurs arguments en une série de fonctions à un seul argument. |
| Tacit programming / Point-free style | Technique consistant à créer une fonction, la déclarer, sans pour autant définir les arguments de celle-ci de façon explicite. |
| Monades | Design pattern combinant les éléments de langages fonctionnels avec des éléments procéduraux. C'est un concept abstrait qui peut être utilisé pour modéliser l'état et les opérations qui gèrent les effets de bord dans les programmes. C'est aussi un constructeur de conteneurs qui permet de faire des opérations séquentielles de manière contrôlée et prévisible. |
| Foncteur | Structure de données qui peut être vue comme une fonction qui prend un type et renvoie un autre type. Ils permettent de modéliser des transformations de types de manière abstraite et de les utiliser pour des opérations génériques telles que la transformation de conteneurs de données. |
| Chaining | Désigne la pratique d'enchaîner plusieurs appels de méthodes ou fonctions les uns après les autres sur une même instance d'objet ou sur un même résultat. |
| Lois monadiques | Contraintes sémantiques qui définissent comment une monade doit être implémentée et utilisée. Elles garantissent que les opérations de la monade se comportent de manière cohérente et prévisible, en respectant des propriétés associatives et unitaires. On peut alors faire confiance aux résultats produits par les opérations monadiques sans avoir à connaître les détails de leur implémentation. |
| Théorie des catégories | Domaine de la mathématique abstraite qui étudie les relations et les transformations entre objets mathématiques. Elle définit une catégorie comme un |

| | |
|--|---|
| | ensemble d'objets et de morphismes (fonctions) entre ces objets, qui respectent des relations de composition et d'identité. Permet de formaliser des concepts mathématiques de manière générale et de les utiliser pour développer des méthodes de résolution de problèmes mathématiques complexes. |
| Flatmap | Opération qui combine la mappage et la flattening (ou "aplatissement") de conteneurs de données. Il prend en entrée une fonction qui mappe chaque élément d'un conteneur à un autre conteneur, puis concatène les conteneurs résultants en un seul. |
| Design Pattern (ou Patron de Conception) | Méthodes de résolution d'un problème connu grâce à l'arrangement et l'utilisation de modules et d'architecture. Peut être vu comme une réponse à un problème de conception d'un programme. |

Références

Liens :

Présentation functional programming : <https://www.youtube.com/watch?v=0if71HOyVjY>

Vocabulaire en français : <https://github.com/marcwrobel/functional-programming-jargon-fr>

Remplacement de la lib lodash en PF pour comprendre les utilisations et application de la PF :
<https://ramdajs.com/>

Conférence sur le pourquoi du comment la PF n'est pas la norme :
<https://www.youtube.com/watch?v=QyJZzq0v7Z4>

Functional-Light JavaScript : <https://github.com/getify/Functional-Light-JS>

Se faire une idée de la PF :
<https://dev.to/pryl/explain-functional-programming-paradigm-like-im-five-2pi4>

Définition programmation et ses paradigmes :
https://www.clear.rice.edu/comp310/course/design/prog_paradigms.html

Notions de PF :
<https://ingin.fr/blog/2021/02/14/programmation-fonctionnelle-en-javascript>

Monads :
https://fr.wikipedia.org/wiki/Foncteur#Foncteurs_conservatifs
[https://en.wikipedia.org/wiki/Monad_\(functional_programming\)#Overview](https://en.wikipedia.org/wiki/Monad_(functional_programming)#Overview)
<https://ingin.fr/blog/2022/04/27/javascript-programmation-fonctionnelle-avec-les-monades>

Théorie des Catégories :
https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_cat%C3%A9gories

Livres:
Learn You a Haskell for Great Good!: A Beginner's Guide

Paradigmes
<https://www.invivoo.com/paradigme-de-programmation/>

<https://librecours.net/module/js/js02/evenementielle.xhtml>

<https://www.lyceum.fr/tg/nsi/4-langages-et-programmation/4-paradigmes-de-programmation/>

(Pensez à mettre les liens de ref)