An introduction to functional programming in JavaScript

Explore functional programming and how using it can make your programs easier to read and debug.

27 Jun 2017 | Matt Banz (/users/battmanz) | 479 | 2 comments

Image credits : *Steve Jurvetson via Flickr (CC-BY-2.0)* dialect of Lisp, is a functional programming language. Things changed when Eich was told that the new language should be the scripting language companion to Java. Eich eventually settled on a language that has a C-style

first-class functions. Java technically did not have n 8, however you could simulate first-class ısses. Those first-class functions are what makes le in JavaScript.

language that allows you to freely mix and match d functional paradigms. Recently there has been ıal programming. In frameworks such as Angular .rangle.io/handout/change-rategy_onpush.html) and React t/docs/optimizing-performance.html#the-power-of-not-mutating-data), you'll actually get a performance boost by using immutable data structures. Immutability is a core tenet of functional programming. It, along with pure functions, makes it easier to reason about and debug your programs. Replacing procedural loops with functions can increase the readability of your program and make it more elegant. Overall, there are many advantages to functional programming.

What functional programming isn't

Before we talk about what functional programming is, let's talk about what it is not. In fact, let's talk about all the language constructs you should throw out (goodbye, old friends):

- Red Hat Developers Blog (https://developers.redhat.com/?intcmp=7016000000127cYAAQ&src=programming_resource_menu4)

ar or **let**

e: **o.x = 5;**)

- Array mutator methods
  - **copyWithin**
  - **fill**
  - **pop**
  - **push**
  - **reverse**
  - **shift**
  - **sort**
  - **splice**
  - **unshift**
- Map mutator methods
  - **clear**
  - **delete**
  - **set**
- Set mutator methods
  - **add**
  - **clear**
  - **delete**

Pure functions

Just because your program contains functions does not necessarily mean that

amming. Functional programming distinguishes tions. It encourages you to write pure functions. A of the following properties:

The function always gives the same return value s means that the function cannot depend on any

on cannot cause any side effects. Side effects may ne console or a log file), modifying a mutable e, etc.

Let's illustrate with a few examples. First, the **multiply** function is an example of a pure function. It always returns the same output for the same input, and it causes no side effects.

```
function multiply(a, b) {
  return a * b;
}
```

e29d99e80ba8db2b1/raw/fd586c5da7c936235a6d99b11cb80c9c67e4deaf/pure-

pure-function-example.js (https://gist.github.com/battmanz/62fa0a78841aa0fe29d99e80ba8db2b1#file-pure-function-example-js) hosted with ❤ by GitHub (https://github.com)

The following are examples of impure functions. The **canRide** function depends on the captured **heightRequirement** variable. Captured variables do not necessarily make a function impure, but mutable (or re-assignable) ones do. In this case it was declared using **let**, which means that it can be reassigned. The **multiply** function is impure because it causes a side-effect by logging to the console.

```
let heightRequirement = 46;

// Impure because it relies on a mutable (reassignable) variable.
// Impure because it causes a side-effect by logging to the console.
function multiply(a, b) {
  console.log('Arguments: ', a, b);
```

Subscribe now

X

Get the highlights in your inbox every week.

5c36c5bde69d4000b6ecb8fee98c9edcd3/impure-

anz/459c13138ea8e333fc6603ae688b7992#file-

by **GitHub** (https://github.com)

eral built-in functions in JavaScript that are impure. properties each one does not satisfy?

- **console.log**
- **element.addEventListener**
- **Math.random**
- **Date.now**
- **$.ajax** (where **$** == the Ajax library of your choice)

Living in a perfect world in which all our functions are pure would be nice, but as you can tell from the list above, any meaningful program will contain impure functions. Most of the time we will need to make an Ajax call, check the current date, or get a random number. A good rule of thumb is to follow the 80/20 rule: 80% of your functions should be pure, and the remaining 20%, of necessity, will be impure.

There are several benefits to pure functions:

- They're easier to reason about and debug because they don't depend on mutable state.
- The return value can be cached or "memoized" to avoid recomputing it in the future.
- They're easier to test because there are no dependencies (such as logging, Ajax, database, etc.) that need to be mocked.

it's causing some side effect. Along the same lines, if you call a function but do not use its return value, again, you're probably relying on it to do some side effect, and it is an impure function.

aptured variables. Above, we looked at the
that it is an impure function, because the
reassigned. Here is a contrived example of how it
ictable results:

```javascript
    return height >= heightRequirement;
}

// Every half second, set heightRequirement to a random number between 0 and 200.
setInterval(() => heightRequirement = Math.floor(Math.random() * 201), 500);

const mySonsHeight = 47;

// Every half second, check if my son can ride.
// Sometimes it will be true and sometimes it will be false.
setInterval(() => console.log(canRide(mySonsHeight)), 500);
```

b3f7b0c12ab8e4/raw/e2b6365e79def9b80bd7652cf15078d613ed686f/mutable-

mutable-state.js (https://gist.github.com/battmanz/bc13c4cf24b380cbd7b3f7b0c12ab8e4#file-mutable-state-js) hosted with ❤ by GitHub (https://github.com)

Let me reemphasize that captured variables do not necessarily make a function impure. We can rewrite the **canRide** function so that it is pure by simply changing how we declare the **heightRequirement** variable.

```javascript
const heightRequirement = 46;

function canRide(height) {
  return height >= heightRequirement;
```

a51c2678983/raw/6792dd568e0fc3e6b372d078735d5b74857dbae4/immutable-

immutable-state.js

Subscribe now

X

Get the highlights in your inbox every week.

anz/b65416550d62da94a69eea51c2678983#file- by **GitHub** (https://github.com)

nst means that there is no chance that it will be ade to reassign it, the runtime engine will throw an of a simple number we had an object that stored

```
here
```

```
function canRide(height) {
  return height >= constants.heightRequirement;
}
```

019ba4d070c25b/raw/f7318904effbef28e3a47989d4899ab019127c05/captured-

captured-mutable-object.js (https://gist.github.com/battmanz/d32f2be485f4224121019ba4d070c25b#file-captured-mutable-object-js) hosted with ❤ by **GitHub** (https://github.com)

We used **const** so the variable cannot be reassigned, but there's still a problem. The object can be mutated. As the following code illustrates, to gain true immutability, you need to prevent the variable from being reassigned, and you also need immutable data structures. The JavaScript language provides us with the **Object.freeze** method to prevent an object from being mutated.

```
'use strict';

// CASE 1: The object is mutable and the variable can be reassigned.
let o1 = { foo: 'bar' };

// Mutate the object
o1.foo = 'something different';

```

```
// CASE 2: The object is still mutable but the variable cannot be reassigned.
const o2 = { foo: 'baz' };
```

Subscribe now

X

Get the highlights in your inbox every week.

```
ect
nt, yet again';

able
cause an error if you uncomment me' }; // Error!


nutable but the variable can be reassigned.
): "Can't mutate me" });


nent me. I dare ya!'; // Error!

ariable
o3 = { message: "I'm some other object, and I'm even mutable -- so take that!" };


// CASE 4: The object is immutable and the variable cannot be reassigned. This is w
const o4 = Object.freeze({ foo: 'never going to change me' });

// Cannot mutate the object
// o4.foo = 'talk to the hand' // Error!

// Cannot reassign the variable
// o4 = { message: "ain't gonna happen, sorry" }; // Error
```

178ab9cb007/raw/6cfd1b9644486f257357e611b2eeb148b3956baf/immutability-

immutability-vs-reassignment.js (https://gist.github.com/battmanz/c5046c2d0af45938190e1178ab9cb007#file-immutability-vs-reassignment-js) hosted with ❤ by GitHub (https://github.com)

Immutability pertains to all data structures, which includes arrays, maps, and sets. That means that we cannot call mutator methods such as **array.prototype.push** because that modifies the existing array. Instead of pushing an item onto the existing array, we can create a new array with all the same items as the original array, plus the one additional item. In fact, every mutator method can be replaced by a function that returns a new array with the

```
'use strict';

const a = Object.freeze([4, 5, 6]);
```

Subscribe now X

Get the highlights in your inbox every week.

```
 9);
 2, 3);
a);
pareFunction);
unction, a); // R = Ramda

// Instead of: a.reverse();
const g = R.reverse(a); // R = Ramda

// Exercise for the reader:
// copyWithin
// fill
// splice
```

c3a2ff87c0bf7da68/raw/6771481278c95942b4627a709c377f62856a0a3a/array-

array-mutator-method-replacement.js (https://gist.github.com/battmanz/a400ad93d9922fdc3a2ff87c0bf7da68#file-array-mutator-method-replacement-js) hosted with ❤ by GitHub (https://github.com)

The same thing goes when using a **Map** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) or a **Set** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set). We can avoid mutator methods by returning a new **Map** or **Set** with the desired changes.

```
const map = new Map([
  [1, 'one'],
  [2, 'two'],
// Instead of: map.set(4, 'four');
const map2 = new Map([...map, [4, 'four']]);

// Instead of: map.delete(1);
```

Subscribe now X

Get the highlights in your inbox every week.

```
]].filter(([key]) => key !== 1));
```

55ebe3d2a7fda7ac1f434c2d56ad98a04ce0/map-

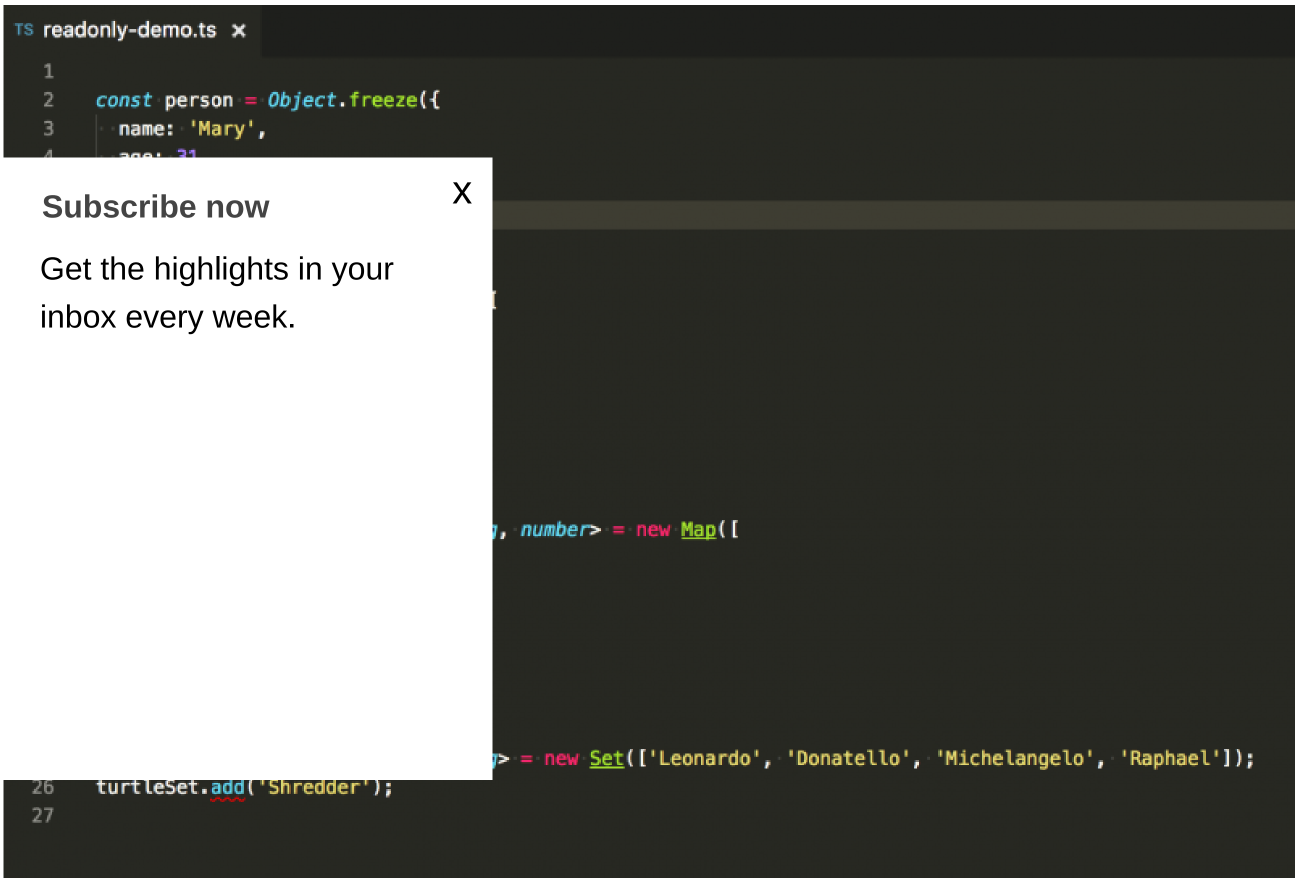
anz/9ffbac3c18c6cf33d97a4ad511129e94#file-ment-js) hosted with ❤ by GitHub (https://github.com)

```
3', 'C']);

const set2 = new Set([...set, 'D']);

// Instead of: set.delete('B');
const set3 = new Set([...set].filter(key => key !== 'B'));

// Instead of: set.clear();
const set4 = new Set();
```

l780f68daaa6a87338/raw/111578801a4120726a369a43f87d33a64b39dc83/set-

set-mutator-method-replacement.js (https://gist.github.com/battmanz/d42d3224c99d76d780f68daaa6a87338#file-set-mutator-method-replacement-js) hosted with ❤ by GitHub (https://github.com)

I would like to add that if you are using TypeScript (I am a huge fan of TypeScript), then you can use the **Readonly<T>**, **ReadonlyArray<T>**, **ReadonlyMap<K, V>**, and **ReadonlySet<T>** interfaces to get a compile-time error if you attempt to mutate any of those objects. If you call **Object.freeze** on an object literal or an array, then the compiler will automatically infer that it is read-only. Because of how Maps and Sets are represented internally, calling **Object.freeze** on those data structures does not work the same. But it's easy enough to tell the compiler that you would like them to be read-only.


readonly-demo.ts

```
const person = Object.freeze({
  name: 'Mary',
  age: 31
```

```
, number> = new Map([
```

```
> = new Set(['Leonardo', 'Donatello', 'Michelangelo', 'Raphael']);
turtleSet.add('Shredder');
```

TypeScript read-only interfaces

Okay, so we can create new objects instead of mutating existing ones, but won't that adversely affect performance? Yes, it can. Be sure to do performance testing in your own app. If you need a performance boost, then consider using Immutable.js (https://facebook.github.io/immutable-js/). Immutable.js implements Lists (https://facebook.github.io/immutable-js/docs/#/List), Stacks (https://facebook.github.io/immutable-js/docs/#/Stack), Maps (https://facebook.github.io/immutable-js/docs/#/Map), Sets (https://facebook.github.io/immutable-js/docs/#/Set), and other data structures using persistent data structures (https://en.wikipedia.org/wiki/Persistent_data_structure). This is the same technique used internally by functional programming languages such as Clojure and Scala.

```
// Use in place of `[]`.
const list1 = Immutable.List(['A', 'B', 'C']);
const list2 = list1.push('D', 'E');

console.log([...list1]); // ['A', 'B', 'C']
```

```

// Use in place of `new Map()`
const map1 = Immutable.Map([
  ['one', 1]
```

```
', 4);

[['one', 1], ['two', 2], ['three', 3]]
[['one', 1], ['two', 2], ['three', 3], ['four', 4]]

()`
[1, 2, 3, 3, 3, 3, 3, 4]);

[1, 2, 3, 4]
console.log([...set2]); // [1, 2, 3, 4, 5]
```

c478236892de/raw/2cad9d5441ebef816e3d1cbc03af883451e68dc3/immutable-

immutable-js-demo.js (https://gist.github.com/battmanz/7cec8c2f22ee55f60dd0c478236892de#file-immutable-js-demo-js) hosted with ❤ by GitHub (https://github.com)

Function composition

Remember back in high school when you learned something that looked like **(f ∘ g)(x)**? Remember thinking, "When am I ever going to use this?" Well, now you are. Ready? **f ∘ g** is read "**f composed with g**." There are two equivalent ways of thinking of it, as illustrated by this identity: **(f ∘ g)(x) = f(g(x))**. You can either think of **f ∘ g** as a single function or as the result of calling function **g** and then taking its output and passing that to **f**. Notice that the functions get applied from right to left—that is, we execute **g**, followed by **f**.

A couple of important points about function composition:

1. We can compose any number of functions (we're not limited to two).
2. One way to compose functions is simply to take the output from one function

```
// h(x) = x + 1
// number -> number
function h(x) {
  return x + 1;
```

Subscribe now X

Get the highlights in your inbox every week.

ıg

```
// y = (f ∘ g∘h)(1)
const y = f(g(h(1)));
console.log(y); // '4'
```

:0f5652785430381/raw/28a6dc814aaf7d023cebefb6d7f694d76f99f9da/function-

function-composition-basic.js (https://gist.github.com/battmanz/99325b35a147c37b20f5652785430381#file-function-composition-basic-js) hosted with ❤ by GitHub (https://github.com)

There are libraries such as Ramda (http://ramdajs.com/) and lodash (https://github.com/lodash/lodash/wiki/FP-Guide) that provide a more elegant way of composing functions. Instead of simply passing the return value from one function to the next, we can treat function composition in the more mathematical sense. We can create a single composite function made up from the others (i.e., **(f ∘ g)(x)**).

```
// h(x) = x + 1
// number -> number
function h(x) {
  return x + 1;
}

// g(x) = x^2
```

```
}

// f(x) = convert x to string
// number -> string
```

```
(f, g, h);

o get the result.
```

ac718d046ea74e/raw/a0c22d4a1afaf68c6297df3de4736c62e58cb028/function-

function-composition-elegant.js (https://gist.github.com/battmanz/e250ae6c628550f6f0ac718d046ea74e#file-function-composition-elegant-js) hosted with ❤ by GitHub (https://github.com)

Okay, so we can do function composition in JavaScript. What's the big deal? Well, if you're really onboard with functional programming, then ideally your entire program will be nothing but function composition. There will be no loops (**for**, **for...of**, **for...in**, **while**, **do**) in your code. None (period). But that's impossible, you say! Not so. That leads us to the next two topics: recursion and higher-order functions.

Recursion

Let's say that you would like to implement a function that computes the factorial of a number. Let's recall the definition of factorial from mathematics:

n! = n * (n-1) * (n-2) * ... * 1.

That is, **n!** is the product of all the integers from **n** down to **1**. We can write a loop that computes that for us easily enough.

```
  for (let i = 1; i <= n; i++) {
    product *= i;
  }
  return product;
```

Subscribe now X

Get the highlights in your inbox every week.

bc2d40f4fc6e137f7426b803337c5fa3bb/iterative-

anz/bc225959e1328e73b08c1fe4ab59b630#file-
by GitHub (https://github.com)

are repeatedly being reassigned inside the loop. pproach to solving the problem. How would we oach? We would need to eliminate the loop and es being reassigned. Recursion is one of the most powerful tools in the functional programmer's toolbelt. Recursion asks us to break down the overall problem into sub-problems that resemble the overall problem.

Computing the factorial is a perfect example. To compute **n!**, we simply need to take **n** and multiply it by all the smaller integers. That's the same thing as saying:

n! = n * (n-1)!

A-ha! We found a sub-problem to solve **(n-1)!** and it resembles the overall problem **n!**. There's one more thing to take care of: the base case. The base case tells us when to stop the recursion. If we didn't have a base case, then recursion would go on forever. In practice, you'll get a stack overflow error if there are too many recursive calls.

What is the base case for the factorial function? At first you might think that it's when **n == 1**, but due to some complex math stuff (https://math.stackexchange.com/questions/20969/prove-0-1-from-first-principles), it's when **n == 0**. **0!** is defined to be **1**. With this information in mind, let's write a recursive factorial function.

```
    return 1; // 0! is defined to be 1.
  }
  return n * recursiveFactorial(n - 1);
}
```

Subscribe now X

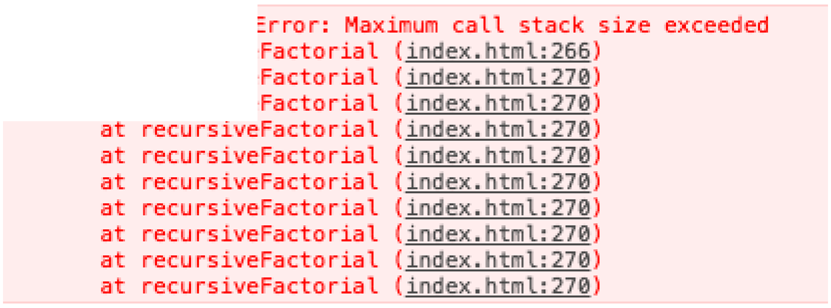
Get the highlights in your inbox every week.

ecc5adc21ab38f281a788e286abc107a/recursive-

anz/63961ad6fa380463785b69a1b34e7997#file-

❤ by GitHub (https://github.com)

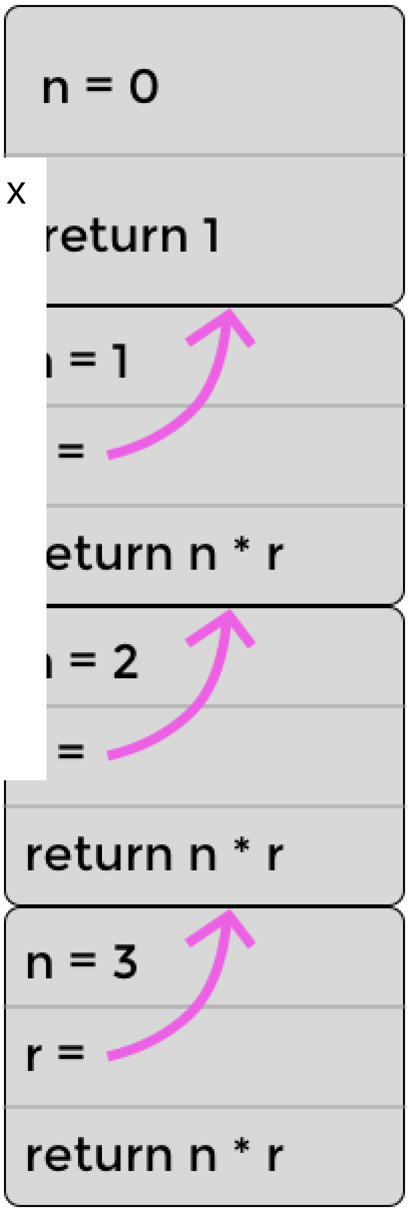
veFactorial(20000), because... well, why not!



```
Error: Maximum call stack size exceeded
Factorial (index.html:266)
Factorial (index.html:270)
Factorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
```

Stack overflow error

So what's going on here? We got a stack overflow error! It's not because of infinite recursion. We know that we handled the base case (i.e., **n === 0**). It's because the browser has a finite stack and we have exceeded it. Each call to **recursiveFactorial** causes a new frame to be put on the stack. We can visualize the stack as a set of boxes stacked on top of each other. Each time **recursiveFactorial** gets called, a new box is added to the top. The following diagram shows a stylized version of what the stack might look like when computing **recursiveFactorial(3)**. Note that in a real stack, the frame on top would store the memory address of where it should return after executing, but I have chosen to depict the return value using the variable **r**. I did this because JavaScript developers don't normally need to think about memory addresses.



The stack for recursively calculating 3! (three factorial)

You can imagine that the stack for **n = 20000** would be much higher. Is there anything we can do about that? It turns out that, yes, there is something we can do about it. As part of the **ES2015** (aka **ES6**) specification, an optimization was added to address this issue. It's called the *proper tail calls optimization* (PTC). It allows the browser to elide, or omit, stack frames if the last thing that the recursive function does is call itself and return the result. Actually, the optimization works for mutually recursive functions as well, but for simplicity we're just going to focus on a single recursive function.

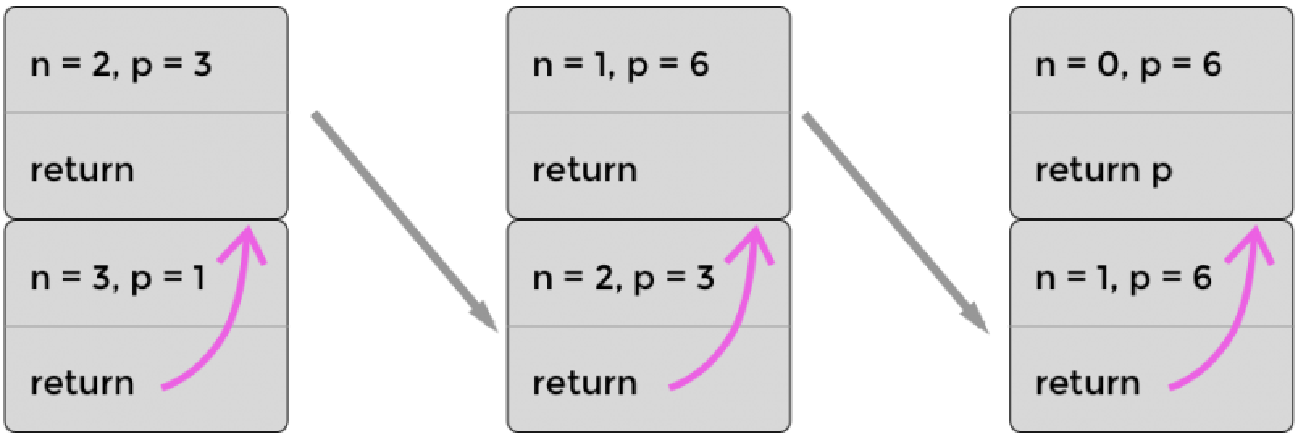
You'll notice in the stack above that after the recursive function call, there is still an additional computation to be made (i.e., **n * r**). That means that the browser so that the last step is the recursive call. One trick to doing that is to pass the intermediate result (in this case the **product**) into the function as an argument.

```
ptimization.
: = 1) {



pduct * n)
```

3b9d262dcd35e534145cba0f52b5d5d67/factorial-

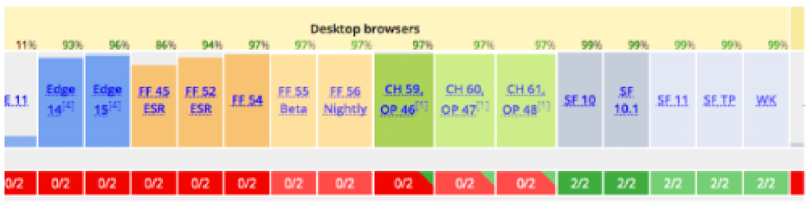
anz/26ecb25247a01030ca4ab0cd1ebfc5b3#file-factorial-tail-recursion-js) hosted with ❤ by GitHub (https://github.com)

Let's visualize the optimized stack now when computing **factorial(3)**. As the following diagram shows, in this case the stack never grows beyond two frames. The reason is that we are passing all necessary information (i.e., the **product**) into the recursive function. So, after the **product** has been updated, the browser can throw out that stack frame. You'll notice in this diagram that each time the top frame falls down and becomes the bottom frame, the previous bottom frame gets thrown out. It's no longer needed.



The optimized stack for recursively calculating 3! (three factorial) using PTC

Now run that in a browser of your choice, and assuming that you ran it in Safari overflow error, so that's good! Now what about all the other browsers? It turns out that Safari is the only browser that has implemented PTC, and it might be the only browser to ever implement it. See the following compatibility table:



Desktop browsers															
11%	93%	96%	86%	94%	97%	97%	97%	97%	97%	97%	99%	99%	99%	99%	99%
E 11	Edge 14[4]	Edge 15[4]	FF 45 ESR	FF 52 ESR	FF 54	FF 55 Beta	FF 56 Nightly	CH 59, OP 46[1]	CH 60, OP 47[1]	CH 61, OP 48[1]	SF 10	SF 10.1	SF 11	SF TP	WK
0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2

PTC compatibility

a competing standard called syntactic tail calls ;al-ptc-syntax#syntactic-tail-calls-stc) (STC). have to specify via new syntax that you would like e tail-call optimization. Even though there is not widespread browser support yet, it's still a good idea to write your recursive functions so that they are ready for tail-call optimization whenever (and however) it comes.

Higher-order functions

We already know that JavaScript has first-class functions that can be passed around just like any other value. So, it should come as no surprise that we can pass a function to another function. We can also return a function from a function. Voilà! We have higher-order functions. You're probably already familiar with several higher order functions that exist on the **Array.prototype**. For example, **filter**, **map**, and **reduce**, among others. One way to think of a higher-order function is: It's a function that accepts (what's typically called) a callback function. Let's see an example of using built-in higher-order functions:

```
const vehicles = [
  { make: 'Honda', model: 'CR-V', type: 'suv', price: 24045 },
  { make: 'Honda', model: 'Accord', type: 'sedan', price: 22455 },
  { make: 'Mazda', model: 'Mazda 6', type: 'sedan', price: 24195 },
  { make: 'Mazda', model: 'CX-9', type: 'suv', price: 31520 },
  { make: 'Toyota', model: '4Runner', type: 'suv', price: 34210 },
  { make: 'Toyota', model: 'Sequoia', type: 'suv', price: 45560 },
  { make: 'Ford', model: 'Explorer', type: 'suv', price: 31660 }
];

const averageSUVPrice = vehicles
                      suv')

                      ray) => sum + price / array.length, 0);

                      ); // 33399
```

Subscribe now X

Get the highlights in your inbox every week.

52ef116d5a2ae430e68c6b5650d1dd6f44a4/built-

anz/9eda50362457362f2d8a28384bf1adfc#file-

-js) hosted with ❤ by GitHub (https://github.com)

ıods on an array object, which is characteristic of object-oriented programming. If we wanted to make this a bit more representative of functional programming, we could use functions provided by Ramda or lodash/fp instead. We can also use function composition, which we explored in a previous section. Note that we would need to reverse the order of the functions if we use **R.compose**, since it applies the functions right to left (i.e., bottom to top); however, if we want to apply them left to right (i.e., top to bottom), as in the example above, then we can use **R.pipe**. Both examples are given below using Ramda. Note that Ramda has a **mean** function that can be used instead of **reduce**.

```
const vehicles = [
  { make: 'Honda', model: 'CR-V', type: 'suv', price: 24045 },
  { make: 'Honda', model: 'Accord', type: 'sedan', price: 22455 },
  { make: 'Mazda', model: 'Mazda 6', type: 'sedan', price: 24195 },
  { make: 'Mazda', model: 'CX-9', type: 'suv', price: 31520 },
  { make: 'Toyota', model: '4Runner', type: 'suv', price: 34210 },
  { make: 'Toyota', model: 'Sequoia', type: 'suv', price: 45560 },
  { make: 'Toyota', model: 'Tacoma', type: 'truck', price: 24320 },
  { make: 'Ford', model: 'F-150', type: 'truck', price: 27110 },
  { make: 'Ford', model: 'Fusion', type: 'sedan', price: 22120 },
  { make: 'Ford', model: 'Explorer', type: 'suv', price: 31660 }
];

// Using `pipe` executes the functions from top-to-bottom.
```

```
  R.mean
)(vehicles);

console.log(averageSUVPrice1); // 33399
```

```
the functions from bottom-to-top.
compose(

'suv')

2); // 33399
```

37d515f4166290802d4a3d89f80210c/composing-

anz/bee10f02a076f064f72e20cd4aea6b85#file-composing-higher-order-functions-js) hosted with ❤ by GitHub (https://github.com)

The advantage of the functional programming approach is that it clearly separates the data (i.e., **vehicles**) from the logic (i.e., the functions **filter**, **map**, and **reduce**). Contrast that with the object-oriented code that blends data and functions in the form of objects with methods.

Currying

Informally, **currying** is the process of taking a function that accepts **n** arguments and turning it into **n** functions that each accepts a single argument. The **arity** of a function is the number of arguments that it accepts. A function that accepts a single argument is **unary**, two arguments **binary**, three arguments **ternary**, and **n** arguments is **n-ary**. Therefore, we can define currying as the process of taking an **n-ary** function and turning it into **n** unary functions. Let's start with a simple example, a function that takes the dot product of two vectors. Recall from linear algebra that the dot product of two vectors **[a, b, c]** and **[x, y, z]** is equal to **ax + by + cz**.

```
function dot(vector1, vector2) {
  return vector1.reduce((sum, element, index) => sum += element * vector2[index], 0)
```

```
const v2 = [4, -2, -1];

console.log(dot(v1, v2)); // 1(4) + 3(-2) + (-5)(-1) = 4 - 6 + 5 = 3
```

Subscribe now X

Get the highlights in your inbox every week.

e5489652e1f4815bf810f98b4aba6f5ec934e6/dot-

anz/e28311f765a18fc6a841201912422d60#file-
Hub (https://github.com)

e it accepts two arguments; however, we can
ary functions, as the following code example
: is a unary function that accepts a vector and
that then accepts the second vector.

```
function curriedDot(vector1) {
  return function(vector2) {
    return vector1.reduce((sum, element, index) => sum += element * vector2[index],
  }
}

// Taking the dot product of any vector with [1, 1, 1]
// is equivalent to summing up the elements of the other vector.
const sumElements = curriedDot([1, 1, 1]);

console.log(sumElements([1, 3, -5])); // -1
console.log(sumElements([4, -2, -1])); // 1
```

)752e8fe3e3a0b8d/raw/c886e5ea1fd7b6e4a130925634c1d1d6f8ffc689/manual-

manual-currying.js (https://gist.github.com/battmanz/3a3694f87b9c48ac0752e8fe3e3a0b8d#file-manual-currying-js) hosted with ❤ by GitHub (https://github.com)

Fortunately for us, we don't have to manually convert each one of our functions to a curried form. Libraries including Ramda (http://ramdajs.com/docs/#curry) and lodash (https://lodash.com/docs/4.17.4#curry) have functions that will do it for us. In fact, they do a hybrid type of currying, where you can either call the function one argument at a time, or you can continue to pass in all the arguments at once, just like the original.

```
  return vector1.reduce((sum, element, index) => sum += element * vector2[index], 0
}

const v1 = [1, 3, -5];
```

```
ving for us!
dot);

Dot([1, 1, 1]);

); // -1
); // 1

l call the curried function with two arguments.
/2)); // 3
```

69c359774b76ee35/raw/99a2997e4609e9ca294d7b58e330f2cf6dbaefcb/fancy-

fancy-curying.js (https://gist.github.com/battmanz/3335a949ea88b8d969c359774b76ee35#file-fancy-currying-js) hosted with ❤ by GitHub (https://github.com)

Both Ramda and lodash also allow you to "skip over" an argument and specify it later. They do this using a placeholder. Because taking the dot product is commutative, it won't make a difference in which order we passed the vectors into the function. Let's use a different example to illustrate using a placeholder. Ramda uses a double underscore as its placeholder.

```
const giveMe3 = R.curry(function(item1, item2, item3) {
  return `
    1: ${item1}
    2: ${item2}
    3: ${item3}
  `;
});

const giveMe2 = giveMe3(R.__, R.__, 'French Hens');   // Specify the third argument
const giveMe1 = giveMe2('Partridge in a Pear Tree');  // This will go in the first
const result = giveMe1('Turtle Doves');               // Finally fill in the second

console.log(result);
```

557c78e43a9b5/raw/9b6556bd9111efd03dd69bee5596c29002e2279a/currying-

Subscribe now

Get the highlights in your inbox every week.

X

anz/ea5e1f34cf468214039557c78e43a9b5#file-
with ❤ by **GitHub** (https://github.com)

plete the topic of currying, and that is partial and currying often go hand in hand, though they curried function is still a curried function even if it nts. Partial application, on the other hand, is when e, but not all, of its arguments. Currying is often out it's not the only way.

a built-in mechanism for doing partial application without currying. This is done using the **function.prototype.bind** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) method. One idiosyncrasy of this method is that it requires you to pass in the value of **this** as the first argument. If you're not doing object-oriented programming, then you can effectively ignore **this** by passing in **null**.

```
function giveMe3(item1, item2, item3) {
  return `
    1: ${item1}
    2: ${item2}
    3: ${item3}
  `;
}

const giveMe2 = giveMe3.bind(null, 'rock');
const giveMe1 = giveMe2.bind(null, 'paper');
const result = giveMe1('scissors');

console.log(result);
// 1: rock
// 2: paper
// 3: scissors
```

e85af0aecfd6f6e3/raw/3d248b9810cab7d02ecd050ec549247499de6f31/partial-

partial-application-using-bind-js) hosted with ❤ by GitHub (https://github.com)

Subscribe now X

Get the highlights in your inbox every week.

ınctional programming in JavaScript with me! For
ew programming paradigm, but I hope you will
nd that your programs are easier to read and
low you to take advantage of performance
eact.

)penWest talk, [JavaScript the Good-er Parts](edule/#talk-5). OpenWest
ll be held July 12-15, 2017 in Salt Lake City, Utah.

Topics :

Programming (/tags/programming) **JavaScript** (/tags/javascript)
OpenWest (/tags/openwest)

(/users/battmanz)

About the author

Matt Banz - Matt graduated from the University of Utah with a degree in mathematics in May of 2008. One month later he landed a job as a web developer, and he's been loving it ever since! In 2013, he earned a Master of Computer Science degree from North Carolina State University. He has taught courses in web development for LDS Business College and the Davis School District Community Education program. He is currently a Senior Front-End Developer for Motorola Solutions. Matt is enthralled by functional...

• More about me (/users/battmanz)

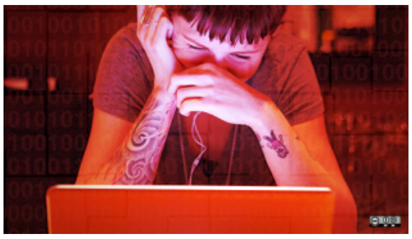
Recommended reading

Subscribe now

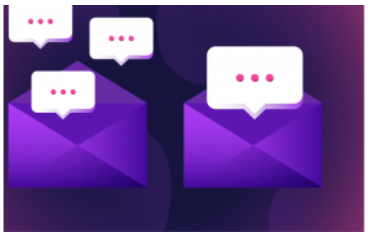
Get the highlights in your inbox every week.

X

[**An introduction to iting your own HTML web pages**]



[**When do you code?** (/article/20/3/when-do-you-code?]



[**line** (/article/20/3/markdown-apps-linux-command-line?]

[**hare data between C nd Python with this messaging library** (/article/20/3/zeromq-c-python? utm_campaign=intrel)]

[**Using C and C++ for data science** (/article/20/2/c-data-science? utm_campaign=intrel)]

2 Comments

Lewis Cowles (/users/lewiscowles1986) on 04 Aug 2017 3

Wow thanks. That is a lot of information. (31 screens deep. 29 without images, 17 with no images and no gists) I think this needs to be bookmarked and re-read several times for me to wrap my head around it.

I Have to something to ask after my first skim-though. If mutability & state are such bad ideas. Why are so many using them?

I'm aware there are situations where mutability is bad. Accidentally transforming the value of a users e-commerce cart could be bad for example. I'm just not sure the hard-line strict purity is practical for a broader system. I'd be happy to hear my initial feelings are incorrect, but would appreciate some links.

Cliff Stamp on 06 Sep 2017 3

common JS applications. If you consider some of the main uses of JS, they seem centered on impure functions as state has to change in web page updating.

It would be nice to some some simple, yet common applications, with a functional approach.