Understanding Functional Programming in Javascript — A Complete Guide

Andrew Davis Escalona
Jan 15 · 20 min read

When it comes to programming paradigms, the most popular choice has been Object-Oriented Programming. It's not difficult to find developers who can describe what a Class does or how to instantiate an object.

Functional Programming has been around for much longer than OOP. The best example is LISP, which its first specification was written in 1958. However, unlike OOP it's not as simple to find developers who can understand functional concepts such as Purity, Currying, or Function Composition.

Javascript is not a functional programming language, or at least, that's not its main oriented paradigm. That doesn't mean we cannot work in a Functional way by using libraries like Lodash, Underscore, RambdaJS or only using Vanilla Javascript.

This year I had the chance to read this book Mastering Javascript Functional Programming by Federico Kereki. This is a book I personally recommend to whoever wants to have a deep understanding of Functional Programming concepts and their applications using Javascript. In this article, I'll explain what I learned from Kereki's book. My intention is not to repeat everything from the book — If you want that, read the book! — but rather offer a summary and the main points.

> ***NOTE**: This article is addressed to Javascript developers at an Intermediate/Advanced level.*

. . .

1. Functions as First-Class Objects

When doing Functional Programming, functions are first-class objects. This means you can use functions as if they were variables or constants. You can combine functions with other functions and generate new functions in the process. You can chain functions together to make complex calculations. In general, functions are everything!

In Javascript, you can define functions in several ways. I find it convenient to always use the arrow form — unless you want to use the `this` state. Example:

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

If you are an experienced JS developer, more than likely you have come across callbacks functions like the ones used with `setTimeout` and `setInterval` . Those are perfect examples of how to use functions as parameters. Another example:

```
var doSomething = function(status) {
// Doing something
};

var foo = function(data, func) { // Passing function as a parameter
    func(data);
}

foo("some data", doSomething);
```

2. The importance of Pure Functions

Working with functions is not the only thing you need to do in order to work in a functional way. You also need to keep your functions **pure**. But what does that mean? Well, according to Federico Kereki, you have a pure function when it meets the following criteria:

> *- Given the same arguments, the function always calculates and returns the same result*
>
> *- When calculating its result, the function doesn't cause any observable side effect, including output to I/O devices, mutation of objects, change to program state outside of the function, and so on.*

Let's see an example:

```
const getRectangleArea = (sideA, sideB) => sideA * sideB;

console.log(getRectangleArea(2, 3)); // 6
```

This function is pure because it will always return the result "6" for the given arguments "2" and "3". This function is not affecting its external context at all. Everything is happening inside of it and it's producing a new result without altering (mutating) its arguments. So we can confidently say: **It does not have side effects, therefore the function is pure.**

That's pretty much it. Your function is pure when it does not have side effects. If it does not have side effects, then your function is pure. Easy right? *But what that hell is a side effect anyway?* Here there's a list of possible side effects you might find around:

- Using global variables (unless they're constants).
- Mutating objects received as arguments.
- Operations like doing any kind of I/O, working with, changing, the filesystem, updating a database, calling an external API, etc.

And last but not least, using a function that happens to be impure. Federico says that impure functions are "contagious". So if your function uses something that causes a side effect, impure your function will become.

3. Dealing with Side Effects

Pure functions sound nice but, unless you're developing a simple calculator, it's more than likely you will need to work with asynchronous operations like accessing a database or calling an external API. Like Thanos, side effects are *inevitable*.

So how can we manage to use impure functions at the same time we work in a functional way? First, you need to accept the fact that it is not possible to achieve 100% pure-functional programming in your day-to-day work. Nevertheless, that shouldn't be your goal either. As Federico says in his book:

> *However, don't fall into the trap of considering FP as a goal! Think of FP only as a means towards an end, as with all software tools. Functional code isn't good just for being functional… and writing bad code is just as possible with FP as with any other techniques!*

Federico has a name for it: "Sorta Functional Programming". In general lines, our goal should be to decouple the impure parts of our code from the pure parts. Here you can find an example:

```
/*
  file.txt content:
  Hello file!
*/

const fs = require("fs").promises;
const path = require("path");

const getFileContent = fileName =>
  fs.readFile(path.join(__dirname, fileName), "utf-8");

```

```javascript
const getWordsAmount = content => content.split(" ").length;

// Let's pretend this is doing an API call
const sendWordAmount = wordAmount => Promise.resolve({ statusCode: 200 });

(async () => {
  // Reading file (impure function)
  const fileContent = await getFileContent("file.txt");

  // Counting words (pure function)
  const wordAmount = getWordsAmount(fileContent);

  console.log(`File has ${wordAmount} words`);

  // Sending result to API (impure function)
  const response = await sendWordAmount(wordAmount);
  console.log("Sending to API", response);
})();
```

example1.js hosted with ❤ by **GitHub** **view raw**

Live Example: Here

This Node script is reading a text file, counting its words, and sending the result to an external API — well, not exactly an "actual" API, but let's pretend that it's real — There's only one pure function (counting words), the rest are impure. In this way, we can keep the impure parts of our code separate from the pure ones.

The past strategy works well. However, the function calls are made in an **imperative** form (instructions in a sequence). A more functional alternative is **to inject** impure functions into the pure ones. Let's see another example explaining this.

Let's suppose we want to generate random integer numbers within a range. Using the solution suggested by the Mozilla Documentation, you can do something like this:

```javascript
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min;
}
```

The problem with this solution is the impurity of the `Math.random` method. So in order to decouple this method from the `getRandomInt` function, we can do something like this:

```
function getRandomInt(min, max, random = Math.random) {
  return Math.floor(random() * (max - min)) + min;
}
```

That's it. We have decoupled our `getRandomInt` function. Notice how we're using ES6 default parameters so that we don't have to pass the `Math.random` reference every time we make a call. However, I know what you're thinking…

> You told me that if a pure function uses something impure, this one will become impure too.

Yes, you're right. This function will not behave in a pure way as long as the function passed is impure. However, this gives us a great advantage when running tests. Let's see this example using Jest:

```
function getRandomInt(min, max, random = Math.random) {
  return Math.floor(random() * (max - min)) + min;
}

describe("My Tests", () => {
  it("should generate a random number", () => {
    expect(getRandomInt(0, 10, () => 0.7)).toBe(7);
    expect(getRandomInt(0, 10, () => 0.6)).toBe(6);
    expect(getRandomInt(0, 10, () => 0.5)).toBe(5);
  });
});
```

randomTest.js hosted with ❤ by **GitHub** **view raw**

Run this using Jest

In the example above, we're overriding the default function to use the one we pass as the third parameter (an arrow function that only returns a value). By doing this, we are making the random function **deterministic**, and therefore **pure,** which makes it much easier for us to make test assertions.

4. Using High Order Functions

If you have been working with JS for a while, more than likely you have stumbled upon functions like map, filter, reduce, etc. These are good examples of High Order

Functions or **HOF**. They are functions that take other functions as parameters. They can either return a new function or a result based on the function that was passed to it.

I'll be describing map, filter, and other built-in HOF in the next part. But for now, we're going to focus on creating our own implementation of HOF.

Measuring Time

Let's say you want to log the time a function takes to complete. Your first idea might be to do something like this:

```
const calculateRectangleArea = (sideA, sideB) => sideA * sideB;

(() => {
  const startTime = Date.now();
  const rectangleArea = calculateRectangleArea(2, 3);
  const time = Date.now() - startTime;
  console.log(`Function calculateRectangleArea took ${time} to
complete`);
})();
```

This solution works well, but what if we wanted to measure the time of other functions? Would we have to repeat this code all over again? *No way!* So let's implement a HOF that takes any function as a parameter and generates a new function that logs the execution time. We are going to call it `addTiming`.

```
const logTime = (message, fname, startTime, endTime) =>
  console.log(`${fname}: ${message} - ${startTime - endTime}`);

const getCurrentTime = () => Date.now();

const addTiming = (fn, getTime = getCurrentTime, logFnTime = logTime) => (
  ...args
) => {
  const startTime = getCurrentTime();
  try {
    const valueToReturn = fn(...args);
    logFnTime("Normal execution", fn.name, startTime, getTime());
    return valueToReturn;
  } catch (err) {
    logFnTime("Error thrown", fn.name, startTime, getTime());
  }
};

```

```
// Example
const calculateRectangleArea = (sideA, sideB) => sideA * sideB;
addTiming(calculateRectangleArea)(2, 3);
// -> calculateRectangleArea: Normal execution - 0
```

addTiming.js hosted with ❤ by GitHub view raw

Live Example: Here

As you can see above, we're implementing a HOF that takes three parameters, but just one of them is mandatory (the first one). Which is the targeted function we want to wrap around, the second one is the function used to get the current time and the third one is the function used to log the time.

This function will first get the current time, then it will call the targeted function with its corresponding arguments — that we get using the spread operator — and right after that, it will execute and log the time that the targeted function takes to complete. Finally, it will return the corresponding result. There is also some error handling just in case the function doesn't end as expected.

Memoizing Functions

Memoizing (or caching results) is another interesting feature of HOF that usually is overlooked. As we already mentioned, when working with **Pure Functions**, we can be sure that for any given set of parameters, it will **always** return the same specific result. This means we can leverage to save those results in-memory so that we can use them in future calls later on. This **Memoization** technique is especially important when dealing with expensive calculations that take too much processing time to complete.

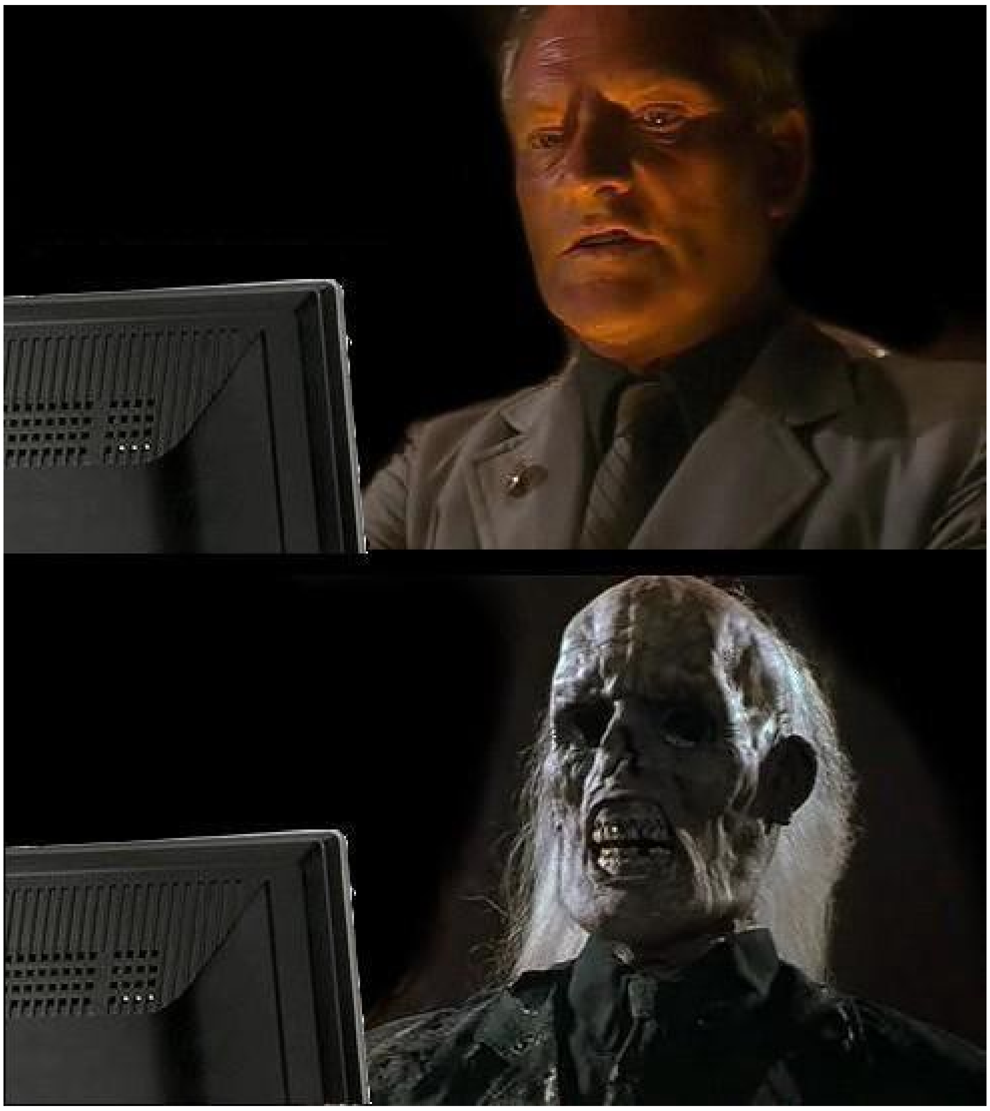
Let's take as an example the following Fibonacci function:

```
const fib = n => {
  if (n === 0) {
    return 0;
  } else if (n === 1) {
    return 1;
  } else {
    return fib(n - 2) + fib(n - 1);
  }
};
```

This Fibonacci function is using recursion and it will increase its execution time as it increases its input. We can use our recently created `addTiming` function to verify that:

```
addTiming(fib)(10); // fib: Normal execution - 0
addTiming(fib)(20); // fib: Normal execution - 1
addTiming(fib)(30); // fib: Normal execution - 11
addTiming(fib)(40); // fib: Normal execution - 1447
addTiming(fib)(50); // fib: Normal execution - 181611
```

The times shown above can vary from the ones you see in your PC, but unless you have a quantum computer — and I bet you don't — running this Fibonacci function with an input of 50 will take a good time to complete.


Waiting for the Fibonacci(50) function to complete be like...

So how can we optimize this? By using memoization of course. But instead of implementing our own memoizing solution — and reinventing the wheel — we're going to use the HOF that Lodash provides.

> *NOTE: If you have never used Lodash, you can check its documentation here.*

The complete solution with a live example is found below:

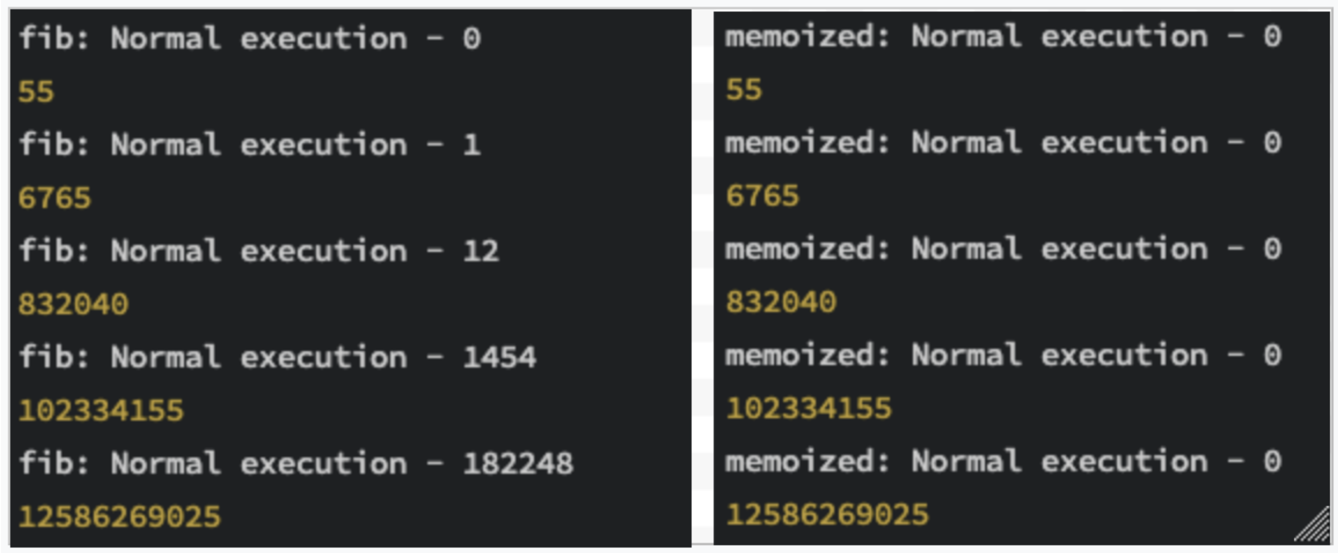
```
const _ = require("lodash");

const logTime = (message, fname, startTime, endTime) =>
  console.log(`${fname}: ${message} - ${endTime - startTime} ms`);

const getCurrentTime = () => Date.now();

const addTiming = (fn, getTime = getCurrentTime, logFnTime = logTime) => (
  ...args
) => {
  const startTime = getCurrentTime();
  try {
    const valueToReturn = fn(...args);
    logFnTime("Normal execution", fn.name, startTime, getTime());
    return valueToReturn;
  } catch (err) {
    logFnTime("Error thrown", fn.name, startTime, getTime());
  }
};

// fib cannot be constant anymore because we need to overwrite its reference
let fib = n => {
  if (n === 0) {
    return 0;
  } else if (n === 1) {
    return 1;
  } else {
    return fib(n - 2) + fib(n - 1);
  }
};

fib = _.memoize(fib);

console.log(addTiming(fib)(10));
console.log(addTiming(fib)(20));
console.log(addTiming(fib)(30));
console.log(addTiming(fib)(40));
console.log(addTiming(fib)(50));
```

memoizingFib.js hosted with ❤ by GitHub view raw

Live Example: Here

As you can see, we are using `let` instead of `const` to define the Fibonacci function. We are doing this because we need the original function reference to be overwritten by the memoized call. Otherwise, it won't work correctly.

Let's compare side by side both execution outputs. I ran these examples using CodeSandbox.io:



```
fib: Normal execution - 0
55
fib: Normal execution - 1
6765
fib: Normal execution - 12
832040
fib: Normal execution - 1454
102334155
fib: Normal execution - 182248
12586269025
```

```
memoized: Normal execution - 0
55
memoized: Normal execution - 0
6765
memoized: Normal execution - 0
832040
memoized: Normal execution - 0
102334155
memoized: Normal execution - 0
12586269025
```

Left: Unaltered Fibonacci output. Right: Memoized Fibonacci output

As you can see, the memoized function takes almost no time (less than 1 ms) to complete for each input used. Whereas the unaltered function takes up to 182,248 milliseconds (more than 2 minutes). It's a very striking difference indeed.

5. Avoid loops by working Declaratively

As I mentioned in the previous part, Javascript already counts with a series of built-in high order functions (HOF). Most of these functions are used to handle Arrays or object collections.

If you are reading this article, chances are you are already an experienced JS developer. So I will not take too much time explaining each one of them in detail — *that would be boring as hell.*

The functions I'll be describing are:

- Reduce
- Map
- ForEach
- Filter

- Find
- Every and Some

> *NOTE: Skip to part 6 if you already know how all of these methods work.*

Calculating results with the Reduce method

Let's suppose you have an array of numbers and you want to calculate its average. The best functional way to achieve this is by using the built-in HOF `reduce`. For example:

```
const getAverage = myArray =>
  myArray.reduce((sum, val, ind, arr) => {
    sum += val;
    return ind === arr.length - 1 ? sum / arr.length : sum;
  }, 0);

console.log("Average:", getAverage([22, 9, 60, 12, 4, 56]));
// Average: 27.166666666666668
```

> *The* `reduce()` *method executes a* ***reducer*** *function (that you provide) on each element of the array, resulting in a single output value. (Mozilla Reference)*

So in the example above, we are implementing a function that receives an array of numbers. Then, we proceed to apply the reduce method upon this array and pass the (reducer) function that will sum all the array values. Once the last one is reached, we will just return the division (average) using the array length.

Creating new arrays with the Map method

Let's suppose you have the following array of objects containing the latitude and longitude of several countries.

```
const markers = [
  { name: "UY", lat: -34.9, lon: -56.2 },
  { name: "AR", lat: -34.6, lon: -58.4 },
  { name: "BR", lat: -15.8, lon: -47.9 },
  { name: "BO", lat: -16.5, lon: -68.1 }
];
```

Now let's suppose we want to create an Array with only the latitude values. This is easily achievable using the `map` method like so:

```
console.log("Lat values:", markers.map(x => x.lat));
// Lat values: [ -34.9, -34.6, -15.8, -16.5 ]
```

> *The `map()` method **creates a new array** populated with the results of calling a provided function on every element in the calling array. (Mozilla Reference)*

In the example above, the provided function only returns the latitude values.

Looping using the ForEach method

Sometimes, the only thing we want to do is to loop through a series of values or objects. We can use `forEach` in such situations. Example, let’s suppose we want to log all the markers array data:

```
const logMarkersData = markers => {
  console.log("Data provided:");

markers.forEach(marker => console.log(`Country: ${marker.name}
Latitude: ${marker.lat} Longitude: ${marker.lon}`));
};

logMarkersData(markers);
/*
Data provided:
Country: UY Latitude: -34.9 Longitude: -56.2
Country: AR Latitude: -34.6 Longitude: -58.4
Country: BR Latitude: -15.8 Longitude: -47.9
Country: BO Latitude: -16.5 Longitude: -68.1
*/
```

> *The `forEach()` method executes a provided function once for each array element. (Mozilla Reference)*

This example is quite easy to understand. The `forEach` method will iterate over each element and log the accessed data.

Filtering array elements with the Filter method

Let’s continue using the same markers array we used before. Suppose we want to filter the data from countries whose initial letter is “B”. The `filter` method can help you with that:

```
console.log(
  "Data of countries starting with B:",
  markers.filter(mark => mark.name.charAt(0) === "B")
);

/*
Data of countries starting with B: [ { name: 'BR', lat: -15.8, lon:
-47.9 }, { name: 'BO', lat: -16.5, lon: -68.1 } ]
*/
```

> *The* `filter()` *method* ***creates a new array*** *with all elements that pass the test implemented by the provided function. (Mozilla Reference)*

Finding a specific element using the Find and FindIndex method

Now the situation is even more specific. We only require the data from Brasil (BR). So we use the `find` method:

```
console.log("Brazil Data:", markers.find(m => m.name === "BR"));

// Brazil Data: { name: 'BR', lat: -15.8, lon: -47.9 }
```

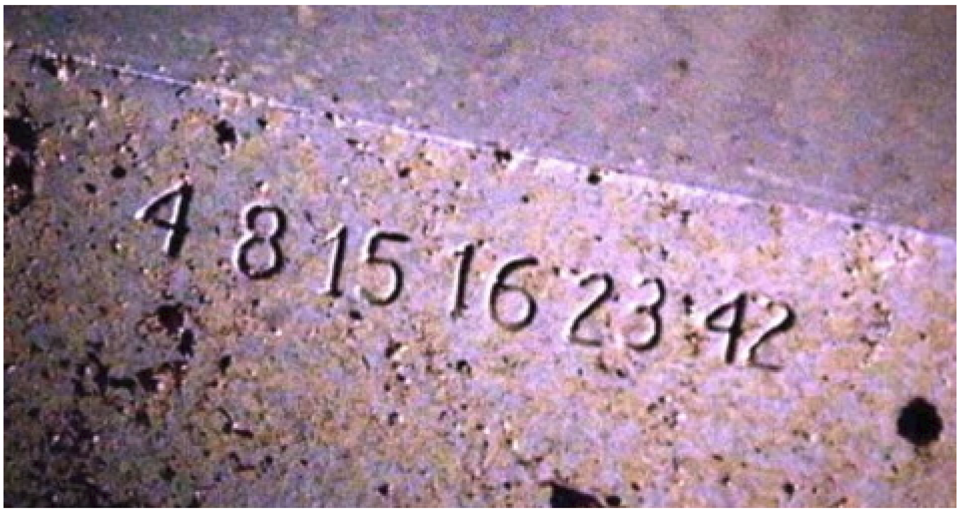
> *The* `find()` *method returns the* ***value*** *of the* ***first element*** *in the provided array that satisfies the provided testing function. (Mozilla Reference)*

But what if you wanted to only know the index? We would use the `findIndex` method instead.

```
console.log("Brazil Data Index:", markers.findIndex(m => m.name ===
"BR"));

// Brazil Data Index: 2
```

> *The* `findIndex()` *method returns the* ***index*** *of the first element in the array* ***that satisfies the provided testing function****. Otherwise, it returns -1, indicating that no element passed the test. (Mozilla Reference)*

Chaining logic operations with the Every and Some method



Do you remember Lost?

I came across these two methods very recently. They are especially useful when you need to determine whether or not all the elements from an array meet some specific logic. It's equivalent to using the AND/OR operator in sequence. So instead of doing something like this:

```
if (arr[0] > 0 && arr[1] > 0 ..... arr[n] > 0) {...}
```

You can do something like this:

```
if (arr.every(item => item > 0)) {...}
```

The same applies to the OR operator but this time using the `some` method instead.

Let's see this through an example. Imagine we have a sequence of numbers (e.g: 4, 8, 15, 16, 23 and 42) and we would like to determine whether there is at least one **even** number or not. For this case, we need to use the `some` method:

```
const lostNumbers = [4, 8, 15, 16, 23, 42];

console.log(
  "Does it contain even numbers?",
  lostNumbers.some(n => n % 2 === 0)
);

// Does it contain even numbers? true
```

> *The* **`some()`** *method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.*

In the other hand, if we wanted to determine whether or not all the array numbers are even, we would need to use the `every` method like so:

```
console.log("Are all even numbers?", lostNumbers.every(n => n % 2
=== 0));

// Are all even numbers? false
```

> *The* **`every()`** *method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.*

6. Using Function Composition to combine functions

Ideally, our functions should be small. They should be capable to only do one thing and do it right. So how can we combine all of these "small" functions to work together? One way to achieve this is through **Function Composition**. This sounds like a *fancy mathematical concept* and somehow — it is. However, it is not that hard to understand. It consists of **making the result of one function be the parameter of the next one**.

Function Composition comes in two flavors: **Pipelining and Composing**. They are pretty much the same, *but different*. One works from left to right whereas the other one works in the opposite direction.

Pipelining and Composing in a nutshell

Pipelining

> *NOTE: In Federico Kereki's book, he teaches how to implement a "pipeline" function from scratch. But here we are going to use the solution(s) provided by* ***Lodash****.*

With Lodash, we have two alternatives. The first one, by using the method `flow` from the standard API and the second one, by using the method `pipe` from the FP API. Let's explore an example in both ways.

Suppose we need to connect to a legacy web service (SOAP WS or similar) that still uses XML as a data format. This WS is providing some books information like the one below:

```xml
<?xml version="1.0"?>
<Library>
    <Book>
        <Author>Sam Newman</Author>
        <Title>Building Microservices: Designing Fine-Grained Systems</Title>
        <Year>2015</Year>
        <Price>30.38</Price>
    </Book>
    <Book>
        <Author>Federico Kereki</Author>
        <Title>Mastering JavaScript Functional Programming: In-depth guide for writing
        <Year>2017</Year>
        <Price>22.39</Price>
    </Book>
    <Book>
        <Author>Robert C. Martin </Author>
        <Title>Clean Code: A Handbook of Agile Software Craftsmanship</Title>
        <Year>2008</Year>
        <Price>26.39</Price>
    </Book>
</Library>
```

books.xml hosted with ❤ by **GitHub** **view raw**

Sample of books WS response

Our task is the following. Take the response above and:

- Organize books by price (ascending order) and…
- Print the information in JSON format (using `console.log`)

It turns out that (in JS) dealing with data in XML format is a complete *pain in the a***. Lucky for us, there are several libraries that we can use to translate XML into a handy JS object. In this example, we're going to use one of them called xml-js.

First, let's get the data from the "web service". Unfortunately, we don't have one on hand. So let's fake it:

```
// Let's pretend this is a Webservice call
const getingWsData = () =>
   Promise.resolve(fs.readFileSync(`${__dirname}/books.xml`,
"utf8"));
```

Next, let's extract the required information from the XML response. We're going to use a utility Lodash function called `get`, to easily obtain values by providing an object path, the `map` method, which we already described in the previous part and the `xml2js` library method to parse the initial XML text.

```
const transformToJsObject = xmlData =>
  _(xmlJs.xml2js(xmlData, { compact: true })) // Transforming to JS
    .get("Library.Book") // Getting Books reference
    .map(book => ({ // Interating books array to generate new one
      author: book.Author._text,
      title: book.Title._text,
      year: Number(book.Year._text),
      price: Number(book.Price._text)
    }));
```

> *NOTE: I'm not including any validation upon the XML data. So let's assume it is well-formed and the year and price values are numeric. But you probably want to double-check this for a production app.*

Finally, let's sort by price amount. Here, we could use the built-in `sort` method (reference here). But that one mutates the targeted array which makes our function **impure**. So we'd better use the Lodash equivalent `sortBy` as follows:

```
const sortByPrice = booksArray => _.sortBy(booksArray, "price");
```

We have already defined all the required functions to do the assigned task. But how do we connect them together? Your first guess might be to do it in the well-known imperative form:

```
let booksArray = transformToJsObject(xmlData);
booksArray = sortByPrice(booksArray);
const jsonData = JSON.stringify(booksArray);
console.log(jsonData);
```

It works just fine, but it takes too many lines! Besides, notice how many intermediate variables/constants we need to use. You might have the idea to simplify everything by doing something within one single line like this:

```
console.log(JSON.stringify(sortByPrice(transformToJsObject(xmlData))
));
```

But now everything turns into an *un-readable parenthesis hell*. So let's apply the pipelining technique. First with the standard Lodash `flow` method:

```
_.flow(
   transformToJsObject,
   sortByPrice,
   JSON.stringify,
   console.log
)(xmlData);
```

And now with the FP Lodash `pipe` :

```
fp.pipe(
   transformToJsObject,
   sortByPrice,
   JSON.stringify,
   console.log
)(xmlData);
```

As you can see both high-order functions are pretty much identical. Both receive their arguments in the same order, which is left-to-right, and the most left function would be the one that is called at first. Here you have the full example:

```
const _ = require("lodash");
const fp = require("lodash/fp");
const xmlJs = require("xml-js");
const fs = require("fs");

// Let's pretend this is a Webservice call
const getingWsData = () =>
  Promise.resolve(fs.readFileSync(`${__dirname}/books.xml`, "utf8"));

const transformToJsObject = xmlData =>
  _(xmlJs.xml2js(xmlData, { compact: true }))
    .get("Library.Book") // Let's assume the XML is validated
    .map(book => ({
      author: book.Author._text,
      title: book.Title._text,
      year: Number(book.Year._text),
      price: Number(book.Price._text)
    }));

const sortByPrice = booksArray => _.sortBy(booksArray, "price");

(async () => {
  const xmlData = await getingWsData(); // This function is impure so it's out of the

  console.log("\nImperative calls... eww!!");

  let booksArray = transformToJsObject(xmlData);
  booksArray = sortByPrice(booksArray);
  const jsonData = JSON.stringify(booksArray);
  console.log(jsonData);

  console.log("\nPipelining using flow (Lodash standard API)");
  _.flow(
    transformToJsObject,
    sortByPrice,
    JSON.stringify,
    console.log
  )(xmlData);

  console.log("\nPipelining using pipe (Lodash FP API)");
  fp.pipe(
    transformToJsObject,
    sortByPrice,
    JSON.stringify,
    console.log
  )(xmlData);
})();
```

pipelining is hosted with ❤ by GitHub view raw

Live Example: Here

Composing

As I mentioned before, composing works in a very similar way as pipelining. So let's take a look at the same example but this time using `compose` and `flowRight`.

```
const _ = require("lodash");
const fp = require("lodash/fp");
const xmlJs = require("xml-js");
const fs = require("fs");

// Let's pretend this is a Webservice call
const getingWsData = () =>
  Promise.resolve(fs.readFileSync(`${__dirname}/books.xml`, "utf8"));

const transformToJsObject = xmlData =>
  _(xmlJs.xml2js(xmlData, { compact: true }))
    .get("Library.Book") // Let's assume the XML is validated
    .map(book => ({
      author: book.Author._text,
      title: book.Title._text,
      year: Number(book.Year._text),
      price: Number(book.Price._text)
    }));

const sortByPrice = booksArray => _.sortBy(booksArray, "price");

(async () => {
  const xmlData = await getingWsData(); // This function is impure so it's out of the

  console.log("\nImperative calls... eww!!");

  let booksArray = transformToJsObject(xmlData);
  booksArray = sortByPrice(booksArray);
  const jsonData = JSON.stringify(booksArray);
  console.log(jsonData);

  console.log("\nComposing using flowRight (Lodash standard API)");
  _.flowRight(
    console.log,
    JSON.stringify,
    sortByPrice
```

```
      sortByPrice,
      transformToJsObject
    )(xmlData);

    console.log("\nComposing using compose (Lodash FP API)");
    fp.compose(
      console.log,
      JSON.stringify,
      sortByPrice,
      transformToJsObject
    )(xmlData);
  })();
```

composing is hosted with ❤ by GitHub view raw

Live Example: Here

As you can see, the `flowRight` and `compose` methods work similarly to `flow` and `pipe`. With the only difference being that the order of the parameters is now inverted (right-to-left). But the output result is exactly the same as before.

Having seen this, you might wonder: **Which one should I use?** And the answer is *The one you prefer!* In my personal opinion, I rather use pipelining since we usually read from left to right so it's more natural for me to go in that direction. But other people may prefer to use composing since the initial parameter is closer to the function that receives it. Whichever option you use, is fine as long as you understand what you are doing.

7. Reusability with Currying

In the past part, we saw how we can combine functions to work together using Function Composition. Now let me ask you a question about it:

> Did you notice a particular characteristic of those functions?

I'm talking about the functions used within the pipelines and composers. Did you notice anything about them? Well, it turns out they always had one single argument. Since the previous function could only have one single return value. So in order to make our functions work with pipelines or composers, they are required to be **unary** (one argument) by default. Currying is a simple way to transform non-unary functions to work in a unary form.

Just to clarify, I'm not talking about this kind of curry…

Let's see this throughout an example. Here you can see a function that adds three numbers, nothing new:

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

A **curried** version of this function would be something like this:

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

Now we have a function that returns a function that returns **another** function. In order to call this curried version of "sum", you could do this:

```
sum(1)(2)(3); // Result is 6
```

I know what you're thinking… *What the hell do we gain from this?* And the answer is simple: **You can create curried versions of your functions if their initial**

parameters don't change throughout multiple calls. For example, let's say you have a function that calculates discounts given a certain percentage. It would look like this:

```
comesconst calculateDiscount = (discount, price) => price - (price *
discount) / 100;

console.log(calculateDiscount(10, 2000)); // 1800
```

Now, let's imagine a situation where we need to apply always the same discount to different products (with different prices). That would mean to repeat multiple times the same first parameter:

```
console.log(calculateDiscount(10, 2000)); // 1800

console.log(calculateDiscount(10, 500)); // 450

console.log(calculateDiscount(10, 750)); // 675

console.log(calculateDiscount(10, 900)); // 810
```

Probably it's not a big deal to repeat one single parameter multiple times, but imagine situations where the number of repeated parameters is higher (2, 4, 5, etc) So let's apply currying to this `calculateDiscount` function. For that, we are going to use the `curry` method from Lodash:

```
const _ = require("lodash");

const calculateDiscount = (discount, price) => price - (price * discount) / 100;
const apply10percDiscount = _.curry(calculateDiscount)(10);

console.log(apply10percDiscount(1000)); // 900
console.log(apply10percDiscount(500)); // 450
console.log(apply10percDiscount(750)); // 675
console.log(apply10percDiscount(900)); // 810
```

curry.js hosted with ❤ by GitHub view raw

Live Example: Here

As you can see above, we applied currying to the `calculateDiscount` function and at the same time, we passed the first parameter (discount percentage) to the resulting function. Now we have a new function whose only purpose is to calculate a 10% discount on any given price.

Of course, this is not the only practical case where you can use currying. Think of situations where you have to create multiple listeners to one single event. You can apply currying to the function responsible to add listeners and use that (curried) function as you require it. Like this, there are many other examples.

8. Reusability with Partial Application

Currying is useful when our constant parameters come in order. But sometimes this is not the case and we have to deal with situations where the fixed parameters don't follow any order in particular. Here it's where the **Partial Application** technique comes handy.

Let's check a new example to see how this works. Suppose you have a simple Winston logger instance like the one next:

```
const logger = winston.createLogger({
  transports: [new winston.transports.Console()]
});
```

> *NOTE: Winson is a library to manage NodeJS logs easily. If you want to know more about it, check its github repo here.*

Now let's suppose we are developing a system with multiple steps — like a pipeline. And we would like to log messages corresponding to a specific step, maybe because we want to provide some "trackability" for our system. For example: `step: "Order Preparation", message: "Assigning Carrier to order XXX"` . Let's define a function that does just that:

```
const logMessage = (level, message, step, loggerFn = logger) =>
  loggerFn[level]({ message, step });
```

As you can see, we are injecting the logger instance into the `logMessage` function. Now we can start logging message by calling this function like this:

```
logMessage("info", "Assigning Carrier to order XXX", "Order
Preparation")
```

The problem with this approach is that if we have to log multiple messages at the same step and with the same (info) level, we are going to need to repeat multiple times the same two parameters `"info"` and `"Order Preparation"` . So let's apply partial application (using Lodash) to fix those two parameters.

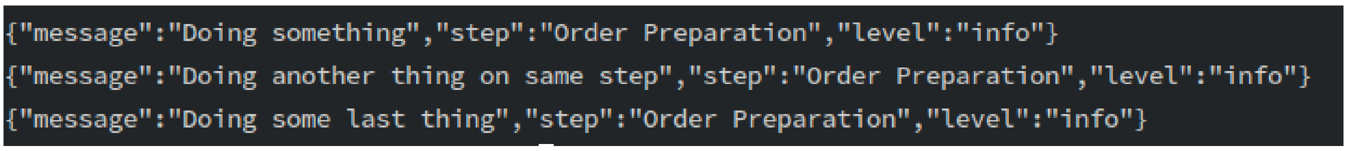
```
const loggerStepInfo = _.partial(logMessage, "info", _, "Order
Preparation");
```

The `_` is used by Lodash to indicate a "placeholder" parameter, which (in this case) is the only one that is changeable. However, we could set multiple placeholder parameters if we wanted it. This is one of the main differences between currying and partial application. **The resulting function doesn't have to be unary.**

Now let's test our resulting function with some sample calls:

```
loggerStepInfo("Doing something");
loggerStepInfo("Doing another thing on same step");
loggerStepInfo("Doing some last thing");
```

The output should be something like this:



```
{"message":"Doing something","step":"Order Preparation","level":"info"}
{"message":"Doing another thing on same step","step":"Order Preparation","level":"info"}
{"message":"Doing some last thing","step":"Order Preparation","level":"info"}
```

Here, it's the full code with its live example:

```
const winston = require("winston");
const _ = require("lodash");

// Creating an instance of winston
const logger = winston.createLogger({
  transports: [new winston.transports.Console()]
});

```

```javascript
// Defining a log message function
const logMessage = (level, message, step, loggerFn = logger) =>
  loggerFn[level]({ message, step });

// level and step parameters are fixed. Message is the only one to be used
const loggerStepInfo = _.partial(logMessage, "info", _, "Order Preparation");
// Examples:
loggerStepInfo("Doing something");
loggerStepInfo("Doing another thing on same step");
loggerStepInfo("Doing some last thing");
```

partial.js hosted with ❤ by **GitHub** **view raw**

Live example: Here

. . .

Conclusion

Working with Functional Programming is not a silver bullet. Your code is not going to be (or work) better if it is written with a Functional Style. So it's up to you to decide whether or not to use any FP technique you have seen here or in any other place.

In my humble opinion, when it comes to designing and implementing complex algorithms or performing complex calculations that don't require accessing external resources (files, databases, API), I like to do it in a functional and declarative way. Because I can easily unit-test and decouple everything. But If I have to deal intensively with external resources, I will go with the well-know imperative form. *Again, This is just me! You might have a different thought!*

I think the key element here, it's to separate the impure parts of your project from the pure ones. So that you can easily decouple your code and apply FP techniques when you find it convenient.

In this article, I didn't mention other FP related concepts like recursion optimization, monads, functors, etc. I found them quite advanced to understand and with few practical use cases, so I decided to skip them. If you are still curious about them, I will suggest you (again) to read the Federico Kereki's Book **Mastering JavaScript Functional Programming.**

Happy Codding!

Functional Programming JavaScript High Order Functions Web Development Programming