



Effets de bord

Concepts clés de la Programmation Fonctionnelle

Auteur / Enseignant :

Alexandre Leroux (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.

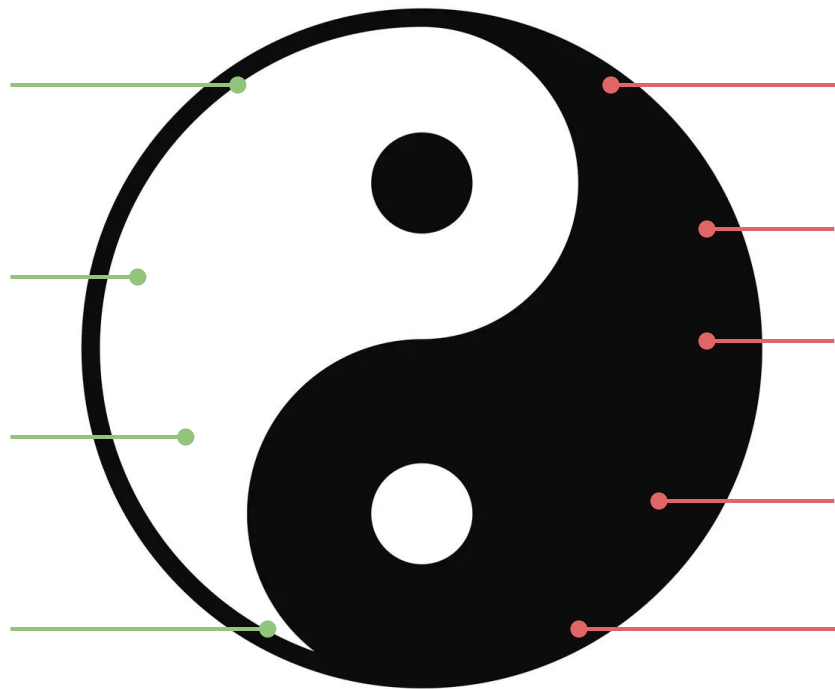
Monde pur & Monde impur

Fonctions
Idempotentes
et Totales

Algorithmique
et Logique
Métier

Fonctions
Unaires

Tests
Unitaires



I/O :
Logs, API, DB,
FileSystem...

UI

Mutations
de données

Exception

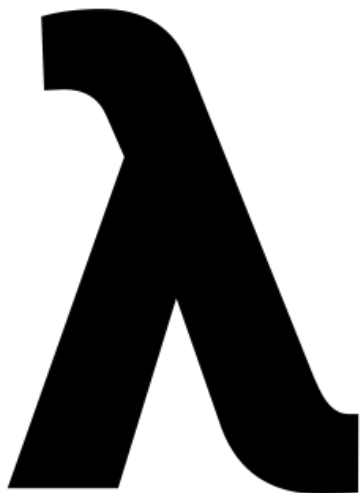
Random,
Timestamp...

L'opposition entre "*monde pur*"
et "*monde impur*" en Programmation Fonctionnelle

Monde pur

En Programmation Fonctionnelle, l'objectif est de réaliser des **logiciels fiables, composés de fonctions produisant des effets prédictibles, donc testables.**

Le "*monde pur*" correspond à un espace du programme maîtrisé, contrairement au "*monde impur*" en proie aux erreurs d'exécution (accès réseau, système de fichiers, base de données...).



Logique métier
Fonctions pures et totales
Résultats prédictibles
Code testable
Immutabilité
Pipeline d'effets
...

Monde pur

Monde impur

A contrario, le "*monde impur*" regroupe toutes les opérations qui ne sont pas prédictibles / de nature à produire des erreurs.

- Appels I/O (BDD, API, Système de fichier...),
- Rendu de la vue / interactions avec l'interface,
- Obtention de données contextuelles ou liées au hasard (date, timestamp, valeur random...).



Opérations I/O
Communication réseau
UI
Système de fichier
Base de données
Mutations
Exceptions
Random
Log
Void Functions
...


Monde impur

Un peu d'impur dans le monde pur

Pour qu'un logiciel réponde aux besoins du monde réel tout en respectant le Paradigme Fonctionnel, **il est nécessaire d'effectuer des compromis.**

Ainsi le programme qui permet au logiciel de fonctionner se divise en 2 mondes :

- le ***monde pur*** gère la logique métier et l'état des données,
- le ***monde impur*** fait l'interface avec l'extérieur : API tierces, BDD, système de fichiers, messages de logs, UI...



Pour que la logique métier ne soit pas impactée par des effets de bord, les données obtenues par le biais d'opérations "impures", doivent être communiquées aux fonctions en tant que arguments.

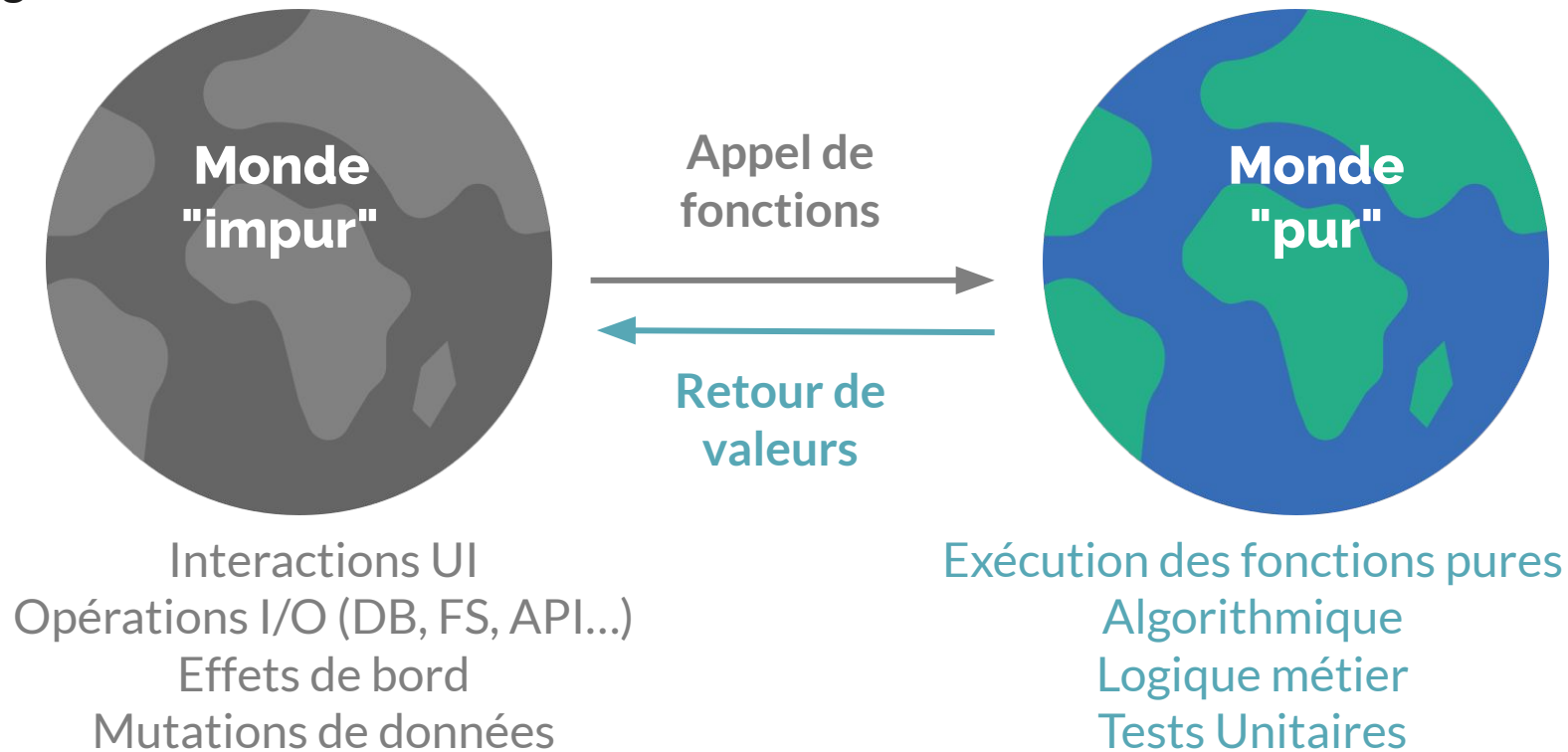


Programmation fonctionnelle et principe de réalité : gérer les effets

Xavier Van de Woestyne - Codeurs en Seine

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

Logiciel



Collaboration entre "monde impur" et "monde pur" au sein du logiciel

```
//monde "impur"  
//non-prédictible  
  
try{  
  
    const response =  
    await fetch(API);  
  
    const data =  
    await response.json();  
  
    const discount = data[0];  
  
    const price =  
    applyDiscount(product.price,  
    discount);  
  
}catch(e){  
    console.error(e);  
}
```

```
//monde "pur",  
//prédictible, testable  
  
const applyDiscount =  
(price,discount)=>  
    price -  
    (price * discount/100);
```

Dans l'exemple précédent, le monde "*impur*" récupère la valeur de l'attribut *price* d'un objet *Product* à l'aide d'une requête auprès d'une API.

Il s'agit d'une **opération asynchrone de type I/O**, considérée "**impure**" et **non-prédictible** (car exposée aux erreurs réseau).

Cette valeur est communiquée en tant qu'argument à une fonction du monde "*pur*" du logiciel.

Le résultat retourné par la fonction est prédictible, idempotent et donc testable.

then / catch

Dans le cadre d'opérations asynchrones, l'emploi des instructions ***then / catch*** avec le concept de ***Promise***, permet de gérer un éventuel problème révélé au runtime sans déclencher d'interruption du programme (et donc d'effet de bord).

En Programmation Fonctionnelle, les instructions ***then / catch*** sont préférées à ***try / catch + async / await*** car elles évitent le déclenchement d'une interruption du programme.

Limites du concept

La couche du Domaine d'un logiciel peut viser l'absence d'effets de bord. En revanche, la couche d'Infrastructure produit forcément des effets de bord en interagissant avec le *"monde extérieur"* :

- rendu graphique,
- interaction avec l'utilisateur,
- communication avec une base de données ou une API,
- production de messages de logs...



Functional Core, l'alternative FP à l'architecture hexagonale

Thomas Pierrain & Bruno Boucard

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

Effets de bord

Side effects (Effets Secondaires)

Une fonction est dite à effet de bord si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que retourner une valeur.

Wikipédia



pixtastock.com - 56251370

Effets de bord indésirables

On nomme "*effet de bord*" **une modification involontaire et indésirable d'une valeur dans un programme** provoquée par une intervention sur une valeur, en lecture ou en écriture.

Effets indésirables

Modification d'une valeur externe / globale depuis le corps de la fonction ou du programme, ou à l'interruption involontaires du programme, des effets de bord volontaires, correspondant à un accès ou à une modification de l'état applicatif...

Effets volontaires

Connexion à une base de données, requêtes auprès d'une API, écriture dans le système de fichiers, envoi d'un message de log...



— Les effets de bord sont inévitables

L'absence d'effets de bord est un des points majeurs de la Programmation Fonctionnelle.

Cependant, si l'on s'en tient au respect strict du principe **cela implique de ne pas interagir avec le monde extérieur** : interactions avec une API ou une Base de données, interactions avec le Système de fichiers, exécutions du résultats de tests unitaires, affichage de messages de logs et autres opérations I/O asynchrones... **ce qui rend le programme inutile.**

Limitation des effets de bord

L'un des objectifs de la Programmation Fonctionnelle est d'éviter de produire des *"effets de bord"* dans un programme.

Pour un langage de programmation "impur" tel que JS, cela passe par :

- l'**Immutabilité des données**,
- la **pureté des fonctions** (idempotence et prédictibilité),
- les **fonctions de 1ère classe** (*First-Class Citizen*),
- les **fonctions d'ordre supérieur** (*High-Order Functions*),
- la **transparence référentielle**.

Type de mutations de données en JS

- **Emploi de variables** déclarées avec *let* ou *var* (Attention, cela est aussi possible avec *const* sur un *Array* ou un *Object*),
- **Mutations de valeurs stockées dans des instances de classes** (via des attributs publics, setters ou par exécution de méthodes publiques),
- **Mutations directes, par référence** ou via l'un des **mutateurs** des types *Object* et *Array* (ex: *splice*, *reverse*, *sort*, *pop*, *push*...),
- **Mutations sur un *Set*** via les mutateurs *add*, *set*, *delete*,
- **Mutations sur un *Map*** via les mutateurs *clear*, *delete*, *set*.

Array : méthodes provoquant une mutation

```
copyWithin() ;  
fill() ;  
pop() ;  
push() ;  
reverse() ;
```

```
shift() ;  
sort() ;  
splice() ;  
unshift() ;
```



```
const fruits = [  
  "lemon", "orange", "apple"  
];  
  
fruits.splice(0,1);  
  
console.log(fruits);  
  
// [object Array] (2)  
// ["orange","apple"]
```

splice
-> méthode impure

```
const fruits = [  
  "lemon", "orange", "apple"  
];  
  
const copy = fruits.slice(0,1);  
  
console.log(fruits);  
// [object Array] (3)  
// ["lemon","orange","apple"]  
  
console.log(copy);  
// [object Array] (2)  
// ["orange","apple"]
```

slice
-> méthode pure



```
const countries = ["Uruguay", "Italy", "Brazil"];

let ref = countries;

ref.push("England");

console.log(countries);

// ["Uruguay", "Italy", "Brazil", "England"]
```

Mutation d'un Array par référence

```
const italy = {capital:"Rome"};  
let objRef = italy;
```

```
objRef.victories = [1934];
```

```
console.log(italy);
```

```
/*  
{  
  "capital": "Rome",  
  "victories": [  
    1934  
  ]  
}  
*/
```

Mutation d'un Objet par référence

En JS, les types primitifs sont immuables

`undefined`

`null`

`boolean`

`number`

`string`

`symbol`

```
const hello = "hello, world!";

hello.toUpperCase();

console.log(hello);

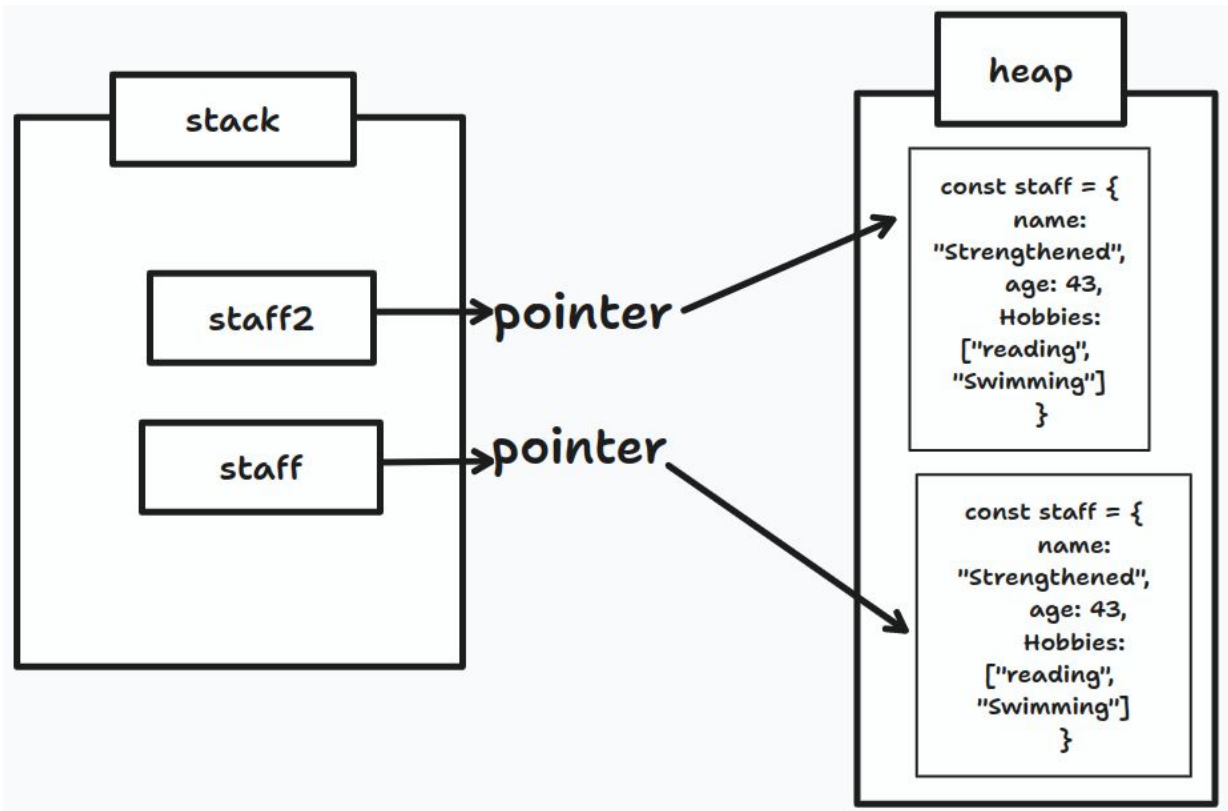
// "hello, world!"

const helloUpperCase = hello.toUpperCase();

console.log(helloUpperCase);

// "HELLO, WORLD!"
```

Démonstration d'immuabilité sur le type primitif string



Immutabilité en JS

<https://www.freecodecamp.org/news/mutability-vs-immutability-in-javascript/>

Origine des effets de bord (en général)

- Application des principes de la POO (système à état),
- Opérations I/O : système de fichiers, base de données, requêtes HTTP,
- Exécution d'exception,
- Ecriture de messages de log (dans la console ou dans un fichier),
- Modification de valeur de variables en-dehors du scope local,
- Fonctions impures
- Void Functions...

Origines des effets de bord en JS

- Boucles *for*, *while*...
- Conditions *if* sans *else*
- *console.log()*
- *alert()*
- Void functions
- Opérations I/O (requêtes HTTP, requêtes SQL, interactions avec le système de fichiers...).


```
let foo = 1;  
  
function increment(){  
  foo+=1;  
}  
  
increment();
```

Quelle est la source
d'un **effet de bord**
dans ce programme ?



```
let foo = 1;

function increment() {
  foo+=1;
}

increment();
//foo = 2
```



Effet de bord provoqué par :

- l'accès et la modification de l'état extérieur (valeur de la variable `foo`) au sein de la fonction `increment`.
- l'emploi d'une **fonction impure** / **void function** (ne retourne pas de valeur)
- l'emploi de l'instruction `+=` qui est un mutateur

```
let foo = 1;

function increment(foo){
  foo+=1;
}

increment(foo);
```

Quelle est la source
d'un **effet de bord**
dans ce programme ?



```
let foo = 1;

function increment(foo) {
  foo+=1;
}

increment(foo);
//foo = 2
```



Effet de bord provoqué par :

- l'accès et la modification de fournie en argument,
- l'emploi d'une **fonction impure / void function** (ne retourne pas de valeur)
- l'emploi de l'instruction **+=** qui est un mutateur

```
let foo = 1;
let incrementValue = 2;

function increment(foo) {
  console.log(foo);
  return foo+=incrementValue;
}

foo = increment(foo);

increment(foo);

foo = increment(foo);
```

Quelle est la source
d'un **effet de bord**
dans ce programme ?



```
let foo = 1;
let incrementValue = 2;

function increment(foo) {
  console.log(foo);
  return foo+=incrementValue;
}

foo = increment(foo); //3

increment(foo); //3

foo = increment(foo); //5
```



Effet de bord provoqué par :

- l'accès à l'état extérieur : **incrementValue**
- la modification de l'état extérieur : variable **foo**
- modification d'un argument
- l'emploi de l'instruction **+=** qui est un mutateur
- l'emploi de l'instruction

console.log()

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

```
let foo = 1;

function increment(target){
  return target+1;
}

const bar = increment(foo);

const baz = increment("2");
```

Quelle est la source
d'un **effet de bord**
dans ce programme ?



```
let foo = 1;

function increment(target) {
  return target+1;
}

const bar = increment(foo); //2

const baz = increment("2"); // "21"
```



Effet de bord provoqué par :

- la non vérification du type de la valeur passée à l'argument **target**
- si une valeur **string** est passée en argument de la fonction **increment**, une concaténation est effectuée à la place d'une incrémentation... et une valeur **string** est retournée.


```
let foo = 1;

function increment(target){
  if(
    !(target instanceof Number)
  ){
    throw new Error(
      `target should be a number`);
  }
  return target+1;
}

const bar = increment("2");
```

Quelle est la source
d'un **effet de bord**
dans ce programme ?



```
let foo = 1;

function increment(target){
  if(!(target instanceof Number)
  ){
    throw new Error(
      `target should be a number`);
  }
  return target+1;
}

const bar = increment("2");//21


/*Uncaught Error: target
should be a number */
```



Effet de bord provoqué par :

- la levée d'exception si le type la valeur passée à l'argument **target** n'est pas de type **Number**
- Emploi de **if** sans **else**.

En alternative, l'emploi d'une condition ternaire est recommandée.



De nombreuses autres situations, parfois plus complexes à détecter, peuvent provoquer des effets de bords indésirables et influencer sur les résultats fournis par un programme.

L'application des principes d'immuabilité et de pureté des fonctions est un premier moyen permettant de réduire les risques.

SIDE EFFECTS IN JAVASCRIPT

HOW TO AVOID IT BY WRITING PURE FUNCTIONS



How to Avoid Side Effects with Pure Functions in JavaScript

<https://javascript.plainenglish.io/how-to-avoid-side-effects-using-pure-functions-in-javascript-366acaafb60c>



Les 7 commandements d'une Fonction

1. Viser **1 seul objectif** à la fois,
2. **Prédictibilité** (= idempotence),
3. **Composabilité** (disposer de fonction unaire),
4. **Immutabilité** (ne pas changer la valeur des inputs),
5. **Toujours retourner une valeur** (ne pas déclencher d'exception, retourner une erreur),
6. **Pureté** : ne pas générer d'effets de bord,
7. **Pas d'état partagé.**



Les habitudes à perdre

- Pas d'**effets de bord**,
- Ne plus déclencher d'**exceptions**,
- Isoler les **opérations I/O** : fichiers, requêtes, logs...,
- Ne plus utiliser de **boucles** *for, while...*
- Privilégier les conditions ternaires aux **instructions if**,
- Ne plus effectuer de **mutations de données**,
- Pas de **fonction sans retour de valeur**,
- Pas de **fonction prenant plus d'un argument**,
- Pas de **modification de valeur d'argument...**