An introduction to Functional Programming with JavaScript

Getting started with the main concepts of Functional Programming in the JavaScript Programming Language

Published Feb 23, 2018

Introduction to Functional Programming

Functional Programming (FP) is a programming paradigm with some particular techniques.

In programming languages, you'll find purely functional programming languages as well as programming languages that support functional programming techniques.

Haskell, Clojure and Scala are some of the most popular purely functional programming languages.

Popular programming languages that support functional programming techniques are JavaScript (https://flaviocopes.com/javascript/) , Python, Ruby and many others.

Functional Programming is not a new concept, actually its roots go back o the 1930's when lamda calculus was born, and has influenced many programming languages.

FP has been gaining a lot of momentum lately, so it's the perfect time to learn about it.

In this course I'll introduce the main concepts of Functional Programming, by using in the code examples JavaScript.

First class functions

In a functional programming language, functions are first class citizens.

They can be assigned to variables

```
const f = (m) => console.log(m)
f('Test')
```

Since a function is assignable to a variable, they can be added to objects:

```
const obj = {
  f(m) {
    console.log(m)
  }
}
obj.f('Test')
```

as well as to arrays:

```
const a = [
  m => console.log(m)
]
a[0]('Test')
```

They can be used as an argument to other functions

```
const f = (m) => () => console.log(m)
const f2 = (f3) => f3()
f2(f('Test'))
```

They can be returned by functions

```
const createF = () => {
  return (m) => console.log(m)
}
const f = createF()
f('Test')
```

Higher Order Functions

Functions that accept functions as arguments or return functions are called **Higher Order Functions**.

Examples in the JavaScript standard library include `Array.map()` , `Array.filter()` and `Array.reduce()` , which we'll see in a bit.

Declarative programming

You may have heard the term "declarative programming".

Let's put that term in context.

The opposite of *declarative* is **imperative**.

Declarative vs Imperative

An imperative approach is when you tell the machine (in general terms), the steps it needs to take to get a job done.

A declarative approach is when you tell the machine what you need to do, and you let it figure out the details.

You start thinking declarative when you have enough level of abstraction to stop reasoning about low level constructs, and think more at a higher UI level.

One might argue that C programming is more declarative than Assembly programming, and that's true.

HTML is declarative, so if you've been using HTML since 1995, you've actually being building declarative UIs since 20+ years.

JavaScript can take both an imperative and a declarative programming approach.

For example a declarative programming approach is to avoid using loops (https://flaviocopes.com/javascript-loops/) and instead use functional programming constructs like `map` , `reduce` and `filter` , because your programs are more abstract and less focused on telling the machine each step of processing.

Immutability

In functional programming data never changes. Data is **immutable**.

A variable can never be changed. To update its value, you create a new variable.

Instead of changing an array, to add a new item you create a new array by concatenating the old array, plus the new item.

An object is never updated, but copied before changing it.

`const`

This is why the ES2015 `const` is so widely used in modern JavaScript, which embraces functional programming concepts: to **enforce** immutability on variables.

`Object.assign()`

ES2015 also gave us `Object.assign()` (https://flaviocopes.com/javascript-object-assign/) , which is key to creating objects:

```
const redObj = { color: 'red' }
const yellowObj = Object.assign({}, redObj, {color: 'yellow'})
```

`concat()`

To append an item to an array in JavaScript we generally use the `push()` method on an array, but that method mutates the original array, so it's not FP-ready.

We instead use the `concat()` method:

```
const a = [1, 2]
const b = [1, 2].concat(3)
// b = [1, 2, 3]
```

or we use the **spread operator** (https://flaviocopes.com/javascript-spread-operator/) :

```
const c = [...a, 3]
// c = [1, 2, 3]
```

`filter()`

The same goes for removing an item from an array: instead of using `pop()` and `splice()`, which modify the original array, use `array.filter()`:

```
const d = a.filter((v, k) => k < 1)
// d = [1]
```

Purity

A **pure function**:

- never changes any of the parameters that get passed to it by reference (in JS, objects and arrays): they should be considered immutable. It can of course change any parameter copied by value
- the return value of a pure function is not influenced by anything else than its input parameters: passing the same parameters always result in the same output
- during its execution, a pure function does not change anything outside of it

Data Transformations

Since immutability is such an important concept and a foundation of functional programming, you might ask how can data change.

Simple: **data is changed by creating copies**.

Functions, in particular, change the data by returning new copies of data.

Core functions that do this are **map** and **reduce**.

`Array.map()`

Calling `Array.map()` on an array will create a new array with the result of a function executed on every item of the original array:

```
const a = [1, 2, 3]
const b = a.map((v, k) => v * k)
// b = [0, 2, 6]
```

`Array.reduce()`

Calling `Array.reduce()` on an array allows us to transform that array on anything else, including a scalar, a function, a boolean, an object.

You pass a function that processes the result, and a starting point:

```
const a = [1, 2, 3]
const sum = a.reduce((partial, v) => partial + v, 0)
// sum = 6
```

```
const o = a.reduce((obj, k) => { obj[k] = k; return obj }, {})
// o = {1: 1, 2: 2, 3: 3}
```

Recursion

Recursion is a key topic in functional programming. when **a function calls itself**, it's called a *recursive function*.

The classic example of recursion is the Fibonacci sequence (N = (N-1 + N-2)) calculation, here in its 2^N totally inefficient (but nice to read) solution:

```
var f = (n) => n <= 1 ? 1 : f(n-1) + f(n-2)
```

Composition

Composition is another key topic of Functional Programming, a good reason to put it into the “key topics” list.

Composition is how we generate a higher order function, by combining simpler functions.

Composing in plain JS

A very common way to compose functions in plain JavaScript is to chain them:

```
obj.doSomething()
   .doSomethingElse()
```

or, also very widely used, by passing a function execution into a function:

```
obj.doSomething(doThis())
```

Composing with the help of `lodash`

More generally, composing is the act of putting together a list of many functions to perform a more complicated operation.

`lodash/fp` comes with an implementation of `compose`: we execute a list of functions, starting with an argument, **each function inherits the argument from the preceding function return value**. Notice how we don’t need to store intermediate values anywhere.

```
import { compose } from 'lodash/fp'

const slugify = compose(
  encodeURIComponent,
  join('-'),
  map(toLowerCase),
  split(' ')
)
```

```
slufigy('Hello World') // hello-world
```