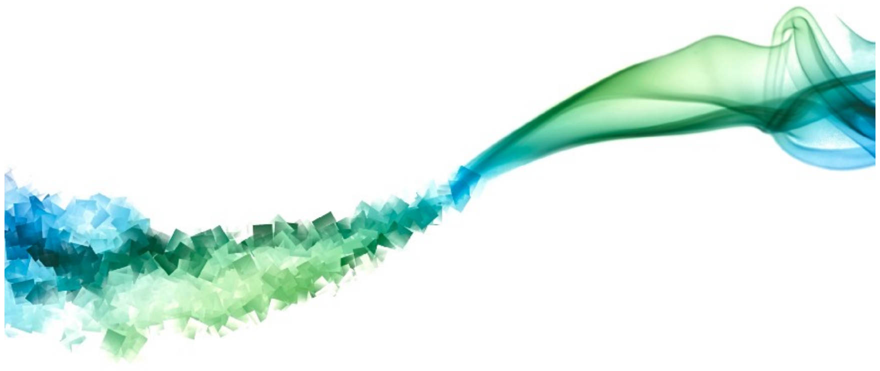
Applaus

Eric Elliott
Make some magic. #JavaScript
Feb 25, 2017 · 13 min read

A Functional Programmer's Introduction to 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 >*

For those unfamiliar with JavaScript or ES6+, this is intended as a brief introduction. Whether you're a beginner or experienced JavaScript developer, you may learn something new. The following is only meant to scratch the surface and get you excited. If you want to know more, you'll just have to explore deeper. There's a lot more ahead.

The best way to learn to code is to code. I recommend that you follow along using an interactive JavaScript programming environment such as CodePen or the Babel REPL.

Alternatively, you can get away with using the Node or browser console REPLs.

Expressions and Values

An expression is a chunk of code that evaluates to a value.

The following are all valid expressions in JavaScript:

```
7;

7 + 1; // 8

7 * 2; // 14

'Hello'; // Hello
```

The value of an expression can be given a name. When you do so, the expression is evaluated first, and the resulting value is assigned to the name. For this, we’ll use the `const` keyword. It's not the only way, but it's the one you'll use most, so we'll stick with `const` for now:

```
const hello = 'Hello';
hello; // Hello
```

var, let, and const

JavaScript supports two more variable declaration keywords: `var`, and `let`. I like to think of them in terms of order of selection. By default, I select the strictest declaration: `const`. A variable declared with the `const` keyword can't be reassigned. The final value must be assigned at declaration time. This may sound rigid, but the restriction is a good thing. It's a signal that tells you, "the value assigned to this name is not going to change". It helps you fully understand what the name means right away, without needing to read the whole function or block scope.

Sometimes it’s useful to reassign variables. For example, if you’re using manual, imperative iteration rather than a more functional approach, you can iterate a counter assigned with `let`.

Because `var` tells you the least about the variable, it is the weakest signal. Since I started using ES6, I have never intentionally declared a `var` in a real software project.

Be aware that once a variable is declared with `let` or `const`, any attempt to declare it again will result in an error. If you prefer more experimental flexibility in the REPL (Read, Eval, Print Loop) environment, you may use `var` instead of `const` to declare variables. Redeclaring `var` is allowed.

This text will use `const` in order to get you in the habit of defaulting to `const` for actual programs, but feel free to substitute `var` for the purpose of interactive experimentation.

Types

So far we’ve seen two types: numbers and strings. JavaScript also has booleans (`true` or `false`), arrays, objects, and more. We'll get to other types later.

An array is an ordered list of values. Think of it as a box that can hold many items. Here’s the array literal notation:

```
[1, 2, 3];
```

Of course, that’s an expression which can be given a name:

```
const arr = [1, 2, 3];
```

An object in JavaScript is a collection of key: value pairs. It also has a literal notation:

```
{
  key: 'value'
}
```

And of course, you can assign an object to a name:

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

If you want to assign existing variables to object property keys of the same name, there's a shortcut for that. You can just type the variable name instead of providing both a key and a value:

```
const a = 'a';
const oldA = { a: a }; // long, redundant way
const oA = { a }; // short an sweet!
```

Just for fun, let's do that again:

```
const b = 'b';
const oB = { b };
```

Objects can be easily composed together into new objects:

```
const c = {...oA, ...oB}; // { a: 'a', b: 'b' }
```

Those dots are the object spread operator. It iterates over the properties in `oA` and assigns them to the new object, then does the same for `oB`, overriding any keys that already exist on the new object. As of this writing, object spread is a new, experimental feature that may not be available in all the popular browsers yet, but if it's not working for you, there is a substitute: `Object.assign()`:

```
const d = Object.assign({}, oA, oB); // { a: 'a', b: 'b' }
```

Only a little more typing in the `Object.assign()` example, and if you're composing lots of objects, it may even save you some typing. Note that when you use `Object.assign()`, you must pass a destination object as the first parameter. It is the object that properties will be copied to. If you forget, and omit the destination object, the object you pass in the first argument will be mutated.

In my experience, mutating an existing object rather than creating a new object is usually a bug. At the very least, it is error-prone. Be careful with `Object.assign()` .

Destructuring

Both objects and arrays support destructuring, meaning that you can extract values from them and assign them to named variables:

```
const [t, u] = ['a', 'b'];
t; // 'a'
u; // 'b'
```

```
const blep = {
  blop: 'blop'
};

// The following is equivalent to:
// const blop = blep.blop;
const { blop } = blep;
blop; // 'blop'
```

As with the array example above, you can destructure to multiple assignments at once. Here's a line you'll see in lots of Redux projects:

```
const { type, payload } = action;
```

Here's how it's used in the context of a reducer (much more on that topic coming later):

```
const myReducer = (state = {}, action = {}) => {
  const { type, payload } = action;
  switch (type) {
    case 'FOO': return Object.assign({}, state, payload);
    default: return state;
  }
};
```

If you don't want to use a different name for the new binding, you can assign a new name:

```
const { blop: bloop } = blep;
bloop; // 'blop'
```

Read: Assign `blep.blop` as `bloop`.

Comparisons and Ternaries

You can compare values with the strict equality operator (sometimes called “triple equals”):

```
3 + 1 === 4; // true
```

There’s also a sloppy equality operator. It’s formally known as the “Equal” operator. Informally, “double equals”. Double equals has a valid use-case or two, but it’s almost always better to default to the `===` operator, instead.

Other comparison operators include:

- `>` Greater than
- `<` Less than
- `>=` Greater than or equal to
- `<=` Less than or equal to
- `!=` Not equal
- `!==` Not strict equal
- `&&` Logical and
- `||` Logical or

A ternary expression is an expression that lets you ask a question using a comparator, and evaluates to a different answer depending on whether or not the expression is truthy:

```
14 - 7 === 7 ? 'Yep!' : 'Nope.'; // Yep!
```

Functions

JavaScript has function expressions, which can be assigned to names:

```
const double = x => x * 2;
```

This means the same thing as the mathematical function `f(x) = 2x`. Spoken out loud, that function reads `f` of `x` equals `2x`. This function is only interesting when you apply it to a specific value of `x`. To use the function in other equations, you'd write `f(2)`, which has the same meaning as `4`.

In other words, `f(2) = 4`. You can think of a math function as a mapping from inputs to outputs. `f(x)` in this case is a mapping of input values for `x` to corresponding output values equal to the product of the input value and `2`.

In JavaScript, the value of a function expression is the function itself:

```
double; // [Function: double]
```

You can see the function definition using the `.toString()` method:

```
double.toString(); // 'x => x * 2'
```

If you want to apply a function to some arguments, you must invoke it with a function call. A function call applies a function to its arguments and evaluates to a return value.

You can invoke a function using `<functionName>(argument1, argument2, ...rest)`. For example, to invoke our double function, just add the parentheses and pass in a value to double:

```
double(2); // 4
```

Unlike some functional languages, those parentheses are meaningful. Without them, the function won't be called:

```
double 4; // SyntaxError: Unexpected number
```

Signatures

Functions have signatures, which consist of:

1. An *optional* function name.
2. A list of parameter types, in parentheses. The parameters may optionally be named.
3. The type of the return value.

Type signatures don't need to be specified in JavaScript. The JavaScript engine will figure out the types at runtime. If you provide enough clues, the signature can also be inferred by developer tools such as IDEs (Integrated Development Environment) and Tern.js using data flow analysis.

JavaScript lacks its own function signature notation, so there are a few competing standards: JSDoc has been very popular historically, but it's awkwardly verbose, and nobody bothers to keep the doc comments up-to-date with the code, so many JS developers have stopped using it.

TypeScript and Flow are currently the big contenders. I'm not sure how to express everything I need in either of those, so I use Rtype, for documentation purposes only. Some people fall back on Haskell's curry-only Hindley–Milner types. I'd love to see a good notation system standardized for JavaScript, if only for documentation purposes, but I don't think any of the current solutions are up to the task, at present. For now, squint and do your best to keep up with the weird type signatures which probably look slightly different from whatever you're using.

```
functionName(param1: Type, param2: Type) => Type
```

The signature for double is:

```
double(x: n) => Number
```

In spite of the fact that JavaScript doesn’t require signatures to be annotated, knowing what signatures *are* and what they *mean* will still be important in order to communicate efficiently about how functions are used, and how functions are composed. Most reusable function composition utilities require you to pass functions which share the same type signature.

Default Parameter Values

JavaScript supports default parameter values. The following function works like an identity function (a function which returns the same value you pass in), unless you call it with `undefined`, or simply pass no argument at all -- then it returns zero, instead:

```
const orZero = (n = 0) => n;
```

To set a default, simply assign it to the parameter with the `=` operator in the function signature, as in `n = 0`, above. When you assign default values in this way, type inference tools such as Tern.js, Flow, or TypeScript can infer the type signature of your function automatically, even if you don't explicitly declare type annotations.

The result is that, with the right plugins installed in your editor or IDE, you’ll be able to see function signatures displayed inline as you’re typing function calls. You’ll also be able to understand how to use a function at a glance based on its call signature. Using default assignments wherever it makes sense can help you write more self-documenting code.

> *Note: Parameters with defaults don’t count toward the function’s* `.length` *property, which will throw off utilities such as autocurry which depend on the* `.length` *value. Some curry utilities (such as* `lodash/curry`*) allow you to pass a custom arity to work around this limitation if you bump into it.*

Named Arguments

JavaScript functions can take object literals as arguments and use destructuring assignment in the parameter signature in order to achieve the equivalent of named arguments. Notice, you can also assign default values to parameters using the default parameter feature:

```
const createUser = ({
  name = 'Anonymous',
  avatarThumbnail = '/avatars/anonymous.png'
}) => ({
  name,
  avatarThumbnail
});

const george = createUser({
  name: 'George',
  avatarThumbnail: 'avatars/shades-emoji.png'
});


george;
/*
{
  name: 'George',
  avatarThumbnail: 'avatars/shades-emoji.png'
}
*/
```

Rest and Spread

A common feature of functions in JavaScript is the ability to gather together a group of remaining arguments in the functions signature using the rest operator: `...`

For example, the following function simply discards the first argument and returns the rest as an array:

```
const aTail = (head, ...tail) => tail;
aTail(1, 2, 3); // [2, 3]
```

Rest gathers individual elements together into an array. Spread does the opposite: it spreads the elements from an array to individual elements. Consider this:

```
const shiftToLast = (head, ...tail) => [...tail, head];
shiftToLast(1, 2, 3); // [2, 3, 1]
```

Arrays in JavaScript have an iterator that gets invoked when the spread operator is used. For each item in the array, the iterator delivers a value. In the expression, `[...tail, head]` , the iterator copies each element in order from the `tail` array into the new array created by the surrounding literal notation. Since the head is already an individual element, we just plop it onto the end of the array and we're done.

Currying

A curried function is a function that takes multiple parameters one at a time: It takes a parameter, and returns a function that takes the next parameter, and so on until all parameters have been supplied, at which point, the application is completed and the final value is returned.

Curry and partial application can be enabled by returning another function:

```
const highpass = cutoff => n => n >= cutoff;
const gt4 = highpass(4); // highpass() returns a new
function
```

You don't have to use arrow functions. JavaScript also has a `function` keyword. We're using arrow functions because the `function` keyword is a lot more typing. This is equivalent to the `highPass()` definition, above:

```
const highpass = function highpass(cutoff) {
  return function (n) {
    return n >= cutoff;
  };
};
```

The arrow in JavaScript roughly means "function". There are some important differences in function behavior depending on which kind of function you use (`=>` lacks its own `this` , and can't be used as a constructor), but we'll get to those differences when we get there. For now, when you see `x => x` , think "a function that takes `x` and returns `x` ". So you can read `const highpass = cutoff => n => n >= cutoff;` as:

"`highpass` is a function which takes `cutoff` and returns a function which takes `n` and returns the result of `n >= cutoff` ".

Since `highpass()` returns a function, you can use it to create a more specialized function:

```
const gt4 = highpass(4);

gt4(6); // true
gt4(3); // false
```

Autocurry lets you curry functions automatically, for maximal flexibility. Say you have a function `add3()` :

```
const add3 = curry((a, b, c) => a + b + c);
```

With autocurry, you can use it in several different ways, and it will return the right thing depending on how many arguments you pass in:

```
add3(1, 2, 3); // 6
add3(1, 2)(3); // 6
add3(1)(2, 3); // 6
add3(1)(2)(3); // 6
```

Sorry Haskell fans, JavaScript lacks a built-in autocurry mechanism, but you can import one from Lodash:

```
$ npm install --save lodash
```

Then, in your modules:

```
import curry from 'lodash/curry';
```

Or, you can use the following magic spell:

```
// Tiny, recursive autocurry
const curry = (
  f, arr = []
) => (...args) => (
  a => a.length === f.length ?
    f(...a) :
    curry(f, a)
)([...arr, ...args]);
```

Function Composition

Of course you can compose functions. Function composition is the process of passing the return value of one function as an argument to another function. In mathematical notation:

```
f . g
```

Which translates to this in JavaScript:

```
f(g(x))
```

It’s evaluated from the inside out:

1. `x` is evaluated
2. `g()` is applied to `x`
3. `f()` is applied to the return value of `g(x)`

For example:

```
const inc = n => n + 1;
inc(double(2)); // 5
```

The value `2` is passed into `double()`, which produces `4`. `4` is passed into `inc()` which evaluates to `5`.

You can pass any expression as an argument to a function. The expression will be evaluated before the function is applied:

```
inc(double(2) * double(2)); // 17
```

Since `double(2)` evaluates to `4`, you can read that as `inc(4 * 4)` which evaluates to `inc(16)` which then evaluates to `17`.

Function composition is central to functional programming. We’ll have a lot more on it later.

Arrays

Arrays have some built-in methods. A method is a function associated with an object: usually a property of the associated object:

```
const arr = [1, 2, 3];
arr.map(double); // [2, 4, 6]
```

In this case, `arr` is the object, `.map()` is a property of the object with a function for a value. When you invoke it, the function gets applied to the arguments, as well as a special parameter called `this`, which gets automatically set when the method is invoked. The `this` value is how `.map()` gets access to the contents of the array.

Note that we’re passing the `double` function as a value into `map` rather than calling it. That's because `map` takes a function as an argument and applies it to each item in the array. It returns a new array containing the values returned by `double()`.

Note that the original `arr` value is unchanged:

```
arr; // [1, 2, 3]
```

Method Chaining

You can also chain method calls. Method chaining is the process of directly calling a method on the return value of a function, without needing to refer to the return value by name:

```
const arr = [1, 2, 3];
arr.map(double).map(double); // [4, 8, 12]
```

A **predicate** is a function that returns a boolean value (`true` or `false`). The `.filter()` method takes a predicate and returns a new list, selecting only the items that pass the predicate (return `true`) to be included in the new list:

```
[2, 4, 6].filter(gt4); // [4, 6]
```

Frequently, you'll want to select items from a list, and then map those items to a new list:

```
[2, 4, 6].filter(gt4).map(double); [8, 12]
```

Note: Later in this text, you'll see a more efficient way to select and map at the same time using something called a *transducer*, but there are other things to explore first.

Conclusion

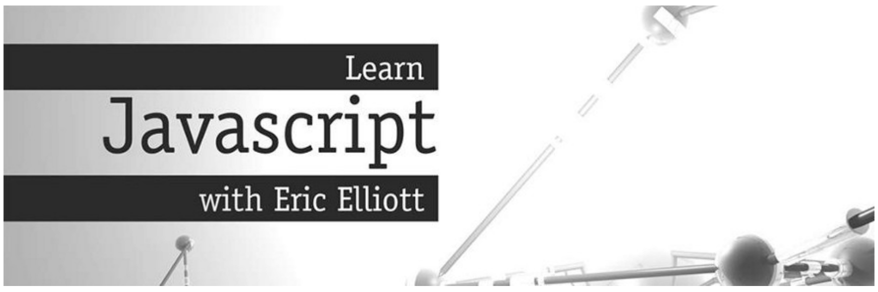
If your head is spinning right now, don't worry. We barely scratched the surface of a lot of things that deserve a lot more exploration and consideration. We'll circle back and explore some of these topics in much more depth, soon.

Continued in "Higher Order Functions"…

Next Steps

Want to learn more about functional programming in JavaScript?

Learn JavaScript with Eric Elliott. If you're not a member, you're missing out!



. . .

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 spends most of his time in the San Francisco Bay Area with the most beautiful woman in the world.