



# Fonctions

## Concepts clés de la Programmation Fonctionnelle

Auteur / Enseignant :

Alexandre Leroux ([alex@shrp.dev](mailto:alex@shrp.dev)) - 2024

Toute reproduction, représentation, modification, publication, adaptation de tout ou partie des éléments de ce support de formation, quel que soit le moyen ou le procédé utilisé, est interdite, sauf autorisation écrite préalable de l'auteur.

Icônes et illustrations libres de droit : <https://www.flaticon.com>

Document à usage personnel. Ne pas diffuser.

# — Les fonctions

---

## Différences fondamentales avec le Paradigme Impératif

Le Paradigme Déclaratif se caractérise par l'expression d'une intention de résultat plutôt que par l'expression d'une procédure de résolution (principe associé au Paradigme Impératif).

Il s'agit de définir le résultat attendu (le "*quoi*") plutôt que la façon d'y parvenir (le "*comment*").

## Les Fonctions sont partout !

Pour qu'un langage de programmation soit adapté à la Programmation Fonctionnelle, il est nécessaire que les fonctions soient des "*citoyens de 1ère classe*" (*First-Class Citizen*) :

- en tant que **paramètre d'entrée** (input) d'une fonction,
- en tant que **valeur de sortie** (output) d'une fonction,
- en tant que **structure de données** (container),
- et évidemment, en tant que fonctions !

---

## Fonction ≠ Méthode

Une **fonction** diffère d'une **méthode** (POO) dans le sens où elle est indépendante d'une classe.

Une méthode représente le comportement d'un objet ou d'une classe. Elle accède à un espace de l'état logiciel, dont le périmètre est délimité par la classe (attributs de l'objet).

**Par essence, une fonction est autonome.**

---

## Fonction $\neq$ Procédure / Routine

Le terme "fonction" est parfois employé à tort, pour désigner une **procédure** regroupant **une séquence d'instructions**.

Dans ce cas, il ne s'agit pas d'une fonction mais d'une **procédure**.

L'emploi de procédure ne s'assimile pas à la notion de fonction promue par la Programmation Fonctionnelle.

# Procedures vs. Functions



- Function:
  - no side effect
  - return a value
  - Function call: expression
- Procedure:
  - side effect, executed for it
  - no return value
  - Procedure call: statement
- No clear distinction made in most languages
  - C/C++: void
  - Ada/FORTRAN/Pascal: procedure/function

## Imbrication de fonctions

Comme son nom le suggère, un programme écrit dans un style fonctionnel correspond à une **imbrication de fonctions**.

```
const pipe = (...funcs) => {  
  return (val) => {  
    return funcs.reduce(  
      (x, func) => func(x), val  
    )  
  }  
}
```



---

## Fonction au sens mathématique

La Programmation Fonctionnelle fait en premier lieu référence à la notion de ***Fonction Mathématique*** plutôt qu'à la notion de fonction en programmation.

$100 + 1$

$f(100, 1) = 101$



$101 + 2$

$f(101, 2) = 103$



$103 + 3$

$f(103, 3) = 106$



$106 + 4$

$f(106, 4) = 110$



$110 + 5$

$f(110, 5) = 115$

## Programmation Fonctionnelle en Python

---

## Fonction Totale

**Les fonctions retournent toujours un résultat** (y compris en cas d'échec, elles ne retournent pas d'exception).

## Fonction Pure

**Les fonctions sont idempotentes : elles retournent toujours le même résultat à partir des mêmes valeurs fournis en entrée.**

Les fonctions ne produisent pas d'effet de bord : pas d'accès à l'état extérieur.

---

## Avantages d'une Fonction Pure

- Prédicible donc testable,
- Transparence référentielle : l'appel de la fonction peut être remplacé par la valeur retournée sans causer d'erreur,
- Sans effet de bord,
- Thread Safety : peut être exécutée simultanément par plusieurs threads parallèles ou concurrents (n'est pas sujet au *Race Conditions*).

## Problème des valeurs contextuelles et/ou liées au hasard

Attention, si la valeur retournée dépend du hasard (valeur random) ou d'une valeur contextuelle (Date, Timestamp...), la fonction est de facto impure, car non prédictible, reproductible et non idempotente.



# IDEMPOTENCE

WHEN PERFORMING AN OPERATION AGAIN GIVES THE SAME RESULT

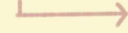
## IDEMPOTENT

LOOK\_AT\_CAKE

LOOK\_AT\_CAKE

LOOK\_AT\_CAKE

LOOK\_AT\_CAKE



## NOT IDEMPOTENT

EAT\_SLICE\_OF\_CAKE

EAT\_SLICE\_OF\_CAKE

EAT\_SLICE\_OF\_CAKE

EAT\_SLICE\_OF\_CAKE



# Idempotency

where performing an operation  
more than once has the same  
effect as performing it once

## Idempotence

```
let x = 1;

function add(y) {
  x += y;
}

add(2);

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

Fonction impure



```
function add(x,y){
  return x + y;
}

let x = add(1,2);

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

Fonction pure





---

## Fonction Déclarative

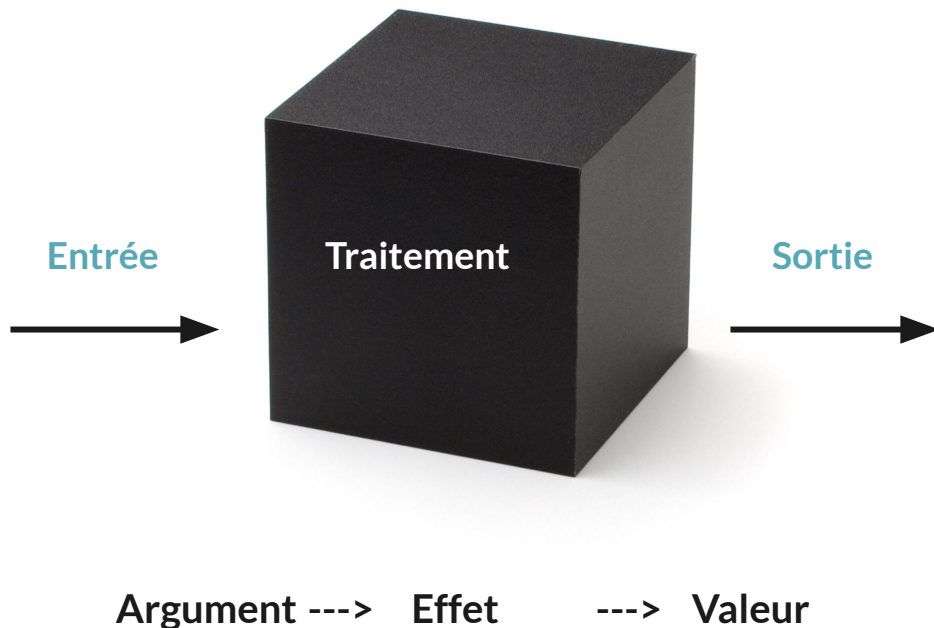
- Retourne toujours une valeur (de type scalaire, erreur ou fonction).
- Accède aux données internes ou passées en paramètres.
- Ne modifie pas les valeurs d'entrée.

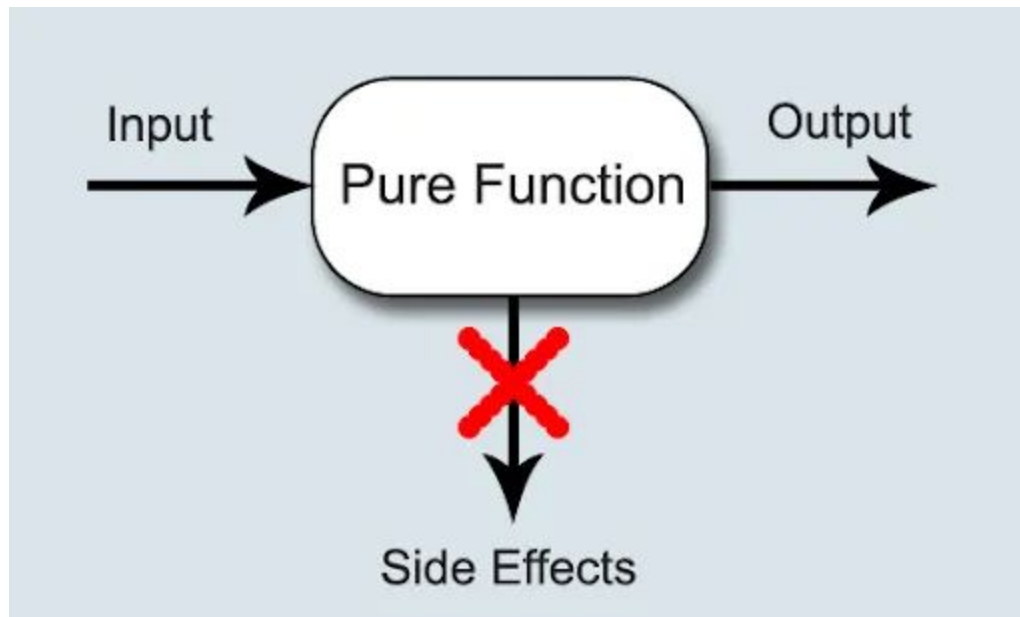
## Fonction impérative

- Regroupe des instructions,
- Peut ne retourner aucun résultat ("Void Function") ou déclencher une exception,
- Peut accéder au scope externe et modifier son état.

## Le principe de la "Boîte Noire"

Par essence, le mécanisme d'une fonction s'apparente à celui d'une "boîte noire" : l'appelant de la fonction communique des **données en entrée** et reçoit une **donnée en sortie**. Le fonctionnement interne reste opaque.





Par définition, une fonction pure  
est exempte d'effets de bord

---

## Entrée / Traitement / Sortie

- **Entrée** (Inputs) : une fonction prend une ou plusieurs valeurs en entrée.
- **Traitement** : Les valeurs communiquées en entrée sont accessibles dans le corps de la fonction sous forme de paramètres.
- **Sortie** (Output) : la fonction retourne une valeur, y compris en cas d'erreur. Pour rester *pure*, une fonction ne doit pas lever d'exception en cas d'erreur.



## Essayons la Programmation Fonctionnelle

[https://medium.com/openclassrooms-produit-design-et-ing%C3%A9nierie/essayons-la-programmation-fonctionnelle-8](https://medium.com/openclassrooms-produit-design-et-ing%C3%A9nierie/essayons-la-programmation-fonctionnelle-85958338ef31)

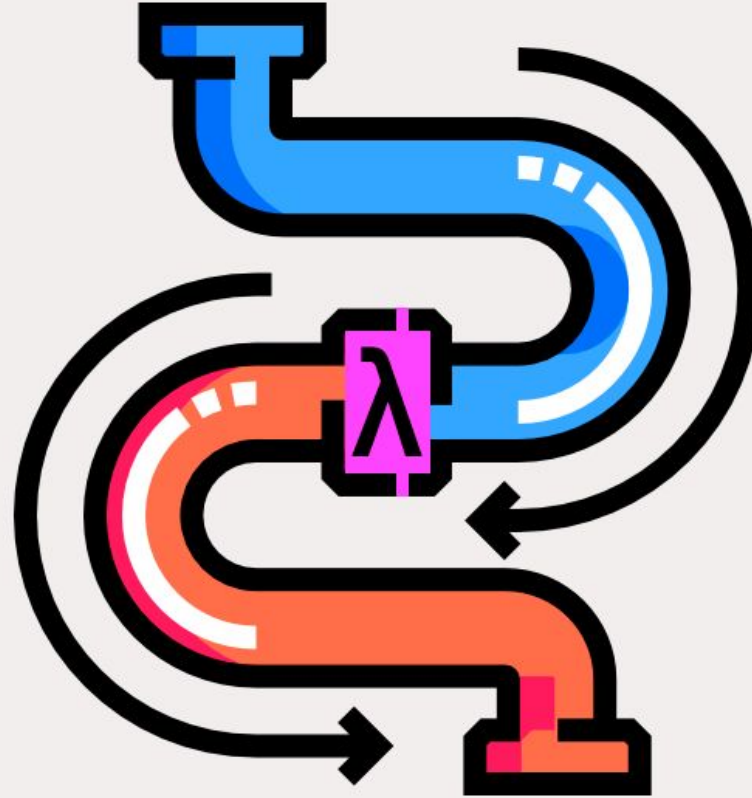
[5958338ef31](https://medium.com/openclassrooms-produit-design-et-ing%C3%A9nierie/essayons-la-programmation-fonctionnelle-85958338ef31)

Auteur : Alexandre Leroux (alex@shrp.dev) - <https://shrp.dev>

---

# Fonction Pure

**SAME INPUT**



**SAME OUTPUT**

---

## Fonction Pure

- Elle n'accède qu'aux données qui se trouvent dans son corps ou qui lui sont passées en paramètre(s),
- Ne modifie pas la valeur de ses paramètres,
- Ne produit pas d'effet de bord (pas d'accès à l'extérieur de la fonction),
- Est prédictible : elle retourne toujours la même valeur en sortie à partir des mêmes paramètres en entrée.



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

```
add(1, 2); //3
```

```
/*
```

```
à chaque fois que l'on appelle la méthode add en fournissant les  
paramètres 1 et 2, on obtient le même résultat : 3.
```

```
*/
```

## Fonction pure en JS (ES6+)

```
let n=1;

const add = (a) => n + a;

add(2); //3

add(2); //5

/*
à chaque fois que l'on appelle la méthode add en fournissant le
paramètre 1, on obtient un résultat différent, car il y a un
effet de bord sur la variable n.
*/
```

## Fonction impure en JS (ES6+)

---

# Fonction Totale

---

## Fonction Totale

Une Fonction Totale **retourne toujours un résultat.**

En cas d'erreur, la fonction **retourne une valeur de type Error** plutôt que de déclencher une **Exception** avec le mot clé *throw*.

```
const add = (a, b) => {  
  if (!Number.isFinite(a)) {  
    return new Error("arg 'a' must be a Number");  
  } else if (!Number.isFinite(b)) {  
    return new Error("arg 'b' must be a Number");  
  } else {  
    return a + b;  
  }  
};
```

```
const result = add(1, 2);  
console.log(result); //3
```

```
const error = add("0", 1);  
console.error(error.message);  
//"arg 'a' must be a Number"
```

# Writing Functionally Pure Code

## Is all my code going to be pure?

Nope. No program can be composed 100% of pure code because every program must do I/O work, which is inherently stateful. Most programs also need to carry around some amount of internal state, to one degree or another. The goal of functional programming, then, is to *maximize* the proportion of pure code to impure code in your program.

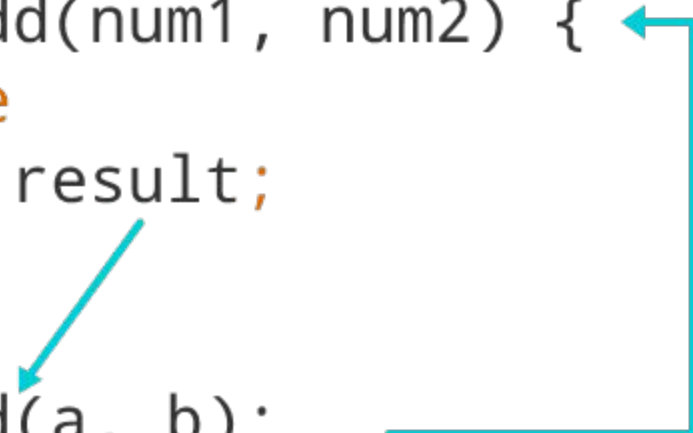
## Pure vs. Impure

What does it mean for data and code to be pure? The short answer is that pure things are immutable, but this is not quite accurate: all pure things are immutable, but not all immutable things are pure. Pure things are not only unmodifiable but also *definitional*.

## Writing Functionally Pure Code

<https://gist.github.com/BrianWill/e122e4cffa000d30f187>

```
function add(num1, num2) {  
    // code  
    return result;  
}  
  
let x = add(a, b);  
// code
```



function call

## Named vs Anonymous, Nested / Scoping / Closure, Arrow Functions...

<https://gist.github.com/BrianWill/ba9addaf887970e4f9c38cde905eae05>

---

# Fonction d'ordre supérieur



---

## Fonction d'ordre supérieur

Une fonction d'ordre supérieur est :

- une fonction qui **peut prendre une fonction en tant que paramètre** (callback, prédicat...),

et/ou

- une fonction qui **peut retourner une fonction en tant que résultat.**

```
function add(x) {  
    return function(y) {  
        return x+y;  
    }  
}  
  
const result = add(1)(1);  
  
console.log(result); //2
```

Fonction d'ordre supérieur  
retournant une fonction  
(syntaxe classique)

```
const add = x => y => x + y;  
  
const result = add(1)(1);  
  
console.log(result); //2
```

Fonction d'ordre supérieur  
retournant une fonction  
(syntaxe raccourcie)



```
const filter = (predicate, list) => list.filter(predicate);  
  
const is = (type) => (x) => Object(x) instanceof type;  
  
const values = [0, '1', 2, null];  
  
filter(is(Number), values) // [0, 2]
```

Fonction d'ordre supérieur  
prenant une fonction (prédicat) en tant qu'argument

## function definition

```
function calculateBill(meal, taxRate = 0.05) {  
  const total = meal * (1 + taxRate);  
  return total;  
}
```

**keyword**

**function name**

**Parameters**  
placeholders

**default value**

**Scope Start**

**function body**

**return statement**

**Scope End**

```
const myTotal = calculateBill(100, 0.13);
```

**variable to capture returned value**

**name or reference**

**call, run or invoke**

**Arguments**  
actual values

async, generator, ...rest and other ways to define a function not included.

🔥 @WesBos

## Anatomie d'une fonction

---

# Arité d'une fonction

---

## Arité d'une fonction

L'arité désigne le nombre de paramètres d'une fonction :

- **nullary** : 0 paramètre,
- **unary** : 1 paramètre,
- **binary** : 2 paramètres,
- **ternary** : 3 paramètres,
- **n-ary** : plus de 3 paramètres.
- **variadic** : nombre variable de paramètres.

```
const add = (a,b) => a + b;  
//fonction binaire  
  
const result = add(1,2);  
//3  
  
//refactorisée en fonction unaire  
const add = a => b => a + b;  
  
const result = add(1)(2);  
//3
```

En Programmation Fonctionnelle, les Fonctions Unaires doivent être privilégiées afin de pouvoir être appliquées dynamiquement dans des patterns avancés.

---

## Fonctions unaires pour le chaînage (pipeline)

L'application de plusieurs fonctions à la chaîne (pipeline) nécessite de disposer de fonctions unaires.

Le résultat d'une fonction est transmis en paramètre de la fonction unaire suivante, et ainsi-de-suite.



## Curryfication

Le concept de "**Curryfication**" (ou "*Currying*") permet de convertir une fonction disposant de plusieurs arguments en fonction unaire (1 seul paramètre).

La programmation fonctionnelle privilégie l'emploi de monades, utiles au concept de "*composition*" (cf. *chaining*, *pipeline*).

---

## Fonction unaire pour le chaînage de fonctions

L'exécution de fonctions en chaîne est rendu possible par le fait que les fonctions employées n'attendent qu'un argument en paramètre.

Dans une **opération chaînée** (compose, pipe), la **valeur générée en sortie (output)** par une fonction fait office de **valeur d'entrée (input)** pour la fonction suivante.

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

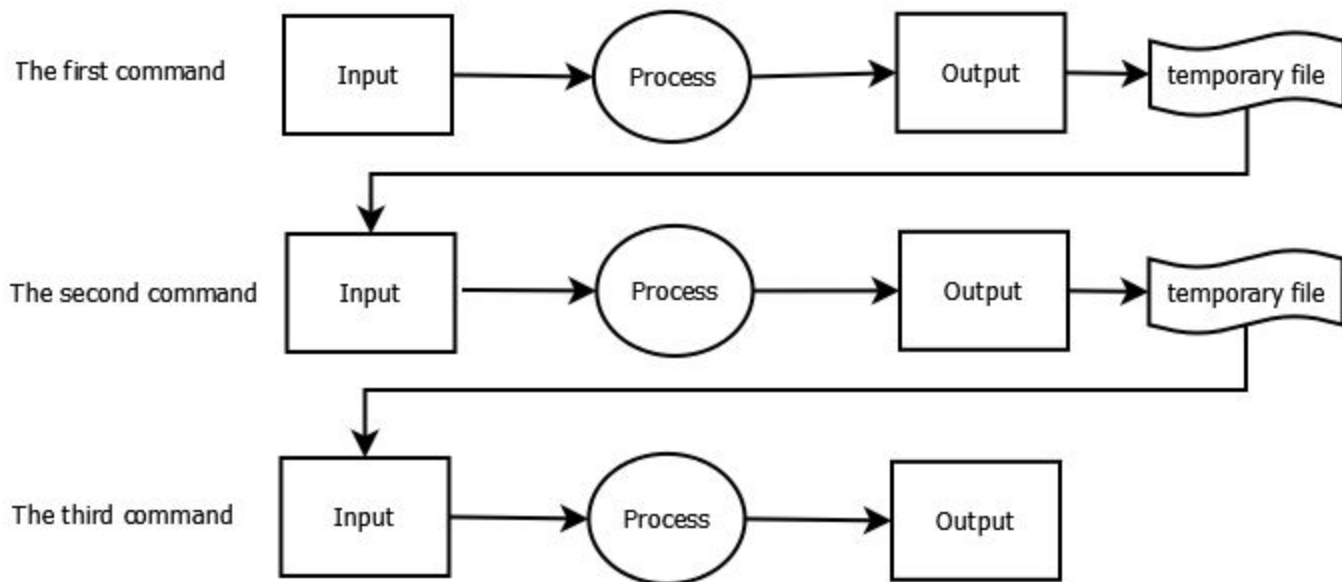
```
sum(1,2);  
//3
```

Fonction binaire

```
const sum = a => b => a+b;
```

```
sum(1)(2);  
//3
```

Fonction unaire  
après curryfication



## Pipes in Linux explained

<https://www.computernetworkingnotes.com/linux-tutorials/pipes-in-linux-explained.html>

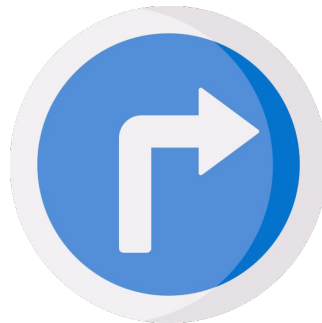
---

## Pipe vs Compose

**Compose** : applique une suite de fonctions, passés en argument, **de la droite vers la gauche**.



**Pipe** : applique une suite de fonctions, passés en argument, **de la gauche vers la droite**.



## Compose

```
const compose =  
  <T>(f: Function,  
    g: Function,  
    h: Function) =>  
    (x: T): T =>  
      f(g(h(x)));
```

## Pipe

```
const pipe =  
  <T>(f: Function,  
    g: Function,  
    h: Function) =>  
    (x: T): T =>  
      h(g(f(x)));
```

---

# Transparence référentielle

---

## Vérification de fonction

En Programmation Fonctionnelle, pour vérifier l'effet produit par l'application d'une fonction :

- on se base sur le résultat retourné par la fonction,
- et non pas sur un changement d'état provoqué par l'application de la fonction.



## Transparence référentielle

En Programmation Fonctionnelle, une fonction est une expression qui doit pouvoir être remplacée par le résultat qu'elle produit sans que cela n'impacte le fonctionnement du logiciel.

```
const square = (a) => a*a;
```

```
square(3)+1 === 10; //true
```

```
9+1 === 10; //true
```

```
10 === 10; //true
```

```
const incrementArray = array => {  
  const incremented = []  
  for(let i = 0; i < array.length; i++){  
    incremented.push(array[i] + 1)  
  }  
  return incremented  
}
```

**Vous faites sûrement de la programmation fonctionnelle sans le savoir**

<https://golb.ch/programmation-fonctionnelle/>

---

# Point Freestyle

---

## Point Freestyle

*Point Freestyle* est un concept de la Programmation Fonctionnelle qui aborde la façon d'invoquer des fonctions unaires ou curryfiées.

L'idée est d'écrire des fonctions plus concises et expressives pour faciliter la lecture du code.





## Point Freestyle

<https://medium.com/dailyjs/functional-js-7-point-free-style-b21a1416ac6a>

Auteur : Alexandre Leroux (alex@shrp.dev) - <https://shrp.dev>

```
const prices = [1,2,3,4,5];

const applyDiscount = a => a - (a*15/100);

const result = prices.map(applyDiscount);

const result2 = prices.map(a => a - (a*15/100));

console.log(prices);
//[1,2,3,4,5]

console.log(result);
//[0.85,1.7,2.55,3.4,4.25]

console.log(result2);
//[0.85,1.7,2.55,3.4,4.25]
```

## Point Freestyle

```
const add1 = x => x + 1;

const numbers = [1, 2, 3, 4];

const incrementedNumbers = numbers.map(add1);

console.log(incrementedNumbers);
//[2, 3, 4, 5]
```

## Point Freestyle

```
const teams = [  
  {name:"Brazil", color:"yellow"},  
  {name: "Germany", color: "white"},  
  {name: "Italy",color: "blue"},  
];
```

```
const getColor = (team) => team.color;
```

```
const colors = teams.map(getColor);
```

```
console.log(colors);
```

```
// [object Array] (3)
```

```
//["yellow","white","blue"]
```

## Point Freestyle : exemple basique



```
const teams = [  
  {name:"Brazil", color:"yellow"},  
  {name: "Germany", color: "white"},  
  {name: "Italy",color: "blue"},  
];  
  
//fonction curryfiée  
const get = (field) => (team) => team[field];  
  
const names = teams.map(get("name"));  
  
console.log(names);  
//[ 'Brazil', 'Germany', 'Italy' ]
```

## Point Freestyle : lecture dynamique d'un champ

```
const strNumList =  
["1","2","one","true"];  
  
const parsedNums_v1 =  
strNumList.map(  
  item=>parseInt(item)  
);  
// [1, 2, NaN, NaN]  
  
const parsedNums_v2 =  
strNumList.map(parseInt);  
// [1, NaN, NaN, NaN]
```

Point Freestyle : attention aux  
bizarreries de JS :)

```
const unary = fn => (...args) =>  
  fn(args[0]);  
  
const parsedNums_v3 =  
strNumList.map(unary(parseInt));  
// [1, 2, NaN, NaN]
```

Emploi d'un convertisseur de  
fonction unaire

```
const pizzas = [  
  {  
    name: "Margherita",  
    price: 6  
  },  
  {  
    name: "Quattro Stagioni",  
    price: 12  
  },  
  {  
    name: "Diavola",  
    price: 9  
  }  
];
```

## Collection de données utiles à l'exemple suivant

(cf. écrans suivants)

```
//fonction curryfiée
//association d'une fonction delegate à un champ field
const compute =
(delegate, field) => (data) => delegate(data, field);

//calcul d'une somme de valeurs provenant d'une liste
const sumOf = (data = [], field) =>
  data.reduce(
    (accumulator, currentValue) =>
      accumulator + currentValue[field] || 0, 0);

//application partielle de la fonction compute
const sumPricesOf = compute(sumOf, "price");

//calcule la somme du prix des pizzas
const sumPizzasPrice = sumPricesOf(pizzas);
```

## Freestyle Computation + Partial Application



## JavaScript Freestyle — The Power of Functional Programming

<https://medium.com/@chaluwa/javascript-freestyle-the-power-of-functional-programming-cf920c1e84be>

Auteur : Alexandre Leroux (alex@shrp.dev) - <https://shrp.dev>

## OO pattern/principle

- Single Responsibility Principle
- Open/Closed principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Factory pattern
- Strategy pattern
- Decorator pattern
- Visitor pattern

## FP equivalent

- Functions
- Functions
- Functions, also
- Functions
- You will be assimilated!
- Functions again
- Functions
- Resistance is futile!

*Seriously, FP patterns are different*

Slides de Scott Wlaschin à propos de la Programmation Fonctionnelle

<https://fr.slideshare.net/slideshow/fp-patterns-ndc-london2014/42373281>