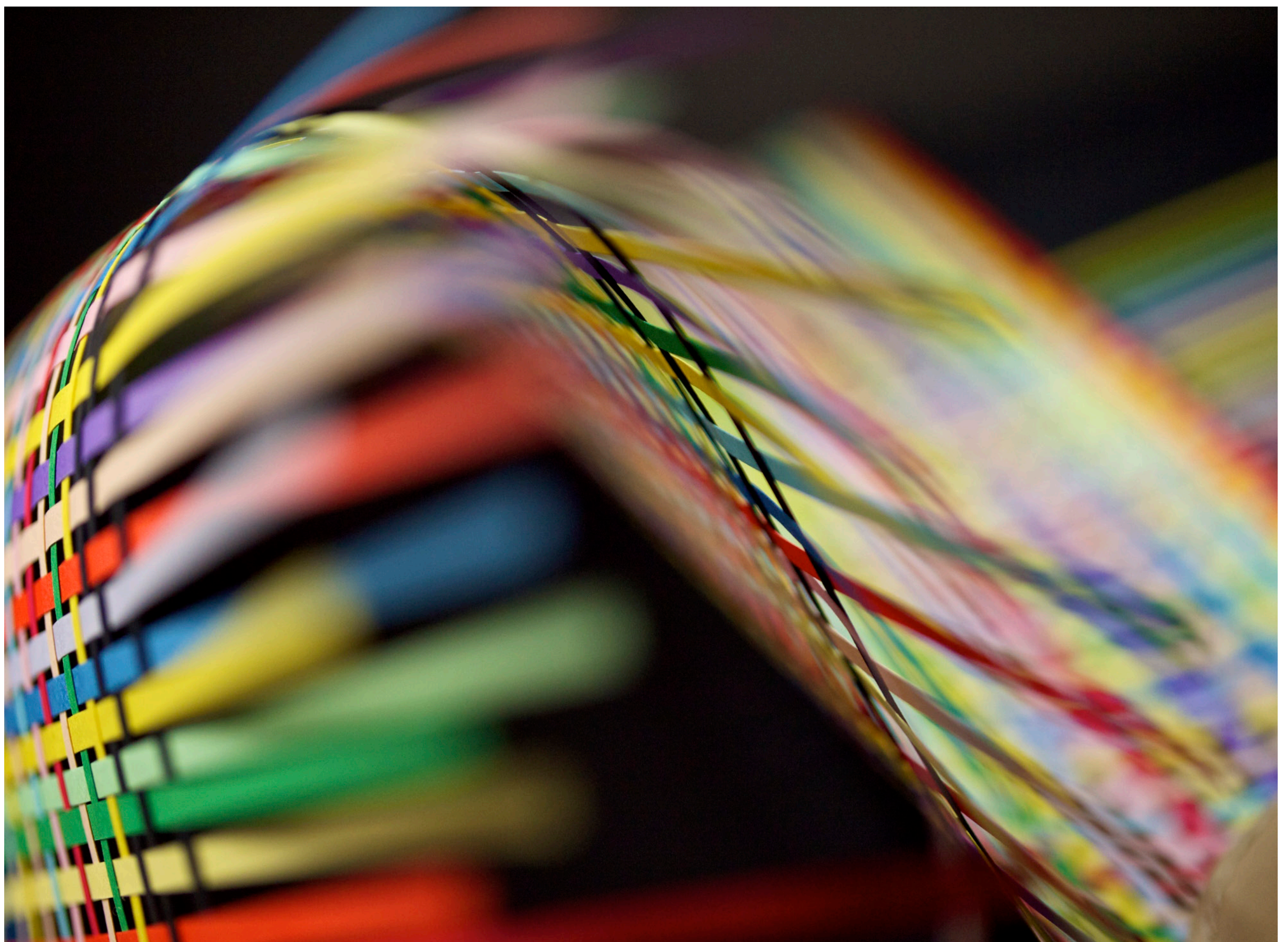
O'REILLY®

/ theory / in / practice

Beautiful JavaScript

Leading Programmers Explain How They Think

Anton Kovalyov



Beautiful JavaScript

Edited by Anton Kovalyov

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

CHAPTER FOURTEEN

Functional JavaScript

Anton Kovalyov

Is JavaScript a functional programming language? This question has long been a topic of great debate within our community. Given that JavaScript's author was recruited to do "Scheme in the browser," one could argue that JavaScript was designed to be used as a functional language. On the other hand, JavaScript's syntax closely resembles Java-like object-oriented programming, so perhaps it should be utilized in a similar manner. Then there might be some other arguments and counterarguments, and the next thing you know the day is over and you didn't do anything useful.

This chapter is not about functional programming for its own sake, nor is it about altering JavaScript to make it resemble a pure functional language. Instead, this chapter is about taking a pragmatic approach to functional programming in JavaScript, a method by which programmers use elements of functional programming to simplify their code and make it more robust.

Functional Programming

Programming languages come in several varieties. Languages like Go and C are called *procedural*: their main programming unit is the procedure. Languages like Java and SmallTalk are object oriented: their main programming unit is the object. Both these approaches are imperative, which means they rely on commands that act upon the machine state. An imperative program executes a sequence of commands that change the program's internal state over and over again.

Functional programming languages, on the other hand, are oriented around expressions. Expressions—or rather, pure expressions—don't have a state, as they merely compute a value. They don't change the state of something outside their scope, and they don't rely on data that can change outside their scope. As a result, you should be

able to substitute a pure expression with its value without changing the behavior of a program. Consider an example:

```
function add(a, b) {
  return a + b
}

add(add(2, 3), add(4, 1)) // 10
```

To illustrate the process of substituting expressions, let's evaluate this example. We start with an expression that calls our `add` function three times:

```
add(add(2, 3), add(4, 1))
```

Since `add` doesn't depend on anything outside its scope, we can replace all calls to it with its contents. Let's replace the first argument that is not a primitive value—`add(2, 3)`:

```
add(2 + 3 , add(4, 1))
```

Then we replace the second argument:

```
add(2 + 3, 4 + 1)
```

Finally, we replace the last remaining call to our function and calculate the result:

```
(2 + 3) + (4 + 1) // 10
```

This property that allows you to substitute expressions with their values is called *referential transparency*. It is one of the essential elements of functional programming.

Another important element of functional programming is *functions as first-class citizens*. Michael Fogus gave a great explanation of functions as first-class citizens in his book, *Functional JavaScript*. His definition is one of the best I've seen:

> The term "first-class" means that something is just a value. A first-class function is one that can go anywhere that any other value can go—there are few to no restrictions. A number in JavaScript is surely a first-class thing, and therefore a first-class function has a similar nature:
>
> - A number can be stored in a variable and so can a function:
>
>   ```
>   var fortytwo = function() { return 42 };
>   ```
>
> - A number can be stored in an array slot and so can a function:
>
>   ```
>   var fortytwos = [42, function() { return 42 }];
>   ```
>
> - A number can be stored in an object field and so can a function:
>
>   ```
>   var fortytwos = {number: 42, fun: function() { return 42 }};
>   ```
>
> - A number can be created as needed and so can a function:
>
>   ```
>   42 + (function() { return 42 })(); // => 84
>   ```

- A number can be passed to a function and so can a function:

```
function weirdAdd(n, f) { return n + f() }

weirdAdd(42, function() { return 42 }); // => 84
```

- A number can be returned from a function and so can a function:

```
return 42;

return function() { return 42 };
```

Having functions as first-class citizens enables another important element of functional programming: higher-order functions. A *higher-order function* is a function that operates on other functions. In other words, higher-order functions can take other functions as their arguments, return new functions, or do both. One of the most basic examples is a higher-order `map` function:

```
map([1, 2, 3], function (n) { return n + 1 }) // [2, 3, 4]
```

This function takes two arguments: a collection of values and another function. Its result is a new list with the provided function applied to each element from the list.

Note how this `map` function uses all three elements of functional programming described previously. It doesn't change anything outside of its scope, nor does it use anything from the outside besides the values of its arguments. It also treats functions as first-class citizens by accepting a function as its second argument. And since it uses that argument to compute the value, one can definitely call it a higher-order function.

Other elements of functional programming include recursion, pattern matching, and infinite data structures, although I will not elaborate on these elements in this chapter.

Functional JavaScript

So, is JavaScript a truly functional programming language? The short answer is no. Without support for tail-call optimization, pattern matching, immutable data structures, and other fundamental elements of functional programming, JavaScript is not what is traditionally considered a truly functional language. One can certainly try to treat JavaScript as such, but in my opinion, such efforts are not only futile but also dangerous. To paraphrase Larry Paulson, author of the *Standard ML for the Working Programmer*, a programmer whose style is "almost" functional had better not be lulled into a false sense of referential transparency. This is especially important in a language like JavaScript, where one can modify and overwrite almost everything under the sun.

Consider `JSON.stringify`, a built-in function that takes an object as a parameter and returns its JSON representation:

```
JSON.stringify({ foo: "bar" }) // -> "{"foo":"bar"}"
```

One might think that this function is pure, that no matter how many times we call it or in what context we call it, it always returns the same result for the same arguments. But what if somewhere else, most probably in code you don't control, someone overwrites the `Object.prototype.toJSON` method?

```
JSON.stringify({ foo: "bar" })
// -> "{"foo":"bar"}"

Object.prototype.toJSON = function () {
  return "reality ain't always the truth"
}

JSON.stringify({ foo: "bar" })
// -> ""reality ain't always the truth""
```

As you can see, by slightly modifying a built-in `Object`, we managed to change the behavior of a function that looks pretty pure and functional from the outside. Functions that read mutable references and properties aren't pure, and in JavaScript, most nontrivial functions do exactly that.

My point is that functional programming, especially when used with JavaScript, is about reducing the complexity of your programs and not about adhering to one particular ideology. Functional JavaScript is not about eliminating all the mutations; it's about reducing occurrences of such mutations and making them very explicit. Consider the following function, `merge`, which merges together two arrays by pairing their corresponding members:

```
function merge(a, b) {
  b.forEach(function (v, i) { a[i] = [a[i], b[i]] })
}
```

This particular implementation does the job just fine, but it also requires intimate knowledge of the function's behavior: does it modify the first argument, or the second?

```
var a = [1, 2, 3]
var b = ["one", "two", "three"]

merge(a, b)
a // -> [ [1, "one"], [2, "two"],.. ]
```

Imagine that you're unfamiliar with this function. You skim the code to review a patch, or maybe just to familiarize yourself with a new codebase. Without reading the function's source, you have no information regarding whether it merges the first argument into the second, or vice versa. It's also possible that the function is not destructive and someone simply forgot to use its value.

Alternatively, you can rewrite the same function in a nondestructive way. This makes the state change explicit to everyone who is going to use that function:

```
function merge(a, b) {
  return a.map(function (v, i) { return [v, b[i]] })
}
```

Since this new implementation doesn't modify any of its arguments, all mutations will have to be explicit:

```
var a = [1, 2, 3]
var b = ["one", "two", "three"]

merge(a, b) // -> [ [1, "one"], [2, "two"],.. ]

// a and b still have their original values.
// Any change to the value of a will have to
// be explicit through an assignment:
a = merge(a, b)
```

To further illustrate the difference between the two approaches, let's run that function three times without assigning its value:

```
var a = [1, 2]
var b = ["one", "two"]

merge(a, b)
// -> [ [1, "one"], [2, "two"] ]; a and b are the same
merge(a, b)
// -> [ [1, "one"], [2, "two"] ]; a and b are the same
merge(a, b)
// -> [ [1, "one"], [2, "two"] ]; a and b are the same
```

As you can see, the return value never changes. It doesn't matter how many times you run this function; the same input will always lead to the same output. Now let's go back to our original implementation and perform the same test:

```
var a = [1, 2]
var b = ["one", "two"]

merge(a, b)
// -> undefined; a is now [ [1, "one"], [2, "two"] ]
merge(a, b)
// -> undefined; a is now [ [[1,"one"], "one"], [[2, "two"],"two"] ]
merge(a, b)
// -> undefined; a is even worse now; the universe imploded
```

Even better is that this version of `merge` allows us to use it *as a value itself*. We can return the result of its computation or pass it around without creating temporary variables, just like we would do with any other variable holding a primitive value:

```
function prettyTable(table) {
  return table.map(function (row) {
    return row.join(" ")
  }).join("\n")
}
```

```
console.log(prettyTable(merge([1, 2, 3], ["one", "two", "three"])))
// prints:
//   1 "one"
//   2 "two"
//   3 "three"
```

This type of function, known as a *zip* function, is quite popular in the functional programming world. It becomes useful when you have multiple data sources that are coordinated through matching array indexes. JavaScript libraries such as Underscore and LoDash provide implementations of `zip` and other useful helper functions so you don't have to reinvent the wheel within your projects.

Let's look at another example where explicit code reads better than implicit. JavaScript—at least, its newer revisions—allows you to create constants in addition to variables. Constants can be created with a `const` keyword. While everyone else (including yours truly) primarily uses this keyword to declare module-level constants, my friend Nick Fitzgerald uses `consts` virtually everywhere to make clear which variables are expected to be mutated and which are not:

```
function invertSourceMaps(code, mappings) {
  const generator = new SourceMapGenerator(...)

  return DevToolsUtils.yieldingEach(mappings, function (m) {
    // ...
  })
}
```

With this approach, you can be sure that a `generator` is always an instance of `SourceMapGenerator`, regardless of where it is being used. It doesn't give us immutable data structures, but it *does* make it impossible to point this variable to a new object. This means there's one less thing to keep track of while reading the code.

Here's a bigger example of a functional approach to programming: a few weeks ago, I wrote a static site generator in JavaScript for the JSHint website and my personal blog. The main module that actually reads all the templates, generates a new site, and writes it back to disk consists of only three small functions. The first function, `read`, takes a path as an argument and returns an object that contains the whole directory tree plus the contents of the source files. The second function, `build`, does all the heavy work: it compiles all the templates and Markdown files into HTML, compresses static files, and so on. The third function, `write`, takes the site structure and saves it to disk.

There's absolutely no shared state between those three functions. Each has a set of arguments it accepts and some data it returns. An executable script I use from my command line does precisely the following:

```
#!/usr/bin/env node

var oddweb = require("./index.js")
var args   = process.argv.slice(2)

oddweb.write(oddweb.build(oddweb.read(args[1])))
```

I also get plug-ins for free. If I need a plug-in that deletes all files with names ending with *.draft*, all I do is write a function that gets a site tree as an argument and returns a new site tree. I then plug in that function somewhere between `read` and `write`, and I'm golden.

Another benefit of using a functional programming style is simpler unit tests. A pure function takes in some data, computes it, and returns the result. This means that all that's needed in order to test that function is input data and an expected return value. As a simple example, here's a unit test for our function `merge`:

```
function testMerge() {
  var data = [
    { // Both lists have the same size
      a: [1, 2, 3],
      b: ["a", "b", "c"],
      ret: [ [1, "a"], [2, "b"], [3, "c"] ]
    },

    { // Second list is larger
      a: [1, 2],
      b: ["a", "b", "c"],
      ret: [ [1, "a"], [2, "b"] ]
    },

    { // Etc.
      ...
    }
  ]

  data.forEach(function (test) {
    isEqual(merge(test.a, test.b), test.ret)
  })
}
```

This test is almost fully declarative. You can clearly see what input data is used and what is expected to be returned by the `merge` function. In addition, writing code in a functional way means you have less testing to do. Our original implementation of `merge` was modifying its arguments, so that a proper test would have had to cover cases where one of the arguments was frozen using `Object.freeze`.

All functions involved in the preceding example—`forEach`, `isEqual`, and `merge`—were designed to work with only simple, built-in data types. This approach, where you build your programs around composable functions that work with simple data types, is

called *data-driven programming*. It allows you to write programs that are clear and elegant and have a lot of flexibility for expansion.

Objects

Does this mean you shouldn't use objects, constructors, and prototype inheritance? Of course not! If something makes your code easier to understand and maintain, it'd be silly not to use it. However, JavaScript developers often start making overcomplicated object hierarchies without even considering whether there are simpler ways to solve the problem.

Consider the following object that represents a robot. This robot can walk and talk, but otherwise it's pretty useless:

```
function Robot(name) {
  this.name = name
}

Robot.prototype = {
  talk: function (what)  { /* ... */ },
  walk: function (where) { /* ... */ }
}
```

What would you do if you wanted two more robots: a guard robot to shoot things and a housekeeping robot to clean things? Most people would immediately create child objects `GuardRobot` and `HousekeeperRobot` that inherit methods and properties from the parent `Robot` object and add their own methods. But what if you then decided you wanted a robot that can both clean and shoot things? This is where hierarchy gets complicated and software fragile.

Consider the alternative approach, where you extend instances with functions that define their behavior and not their type. You don't have a `GuardRobot` and a `HousekeeperRobot` anymore; instead, you have an instance of a `Robot` that can clean things, shoot things, or do both. The implementation will probably look something like this:

```
function extend(robot, skills) {
  skills.forEach(function (skill) {
    robot[skill.name] = skill.fn.bind(null, rb)
  })

  return robot
}
```

To use it, all you have to do is to implement the behavior you need and attach it to the instance in question:

```
function shoot(robot) { /* ... */ }
function clean(robot) { /* ... */ }
```

```
var rdo = new Robot("R. Daniel Olivaw")
extend(rdo, { shoot: shoot, clean: clean })

rdo.talk("Hi!") // OK
rdo.walk("Mozilla SF") // OK
rdo.shoot() // OK
rdo.clean() // OK
```

NOTE

My friend Irakli Gozalishvili, after reading this chapter, left a comment saying that his approach would be different. What if objects were used only to store data?

```
function talk(robot)  { /* ... */ }
function shoot(robot) { /* ... */ }
function clean(robot) { /* ... */ }

var rdo = { name: "R. Daniel Olivaw" }

talk(rdo, "Hi!") // OK
walk(rdo, "Mozilla SF") // OK
shoot(rdo) // OK
clean(rdo) // OK
```

With his approach you don't even need to extend anything: all you need to do is pass the correct object.

At the beginning of this chapter, I warned JavaScript programmers against being lulled into the false sense of referential transparency that can result from using a pure functional programming language. In the example we just looked at, the function `extend` takes an object as its first argument, modifies it, and returns the modified object. The problem here is that JavaScript has a very limited set of immutable types. Strings are immutable. So are numbers. But objects—such as an instance of `Robot`—are mutable. This means that `extend` is not a pure function, since it mutates the object that was passed into it. You can call `extend` without assigning its return value to anything, and `rdo` will still be modified.

Now What?

> The major evolution that is still going on for me is towards a more functional programming style, which involves unlearning a lot of old habits, and backing away from some OOP directions.
>
> —John Carmack

JavaScript is a multiparadigm language supporting object-oriented, imperative, and functional programming styles. It provides a framework in which you can mix and match different styles and, as a result, write elegant programs. Some programmers, however, forget about all the different paradigms and stick only with their favorite

one. Sometimes this rigidity is due to fear of leaving a comfort zone; sometimes it's caused by relying too heavily on the wisdom of elders. Whatever the reason, these people often limit their options by confining themselves to a small space where it's their way or the highway.

Finding the right balance between different programming styles is hard. It requires many hours of experimentation and a healthy number of mistakes. But the struggle is worth it. Your code will be easier to reason about. It will be more flexible. You'll ultimately find yourself spending less time debugging, and more time creating something new and exciting.

So don't be afraid to experiment with different language features and paradigms. They're here for you to use, and they aren't going anywhere. Just remember: there's no single true paradigm, and it's never too late to throw out your old habits and learn something new.