JANUARY 9, 2018 / #JAVASCRIPT

# A quick introduction to pipe() and compose() in JavaScript



Yazeed Bzadough



Functional programming's been quite the eye-opening journey for me. This post, and posts like it, are an attempt to share my insights and perspectives as I trek new functional programming lands.

# **Pipe**

The concept of pipe is simple—it combines n functions. It's a pipe flowing left-to-right, calling each function with the output of the last one.

Let's write a function that returns someone's name.

```
getName = (person) => person.name;

getName({ name: 'Buckethead' });
// 'Buckethead'
```

Let's write a function that uppercases strings.

```
uppercase = (string) => string.toUpperCase();

uppercase('Buckethead');
// 'BUCKETHEAD'
```

So if we wanted to get and capitalize person 's name, we could do this:

```
name = getName({ name: 'Buckethead' });
uppercase(name);
// 'BUCKETHEAD'
```

That's fine but let's eliminate that intermediate variable name.

```
uppercase(getName({ name: 'Buckethead' }));
```

to add a function that gets the first 6 characters of a string?

```
get6Characters = (string) => string.substring(0, 6);

get6Characters('Buckethead');
// 'Bucket'
```

# Resulting in:

```
get6Characters(uppercase(getName({ name: 'Buckethead' })));
// 'BUCKET';
```

Let's get really crazy and add a function to reverse strings.

```
reverse = (string) =>
  string
    .split('')
    .reverse()
    .join('');

reverse('Buckethead');
// 'daehtekcuB'
```

#### Now we have:

```
reverse(get6Characters(uppercase(getName({ name: 'Buckethead' }))));
// 'TEKCUB'
```

Instead of jamming functions within functions or creating a bunch of intermediate variables, let's pipe all the things!

```
pipe(
  getName,
  uppercase,
  get6Characters,
  reverse
)({ name: 'Buckethead' });
// 'TEKCUB'
```

Pure art. It's like a todo list!

Let's step through it.

For demo purposes, I'll use a pipe implementation from one of <u>Eric Elliott</u>'s <u>functional programming articles</u>.

```
pipe = (...fns) => (x) => fns.reduce((v, f) => f(v), x);
```

I love this little one-liner.

Using *rest* parameters, <u>see my article on that</u>, we can pipe n functions. Each function takes the output of the previous one and it's all *reduced*? to a single value.

And you can use it just like we did above.

```
pipe(
  getName,
```

```
// 'TEKCUB'
```

I'll expand pipe and add some debugger statements, and we'll go line by line.

```
pipe = (...functions) => (value) => {
  debugger;

  return functions.reduce((currentValue, currentFunction) => {
    debugger;

    return currentFunction(currentValue);
  }, value);
};
```

Call pipe with our example and let the wonders unfold.

```
5
             .reduce((currentValue, currentFunction) => {
 6
                 debugger;
 7
 8
                 return currentFunction(currentValue);
 9
            }, value)
 10 }
{}
    Line 2, Column 2
    Console
0
                             Filter
                                                           Default levels ▼
     top
> functions

√ ▼ (4) [f, f, f, f] 
⑤

    ▶ 0: (person) => person.name
    ▶ 1: (string) => string.toUpperCase()
    ▶ 2: (string) => string.substring(0, 6)
    ▶ 3: (string) => string .split('') .reverse() .join('')
      length: 4
    ▶ __proto__: Array(0)
 value
  ▶ {name: "Buckethead"}
```

Check out the local variables. functions is an array of the 4 functions, and value is { name: 'Buckethead' }.

Since we used *rest* parameters, pipe allows any number of functions to be used. It'll just loop and call each one.

```
4
        return functions
  5
             .reduce((currentValue, currentFunction) => {
  6
                 debugger;
  7
  8
                 return currentFunction(currentValue);
  9
             }, value)
    }
 10
{}
    Line 6, Column 4
    Console
0
                             Filter
     top
                                                           D
> currentValue
  ► {name: "Buckethead"}
  currentFunction
  (person) => person.name
  currentFunction(currentValue)
  "Buckethead"
```

On the next debugger, we're inside reduce. This is where currentValue is passed to currentFunction and returned.

We see the result is 'Buckethead' because currentFunction returns the .name property of any object. That will be returned in reduce, meaning it becomes the new currentValue next time. Let's hit the next debugger and see.

```
4
        return functions
             .reduce((currentValue, currentFunction) => {
  5
  6
                 debugger;
  7
  8
                 return currentFunction(currentValue);
            }, value)
  9
 10
    }
{}
    Line 6, Column 4
    Console
0
                             Filter
     top
                                                           D
  currentValue
  "Buckethead"
  currentFunction
<- (string) => string.toUpperCase()
> currentFunction(currentValue)
  "BUCKETHEAD"
```

Now currentValue is 'Buckethead' because that's what got returned last time. currentFunction is uppercase, so 'BUCKETHEAD' will be the next currentValue.

```
4
        return functions
  5
6
             .reduce((currentValue, currentFunction) => {
                 debugger;
  7
  8
                 return currentFunction(currentValue);
             }, value)
  9
 10
    }
{}
    Line 6, Column 4
    Console
0
                             Filter
     top
                                                           D
 currentValue
  "BUCKETHEAD"
> currentFunction
<- (string) => string.substring(0, 6)
> currentFunction(currentValue)
  "BUCKET"
```

The same idea, pluck 'BUCKETHEAD' 's first 6 characters and hand them off to the next function.

```
4
         return functions
  5
             .reduce((currentValue, currentFunction) => {
  6
                 debugger;
  7
  8
                 return currentFunction(currentValue):
             }. value)
  9
 10
    }
{}
     Line 6, Column 4
     Console
                              Filter
0
     top
                                                            D
  currentValue
"BUCKET"
  currentFunction
<- (string) => string
     .split('')
     .reverse()
     .join('')
 currentFunction(currentValue)
reverse('.aedi emaS')
```

```
pipe = (...functions) => (value) => {
  2
        debugger;
  3
  4
         return functions
  5
             .reduce((currentValue, currentFunction) => {
  6
                 debugger;
  7
                 return currentFunction(currentValue);
  8
             }, value)
  9
 10 }
{}
    Line 6, Column 4
    Console
0
                              Filter
                                                             Default
     top
  "TEKCUB"
```

And you're done!

# What about compose()?

It's just pipe in the other direction.

So if you wanted the same result as our pipe above, you'd do the opposite.

```
compose(
  reverse,
  get6Characters,
  uppercase,
  getName
)({ name: 'Buckethead' });
```

Notice how getName is last in the chain and reverse is first?

Here's a quick implementation of compose, again courtesy of the Magical <u>Eric</u> <u>Elliott</u>, from <u>the same article</u>.

```
compose = (...fns) \Rightarrow (x) \Rightarrow fns.reduceRight((v, f) \Rightarrow f(v), x);
```

I'll leave expanding this function with debugger s as an exercise to you. Play around with it, use it, appreciate it. And most importantly, have fun!



## Yazeed Bzadough

Front-End Developer creating content at https://yazeedb.com.

If you read this far, tweet to the author to show them you care.

Tweet a thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. Get started

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) nonprofit organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Git Reset to Remote

#### **Trending Guides**

ISO File

10 to the Power of 0 Recursion

R Value in Statistics ADB

What is Economics? MBR VS GPT

Module Exports Debounce

Python VS JavaScript Helm Chart

Model View Controller 80-20 Rule

Inductive VS Deductive JavaScript Keycode List

JavaScript Empty Array JavaScript Reverse Array

Best Instagram Post Time How to Screenshot on Mac

Garbage Collection in Java How to Reverse Image Search

Auto-Numbering in Excel Ternary Operator JavaScript

#### **Our Nonprofit**

About Alumni Network Open Source Shop Support Sponsors Academic Honesty Code of Conduct

Privacy Policy Terms of Service Copyright Policy