Concepts of Functional Programming in Javascript

TK Follow
Nov 25, 2018 · 12 min read ★

. . .

After a long time learning and working with object-oriented programming, I took a step back to think about system complexity.

> “Complexity is anything that makes software hard to understand or to modify." — John Outerhout

Doing some research, I found functional programming concepts like immutability and pure function. Those concepts are big advantages to build side-effect-free functions, so it is easier to maintain systems — with some other benefits.

In this post, I will tell you more about functional programming, and some important concepts, with a lot of code examples. In Javascript!

What is functional programming?

> Functional programming is a programming paradigm — a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data — Wikipedia

Pure functions

"water drop" by Mohan Murugesan on Unsplash

The first fundamental concept we learn when we want to understand functional programming is **pure functions**. But what does that really mean? What makes a

function pure?

So how do we know if a function is `pure` or not? Here is a very strict definition of purity:

- It returns the same result if given the same arguments (it is also referred as `deterministic`)
- It does not cause any observable side effects

It returns the same result if given the same arguments

Imagine we want to implement a function that calculates the area of a circle. An impure function would receive `radius` as the parameter, and then calculate `radius * radius * PI`:

```
let PI = 3.14;

const calculateArea = (radius) => radius * radius * PI;

calculateArea(10); // returns 314.0
```

not_pure_function.js hosted with ❤️ by GitHub view raw

Why is this an impure function? Simply because it uses a global object that was not passed as a parameter to the function.

Now imagine some mathematicians argue that the `PI` value is actually `42` and change the value of the global object.

Our impure function will now result in `10 * 10 * 42` = `4200`. For the same parameter (`radius = 10`), we have a different result. Let's fix it!

```
let PI = 3.14;

const calculateArea = (radius, pi) => radius * radius * pi;

calculateArea(10, PI); // returns 314.0
```

pure_function.js hosted with ❤️ by GitHub view raw

TA-DA🎉! Now we'll always pass the `PI` value as a parameter to the function. So now we are just accessing parameters passed to the function. No `external object`.

- For the parameters `radius = 10` & `PI = 3.14`, we will always have the same the result: `314.0`
- For the parameters `radius = 10` & `PI = 42`, we will always have the same the result: `4200`

Reading Files

If our function reads external files, it's not a pure function — the file's contents can change.

```
const charactersCounter = (text) => `Character count: ${text.length}`;

function analyzeFile(filename) {
  let fileContent = open(filename);
  return charactersCounter(fileContent);
}
```

external_file.js hosted with ❤️ by GitHub view raw

Random number generation

Any function that relies on a random number generator cannot be pure.

```
function yearEndEvaluation() {
  if (Math.random() > 0.5) {
    return "You get a raise!";
  } else {
    return "Better luck next year!";
  }
}
```

random_number_generator.js hosted with ❤️ by GitHub view raw

It does not cause any observable side effects

Examples of observable side effects include modifying a global object or a parameter passed by reference.

Now we want to implement a function to receive an integer value and return the value increased by 1.

```
let counter = 1;

function increaseCounter(value) {
  counter = value + 1;
}

increaseCounter(counter);
console.log(counter); // 2
```

modifying_global_variable.js hosted with ❤️ by GitHub view raw

We have the `counter` value. Our impure function receives that value and re-assigns the counter with the value increased by 1.

Observation: mutability is discouraged in functional programming.

We are modifying the global object. But how would we make it `pure`? Just return the value increased by 1. Simple as that.

```
let counter = 1;

const increaseCounter = (value) => value + 1;

increaseCounter(counter); // 2
console.log(counter); // 1
```

returns_increased_global_variable.js hosted with ❤️ by GitHub view raw

See that our pure function `increaseCounter` returns 2, but the `counter` value is still the same. The function returns the incremented value without altering the value of the variable.

If we follow these two simple rules, it gets easier to understand our programs. Now every function is isolated and unable to impact other parts of our system.

Pure functions are stable, consistent, and predictable. Given the same parameters, pure functions will always return the same result. We don't need to think of situations when the same parameter has different results — because it will never happen.

Pure functions benefits

The code's definitely easier to test. We don't need to mock anything. So we can unit test pure functions with different contexts:

- Given a parameter `A` → expect the function to return value `B`
- Given a parameter `C` → expect the function to return value `D`

A simple example would be a function to receive a collection of numbers and expect it to increment each element of this collection.

```
1  let list = [1, 2, 3, 4, 5];
2
3  const incrementNumbers = (list) => list.map(number => number + 1);
```

increment_numbers.js hosted with ❤️ by GitHub view raw

We receive the `numbers` array, use `map` incrementing each number, and return a new list of incremented numbers.

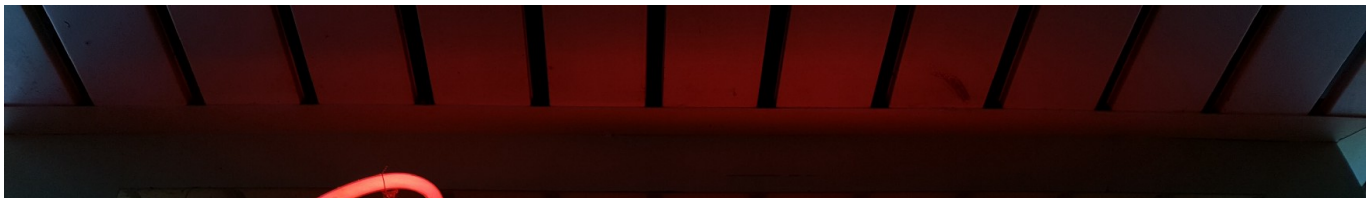
```
1  incrementNumbers(list); // [2, 3, 4, 5, 6]
```

test_increment_numbers.js hosted with ❤️ by GitHub view raw

For the `input` `[1, 2, 3, 4, 5]`, the expected `output` would be `[2, 3, 4, 5, 6]`.

Immutability

> Unchanging over time or unable to be changed.



"Change neon light signage" by Ross Findon on Unsplash

When data is immutable, its **state cannot change after it's created.** If you want to change an immutable object, you can't. Instead, **you create a new object with the new value.**

In Javascript we commonly use the `for` loop. This next `for` statement has some mutable variables.

```
var values = [1, 2, 3, 4, 5];
var sumOfValues = 0;

for (var i = 0; i < values.length; i++) {
  sumOfValues += values[i];
}

sumOfValues // 15
```

mutable_for.js hosted with ❤️ by GitHub view raw

For each iteration, we are changing the `i` and the `sumOfValue` **state.** But how do we handle mutability in iteration? Recursion!

```
let list = [1, 2, 3, 4, 5];
let accumulator = 0;

function sum(list, accumulator) {
  if (list.length == 0) {
```

```
    return accumulator;
  }

  return sum(list.slice(1), accumulator + list[0]);
}

sum(list, accumulator); // 15
list; // [1, 2, 3, 4, 5]
accumulator; // 0
```

recursive_sum.js hosted with ❤️ by GitHub view raw

So here we have the `sum` function that receives a vector of numerical values. The function calls itself until we get the list empty (our recursion `base case`). For each "iteration" we will add the value to the `total` accumulator.

With recursion, we keep our **variables** immutable. The `list` and the `accumulator` variables are not changed. It keeps the same value.

Observation: Yes! We can use `reduce` to implement this function. We will cover this in the `Higher Order Functions` topic.

It is also very common to build up the final **state** of an object. Imagine we have a string, and we want to transform this string into a `url slug`.

In OOP in Ruby, we would create a class, let's say, `UrlSlugify`. And this class will have a `slugify!` method to transform the string input into a `url slug`.

```ruby
class UrlSlugify
  attr_reader :text

  def initialize(text)
    @text = text
  end

  def slugify!
    text.downcase!
    text.strip!
    text.gsub!(' ', '-')
  end
end
```

```ruby

UrlSlugify.new(' I will be a url slug   ').slugify! # "i-will-be-a-url-slug"
```

url_slugify.rb hosted with ❤️ by GitHub view raw

Beautiful! It's implemented! Here we have imperative programming saying exactly what we want to do in each `slugify` process — first lower case, then remove useless white spaces and, finally, replace remaining white spaces with hyphens.

But we are mutating the input state in this process.

We can handle this mutation by doing function composition, or function chaining. In other words, the result of a function will be used as an input for the next function, without modifying the original input string.

```javascript
const string = " I will be a url slug   ";

const slugify = string =>
  string
    .toLowerCase()
    .trim()
    .split(" ")
    .join("-");

slugify(string); // i-will-be-a-url-slug
```

slugify_composition.js hosted with ❤️ by GitHub view raw

Here we have:

- `toLowerCase`: converts the string to all lower case
- `trim`: removes whitespace from both ends of a string
- `split` and `join`: replaces all instances of match with replacement in a given string

We combine all these 4 functions and we can `"slugify"` our string.

Referential transparency

"person holding eyeglasses" by Josh Calabrese on Unsplash

Let's implement a `square function`:

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

referential_transparency.js hosted with ❤ by GitHub view raw

This pure function will always have the same output, given the same input.

```
square(2); // 4
square(2); // 4
square(2); // 4
// ...
```

referential_transparency_output.js hosted with ❤ by GitHub view raw

Passing `2` as a parameter of the `square function` will always returns 4. So now we can replace the `square(2)` with 4. That's it! Our function is `referentially transparent`.

Basically, if a function consistently yields the same result for the same input, it is referentially transparent.

pure functions + immutable data = referential transparency

With this concept, a cool thing we can do is to memoize the function. Imagine we have this function:

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

memoization.js hosted with ❤ by GitHub — view raw

And we call it with these parameters:

```
1  sum(3, sum(5, 8));
```

memoization_1.js hosted with ❤ by GitHub — view raw

The `sum(5, 8)` equals `13`. This function will always result in `13`. So we can do this:

```
1  sum(3, 13);
```

memoization_2.js hosted with ❤ by GitHub — view raw

And this expression will always result in `16`. We can replace the entire expression with a numerical constant and memoize it.

Functions as first-class entities

"first-class" by Andrew Neel on Unsplash

The idea of functions as first-class entities is that functions are **also** treated as values **and** used as data.

Functions as first-class entities can:

- refer to it from constants and variables
- pass it as a parameter to other functions
- return it as result from other functions

The idea is to treat functions as values and pass functions like data. This way we can combine different functions to create new functions with new behavior.

Imagine we have a function that sums two values and then doubles the value. Something like this:

```
1  const doubleSum = (a, b) => (a + b) * 2;
```

double_sum.js hosted with ❤️ by GitHub view raw

Now a function that subtracts values and the returns the double:

```
1  const doubleSubtraction = (a, b) => (a - b) * 2;
```

double_subtraction.js hosted with ❤️ by GitHub view raw

These functions have similar logic, but the difference is the operators functions. If we can treat functions as values and pass these as arguments, we can build a function that receives the operator function and use it inside our function. Let's build it!

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

const doubleOperator = (f, a, b) => f(a, b) * 2;

doubleOperator(sum, 3, 1); // 8
doubleOperator(subtraction, 3, 1); // 4
```

double_operator.js hosted with ❤️ by GitHub view raw

Done! Now we have an `f` argument, and use it to process `a` and `b`. We passed the `sum` and `subtraction` functions to compose with the `doubleOperator` function and create a new behavior.

Higher-order functions

When we talk about higher-order functions, we mean a function that either:

- takes one or more functions as arguments, or
- returns a function as its result

The `doubleOperator` function we implemented above is a higher-order function because it takes an operator function as an argument and uses it.

You've probably already heard about `filter`, `map`, and `reduce`. Let's take a look at these.

Filter

Given a collection, we want to filter by an attribute. The filter function expects a `true` or `false` value to determine if the element **should or should not** be included in the result collection. Basically, if the callback expression is `true`, the filter function will include the element in the result collection. Otherwise, it will not.

A simple example is when we have a collection of integers and we want only the even numbers.

Imperative approach

An imperative way to do it with Javascript is to:

- create an empty array `evenNumbers`
- iterate over the `numbers` array
- push the even numbers to the `evenNumbers` array

```
var numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
var evenNumbers = [];

for (var i = 0; i < numbers.length; i++) {
  if (numbers[i] % 2 == 0) {
    evenNumbers.push(numbers[i]);
  }
}

console.log(evenNumbers); // (6) [0, 2, 4, 6, 8, 10]
```

imperative-even-numbers.js hosted with ❤️ by GitHub view raw

We can also use the `filter` higher order function to receive the `even` function, and return a list of even numbers:

```
const even = n => n % 2 == 0;
const listOfNumbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
listOfNumbers.filter(even); // [0, 2, 4, 6, 8, 10]
```

filter_even_numbers.js hosted with ❤️ by GitHub view raw

One interesting problem I solved on Hacker Rank FP Path was the **Filter Array problem**. The problem idea is to filter a given array of integers and output only those values that are less than a specified value `X`.

An imperative Javascript solution to this problem is something like:

```
var filterArray = function(x, coll) {
  var resultArray = [];

  for (var i = 0; i < coll.length; i++) {
    if (coll[i] < x) {
      resultArray.push(coll[i]);
    }
  }

  return resultArray;
}

console.log(filterArray(3, [10, 9, 8, 2, 7, 5, 1, 3, 0])); // (3) [2, 1, 0]
```

imperative-filter-array.js hosted with ❤️ by GitHub view raw

We say exactly what our function needs to do — iterate over the collection, compare the collection current item with `x`, and push this element to the `resultArray` if it pass the condition.

Declarative approach

But we want a more declarative way to solve this problem, and using the `filter` higher order function as well.

A declarative Javascript solution would be something like this:

```
function smaller(number) {
  return number < this;
}

function filterArray(x, listOfNumbers) {
  return listOfNumbers.filter(smaller, x);
}

let numbers = [10, 9, 8, 2, 7, 5, 1, 3, 0];

filterArray(3, numbers); // [2, 1, 0]
```

declarative_filter_array.js hosted with ❤️ by GitHub view raw

Using `this` in the `smaller` function seems a bit strange in the first place, but is easy to understand.

`this` will be the second parameter in the `filter` function. In this case, `3` (the `x`) is represented by `this`. That's it.

. . .

We can also do this with maps. Imagine we have a map of people with their `name` and `age`.

```
let people = [
  { name: "TK", age: 26 },
  { name: "Kaio", age: 10 },
  { name: "Kazumi", age: 30 }
];
```

people.js hosted with ❤ by GitHub view raw

And we want to filter only people over a specified value of age, in this example people who are more than 21 years old.

```
const olderThan21 = person => person.age > 21;
const overAge = people => people.filter(olderThan21);
overAge(people); // [{ name: 'TK', age: 26 }, { name: 'Kazumi', age: 30 }]
```

over_age.js hosted with ❤ by GitHub view raw

Summary of code:

- we have a list of people (with `name` and `age`).
- we have a function `olderThan21`. In this case, for each person in people array, we want to access the `age` and see if it is older than 21.
- we filter all people based on this function.

Map

The idea of map is to transform a collection.

> The `map` method transforms a collection by applying a function to all of its elements and building a new collection from the returned values.

Let's get the same `people` collection above. We don't want to filter by "over age" now. We just want a list of strings, something like `TK is 26 years old`. So the final string might be `:name is :age years old` where `:name` and `:age` are attributes from each element in the `people` collection.

In a imperative Javascript way, it would be:

```
var people = [
  { name: "TK", age: 26 },
  { name: "Kaio", age: 10 },
  { name: "Kazumi", age: 30 }
];

var peopleSentences = [];

for (var i = 0; i < people.length; i++) {
  var sentence = people[i].name + " is " + people[i].age + " years old";
  peopleSentences.push(sentence);
}

console.log(peopleSentences); // ['TK is 26 years old', 'Kaio is 10 years old', 'Kazumi
```

imperative-map-people.js hosted with ❤️ by GitHub view raw

In a declarative Javascript way, it would be:

```
const makeSentence = (person) => `${person.name} is ${person.age} years old`;

const peopleSentences = (people) => people.map(makeSentence);

peopleSentences(people);
// ['TK is 26 years old', 'Kaio is 10 years old', 'Kazumi is 30 years old']
```

map_people.js hosted with ❤️ by GitHub view raw

The whole idea is to transform a given array into a new array.

Another interesting Hacker Rank problem was the **update list problem**. We just want to update the values of a given array with their absolute values.

For example, the input `[1, 2, 3, -4, 5]` needs the output to be `[1, 2, 3, 4, 5]`. The absolute value of `-4` is `4`.

A simple solution would be an in-place update for each collection value.

```
var values = [1, 2, 3, -4, 5];

for (var i = 0; i < values.length; i++) {
  values[i] = Math.abs(values[i]);
}

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

in-place-update-list.js hosted with ❤️ by GitHub view raw

We use the `Math.abs` function to transform the value into its absolute value, and do the in-place update.

This is **not** a functional way to implement this solution.

First, we learned about immutability. We know how immutability is important to make our functions more consistent and predictable. The idea is to build a new collection with all absolute values.

Second, why not use `map` here to "transform" all data?

My first idea was to test the `Math.abs` function to handle only one value.

```
Math.abs(-1); // 1
Math.abs(1); // 1
Math.abs(-2); // 2
Math.abs(2); // 2
```

testing_math_abs.js hosted with ❤️ by GitHub view raw

We want to transform each value into a positive value (the absolute value).

Now that we know how to do `absolute` for one value, we can use this function to pass as an argument to the `map` function. Do you remember that a `higher order function` can receive a function as an argument and use it? Yes, map can do it!

```
let values = [1, 2, 3, -4, 5];

const updateListMap = (values) => values.map(Math.abs);

updateListMap(values); // [1, 2, 3, 4, 5]
```

update_list.js hosted with ❤️ by GitHub view raw

Wow. So beautiful! 😍

Reduce

The idea of reduce is to receive a function and a collection, and return a value created by combining the items.

A common example people talk about is to get the total amount of an order. Imagine you were at a shopping website. You've added `Product 1`, `Product 2`, `Product 3`, and `Product 4` to your shopping cart (order). Now we want to calculate the total amount of the shopping cart.

In imperative way, we would iterate the order list and sum each product amount to the total amount.

Using `reduce`, we can build a function to handle the `amount sum` and pass it as an argument to the `reduce` function.

Here we have `shoppingCart`, the function `sumAmount` that receives the current `currentTotalAmount`, and the `order` object to `sum` them.

The `getTotalAmount` function is used to `reduce` the `shoppingCart` by using the `sumAmount` and starting from `0`.

Another way to get the total amount is to compose `map` and `reduce`. What do I mean by that? We can use `map` to transform the `shoppingCart` into a collection of `amount` values, and then just use the `reduce` function with `sumAmount` function.

The `getAmount` receives the product object and returns only the `amount` value. So what we have here is `[10, 30, 20, 60]`. And then the `reduce` combines all items by adding up. Beautiful!

We took a look at how each higher order function works. I want to show you an example of how we can compose all three functions in a simple example.

Talking about `shopping cart`, imagine we have this list of products in our order:

We want the total amount of all books in our shopping cart. Simple as that. The algorithm?

- **filter** by book type
- transform the shopping cart into a collection of amount using **map**
- combine all items by adding them up with **reduce**

Done! 🎉

Resources

I've organised some resources I read and studied. I'm sharing the ones that I found really interesting. For more resources, visit my **Functional Programming Github repository**.

- EcmaScript 6 course by Wes Bos
- JavaScript by OneMonth
- Ruby specific resources
- Javascript specific resources

- Clojure specific resources

Intros

- Learning FP in JS
- Intro do FP with Python
- Overview of FP
- A quick intro to functional JS
- What is FP?
- Functional Programming Jargon

Pure functions

- What is a pure function?
- Pure Functional Programming 1
- Pure Functional Programming 2

Immutable data

- Immutable DS for functional programming
- Why shared mutable state is the root of all evil

Higher-order functions

- Eloquent JS: Higher Order Functions
- Fun fun function Filter
- Fun fun function Map
- Fun fun function Basic Reduce
- Fun fun function Advanced Reduce
- Clojure Higher Order Functions
- Purely Function Filter

- Purely Functional Map
- Purely Functional Reduce

Declarative Programming

- Declarative Programming vs Imperative

. . .

That's it!

Hey people, I hope you had fun reading this post, and I hope you learned a lot here! This was my attempt to share what I'm learning.

Here is the repository with all codes from this article.

Come learn with me. I'm sharing resources and my code in this **Learning Functional Programming repository**.

I also wrote an **FP post but using mainly Clojure ❤.**

If you want a complete Javascript course, learn more real-world coding skills and build projects, try ***One Month Javascript Bootcamp***. See you there ☺

I hope you saw something useful for you here. And see you next time! :)

. . .

> I hope you liked this content. Support my work on Ko-Fi

. . .

My Twitter & Github. ☺

TK.

Functional Programming | Programming | Coding | Software Development | JavaScript