

Programmation fonctionnelle

Les bases

Developper Fullstack Javascript

Fullstack developer for about fourteen years, I like making software in all its aspects: user needs, prototypes, UI, frontend developments, API developments, ...

Team & Projects


I work on DataClient teams where we work for tooling around data client and perform the data client quality.

Hobbies


I like drawing, UX,
Web Design



SAMUEL GOMEZ

 samuel-gomez

 @gamuez

 www.samuelgomez.fr

 gamuez

Séance 1 : Base de JavaScript

Rappel rapide des bases.

Séance 2 : JavaScript avancé

Manipulation du DOM, appels HTTP, nouveauté ES6, ...

Séance 3 : Programmation fonctionnelle (bases)

Les concepts de bases.

Séance 4 : Programmation fonctionnelle (avancée)

Les concepts plus avancés.

Séance 5 : Les bases de React

JSX, composants, ...

Séance 6 : Gestion d'état

Etat local, Context API, Fetch, ...

Introduction

- Définition
- Quels langages
- Et le JavaScript ?

L'immutabilité

- Le principe
- Quels sont les bénéfices ?
- Exercices
- Corrigés

Les fonctions pures

- Définition
- Intérêts et avantages
- Exercices
- Corrigés

HOF : High Order Function

- Définition
- Avantages
- Exercices
- Corrigés

Composition de fonction

- Le principe
- Avantages
- La fonction compose
- Exercices
- Corrigés

Introduction

Introduction

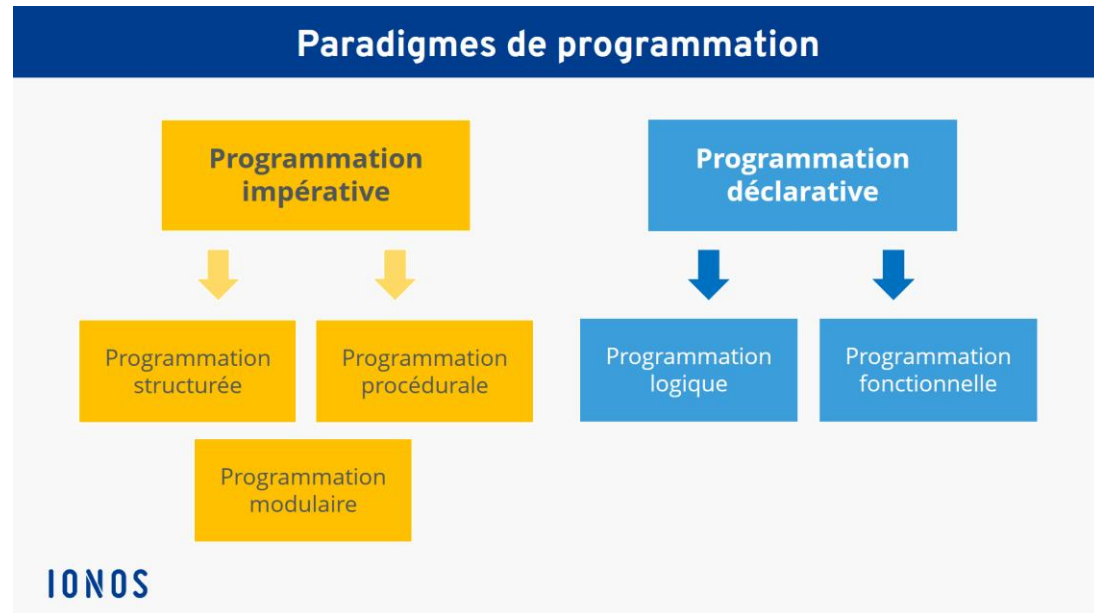
Définition

La **programmation fonctionnelle** est un **paradigme de programmation de type déclaratif** qui considère le calcul en tant qu'évaluation de fonctions mathématiques.

Comme le changement d'état et la mutation des données ne peuvent pas être représentés par des évaluations de fonctions.

La PF met en avant l'application des fonctions, alors que la **programmation impérative** qui met en avant les changements d'état.

Wikipédia



La **programmation déclarative** est à base d'expressions, c'est-à-dire qui retourne une **valeur** et non des instructions qui expriment une **action** utilisées pour leur effets de bord.

L'avantage de la programmation fonctionnelle est d'aboutir à un code **plus expressif**, facilement maintenable et testable.

Par exemple, le CSS ou l'HTML sont déclaratifs

Quels langages ?

La programmation fonctionnelle étant une façon de programmer, il n'y a pas de langage qui ne permette pas d'en faire. Cependant, certains langages ont été créés sur la base de cette méthode de développement, et permettent donc d'implémenter les concepts plus facilement.



Ocaml



Haskell



Clojure



Python



F#



ReasonML

Et JavaScript ?

A l'origine

JavaScript n'a clairement pas été pensé pour faire de la programmation fonctionnelle mais bien de la programmation orienté objet.

La tendance

Mais on l'a vu JavaScript évolue et de plus en plus d'améliorations tendent vers la programmation fonctionnelle.

De plus, au delà des fonctions natives, certains principes peuvent malgré tout être appliqués et améliorer significativement la qualité de votre code.

Un peu d'aide

Il existe également des bibliothèques JavaScript qui permettent de combler certaines lacunes encore présentes dans le langage.

L'immuabilité

L'immutabilité

Le principe

En PF, il n'y a pas de variables à proprement parler. Lorsqu'une variable prend une valeur, **elle ne doit plus changer**.

Voilà pour le principe, maintenant l'appliquer en JS, ce n'est pas aussi évident.

Le JavaScript est un [langage orienté objet à prototype](#) et n'a pas vraiment été conçu pour **l'immutabilité**.

Il existe différentes techniques ou bibliothèques qui permettent d'obtenir une immutabilité relative d'une application en JS.

L'immuabilité

Les types

En JS, si vous déclarez des variables avec **const**, certains types sont mutables de base d'autres pas, il faudra bien avoir cette liste en tête pour éviter les surprises.

Types immutables

- Boolean
- Number
- String
- Symbol
- Null
- Undefined

Types mutables

- Object
- Array
- Function

Remarque : Le type string est un cas à part, car on pourrait imaginer pouvoir modifier un élément de la chaîne comme on le ferait pour un tableau mais cela ne fonctionnera pas.


```
const message2 = "blabla";  
message2[1] = 'l';
```

L'immutabilité


Les fonctions natives

En JS, certaines fonctions natives ou opérations natives sont à éviter comme un push sur un tableau, l'incrément d'une valeur avec l'opérateur ++ ou encore changer l'attribut d'un objet.

Voici quelques exemples :



```
// méthodes qui mutent la variable
array.push('newval');
array.pop();
array.sort();
```



```
// méthodes qui retournent une nouvelle valeur sans toucher à la
variable originale
const merged = array.concat(arr1, arr2);
const filtered = array.filter(fn);
const mapped = array.map(fn);
```

Il faudra donc toujours rester vigilant sur les fonctions natives que l'on utilise, elle ne sont pas proscrites mais elle peuvent être source de bugs.

L'immuabilité

Les techniques

Utilisation du `const`

Le `const` assure une immuabilité pour les types non mutables mais partielle sur les types mutables.

Utilisation du `Object.Seal`

La méthode [Object.seal](#), scelle un objet pour l'ajout de propriété mais les valeurs courantes restent modifiables.

Utilisation du `Object.Freeze`

Pour les objets, on pourra aller un peu loin pour protéger le premier niveau seulement avec [Object.freeze](#), mais ce n'est pas la solution ultime.

Tuples et Records (en cours de validation)

Créer des objets ou des tableaux immutables

```
const obj = #{ a: 1, b: 2 }; // Record pour les objets
const arr = #[1, 2, 3, 4]; // Tuple pour les tableaux
```

<https://github.com/tc39/proposal-record-tuple>

L'immuabilité

Les techniques

La copie avec le spread operator

En JS, une des méthodes pour de faire de l'immuabilité est de faire **une (vraie) copie de l'élément** à modifier. Les nouveautés apportées par l'ES6 vont nous permettre de faire cela sans trop de difficulté .

Remarque : Sur des types mutables comme des tableaux ou des objets, si on essaie de copier une variable à une autre par affectation, on n'effectue pas une réellement une copie, cela créé juste une référence. C'est pour cette raison que lorsque l'on mute la nouvelle variable, cela modifie également la variable d'origine.

```
const notes = [1,2,3];  
const notesCopy = notes;  
notesCopy.push(5);  
console.log('notes', notes); // notes (4) [1, 2, 3, 5]  
console.log('notesCopy', notesCopy); // notesCopy (4) [1, 2, 3, 5]
```



```
const notesCopy = [...notes, 5]; // good
```



La copie avec Object.assign

```
var combined = Object.assign(objA, objB);
```

L'immutabilité

Les bibliothèques

Pourquoi utiliser des bibliothèques ?

Nativement, le meilleur choix éviter de muter un objet ou un tableau, sauf que cela n'est pas gratuit, on consomme plus de mémoire. Elles amènent une autre façon de gérer l'immutabilité par les structures de données immutables persistantes.

Mori

<https://swannodette.github.io/mori/>

- ClojureScript port
- Functional API
- Faster

ImmutableJS

<https://immutable-js.github.io/immutable-js/>

- JS through & through
- Object-oriented API
- A bit smaller

Et d'autres comme Ramdajs, Lodash, Immer ...

Remarque : Nous n'utiliserons pas ces bibliothèques dans ce cours mais je vous invite à y regarder de plus près.

L'immuabilité

Bénéfices

La prévisibilité

Les mutations masquent les changements, qui peuvent créer des **effets de bord** qui eux-mêmes peuvent créer des bugs. Avec l'immuabilité, on peut conserver son architecture d'application et un modèle mental simples, ce qui permet une **meilleure appréhension de la logique applicative**.

Performance

Même si le concept de copie des valeurs peut nécessiter **plus de mémoire**, les objets immuables peuvent utiliser le **partage structurel** pour réduire sa consommation. Grâce à cela, les mises à jour renvoient de nouvelles valeurs mais les structures internes sont partagées.

Suivi des mutations

L'immuabilité permet également d'optimiser votre application en utilisant l'**égalité de référence et de valeur**. Cela permet de voir très facilement si une valeur a changé ou non, auquel cas vous allez créer ou pas une copie de cette valeur.

Par exemple, en **React** on va comparer l'état précédent au nouvel état si on effectue le render uniquement si les états sont différents tant par la structure que par la valeur.



<https://codesandbox.io/s/exo-immutabilite-vqhg9>



Les fonctions pures

Les fonctions pures

Effets de bord

Avant de définir, une fonction pure, il faut d'abord savoir ce qu'est un **effet de bord** (side effect). Un effet de bord est un changement non local à la fonction.

Par exemple, si une fonction modifie une variable qui n'est pas son scope :

```
// A function with a side effect
var x = 10;

const myFunc = function ( y ) {
  x = x + y;
};

myFunc( 3 );
console.log( x ); // 13

myFunc( 3 );
console.log( x ); // 16
```

Quand une fonction produit un effet de bord, cela vous oblige à connaître les entrées/sorties pour comprendre ce que fait la fonction. Il faut connaître le contexte, l'historique du programme ce qui complexifie la lecture.

De plus, les effets peuvent provoquer des bugs imprévisibles et difficiles à corriger.

Minimiser les effets de bord est un des principes fondamentaux de la programmation fonctionnelle.

Quelques exemples :

- Ecriture sur le disque
- Ecriture réseau
- Dessin sur écran
- Loguer dans la console
- Lancer des erreurs
- Muter the DOM
- Muter des objets ou des tableaux passés en arguments
- ...

Les fonctions pures

Définition

Un des outils qui permet de minimiser les effets de bord au sein d'un programme est la **fonction pure**, pour cela il faudra que la fonction **ne modifie jamais** quelque chose qui est en **dehors de son scope**.

Cela facilite la lecture de la fonction et permet d'identifier très rapidement les causes et les effets.

Une fonction pure prend donc une ou des valeurs en arguments et un retour une valeur.

```
const userAge = 18;
const canBuyAlcohol = userAge => userAge >= 18 ? true : false;

// on peut remplacer l'appel
if (canBuyAlcohol(userAge)) { /* on procede à la vente */ }

// par son résultat
if (true) { /* on procede à la vente */ }
```

```
const userAge = 20;
function addYear(userAge)
  return userAge + 1;
}
console.log(addYear(userAge)); // 21
console.log(addYear(userAge)); // 21
console.log(userAge); // 20
```

Une fonction pure a toujours la **transparence référentielle** comme qualité. Ce terme signifie qu'il est possible de **remplacer l'expression par son résultat** sans changer le fonctionnement du programme.

Les fonctions pures

Déterministe

Une autre caractéristique d'une fonction pure est qu'elle **déterministe**, en d'autres termes, elle renverra toujours le **même résultat** pour les **mêmes paramètres d'entrée**.

```
const add = (a,b) => a+b;
```

Exemple simple : si on appelle la fonction **add** avec les mêmes valeurs d'entrée, on obtiendra toujours le même résultat

Évidemment, il est **impossible** en JavaScript de n'avoir que des fonctions pures, par exemple modifier le DOM ou faire des requêtes réseau sont des effets de bords que l'on a besoin d'effectuer.

Enfin, les fonctions pures sont [thread safe](#) (peut s'exécuter sur un même espace d'adressage sur plusieurs threads) puisqu'elles ne dépendent pas de la mémoire partagée et ne souffrent pas de [race conditions](#) dû à l'absence d'effets de bord. Il est donc très simple de les paralléliser.

Intérêts et avantages

L'intérêt des fonctions pures est d'avoir le **moins de mutabilité** possible car la celle-ci est souvent responsable de bugs ou de comportements non souhaités.

De même, une fonction pure sera le plus souvent destinée à ne remplir **qu'une fonction** et c'est la composition de plusieurs fonctions pures qui nous permettra d'élaborer des algorithmes plus complexes mais très maintenables.

Avantages :

- N'a pas d'effets secondaires inattendus.
- Ne cassera rien d'autre en dehors de la fonction.
- Garantit l'ordre d'exécution.
- Les tests unitaires sont plus faciles à écrire
- La mise en cache devient plus facile. Si les mêmes entrées produisent les mêmes sorties, vous pouvez facilement modifier les résultats si l'exécution de la fonction est longue.
- Plus facile à comprendre pour les développeurs, car tout le code est autonome

Les fonctions pures

Exercices



<https://codesandbox.io/s/exo-fonctions-pures-wywff>



HOF : High Order Function

HOF : High Order Function

Définition

Les fonctions d'ordre supérieur sont des fonctions qui ont au moins une des propriétés suivantes :

- *Elles prennent une ou plusieurs fonctions en entrée ;*
- *Elles renvoient une fonction.*

Wikipédia

En JS, les fonctions sont des objets et peuvent donc être passées en paramètre, les callback en sont un bon exemple.

```
function hashPasswd(passwd, callback) {  
  // compute hash  
  callback(computedHash);  
}  
  
hashPasswd(passwd, (hash) => {  
  // on fait des choses avec le hash  
})
```

HOF : High Order Function

Définition

On est peu habitués à avoir des fonctions comme valeur de retour. Cela dit, les HOF sont vraiment très pratiques pour DRY.

```
// dictionnaire
const l18n = {
  it: { hello: 'Bonjournò', bye: 'Ciao' },
  es: { hello: 'Hola', bye: 'Hasta luego' },
  fr: { hello: 'Bonjour', bye: 'À bientôt' }
};
```

```
function saySomethingToUser(l18n, locale, sayWhat) {
  return userName => `${l18n[locale][sayWhat]} ${userName}`;
}
// sayHelloInFrench et sayCiaoInSpanish sont des closures
const sayHelloInFrench = saySomethingToUser(l18n, 'fr', 'hello');
const sayByeInSpanish = saySomethingToUser(l18n, 'es', 'ciao');

console.log(sayHelloInFrench('Jean'));
console.log(sayByeInSpanish('Juan'));
```

La fonction « saySomethingToUser » est une HOF qui renvoie une fonction.

Cette manière de retourner des fonctions permet très facilement de construire des fonctions plus spécifiques et d'éviter d'avoir à passer trop de paramètres.

En effet, la fonction retournée “se souvient” des paramètres passés à la fonction parente, il s'agit d'[une closure](#).

Cela permet de construire plus petites avec moins de paramètres et de les chaîner si besoin. La fonction retournée a accès mémorise les paramètres de la fonction parente, c'est ce qu'on appelle une closure

HOF : High Order Function

Exercices



<https://codesandbox.io/s/exo-high-order-fonction-zb1vs>



Composition de fonctions

Composition de fonctions

Définition

Que l'on soit en PF ou pas, le principe **DRY** (Dont Repeat Yourself) s'applique tout le temps.
Avec des HOF, on peut bien **découper notre logique** en petites fonctions simples et testables.
Ensuite, il faut pouvoir les chaîner facilement pour pouvoir effectuer des traitements plus complexes sans devoir aller voir ce que fait chaque fonction.
Pour cela, on va utiliser le principe de composition de fonctions (emprunté aux mathématiques).

Exemple, prenons 2 fonctions pures, **f** incrémente de 2 et **g** multiplie par 4 : `const f = x => x + 2` et `const g = x => x * 4`

On pourrait écrire la fonction suivante : `const add2AndMult4 = x => (x + 2) * 4`

On va plutôt composer nos fonctions, on va d'abord appliquer **f** à **x** : `const res = f(x);`

Ensuite, on va appliquer **g** au résultat : `const composition = g(res);`

soit `const composition = g(f(x));`

Composition de fonctions

Compose

La fonction compose

En FP, on utilise une fonction bien connue qui s'appelle **compose**.
C'est une fonction qui crée une fonction à partir de 2 à n fonctions passées en paramètre.

```
const compose = (...fns) => (args) => fns.reduceRight((arg, fn) => fn(arg), args);
```

Par convention en PF, on compose les fonctions dans le même ordre dans lequel on les écrirait.
Il faut donc lire le chainage de droite à gauche (intérieur vers extérieur).

```
const composition = g(f(x));
```

 devient

```
const composition = compose(g,f);
```

```
const composition = h(g(f(x)));
```

 devient

```
const composition = compose(h,g,f);
```

Remarque : Attention, il faut garder une certaine logique de chainage, la valeur passée d'une fonction à l'autre doit pour pouvoir être traitée

Composition de fonctions

Exercices



<https://codesandbox.io/s/exo-composition-47idx>





MERCI