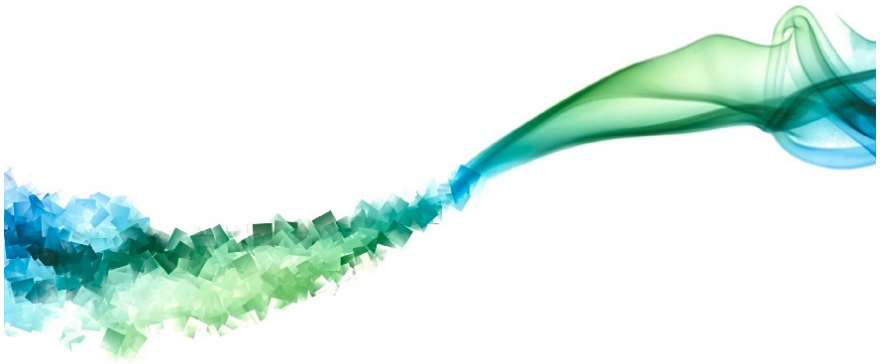
Applaus

Eric Elliott
Make some magic. #JavaScript
Feb 20, 2017 · 10 min read

Why Learn Functional Programming in JavaScript? (Composing Software)



Smoke Art Cubes to Smoke — MattysFlicks — (CC BY 2.0)

> *Note: This is part of the "Composing Software" series on learning functional programming and compositional software techniques in JavaScript ES6+ from the ground up. Stay tuned. There's a lot more of this to come!*
> < Previous | << Start over at Part 1 | Next >

Forget whatever you think you know about JavaScript, and approach this material with a beginner's mind. To help you do that, we're going to review the JavaScript basics from the ground up, as if you've never seen JavaScript before. If you're a beginner, you're in luck. Finally something exploring ES6 and functional programming from scratch! Hopefully all the new concepts are explained along the way—but don't count on too much pampering.

If you're a seasoned developer already familiar with JavaScript, or a pure functional language, maybe you're thinking that JavaScript is a funny choice for an exploration of functional programming. Set those thoughts aside, and try to approach the material with an open mind. You may find that there is another level to JavaScript programming. One you never knew existed.

Since this text is called “Composing Software”, and functional programming is the obvious way to compose software (using function composition, higher order functions, etc…), you may be wondering why I’m not talking about Haskell, ClojureScript, or Elm, instead of JavaScript.

JavaScript has the most important features needed for functional programming:

1. **First class functions:** The ability to use functions as data values: pass functions as arguments, return functions, and assign functions to variables and object properties. This property allows for higher order functions, which enable partial application, currying, and composition.
2. **Anonymous functions and concise lambda syntax:** `x => x * 2` is a valid function expression in JavaScript. Concise lambdas make it easier to work with higher-order functions.
3. **Closures:** A closure is the bundling of a function with its lexical environment. Closures are created at function creation time. When a function is defined inside another function, it has access to the variable bindings in the outer function, even after the outer function exits. Closures are how partial applications get their fixed arguments. A fixed argument is an argument bound in the closure scope of a returned function. In `add2(1)(2)`, `1` is a fixed argument in the function returned by `add2(1)`.

What JavaScript is Missing

JavaScript is a multi-paradigm language, meaning that it supports programming in many different styles. Other styles supported by JavaScript include procedural (imperative) programming (like C), where functions represent a subroutine of instructions that can be called repeatedly for reuse and organization, object-oriented programming, where objects—not functions—are the primary building blocks, and of course, functional programming. The disadvantage of a multi-paradigm language is that imperative and object-oriented programming tend to imply that almost everything needs to be mutable.

Mutation is a change to data structure that happens in-place. For example:

```
const foo = {
  bar: 'baz'
};

foo.bar = 'qux'; // mutation
```

Objects usually need to be mutable so that their properties can be updated by methods. In imperative programming, most data structures are mutable to enable efficient in-place manipulation of objects and arrays.

Here are some features that some functional languages have, that JavaScript does not have:

1. **Purity:** In some FP languages, purity is enforced by the language. Expressions with side-effects are not allowed.
2. **Immutability:** Some FP languages disable mutations. Instead of mutating an existing data structure, such as an array or object, expressions evaluate to new data structures. This may sound inefficient, but most functional languages use trie data structures under the hood, which feature structural sharing: meaning that the old object and new object share references to the data that is the same.
3. **Recursion:** Recursion is the ability for a function to reference itself for the purpose of iteration. In many FP languages, recursion is the only way to iterate. There are no loop statements like `for`, `while`, or `do` loops.

Purity: In JavaScript, purity must be achieved by convention. If you're not building most of your application by composing pure functions, you're not programming using the functional style. It's unfortunately easy in JavaScript to get off track by accidentally creating and using impure functions.

Immutability: In pure functional languages, immutability is often enforced. JavaScript lacks efficient, immutable trie-based data structures used by most functional languages, but there are libraries that help, including Immutable.js and Mori. I'm hoping that future versions of the ECMAScript spec will embrace immutable data structures.

There are signs that offer hope, like the addition of the `const` keyword in ES6. A name binding defined with `const` can't be reassigned to refer to a different value. It's important to understand that `const` does not represent an immutable *value*.

A `const` object can't be reassigned to refer to a completely different object, but the object it refers to *can have its properties mutated*. JavaScript also has the ability to `freeze()` objects, but those objects are only frozen at the root level, meaning that a nested object can still have properties of its properties mutated. In other words, there's still a long road ahead before we see true composite immutables in the JavaScript specification.

Recursion: JavaScript technically supports recursion, but most functional languages have a feature called tail call optimization. Tail call optimization is a feature which allows recursive functions to reuse stack frames for recursive calls.

Without tail call optimization, a call stack can grow without bounds and cause a stack overflow. JavaScript technically got a limited form of tail call optimization in the ES6 specification. Unfortunately, only one of the major browser engines implemented it, and the optimization was partially implemented and then subsequently removed from Babel (the most popular standard JavaScript compiler, used to compile ES6 to ES5 for use in older browsers).

Bottom line: It still isn't safe to use recursion for large iterations—even if you're careful to call the function in the tail position.

What JavaScript Has that Pure Functional Languages Lack

A purist will tell you that JavaScript's mutability is its major disadvantage, which is true. However, side effects and mutation are sometimes beneficial. In fact, it's impossible to create most useful modern applications without side effects. Pure functional languages like Haskell use side-effects, but camouflage them from pure functions using boxes called monads, allowing the program to remain pure even though the side effects represented by the monads are impure.

The trouble with monads is that, even though their use is quite simple, explaining what a monad is to somebody unfamiliar with lots of examples is a bit like explaining what the color "blue" looks like to a blind person.

> "A monad is a monoid in the category of endofunctors, what's the problem?" ~ James Iry, fictionally quoting Philip Wadler, paraphrasing a real quote by Saunders Mac Lane. "A Brief, Incomplete, and Mostly Wrong History of Programming Languages"

Typically, parody exaggerates things to make a funny point funnier. In the quote above, the explanation of monads is actually *simplified* from the original quote, which goes like this:

> "A monad in `X` is just a monoid in the category of endofunctors of `X`, with product `×` replaced by composition of endofunctors and unit set by the identity endofunctor." ~ Saunders Mac Lane. "Categories for the Working Mathematician"

Even so, in my opinion, fear of monads is weak reasoning. The best way to learn monads is not to read a bunch of books and blog posts on the subject, but to jump in and start using them. As with most things in functional programming, the impenetrable academic vocabulary is much harder to understand than the concepts. Trust me, you don't have to understand Saunders Mac Lane to understand functional programming.

While it may not be absolutely ideal for every programmming style, JavaScript is unapologetically a general-purpose language designed to be usable by various people with various programming styles and backgrounds.

According to Brendan Eich, this was intentional from the beginning. Netscape had to support two kinds of programmers:

> "…the component authors, who wrote in C++ or (we hoped) Java; and the 'scripters', amateur or pro, who would write code directly embedded in HTML."

Originally, the intent was that Netscape would support two different languages, and the scripting language would probably resemble Scheme (a dialect of Lisp). Again, Brendan Eich:

> "I was recruited to Netscape with the promise of 'doing Scheme' in the browser."

JavaScript had to be a new language:

> "The *diktat* from upper engineering management was that the language must 'look like Java'. That ruled out Perl, Python, and Tcl,

> along with Scheme."

So, the ideas in Brendan Eich's head from the beginning were:

1. Scheme in the browser.
2. Look like Java.

It ended up being even more of a mish-mash:

> "I'm not proud, but I'm happy that I chose Scheme-ish first-class functions and Self-ish (albeit singular) prototypes as the main ingredients. The Java influences, especially y2k Date bugs but also the primitive vs. object distinction (e.g., string vs. String), were unfortunate."

I'd add to the list of "unfortunate" Java-like features that eventually made their way into JavaScript:

- Constructor functions and the `new` keyword, with different calling and usage semantics from factory functions.
- A `class` keyword with single-ancestor `extends` as the primary inheritance mechanism.
- The user's tendency to think of a `class` as if it's a static type (it's not).

My advice: Avoid those whenever you can.

We're lucky that JavaScript ended up being such a capable language, because it turns out that the scripting approach won over the "component" approach (today, Java, Flash, and ActiveX extensions are unsupported in huge numbers of installed browsers).

What we eventually ended up with was one language directly supported by the browser: JavaScript.

That means that browsers are less bloated and less buggy, because they only need to support a single set of language bindings: JavaScript's. You might be thinking that WebAssembly is an exception, but one of the design goals of WebAssembly is to share JavaScript's language bindings using a compatible Abstract Syntax Tree (AST). In fact, the first demonstrations compiled WebAssembly to a subset of JavaScript known as ASM.js.

The position as the only standard general purpose programming language for the web platform allowed JavaScript to ride the biggest language popularity wave in the history of software:

Apps ate the world, the web ate apps, and JavaScript ate the web.

By multiple measures, JavaScript is now the most popular programming language in the world.

JavaScript is not the ideal tool for functional programming, but it's a great tool for building large applications on very large, distributed teams, where different teams may have different ideas about how to build an application.

Some teams may concentrate on scripting glue, where imperative programming is particularly useful. Others may concentrate on building architectural abstractions, where a bit of (restrained, careful) OO thinking may not be a bad idea. Still others may embrace functional programming, reducing over user actions using pure functions for deterministic, testable management of application state. Members on these teams are all using the same language, meaning that they can more easily exchange ideas, learn from each other, and build on each other's work.

In JavaScript, all of these ideas can co-exist, which allows more people to embrace JavaScript, which has led to the largest open-source package registry in the world (as of February, 2017), npm.

The true strength of JavaScript is diversity of thought and users in the ecosystem. It may not be absolutely the ideal language for functional programming purists, but it may be the ideal language for working together using one language that works on just about every platform you can imagine—familiar to people coming from other popular languages such as Java, Lisp, or C. JavaScript won't feel ideally comfortable to users with any of those backgrounds, but they may feel *comfortable enough* to learn the language and become productive quickly.

I agree that JavaScript is not the best language for functional programmers. However, no other functional language can claim that it is a language that everybody can use and embrace, and as demonstrated by ES6: JavaScript can and does get better at serving the needs of users interested in functional programming. Instead of abandoning JavaScript and its incredible ecosystem used by virtually

every company in the world, why not embrace it, and make it a better language for software composition incrementally?

As-is, JavaScript is already a *good enough* functional programming language, meaning that people are building all kinds of useful and interesting things in JavaScript, using functional programming techniques. Netflix (and every app built with Angular 2+) uses functional utilities based on RxJS. Facebook uses the concepts of pure functions, higher-order functions, and higher order components in React to build Facebook and Instagram. PayPal, KhanAcademy, and Flipkart use Redux for state management.

They're not alone: Angular, React, Redux, and Lodash are the leading frameworks and libraries in the JavaScript application ecosystem, and all of them are heavily influenced by functional programming—or in the cases of Lodash and Redux, built for the express purpose of enabling functional programming patterns in real JavaScript applications.

"Why JavaScript?" Because JavaScript is the language that most real companies are using to build real software. Love it or hate it, JavaScript has stolen the title of "most popular functional programming language" from Lisp, which was the standard bearer for decades. True, Haskell is a much more suitable standard bearer for functional programming concepts today, but people just aren't building as many real applications in Haskell.

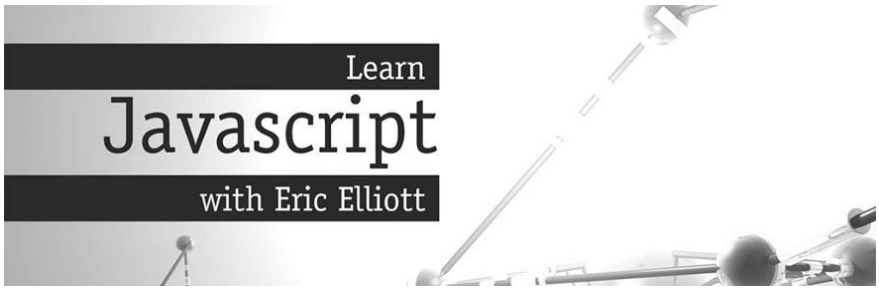
At any given moment, there are close to a hundred thousand JavaScript job openings in the United States, and hundreds of thousands more world-wide. Learning Haskell will teach you a lot about functional programming, but learning JavaScript will teach you a lot about building production apps for real jobs.

Apps ate the world, the web ate apps, and JavaScript ate the web.

Continued in Part 3: A Functional Programmer's Introduction to JavaScript…

Learn More at EricElliottJS.com

Video lessons on functional programming are available for members of EricElliottJS.com. If you're not a member, sign up today.



. . .

Eric Elliott *is the author of "Programming JavaScript Applications" (O'Reilly), and "Learn JavaScript with Eric Elliott". He has contributed to software experiences for* ***Adobe Systems****,* ***Zumba Fitness****,* ***The Wall Street Journal****,* ***ESPN****,* ***BBC****, and top recording artists including* ***Usher****,* ***Frank Ocean****,* ***Metallica****, and many more.*

He works remote from anywhere with the most beautiful woman in the world.