

# JavaScript : Part 4

- 1 : Functional Concepts For JavaScript Developers: Currying**
- 2 : Currying vs partial application**
- 3 : Binding objects to functions**
- 4 : Currying vs Partial Application**
- 5 : What's the difference between Currying and Partial Application?**
- 6 : Approaches to currying in JavaScript**
- 7 : 7 Essential JavaScript Functions**
- 8 : Collection of useful JavaScript functions and patterns**
- 9 : My todays top 10 most useful Javascript Functions**
- 10 : 12 Extremely Useful Hacks for JavaScript**
- 11 : 45 Useful JavaScript Tips, Tricks and Best Practices**
- 12 : JavaScript Architecture for the 23rd Century**
- 13 : 9 New Array Functions in ES6**
- 14 : Implementing EventEmitter in ES6**
- 15 : Getting Functional with Javascript (Part 1)**
- 16 : Getting Functional with Javascript (Part 2)**
- 17 : Understanding Prototypes, Delegation & Composition**
- 18 : Some jQuery Functions And Their JavaScript Equivalents**
- 19 : Is everything in JavaScript an Object?**
- 20 : Partial Application with Function#bind**
- 21 : Partial Application in JavaScript**
- 22 : Composing Functions in JavaScript**
- 23 : A JavaScript Invoke Function**
- 24 : YOU MIGHT NOT NEED JQUERY**

**25 : The JavaScript Bind Function**

**26 : Forcing Function Arity in JavaScript**

**27 : Wrapping JavaScript Functions**

**28 : Javascript : Basic Scope**

**29 : Part 2 : Functions**

**30 : Part 3 : .map, .reduce & .filter, Oh My!**

**31 : Part 4 : Object-ively Javascript**

**32 : Chapter 4: Currying**

**33 : Chapter 5: Coding by Composing**

**34 : Functional Mixins in ECMAScript 2015**

**35 : JavaScript Mixins, Subclass Factories, and Method Advice**

**36 : Compose me That: Function Composition in JavaScript**

**37 : Curry me This: Partial Application in JavaScript**

**38 : Functional Data Structures in JavaScript: The Basics**

**39 : Model-View-Controller (MVC) with JavaScript**

**40 : JavaScript Function Memoization**

**41 : Javascript Memoization**

**42 : Hello World : A "Hello World"-like example of Javascript using the MVC pattern.**

# 1 : Functional Concepts For JavaScript Developers: Currying

<https://blog.simpleblend.net/functional-javascript-concepts-currying/>



Before I understood the concept of currying, I was always filled with anxiety when hearing other developers describe it. They always made it seem so complicated and difficult to understand; as if an ominous thundercloud arrived every time the topic was mentioned. Well I assure you that the concept of currying, at least applied to programming, is not hard to understand at all. It's actually quite easy to grasp and once you do, opens the door to a powerful functional programming technique.

## The Core Idea

Let's first start with a definition from the great book by [Brian Lonsdorf](#)

"The concept is simple: You can call a function with fewer arguments than it expects. It returns a function that takes the remaining arguments."

In other words, a curry function takes and provides additional arguments over time. It does this by returning additional functions to do the work for us.<sup>1</sup> In order to see how this works, let's take a normal function and

curry it.

```
var add = function(x, y) {
    return x + y;
}

add(10, 20); // 30
```

Here we have a function that takes two numbers and returns the sum. Simple. Let's curry it.

```
var add = function(x) {
    return function(y) {
        return x + y;
    }
}

var addFifty = add(50);

addFifty(10); // 60
```

Instead of returning the immediate result, we're returning a function that acts as a closure over variable x. This allows us to keep a reference to that value when we execute the returned function **addFifty** at a later time. Finally, our returned function provides us with the summation of both numbers. In this example we would say the add function is "curried".

## Why is Currying useful?

Curry functions are useful for many reasons. From a functional programming perspective, they allow your program to become "pure". I'll be writing more about pure functions at a later time but suffice it to say for now, pure functions allow your application to be better understood by yourself and other team members.

A more applicable reason can be shown by demonstrating a less contrived example. While adding two numbers together is great for explaining a new concept, it doesn't immediately highlight the enormous benefits of using it within the context of a larger application. Let's do that now.

Let's assume we're working with a list of animals like the below:

```
const animals = [
    {
        id: 1,
        name: "Tails",
        type: "Cat",
```

```
        adoptable: false
    },
    {
        id: 2,
        name: "Soul",
        type: "Cat",
        adoptable: true
    },
    {
        id: 3,
        name: "Fred",
        type: "Dog",
        adoptable: true
    },
    {
        id: 4,
        name: "Fury",
        type: "Lion",
        adoptable: true
    }
];

```

We need a way to filter down these animals based on type. We could do something like this:

```
animals.filter(animal =>
    animal.type === "Cat"
);

/*
[{
    adoptable: false,
    id: 1,
    name: "Tails",
    type: "Cat"
},
{
    adoptable: true,
    id: 2,
    name: "Soul",
    type: "Cat"
}
```

```
 }];
*/
```

Looks pretty good. This grabs all of the cats in the list. However there's a better way to write the same thing.

One major drawback to the above code is that the *type* of animal is tightly coupled to *the act filtering itself*. This prevents reusability and encourages the cultivation of our worst enemy: state.

Let's see if we can do better.

```
const isAnimal =
  type =>
    animal =>
      animal.type === type

  animals.filter(isAnimal("Cat"));

/*
[{
  adoptable: false,
  id: 1,
  name: "Tails",
  type: "Cat"
},
{
  adoptable: true,
  id: 2,
  name: "Soul",
  type: "Cat"
}];
*/
```

Much cleaner. Let's break this down into steps.

- isAnimal expects one argument, the type
- isAnimal then returns a whole new function
- The returned function is used as the callback to .filter()
- Closure then allows our returned function to access the **type** variable which finally allows us to perform our equality comparison

Again, why is this better than the previous example? As an application grows in size, it becomes harder to understand what functions are doing what and where they're coming from within the codebase. Striving to

write pure functions like in our solution eliminates the state chase, and makes our entire app more testable by breaking things into single responsibilities.

## Conclusion & Resources

I hope this inspired you to see the powerful nature behind curry functions. I've personally found them very useful and really inspired me to learn more about functional programming concepts.

If you're interested in further learning, I encourage you to checkout functional libraries like [Ramda.js](#) or [lodash](#) which contain general curry functions that can help build some really exciting functionality.

## 2 : Currying vs partial application

<http://www.jstips.co/en/javascript/curry-vs-partial-application/>

### Currying

Currying takes a function

$f: X * Y \rightarrow R$

and turns it into a function

$f': X \rightarrow (Y \rightarrow R)$

Instead of calling  $f$  with two arguments, we invoke  $f'$  with the first argument. The result is a function that we then call with the second argument to produce the result.

Thus, if the uncurried  $f$  is invoked as

$f(3,5)$

then the curried  $f'$  is invoked as

$f(3)(5)$

For example: Uncurried add()

```
function add(x, y) {  
    return x + y;  
}  
  
add(3, 5); // returns 8
```

Curried add()

```
function addC(x) {  
    return function (y) {  
        return x + y;  
    }  
}  
  
addC(3)(5); // returns 8
```

### The algorithm for currying.

Curry takes a binary function and returns a unary function that returns a unary function.

curry:  $(X \times Y \rightarrow R) \rightarrow (X \rightarrow (Y \rightarrow R))$

Javascript Code:

```
function curry(f) {
  return function(x) {
    return function(y) {
      return f(x, y);
    }
  }
}
```

## Partial application

Partial application takes a function

$f: X * Y \rightarrow R$

and a fixed value for the first argument to produce a new function

$f': Y \rightarrow R$

$f'$  does the same as  $f$ , but only has to fill in the second parameter which is why its arity is one less than the arity of  $f$ .

For example: Binding the first argument of function add to 5 produces the function plus5.

```
function plus5(y) {
  return 5 + y;
}

plus5(3); // returns 8
```

## The algorithm of partial application.\*

partApply takes a binary function and a value and produces a unary function.

partApply :  $((X \times Y \rightarrow R) \times X) \rightarrow (Y \rightarrow R)$

Javascript Code:

```
function partApply(f, x) {
  return function(y) {
```

```
    return f(x, y);  
}  
}
```

## 3 : Binding objects to functions

<http://www.jstips.co/en/javascript/binding-objects-to-functions/>

More than often, we need to bind an object to a function's this object. JS uses the bind method when this is specified explicitly and we need to invoke desired method.

### Bind syntax

```
fun.bind(thisArg[, arg1[, arg2[, ...]]])
```

### Parameters

#### thisArg

`this` parameter value to be passed to target function while calling the `bound` function.

#### arg1, arg2, ...

Prepended arguments to be passed to the `bound` function while invoking the target function.

### Return value

A copy of the given function along with the specified `this` value and initial arguments.

### Bind method in action in JS

```
const myCar = {  
    brand: 'Ford',  
    type: 'Sedan'  
    Color: 'Red'  
};  
  
const getBrand = () => {  
    console.log(this.brand);  
};  
  
const getType = () => {  
    console.log(this.type);  
};  
  
const getColor = () => {  
    console.log(this.color);  
};
```

```
};

getBrand(); // object not bind,undefined

getBrand(myCar); // object not bind,undefined

getType.bind(myCar)(); // Sedan

getColor.bind(myCar); // Red
```

## 4 : Currying vs Partial Application

<http://www.datchley.name/currying-vs-partial-application/>

In this post we'll cover a couple techniques in Javascript that are common among functional languages: *currying* and *partial application*. You don't need a background in functional programming to understand these, so don't worry.

### A quick review about functions...

So, I'll assume you have a basic understanding of functions in Javascript, including [higher-order functions](#), [closures](#) and [call & apply](#); if not, go review those topics quickly and come back.

Now that you're back, let's review a couple of important points before moving ahead.

**Arity** refers to the number of arguments a function can accept. This might be none, one (*unary*), two (*binary*) or more (*polyadic*). You can also have functions that take a variable number of arguments, (*variadic functions*).

Functions allow you to access their arity via the `.length` property of the function. The `.length` of a function never changes - it always matches the number of declared arguments for the function.

```
function howMany(a,b,c) {
  console.log(howmany.length);
}

howMany(1,2);      // 3
howMany(1,2,3,4); // 3
```

Obviously, it's up to you and your function to properly handle the cases of too few and too many arguments.

**Variadic Functions** are those functions which take a variable number of arguments.

Javascript allows us to access all the arguments passed to a function via the `arguments` variable made available inside a function's scope. This variable contains an *array-like* list of all the arguments pass to the function when it was called.

I say, *array-like*, because even though it's a list, it only has a `.length` and none of the other properties you'd expect of a real Array. You can access it via indexing `[]`, get the `.length` and iterate over it using a loop construct - and that's about it.

But we can convert this into a 'real' array which will make the arguments list much more useful to us:

```
function showArgs() {
  var args = [].slice.call(arguments);
```

```
}
```

The `[].slice.call(arguments)` is a short-handed way of doing `Array.prototype.slice.call(arguments)`, we just take advantage of the use of an array literal.

In ES6 we can access and 'unpack' our arguments even more easily with the help of the spread/gather operator:

```
function howMany(...args) {
  console.log("args:", args, ", length:", args.length);
}

howMany(1,2,3,4); // args: [1,2,3,4], length: 4 (a "real" array)!
```

With all that out of the way -- which I'm sure you knew already -- we can move on to our main topic!

## Currying

**Currying** is the process of taking a function that accepts  $N$  arguments and turning it into a chained series of  $N$  functions each taking 1 argument.

If we had an `add()` function that accepted 3 arguments and returned the sum,

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

we can turn it into a curried function as follows:

```
function curriedAdd(a) {
  return function(b) {
    return function(c) {
      return a+b+c;
    }
  }
}
```

How does currying work? It works by nesting functions for each possible argument, using the natural closure created by the nested functions to retain access to each of the successive arguments.



What we want is a way to easily convert an existing function that takes  $N$  arguments into its curried version without having to write-out each curried version of a function as we did with `curriedAdd()`.

Let's see if we can decompose this and build something useful.

## Writing a generic curry()

Ideally, this is the `curry()` function interface we'd like to design:

```
function foo(a,b,c){ return a+b+c; }
var curriedFoo = curry(foo);

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

Our `curry()` returns a new function that allows us to call it with one or more arguments, which it will then partially apply; up until it receives the last argument (*based on the original function's arity*) at which point it will return the evaluation of invoking the original function with all the arguments.

We know we can access the *arity* of a function using the `.length` property of the function. We can use this knowledge to allow us to know how many chained sequences of that function we need to call.

And, we'll need to store the original function passed in as well, so that once we have all the required arguments we can call the original function with the proper arguments and return its results.

Here's our first attempt:

```
function curry(fn) {
  return function curried() {
    var args = [].slice.call(arguments);
    return args.length >= fn.length ?
      fn.apply(null, args) :
      function () {
        var rest = [].slice.call(arguments);
        return curried.apply(null, args.concat(rest));
      };
  };
}
```

Let's break this down in detail...

- **line 2** our `curry` function returns a new function, in this case a named function expression called `curried()`.
- **line 3** every time this function is called, we store the arguments passed to it in `args`
- **line 4** if the number of arguments is equal to the arity of the original function, we have them all, so
- **line 5** return the invocation of the original function with all the arguments

- **line 6** otherwise, return a function that will accept more arguments that, when called, will call our `curried` function again with the original arguments passed previously combined with the arguments passed to the newly returned function.

Let's give this a try with our original `add` function from before.

```
var curriedAdd = curry(add);
curriedAdd(1)(2)(3);
// 6
curriedAdd(1)(2,3);
// 6
```

Excellent! This seems to do exactly what we want it to do. However, suppose we had an object with functions that depended on the proper object being set to the calling context (`this`) of the function. Can we use our `curry` function to curry an object method?

```
var border = {
  style: 'border',
  generate: function(length, measure, type, color) {
    return [this.style + ':', length + measure, type, color].join(' ') +(';';
  }
};

border.curriedGenerate = curry(border.generate);

border.curriedGenerate(2)('px')('solid')('#369')
// => "undefined: 2px solid #369;"
```

Uh oh. That's not what we wanted.

Using our `curry()` function as a method decorator<sup>1</sup> seems to break the object context expected by the method. We'll have to retain the original context and be sure and pass it along to successive calls to the returned `curried` function.

```
function curry(fn) {
  return function curried() {
    var args = toArray(arguments),
      context = this;

    return args.length >= fn.length ?
      fn.apply(context, args) :
```

```

        function () {
            var rest = toArray(arguments);
            return curried.apply(context, args.concat(rest));
        };
    }
}

```

Let's try it again.

```

border.curriedGenerate(2)('px')('solid')('#369')
// => "border: 2px solid #369;"

```

Got it! Now our `curry()` function is context aware and can be used in any number of situations as a function decorator.

## Currying Variadic Functions?

So our current solution works and correctly retains the context when called, as long as those functions being curried only accept exactly the number of arguments they declare - no more, no less. This doesn't help us if we want to curry a function that has optional declared arguments or a variable number of arguments (*variadic functions*).

With variable argument functions, we need a way to tell our `curry()` function *when* it has enough arguments to evaluate the original function that it's currying.

Take the following functions

```

function max(/* variable arguments */) {
    var args = [].slice.call(arguments);
    return Math.max.apply(Math, args);
}

function range(start, end, step) {
    var stop = Math.max(start, end),
        start = Math.min(start, end),
        set = [];

    // step is optional
    step = typeof step !== 'undefined' ? step : 1;

    for (var i=start; i <=stop; i+=step) {
        set.push(i);
    }
}

```

```

    }
    return set;
}

```

In the above, if we tried to use `curry(max)(1)(2)(3)` it evaluates too soon and we get a `TypeError`.

If we try to use `curry(range)(1)(10)` it never evaluates and simply stops by returning us a function that is still expecting another argument.

There is no feasible implementation of `curry` which will suffice for the example of our `max()` function which can take any number of arguments. Without an arity or a minimum number of arguments, there's no efficient way to determine when we should evaluate the original function with the arguments up to that point.

However, we can try to handle the case of optional, trailing arguments as in our `range()` example; and with minimal changes to our original `curry()` implementation.

We can modify our original `curry()` function to take an optional second argument which is the minimum number of arguments to curry.

```

function curry(fn, n) {
  var arity = n || fn.length;
  return function curried() {
    var args = toArray(arguments),
        context = this;

    return args.length >= arity ?
      fn.apply(context, args) :
      function () {
        var rest = toArray(arguments);
        return curried.apply(context, args.concat(rest));
      };
  };
}

```

Now, we can call `curry(range, 2)(1)(10)` as well as `curry(range, 2)(1)(10, 2)`. Unfortunately, though, we can't call it as `curry(range, 2)(1)(10)(2)`, because as soon as it sees the minimum number of arguments, 2 in this case, it will return the results of evaluating the curried function.

## What to do about Currying then?

It's clear from our examination above that currying in Javascript is definitely possible and useful. However, because Javascript allows any function to be variadic by nature, it becomes inefficient to implement a `curry()` function that can handle all possible cases.

**Solution:** If you're writing in a functional style, with unary and/or binary functions; and those functions take specific arguments that are declared, take advantage of currying. Otherwise, using our original implementation of `curry()` with the understanding that there are limitations surrounding functions with optional or variable arguments might be the best approach.

## Partial Application

That was a lot of talking about currying; so, what is *partial application*?

**Partial application** means taking a function and *partially* applying it to one or more of its arguments, but not all, creating a new function in the process.

Javascript already lets you do this with `Function.prototype.bind()`

```
function add3(a, b, c) { return a+b+c; }
add3(2,4,8); // 14

var add6 = add3.bind(this, 2, 4);
add6(8); // 14
```

But, that sounds a lot like what our `curry()` function does internally. If you have `curry()`, you also have partial application!<sup>2</sup>

```
var add6 = curry(add3)(2)(4);
add6(8); // 14
```

The primary difference is in how you use them. We can implement a partial application function fairly easily (*this is applying arguments from left to right*)

```
// Apply left arbitrary number of arguments
function apply(fn /* partial arguments... */) {
    var args = [].slice.call(arguments, 1);
    return function() {
        var _args = [].slice.call(arguments);
        return fn.apply(this, args.concat(_args));
    }
}
```

And we can call this, much like `bind` but without the initial context argument, like `var add6 = apply(add3, 2, 4)`.

## ES6 curry and apply implementations

To wrap things up, I promised to cover the ES6 implementations as well, so here is `curry()`,

```
function curry(fn) {  
    return function curried(...args) {  
        return args.length >= fn.length ?  
            fn.call(this, ...args) :  
            (...rest) => {  
                return curried.call(this, ...args, ...rest);  
            };  
    };  
}
```

and here's our `apply()` function in ES6 as well.

```
// Apply left arbitrary number of arguments  
function apply(fn, ...args) {  
    return (..._args) => {  
        return fn(...args, ..._args);  
    };  
}
```

The `...` (spread/gather) operator and `=>` fat arrow functions simplify the amount of code we need to implement our previous ES5 versions of `curry` and `apply`; and, I think they make the implementation more clear and understandable as well.<sup>3</sup>

# 5 : What's the difference between Currying and Partial Application?

<http://raganwald.com/2013/03/07/currying-and-partial-application.html>

I was participating in [wroc\\_love.rb](#) last week-end, and [Steve Klabnik](#) put up a slide mentioning [partial application](#) and [currying](#). "The difference between them is not important right now," he said, pressing on. And it wasn't.

But here we are, it's a brand new day, and we've already read five different explanations of [this](#) and closures this week, but only three or four about currying, so let's get into it.

## arity

Before we jump in, let's get some terminology straight. Functions have *arity*, meaning the number of arguments they accept. A "unary" function accepts one argument, a "polyadic" function takes more than one argument. There are specialized terms we can use: A "binary" function accepts two, a "ternary" function accepts three, and you can rustle about with greek or latin words and invent names for functions that accept more than three arguments.

Some functions accept a variable number of arguments, we call them *variadic*, although variadic functions and functions taking no arguments aren't our primary focus in this essay.

## partial application

Partial application is straightforward. We could start with addition or some such completely trivial example, but if you don't mind we'll have a look at something from the [allong.es](#) JavaScript library that is of actual use in daily programming.

As a preamble, let's make ourselves a [map](#) function that maps another function over a list:

```
var __map = [].map;

function map (list, unaryFn) {
    return __map.call(list, unaryFn);
};

function square (n) {
    return n * n;
};

map([1, 2, 3], square);
//=> [1, 4, 9]
```

`map` is obviously a binary function, `square` is a unary function. When we called `map` with arguments `[1, 2, 3]` and `square`, we *applied* the arguments to the function and got our result.

Since `map` takes two arguments, and we supplied two arguments, we *fully applied* the arguments to the function. So what's partial application? Supplying fewer arguments. Like supplying one argument to `map`.

Now what happens if we supply one argument to `map`? We can't get a result without the other argument, so what we get back is a unary function that takes the other argument and produces the result we want.

If we're going to apply one argument to `map`, let's make it the `unaryFn`. We'll start with the end result and work backwards. First thing we do, is set up a wrapper around `map`:

```
function mapWrapper (list, unaryFn) {
  return map(list, unaryFn);
};
```

Next we'll break our binary wrapper function into two nested unary functions:

```
function mapWrapper (unaryFn) {
  return function (list) {
    return map(list, unaryFn);
  };
};
```

Now we can supply our arguments one at a time:

```
mapWrapper(square)([1, 2, 3]);
//=> [1, 4, 9]
```

Instead of a binary `map` function that returns our result, we now have a unary function that returns a unary function that returns our result. So where's the partial application? Let's get our hands on the second unary function:

```
var squareAll = mapWrapper(square);
//=> [function]

squareAll([1, 2, 3]);
//=> [1, 4, 9]
squareAll([5, 7, 5]);
//=> [25, 49, 25]
```

We've just partially applied the value `square` to the function `map`. We got back a unary function, `squareAll`, that we could use as we liked. Partially applying `map` in this fashion is handy, so much so that the [allong.es](#) library includes a function called `mapWith` that does this exact thing.

If we had to physically write ourselves a wrapper function every time we want to do some partial application, we'd never bother. Being programmers though, we can automate this. There are two ways to do it.

## currying

First up, we can write a function that returns a wrapper function. Sticking with binary function, we start with this:

```
function wrapper (unaryFn) {
  return function (list) {
    return map(list, unaryFn);
  };
}
```

Rename `map` and the two arguments:

```
function wrapper (secondArg) {
  return function (firstArg) {
    return binaryFn(firstArg, secondArg);
  };
}
```

And now we can wrap the whole thing in a function that takes `binaryFn` as an argument:

```
function rightmostCurry (binaryFn) {
  return function (secondArg) {
    return function (firstArg) {
      return binaryFn(firstArg, secondArg);
    };
  };
}
```

So now we've 'abstracted' our little pattern. We can use it like this:

```
var rightmostCurriedMap = rightmostCurry(map);

var squareAll = rightmostCurriedMap(square);
```

```
squareAll([1, 4, 9]);
//=> [1, 4, 9]
squareAll([5, 7, 5]);
//=> [25, 49, 25]
```

Converting a polyadic function into a nested series of unary functions is called **currying**, after Haskell Curry, who popularized the technique. He actually rediscovered the combinatory logic work of [Moses Schönfinkel](#), so we could easily call it "schönfinkeling."<sup>1</sup>

Our `rightmostCurry` function curries any binary function into a chain of unary functions starting with the second argument. This is a "rightmost" curry because it starts at the right.

The opposite order would be a "leftmost" curry. Most logicians work with leftmost currying, so when we write a leftmost curry, most people just call it "curry":

```
function curry (binaryFn) {
  return function (firstArg) {
    return function (secondArg) {
      return binaryFn(firstArg, secondArg);
    };
  };
}

var curriedMap = curry(map),
  double = function (n) { return n + n; };

var oneToThreeEach = curriedMap([1, 2, 3]);

oneToThreeEach(square);
//=> [1, 4, 9]
oneToThreeEach(double);
//=> [2, 4, 6]
```

When would you use a regular curry and when would you use a rightmost curry? It really depends on your usage. In our binary function example, we're emulating a kind of subject-object grammar. The first argument we want to use is going to be the subject, the second is going to be the object.

So when we use a rightmost curry on `map`, we are setting up a "sentence" that makes the mapping function the subject.

So we read `squareAll([1, 2, 3])` as "square all the elements of the array [1, 2, 3]." By using a rightmost curry, we made "squaring" the subject and the list the object. Whereas when we used a regular curry, the list

became the subject and the mapper function became the object.

Another way to look at it that is a little more like "patterns and programming" is to think about what you want to name and/or reuse. Having both kinds of currying lets you name and/or reuse either the mapping function or the list.

## partial application

So many words about currying! What about "partial application?" Well, if you have currying you don't need partial application. And conversely, if you have partial application you don't need currying. So when you write this kind of essay, it's easy to spend a lot of words describing one of these two things and then explain everything else on top of what you already have.

Let's look at our rightmost curry again:

```
function rightmostCurry (binaryFn) {
  return function (secondArg) {
    return function (firstArg) {
      return binaryFn(firstArg, secondArg);
    };
  };
};
```

You might find yourself writing code like this over and over again:

```
var squareAll = rightmostCurry(map)(square),
  doubleAll = rightmostCurry(map)(double);
```

This business of making a rightmost curry and then immediately applying an argument to it is extremely common, and when something's common we humans like to name it. And it has a name, it's called a *rightmost unary partial application of the map function*.

What a mouthful. Let's take it step by step:

1. rightmost: From the right.
2. unary: One argument.
3. partial application: Not applying all of the arguments.
4. map: To the map function.

So what we're really doing is applying one argument to the map function. It's a binary function, so that means what we're left with is a unary function. Again, functional languages and libraries almost always include a first-class function to do this for us.

We *could* build one out of a rightmost curry:

```
function rightmostUnaryPartialApplication (binaryFn, secondArg) {
  return rightmostCurry(binaryFn)(secondArg);
};
```

However it is usually implemented in more direct fashion.<sup>2</sup>

```
function rightmostUnaryPartialApplication (binaryFn, secondArg) {
  return function (firstArg) {
    return binaryFn(firstArg, secondArg);
  };
};
```

`rightmostUnaryPartialApplication` is a bit much, so we'll alias it `applyLast`:

```
var applyLast = rightmostUnaryPartialApplication;
```

And here're our `squareAll` and `doubleAll` functions built with `applyLast`:

```
var squareAll = applyLast(map, square),
  doubleAll = applyLast(map, double);
```

You can also make an `applyFirst` function (we'll skip calling it "leftmostUnaryPartialApplication"):

```
function applyFirst (binaryFn, firstArg) {
  return function (secondArg) {
    return binaryFn(firstArg, secondArg);
  };
};
```

As with leftmost and rightmost currying, you want to have both in your toolbox so that you can choose what you are naming and/or reusing.

## so what's the difference between currying and partial application?

"Currying is the decomposition of a polyadic function into a chain of nested unary functions. Thus decomposed, you can partially apply one or more arguments,<sup>3</sup> although the curry operation itself does not apply any arguments to the function."

"Partial application is the conversion of a polyadic function into a function taking fewer arguments by providing one or more arguments in advance."

## is that all there is?

Yes. And no. Here are some further directions to explore on your own:

1. We saw how to use currying to implement partial application. Is it possible to use partial application to implement currying? Why? Why not?<sup>4</sup>
2. All of our examples of partial application have concerned converting binary functions into unary functions by providing one argument. Write more general versions of `applyFirst` and `applyLast` that provide one argument to any polyadic function. For example, if you have a function that takes four arguments, `applyFirst` should return a function taking three arguments.
3. When you have `applyFirst` and `applyLast` working with all polyadic functions, try implementing `applyLeft` and `applyRight`: `applyLeft` takes a polyadic function and one *or more* arguments and leftmost partially applies them. So if you provide it with a ternary function and two arguments, it should return a unary function. `applyRight` does the same with rightmost application.
4. Rewrite `curry` and `rightmostCurry` to accept any polyadic function. So just as a binary function curries into two nested unary functions, a ternary function should curry into three nested unary functions and so on.
5. Review the source code for [allong.es](#), the functional programming library extracted from [JavaScript Allongé](#), especially [partial\\_application.js](#).

Thanks for reading, if you discover a bug in the code, please either [fork the repo](#) and submit a pull request, or [submit an issue on Github](#).

## 6 : Approaches to currying in JavaScript

<http://extralogical.net/articles/currying-javascript.html>

JavaScript's dynamic nature makes it hard to straightforwardly apply many functional programming idioms. One example of this is [currying](#): any function may be passed an arbitrary number of arguments, making it impossible to write a truly general currying function.

To recap, currying is a technique for transforming a function which accepts  $n$  parameters into a nest of partially applicable functions. Consider the function  $f = \lambda xyz.M$ , which has three parameters,  $x$ ,  $y$  and  $z$ . By currying, we obtain a new function  $f^* = \lambda x.(\lambda y.(\lambda z.M))$ .

One simple example is currying an `add` function which accepts 2 parameters and returns the result of adding them together.

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

var curriedAdd = function(a) {
    return function(b) {
        return a + b;
    };
};
```

A function which returns the result of evaluating a quadratic expression demonstrates more clearly the 'nesting' of functions which currying produces.

```
var quadratic = function(a, b, c, x) {
    return a * x * x + b * x + c;
};

var curriedQuadratic = function(a) {
    return function(b) {
        return function(c) {
            return function(x) {
                return a * x * x + b * x + c;
            };
        };
    };
};
```

Given a pattern like this, the obvious question is how to generalise it. Ideally, we would write a `curry` function to automatically transform functions like `quadratic` into ones like `curriedQuadratic`. The simplest approach is to make curried functions always return a single wrapping function:

```
var naiveCurry = function(f) {
    var args = Array.prototype.slice.call(arguments, 1);

    return function() {
        var largs = Array.prototype.slice.call(arguments, 0);
        return f.apply(this, args.concat(largs));
    };
};
```

Clearly this is not true currying, except for functions of arity 2. We cannot use it to perform the transformation from `quadratic` to `curriedQuadratic`.

A cleverer approach would be to detect the arity of the function we wish to curry. To do this, we can use the `length` property of the function, which returns the number of named arguments the function accepts.

`Math.tan.length` is 1, while `parseInt.length` is 2.

```
var curryByLength = function(f) {
    var arity = f.length,
        args = Array.prototype.slice.call(arguments, 1),

        accumulator = function() {
            var largs = args;

            if (arguments.length > 0) {
                // We must be careful to copy the `args` array with `concat` rather
                // than mutate it; otherwise, executing curried functions can have
                // strange non-local effects on other curried functions.
                largs = largs.concat(Array.prototype.slice.call(arguments, 0));
            }

            if (largs.length >= arity) {
                return f.apply(this, largs);
            } else {
                return curryByLength.apply(this, [f].concat(largs));
            }
        };
};
```

```
    return args.length >= arity ? accumulator() : accumulator;
};
```

However, the length property of any given JavaScript function can easily mislead. To begin with, we often find it useful to define functions with optional parameters.

```
var someFunction = function(a, flag) {
  if (flag) {
    // Some computation involving a
  } else {
    // Some other computation involving a
  }
};
```

Now consider a variadic function, like `Math.max`, which returns the largest number amongst its arguments. Despite the fact that it can in fact be called with any number of arguments, including 0 and 1, it has a length property of 2. Consequently, our ‘smarter’ curry function will only work with `Math.max` up to a point. This will throw a type error, even though `Math.max` will accept three arguments quite happily:

```
curryByLength(Math.max)(1)(2)(3);
```

Currying `Math.max` limits its utility to discriminating between two numbers, not  $n$  numbers. We can easily think of similar examples—other variadic functions, functions with optional arguments, and similarly clever abuses of JavaScript’s dynamic arguments to create complex APIs. jQuery’s `bind` method could be considered [an example](#) of this: the event handler can be passed to the method as either the second or the third argument, depending on whether the user wishes to use the optional `eventData` parameter or not.

It is easy to see that there is no general way of resolving this issue: currying is essentially at odds with variadic functions and the ability to change the number of arguments a function accepts at runtime. However, one’s choices are not limited simply to the approaches discussed above; there are alternatives, even if they do not fully dispose of the problem of dynamic arity.

Firstly, one can simply leave things as they are, with the `curry` function having a known limitation around functions with dynamic arity. The burden is placed on the user to ensure they take care when currying.

Alternatively, one could make the arity an explicit component of the `curry` function. This differs from the implicit detection of the arity via the curried function’s length property (however, the implementation is almost identical).

```
var curryWithExplicitArity = function(f, n) {
  var args = Array.prototype.slice.call(arguments, 2),
```

```

    accumulator = function() {
        var largs = args;

        if (arguments.length > 0) {
            largs = largs.concat(Array.prototype.slice.call(arguments, 0));
        }

        if (largs.length >= n) {
            return f.apply(this, largs);
        } else {
            return curryByLength.apply(this, [f].concat(largs));
        }
    };

    return args.length >= n ? accumulator() : accumulator;
};

```

Finally, one could have entirely different `curry` functions for each arity. This has the benefit of being explicit, and while it doesn't solve the problem of functions with dynamic arity, it does mean that one doesn't have to specify the arity of the function one wishes to curry each time as an additional parameter. Instead of writing `curry(f, 3)`, one can simply write `curry3(f)`.

In fact, there is a way to combine these last two approaches, by writing a function which generates `curry` functions for any given arity.

```

var ncurry = function(n) {
    var _curry = function(f) {
        var args = Array.prototype.slice.call(arguments, 1),

        return function() {
            var largs = args.concat(Array.prototype.slice.call(arguments, 0));

            if (largs.length < n) {
                largs.unshift(f);
                return _curry.apply(null, largs);
            } else {
                return f.apply(null, largs);
            }
        };
    };
};

```

```
    return _curry;  
};
```

For common use cases such as functions which accept 2 or 3 arguments, one can write simple aliases using `nCurry`, while one can always use `nCurry` 'inline' where necessary.

```
var curry = nCurry(2),  
    curry3 = nCurry(3);  
  
// Presumably `f7` is a function which accepts 7 arguments  
var fc7 = nCurry(7)(f7);
```

However, oftentimes something along the lines of `curryByLength` is preferable. If the library of functions one is working with consists of a set of functions with well-defined lists of parameters, then implicit rather than explicit conversion can be more convenient and more natural; it is, after all, rather nicer to be able to write `curry(f)` than `curry(f, n)` or even `nCurry(n)(f)`.

Ultimately which approach one decides to take must be based on understanding of the properties of the functions one is working with. A choice of currying function will then arise naturally—and after all, one can always use several. Both of these approaches are [available in Udon](#), my library for functional programming in JavaScript, as `Udon.curry` and `Udon.nCurry`.

## 7 : 7 Essential JavaScript Functions

<https://davidwalsh.name/essential-javascript-functions>

I remember the early days of JavaScript where you needed a simple function for just about everything because the browser vendors implemented features differently, and not just edge features, basic features, like `addEventListener` and `attachEvent`. Times have changed but there are still a few functions each developer should have in their arsenal, for performance for functional ease purposes.

### debounce

The debounce function can be a game-changer when it comes to event-fueled performance. If you aren't using a debouncing function with a `scroll`, `resize`, `key*` event, you're probably doing it wrong. Here's a `debounce` function to keep your code efficient:

```
// Returns a function, that, as long as it continues to be invoked, will not
// be triggered. The function will be called after it stops being called for
// N milliseconds. If `immediate` is passed, trigger the function on the
// leading edge, instead of the trailing.
function debounce(func, wait, immediate) {
    var timeout;
    return function() {
        var context = this, args = arguments;
        var later = function() {
            timeout = null;
            if (!immediate) func.apply(context, args);
        };
        var callNow = immediate && !timeout;
        clearTimeout(timeout);
        timeout = setTimeout(later, wait);
        if (callNow) func.apply(context, args);
    };
}

// Usage
var myEfficientFn = debounce(function() {
    // All the taxing stuff you do
}, 250);
window.addEventListener('resize', myEfficientFn);
```

The `debounce` function will not allow a callback to be used more than once per given time frame. This is especially important when assigning a callback function to frequently-firing events.

## poll

As I mentioned with the `debounce` function, sometimes you don't get to plug into an event to signify a desired state -- if the event doesn't exist, you need to check for your desired state at intervals:

```
// The polling function
function poll(fn, timeout, interval) {
    var endTime = Number(new Date()) + (timeout || 2000);
    interval = interval || 100;

    var checkCondition = function(resolve, reject) {
        // If the condition is met, we're done!
        var result = fn();
        if(result) {
            resolve(result);
        }
        // If the condition isn't met but the timeout hasn't elapsed, go again
        else if (Number(new Date()) < endTime) {
            setTimeout(checkCondition, interval, resolve, reject);
        }
        // Didn't match and too much time, reject!
        else {
            reject(new Error('timed out for ' + fn + ': ' + arguments));
        }
    };

    return new Promise(checkCondition);
}

// Usage: ensure element is visible
poll(function() {
    return document.getElementById('lightbox').offsetWidth > 0;
}, 2000, 150).then(function() {
    // Polling done, now do something else!
}).catch(function() {
    // Polling timed out, handle the error!
});
```

Polling has long been useful on the web and will continue to be in the future!

## once

There are times when you prefer a given functionality only happen once, similar to the way you'd use an `onload` event. This code provides you said functionality:

```
function once(fn, context) {
    var result;

    return function() {
        if(fn) {
            result = fn.apply(context || this, arguments);
            fn = null;
        }

        return result;
    };
}

// Usage
var canOnlyFireOnce = once(function() {
    console.log('Fired!');
});

canOnlyFireOnce(); // "Fired!"
canOnlyFireOnce(); // nada
```

The `once` function ensures a given function can only be called once, thus prevent duplicate initialization!

### [getAbsoluteUrl](#)

Getting an absolute URL from a variable string isn't as easy as you think. There's the `URL` constructor but it can act up if you don't provide the required arguments (which sometimes you can't). Here's a suave trick for getting an absolute URL from and string input:

```
var getAbsoluteUrl = (function() {
    var a;

    return function(url) {
        if(!a) a = document.createElement('a');
        a.href = url;

        return a.href;
    };
})();
```

```
// Usage  
getAbsoluteUrl('/something'); // https://davidwalsh.name/something
```

The "burn" element `href` handles all URL nonsense for you, providing a reliable absolute URL in return.

### isNative

Knowing if a given function is native or not can signal if you're willing to override it. This handy code can give you the answer:

```
; (function() {  
  
    // Used to resolve the internal `[[Class]]` of values  
    var toString = Object.prototype.toString;  
  
    // Used to resolve the decompiled source of functions  
    var fnToString = Function.prototype.toString;  
  
    // Used to detect host constructors (Safari > 4; really typed array specific)  
    var reHostCtor = /^[object .+?Constructor\b]/$;  
  
    // Compile a regexp using a common native method as a template.  
    // We chose `Object#toString` because there's a good chance it is not being  
    // mucked with.  
    var reNative = RegExp('^' +  
        // Coerce `Object#toString` to a string  
        String(toString)  
        // Escape any special regexp characters  
        .replace(/[^.*+?^${}()|[\]\/\\\]/g, '\\$&')  
        // Replace mentions of `toString` with `.*?` to keep the template generic.  
        // Replace things like `for ...` to support environments like Rhino which add  
        // extra info  
        // such as method arity.  
        .replace(/toString|(function).*?(?=\\()| for .+?(?=\\])/g, '$1.*?' ) + '$'  
    );  
  
    function isNative(value) {  
        var type = typeof value;  
        return type == 'function'  
            // Use `Function#toString` to bypass the value's own `toString` method  
            // and avoid being faked out.  
    }  
});
```

```

? reNative.test(fnToString.call(value))
// Fallback to a host object check because some environments will represent
// things like typed arrays as DOM methods which may not conform to the
// normal native pattern.
: (value && type == 'object' && reHostCtor.test(toString.call(value))) ||
false;
}

// export however you want
module.exports = isNative;
}());

```

// Usage

```

isNative(alert); // true
isNative(myCustomFunction); // false

```

The function isn't pretty but it gets the job done!

### insertRule

We all know that we can grab a NodeList from a selector (via `document.querySelectorAll`) and give each of them a style, but what's more efficient is setting that style to a selector (like you do in a stylesheet):

```

var sheet = (function() {
    // Create the <style> tag
    var style = document.createElement('style');

    // Add a media (and/or media query) here if you'd like!
    // style.setAttribute('media', 'screen')
    // style.setAttribute('media', 'only screen and (max-width : 1024px)')

    // WebKit hack :(
    style.appendChild(document.createTextNode '');

    // Add the <style> element to the page
    document.head.appendChild(style);

    return style.sheet;
})();

// Usage
sheet.insertRule("header { float: left; opacity: 0.8; }", 1);

```

This is especially useful when working on a dynamic, AJAX-heavy site. If you set the style to a selector, you don't need to account for styling each element that may match that selector (now or in the future).

### [matchesSelector](#)

Oftentimes we validate input before moving forward; ensuring a truthy value, ensuring forms data is valid, etc. But how often do we ensure an element qualifies for moving forward? You can use a `matchesSelector` function to validate if an element is of a given selector match:

```
function matchesSelector(el, selector) {  
    var p = Element.prototype;  
    var f = p.matches || p.webkitMatchesSelector || p.mozMatchesSelector ||  
p.msMatchesSelector || function(s) {  
        return [].indexOf.call(document.querySelectorAll(s), this) !== -1;  
    };  
    return f.call(el, selector);  
  
}  
  
// Usage  
matchesSelector(document.getElementById('myDiv'), 'div.someSelector[some-  
attribute=true]')
```

## 8 : Collection of useful JavaScript functions and patterns

<http://codebits.glennjones.net/cheatsheet/javascript.htm>

This is a collection of useful JavaScript functions and patterns that I often use, when not coding with jQuery. I am still copying example from my projects into this document so it is a bit of a **work in progress**.

- [Strings](#)
- [Arrays](#)
- [Dates](#)
- [Objects](#)
- [DOM](#)
- [Loops](#)
- [Events](#)
- [Type detection](#)
- [Object detection](#)
- [JSON](#)
- [Module](#)
- [Closure](#)

### Strings

```
// Removes any white space to the right and left of the string
```

```
function trim(str) {  
    return str.replace(/^\s+|\s$/g, "");  
}
```

```
// Removes any white space to the left of the string
```

```
function ltrim(str) {  
    return str.replace(/^\s/, "");  
}
```

```
// Removes any white space to the right of the string
```

```
function rtrim(str) {  
    return str.replace(/\s$/, "");  
}
```

```
// Truncate a string to a given length
```

```
function truncate(str, len) {
    if (str.length > len) {
        str = str.substring(0, len);
    }
    return str;
};

// Return a string only containing the letters a to z
function onlyLetters(str) {
    return str.toLowerCase().replace(/[^a-z]/g, "");
};

// Return a string only containing the letters a to z and numbers
function onlyLettersNums(str) {
    return str.toLowerCase().replace(/[^a-z,0-9,-]/g, "");
};
```

## Arrays

```
// Removes an item from a given array
function removeArrayItem(arr, item) {
    var i = 0;
    while (i < arr.length) {
        if (arr[i] == item) {
            arr.splice(i, 1);
        } else {
            i++;
        }
    }
};
```

```
// Does a given array contain a item
function contains(a, obj) {
    var i = a.length;
    while (i--) {
        if (a[i] === obj) {
            return true;
        }
    }
};
```

```
        }
    }
    return false;
};

// The index of an item - Javascript 1.6+
['glenn','john'].indexOf('john')  returns 2
```

## Dates

```
// Converts native date to a ISO date
function isoDateString(d) {
    function pad(n) {
        return n < 10 ? '0' + n : n
    }
    return d.getUTCFullYear() + '-'
        + pad(d.getUTCMonth() + 1) + '-'
        + pad(d.getUTCDate()) + 'T'
        + pad(d.getUTCHours()) + ':'
        + pad(d.getUTCMinutes()) + ':'
        + pad(d.getUTCSeconds()) + 'Z'
}
```

## Objects

```
// Sorts a array of objects by a property
function sortObjectsByProperty(a, field, reverse, primer) {
    return a.sort(sortObjects(field, reverse, primer));
};
```

```
// Object sort
function sortObjects(field, reverse, primer) {
    reverse = (reverse) ? -1 : 1;
    return function (a, b) {
        a = a[field];
        b = b[field];
```

```
    if (primer !== undefined && a !== undefined && b !== undefined) {
        a = primer(a);
        b = primer(b);
    }
    if (a < b) return reverse * -1;
    if (a > b) return reverse * 1;
    return 0;
}
};
```

## DOM

```
// Does the node have a class
function hasClass(node, className) {
    if (node.className) {
        return node.className.match(
            new RegExp('(\s|^)' + className + '(\s|$)'));
    } else {
        return false;
    }
};

// Add a class to an node
function addClass(node, className) {
    if (hasClass(node, className)) node.className += " " + className;
};

// Removes a class from an node
function removeClass(node, className) {
    if (hasClass(node, className)) {
        var reg = new RegExp('(\s|^)' + className + '(\s|$)');
        node.className = node.className.replace(reg, ' ');
    }
};

// Returns first ancestor of required class or a null
function getParentByClass(node, className) {
```

```
if (arguments.length === 2) {
    if (node.parentNode && node.nodeName !== "BODY")
        return getParentByClass(node.parentNode, className, 1);
    else
        return null;
}
// Recursive calls
if (node !== null && node !== undefined) {
    if (hasClass(node, className)) {
        return node;
    } else {
        if (node.parentNode && node.nodeName !== "BODY")
            return getParentByClass(node.parentNode, className, 1);
        else
            return null;
    }
} else {
    return null;
}
};

// Returns the text of a given node
function gettextContent(node) {
    if (typeof node.textContent != "undefined") {
        return node.textContent;
    }
    return node.innerText;
};
```

```
// Removes all child nodes
function removeAllChildren(node) {
    if (node) {
        while (node.firstChild) {
            node.removeChild(node.firstChild);
        }
    }
}
```

```
// Appends a new element to a given node
function append(node, eltName, attArr, strHtml) {
    if (node) {
        var nNode = document.createElement(eltName);
        if (attArr !== undefined) {
            var j = attArr.length;
            while (j--) {
                nNode.setAttribute(attArr[j][0], attArr[j][1]);
            }
        }
        if (strHtml !== undefined) {
            nNode.innerHTML = strHtml;
        }
        node.appendChild(nNode);
        return nNode;
    } else {
        return null;
    }
}

// Get elements by class name (Backwards compatible version)
function getElementsByClassName(rootNode, className) {
    var returnElements = [];
    if (rootNode.getElementsByClassName) {
        // Native getElementsByClassName
        returnElements = rootNode.getElementsByClassName(className);
    } else if (document.evaluate) {
        // XPath
        var xpathExpression;
        xpathExpression = ".//*[@contains(concat(' ', @class, ' '), ' "
            + className + " ')]";
        var xpathResult = document.evaluate(
            xpathExpression, rootNode, null, 0, null);
        var node;
        while ((node = xpathResult.iterateNext())) {
            returnElements.push(node);
        }
    } else {
        // Slower DOM fallback
        className = className.replace(/\-/g, "\\\"");
    }
}
```

```
var elements = rootNode.getElementsByTagName("*");
for (var x = 0; x < elements.length; x++) {
    if (elements[x].className.match("(^|\\s)" + className
        + "(\\s|$)")) {
        returnElements.push(elements[x]);
    }
}
return returnElements;
}
```

```
// Get elements by attribute (Backwards compatible version)
```

```
function getElementsByAttribute(
    rootNode, attributeName, attributeValues) {

    var attributeList = attributeValues.split(" ");
    var returnElements = [];
    if (rootNode.querySelectorAll) {
        var selector = '';
        for (var i = 0; i < attributeList.length; i++) {
            selector += '[' + attributeName
                + '*= "' + attributeList[i] + '"], ';
        }
        returnElements = rootNode.querySelectorAll(
            selector.substring(0, selector.length - 2));
    } else if (document.evaluate) {
        // XPath
        var xpathExpression = ".//*[";
        for (var i = 0; i < attributeList.length; i++) {
            if (i !== 0) {
                xpathExpression += " or ";
            }
            xpathExpression += "contains(
                concat(' ', @" + attributeName
                    + ", ' '), ' " + attributeList[i] + " ')";
        }
        xpathExpression += "]";
        var xpathResult = document.evaluate(
            xpathExpression, rootNode, null, 0, null);
        var node;
```

```

        while ((node = xpathResult.iterateNext())) {
            returnElements.push(node);
        }
    } else {
        // Slower fallback
        attributeName = attributeName.replace(/^-/g, "\\\-");
        var elements = rootNode.getElementsByTagName("*");
        for (var x = 0; x < elements.length; x++) {
            if (elements[x][attributeName]) {
                var found = false;
                for (var y = 0; y < attributeList.length; y++) {
                    if (elements[x][attributeName].match("^(^|\\s)"
                        + attributeList[y] + "(\\s|$)")) {
                        found = true;
                    }
                }
                if (found)
                    returnElements.push(elements[x]);
            }
        }
    }
    return returnElements;
},

```

## Loops

```

// While reverse loop - fast
var i = arr.length;
while (i--) {
    // Do Stuff
}

```

```

// While loop
var i = arr.length;
var x = 0;
while (x < i) {
    // Do stuff
}

```

```
x++;  
}
```

## Events

```
// Add event (Cross browser old school)  
function addEvent(obj, type, fn) {  
    if (obj) {  
        if (obj.attachEvent) {  
            obj['e' + type + fn] = fn;  
            obj[type + fn] = function () {  
                obj['e' + type + fn](window.event);  
            };  
            obj.attachEvent('on' + type, obj[type + fn]);  
        } else {  
            obj.addEventListener(type, fn, false);  
        }  
    }  
};
```

```
// Remove event (Cross browser old school)  
function removeEvent(obj, type, fn) {  
    if (obj) {  
        if (obj.detachEvent) {  
            obj.detachEvent('on' + type, obj[type + fn]);  
            obj[type + fn] = null;  
        } else {  
            obj.removeEventListener(type, fn, false);  
        }  
    }  
};
```

```
// Cancel event bubbling to the rest of DOM  
function cancelBubble(e) {  
    if (window.event)  
        window.event.cancelBubble = true;  
    else
```

```
        e.stopPropagation();
    };

// Prevents the events default action from happening.
u.preventDefault = function (e) {
    if (e.preventDefault)
        e.preventDefault();
    try {
        e.returnValue = false;
    } catch (ex) {
        // Do nothing
    }
};

// Cancel bubbling and prevent default action
function eventStop(e) {
    cancelBubble(e);
    preventDefault(e);
}
```

## Type detection

```
// Is an object a string
function isString(obj) {
    return typeof (obj) == 'string';
};

// Is an object a array
function isArray(obj) {
    return obj && !(obj.propertyIsEnumerable('length'))
        && typeof obj === 'object'
        && typeof obj.length === 'number';
};

// Is an object a int
function isInt(obj) {
    var re = /^\\d+$/;
```

```

        return re.test(obj);
    };

// Is an object a email address

function isEmail(obj) {
    if(isString(obj)){
        return obj.match(/\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b/ig);
    }else{
        return false;
    }
};

// Is an object a URL

function isUrl (obj) {
    if(isString(obj)){
        var re = new RegExp("^(http|https)\://(([a-zA-Z0-9\.\-\-]+(\:\:" +
            "[a-zA-Z0-9\.\&%\$\-\-]+)*)@((25[0-5]|2[0-4][0-9]|" +
            "[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9])\.(25[0-5]|2" +
            "[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\." +
            "(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|" +
            "[1-9]|0)\.(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|" +
            "[1-9]|0)\.(com|edu|gov|int|mil|net|org|biz|arpa|info|name" +
            "|pro|aero|coop|museum|[a-zA-Z]{2}))(:[0-9]+)*(/$|[a-z" +
            "A-Z0-9\.\,\,\?\\"\\+&%\$#\=\_\-\-]+)*$");
        return obj.match(re);
    }else{
        return false;
    }
};

```

## Object/Method detection

```

// Checks for Drag and Drop API support

function hasDragDrop() {
    var element = document.createElement('div');
    var name = 'ondragstart';
    var isSupported = (name in element);

```

```

if (!isSupported && element.setAttribute) {
    element.setAttribute(name, 'return;');
    isSupported = typeof element[name] == 'function';
}
element = null;
return isSupported;
};

```

## JSON

```

// Loads a JSON file into document
functiongetJSON(url) {
    script = document.createElement("script");
    script.setAttribute("type", "text/javascript");
    if (url.indexOf('?') > -1)
        url += '&';
    else
        url += '?';
    url += 'rand=' + Math.random();
    script.setAttribute("src", url);
    document.getElementsByTagName('head')[0].appendChild(script);
};

```

## Module

```

// The module pattern
var PeopleStore = (function (m) {
    m.newModule = {};
    newModule.version = 0.1;

    return m;
} (PeopleStore || {}));

```

## Closure

```
// onClick with a external closure
```

```
removeBtn.onclick = post(obj aValue)
function post(aValue) {
    return function() {
        // do somthing with aValue
    };
}

// onClick with a closure
removeBtn.onclick = function (aValue) {
    return function () {
        // do somthing with aValue
    };
} (obj.aValue)

// addEventListener with a closure
removeBtn.addEventListener('click', function (aValue) {
    return function () {
        // do somthing with aValue
    };
}(obj.aValue), false);
```

I have coded and collected these functions over time. They often follow patterns that are found in the public domain. Everyone sits on the shoulders of others. The RegExp are copied from public sources on the web.

## 9 : My todays top 10 most useful Javascript Functions

---

These are my todays top 10 Javascript functions which I gathered in more than 6 months of casually Javascript Development. Almost each of them helped out of more than one problem, so they deserve to be perpetuated here and help again. Most of these are not written by me, but by some other great programmer on the web, and I really would like to mention all of them here if I would know where I have got all of these great scripts.

Have fun an use wisely.

### 1. Get Elements By Class Name (`getElementsByClassName`)

This is maybe one of the most used and most needed custom Javascript functions of all time. It provides a way to get all elements in a page with a certain class attribute.

```
function getElementsByClassName(classname, node){
    if (!node) {
        node = document.getElementsByTagName('body')[0];
    }

    var a = [], re = new RegExp('\\\\b' + classname + '\\\\b');
    els = node.getElementsByTagName('*');
    for (var i = 0, j = els.length; i < j; i++) {
        if (re.test(els[i].className)) {
            a.push(els[i]);
        }
    }
    return a;
}
```

### 2. Test if Internet Explorer is used and get its version number

I do not know how much functions, CSS selectors or whatever - IE so often acts different or just stupid. It always requires extra work. This after all will help you to detect this nasty browser.

```
// use the class
if(Browser.Version() <8) {
    // make crazy IE shit
```

```
}
```

```
var Browser = {
    Version: function(){
        var version = 999; // we assume a sane browser
        if (navigator.appVersion.indexOf("MSIE") != -1)
            // bah, IE again, lets downgrade version number
            version = parseFloat(navigator.appVersion.split("MSIE")[1]);
        return version;
    }
}
```

### 3. Clear input-field onclick, if value is default

This is not much of javascript, we actually call it dynamic html. It is just the standard way to present the comfortable input fields to users.

### 4. Test if a String is not empty

Can help to avoid a lot of common mistakes with javascript.

```
var bpat    = /\S/;
function isNonblank (s) {
    return String (s).search (bpat) != -1
}
```

### 5. Function Exists

This is a possibility in Javascript to test if a function exists (like `function_exists` in PHP). This can be useful in cases of debugging or when you are not sure if some required script file is loaded, for example.

```
if(typeof yourFunctionName == 'function') {
    yourFunctionName();
}
```

### 6. Get scroll width/height of page visitors' browser window

A browser-safe way to get the number of pixels, which a user scrolled down the webpage currently.

```
function getScrollXY() {
    var scrOfX = 0, scrOfY = 0;

    if( typeof( window.pageYOffset ) == 'number' ) {
        //Netscape compliant
        scrOfY = window.pageYOffset;
        scrOfX = window.pageXOffset;
    } else if( document.body && ( document.body.scrollLeft ||
document.body.scrollTop ) ) {
        //DOM compliant
        scrOfY = document.body.scrollTop;
        scrOfX = document.body.scrollLeft;
    } else if( document.documentElement && ( document.documentElement.scrollLeft ||
document.documentElement.scrollTop ) ) {
        //IE6 standards compliant mode
        scrOfY = document.documentElement.scrollTop;
        scrOfX = document.documentElement.scrollLeft;
    }
    return [ scrOfX, scrOfY ];
}
```

## 7. Get current size of page visitors' browser window

A browser-safe way to get the window height and window width of the current viewers browser in pixels.

```
function getWindowSize() {
    var myWidth = 0, myHeight = 0;

    if( typeof( window.innerWidth ) == 'number' ) {
        //Non-IE
        myWidth = window.innerWidth;
        myHeight = window.innerHeight;
    } else if( document.documentElement && ( document.documentElement.clientWidth ||
document.documentElement.clientHeight ) ) {
        //IE 6+ in 'standards compliant mode'
        myWidth = document.documentElement.clientWidth;
        myHeight = document.documentElement.clientHeight;
    }
}
```

```

} else if( document.body && ( document.body.clientWidth ||
document.body.clientHeight ) ) {
    //IE 4 compatible
    myWidth = document.body.clientWidth;
    myHeight = document.body.clientHeight;
}
return [ myWidth, myHeight ];
}

```

## 8. Scale images by setting a maximum width/height

It is not always the best idea to scale images with javascript. But I came to several applications there this was useful though.

```

var images = document.getElementsByTagName("img"); // get your images somehow
scaleImages(imgs,150,150); // call the function

// scales the images to a maximum width/height
function scaleImages(images,maxh,maxw) {

for(i=0;i ratio){
    // height is the problem
    if (img.height > maxh){
        img.width = Math.round(img.width*(maxh/img.height));
        img.height = maxh;
    }
} else {
    // width is the problem
    if (img.width > maxw){
        img.height = Math.round(img.height*(maxw/img.width));
        img.width = maxw;
    }
}
img.setAttribute("class","showpreview ready");
}
}

```

## 9. Print Javascript Array (like print\_r()-in PHP)

A really useful function for debugging and developing with javascript. Instead of [object Object] or something similar you will see the whole contents of an array in your output.

```
function dump(arr,level) {
    var dumped_text = "";
    if(!level) level = 0;

    //The padding given at the beginning of the line.
    var level_padding = "";
    for(var j=0;j \"" + value + "\"\n";
    }
}
} else { //Strings/Chars/Numbers etc.
    dumped_text = "====>" + arr + "<====(" + typeof(arr) + ")";
}
return dumped_text;
}
```

## 10. XML2Array (parse xml to javascript array)

For the special case of reading XML with Javascript. But very useful, when it comes to the case. It is much easier to handle a huge array in Javascript than reading lots of data from a complex XML-file. Maybe there is a tiny disadvantage concerning speed, but I think it is worth that.

```
/////////////////////////////// xml2array()
///////////////////////////////
// See http://www.openjs.com/scripts/xml_parser/
var not_whitespace = new RegExp(/[^\\s]/|>); //This can be given inside the function
- I made it a global variable to make the script a little bit faster.
var parent_count;
//Process the xml data
function xml2array(xmlDoc,parent_count) {
    var arr;
    var parent = "";
    parent_count = parent_count || new Object;

    var attribute_inside = 0; /*:CONFIG: Value - 1 or 0
    * If 1, Value and Attribute will be shown inside the tag - like this...
    * For the XML string...
```

```
* http://www.bin-co.com/
* The resulting array will be...
* array['guid']['value'] = "http://www.bin-co.com/";
* array['guid']['attribute_isPermaLink'] = "true";
*
* If 0, the value will be inside the tag but the attribute will be outside - like
this...
* For the same XML String the resulting array will be...
* array['guid'] = "http://www.bin-co.com/";
* array['attribute_guid_isPermaLink'] = "true";
*/
if(xmlDoc.nodeName && xmlDoc.nodeName.charAt(0) != "#") {
    if(xmlDoc.childNodes.length > 1) { //If its a parent
        arr = new Object;
        parent = xmlDoc.nodeName;

    }
}
var value = xmlDoc.nodeValue;
if(xmlDoc.parentNode && xmlDoc.parentNode.nodeName && value) {
    if(not_whitespace.test(value)) {//If its a child
        arr = new Object;
        arr[xmlDoc.parentNode.nodeName] = value;
    }
}

if(xmlDoc.childNodes.length) {
    if(xmlDoc.childNodes.length == 1) { //Just one item in this tag.
        arr = xml2array(xmlDoc.childNodes[0],parent_count); //:RECURSION:
    } else { //If there is more than one childNodes, go thru them one by one and get
    their results.
        var index = 0;

        for(var i=0; i2) break;//We just need to know wether it is a single value array
or not
    }

    if(assoc && arr_count == 1) {
        if(arr[key]) { //If another element exists with the same tag name before,
            // put it in a numeric array.
```

```
//Find out how many time this parent made its appearance
if(!parent_count || !parent_count[key]) {
    parent_count[key] = 0;

    var temp_arr = arr[key];
    arr[key] = new Object;
    arr[key][0] = temp_arr;
}

parent_count[key]++;
arr[key][parent_count[key]] = temp[key]; //Members of of a numeric array
} else {
    parent_count[key] = 0;
    arr[key] = temp[key];
    if(xmlDoc.childNodes[i].attributes &&
xmlDoc.childNodes[i].attributes.length) {
        for(var j=0; j
```

# 10 : 12 Extremely Useful Hacks for JavaScript

<https://blog.js scrambler.com/12-extremely-useful-hacks-for-javascript/>

In this post I will share **12 extremely useful hacks for JavaScript**. These hacks reduce the code and will help you to run optimized code. So let's start hacking!

## 1) Converting to boolean using !! operator

Sometimes we need to check if some variable exists or if it has a valid value, to consider them as *true value*. For do this kind of validation, you can use the **!!** (*Double negation operator*) a simple **!variable**, which will automatically convert any kind of data to boolean and this variable will return **false** only if it has some of these values: **0**, **null**, **""**, **undefined** or **NaN**, otherwise it will return **true**. To understand it in practice, take a look this simple example:

```
function Account(cash) {
    this.cash = cash;
    this.hasMoney = !!cash;
}

var account = new Account(100.50);
console.log(account.cash); // 100.50
console.log(account.hasMoney); // true

var emptyAccount = new Account(0);
console.log(emptyAccount.cash); // 0
console.log(emptyAccount.hasMoney); // false
```

In this case, if an **account.cash** value is greater than zero, the **account.hasMoney** will be true.

## 2) Converting to number using + operator

This magic is awesome! And it's very simple to be done, but it only works with string numbers, otherwise it will return **NaN** (*Not a Number*). Have a look on this example:

```
function toNumber(strNumber) {
    return +strNumber;
}

console.log(toNumber("1234")); // 1234
console.log(toNumber("ACB")); // NaN
```

This magic will work with **Date** too and, in this case, it will return the timestamp number:

```
console.log(+new Date()) // 1461288164385
```

### 3) Short-circuits conditionals

If you see a similar code:

```
if (conected) {  
    login();  
}
```

You can shorten it by using the combination of a variable (which will be verified) and a function using the `&&` (AND operator) between both. For example, the previous code can become smaller in one line:

```
conected && login();
```

You can do the same to check if some attribute or function exists in the object. Similar to the below code:

```
user && user.login();
```

### 4) Default values using `||` operator

Today in ES6 there is the default argument feature. In order to simulate this feature in old browsers you can use the `||` (OR operator) by including the default value as a second parameter to be used. If the first parameter returns `false` the second one will be used as a default value. See this example:

```
function User(name, age) {  
    this.name = name || "Oliver Queen";  
    this.age = age || 27;  
}  
  
var user1 = new User();  
console.log(user1.name); // Oliver Queen  
console.log(user1.age); // 27  
  
var user2 = new User("Barry Allen", 25);  
console.log(user2.name); // Barry Allen  
console.log(user2.age); // 25
```

### 5) Caching the `array.length` in the loop

This tip is very simple and causes a huge impact on the performance when processing large arrays during a loop. Basically, almost everybody writes this synchronous `for` to iterate an array:

```
for (var i = 0; i < array.length; i++) {
    console.log(array[i]);
}
```

If you work with smaller arrays – it's fine, but if you process large arrays, this code will recalculate the size of array in every iteration of this loop and this will cause a bit of delays. To avoid it, you can cache the `array.length` in a variable to use it instead of invoking the `array.length` every time during the loop:

```
var length = array.length;
for (var i = 0; i < length; i++) {
    console.log(array[i]);
}
```

To make it smaller, just write this code:

```
for (var i = 0, length = array.length; i < length; i++) {
    console.log(array[i]);
}
```

## 6) Detecting properties in an object

This trick is very useful when you need to check if some attribute exists and it avoids running undefined functions or attributes. If you are planning to write cross-browser code, probably you will use this technique too. For example, let's imagine that you need to write code that is compatible with the old Internet Explorer 6 and you want to use the `document.querySelector()`, to get some elements by their ids. However, in this browser this function doesn't exist, so to check the existence of this function you can use the `in` operator, see this example:

```
if ('querySelector' in document) {
    document.querySelector("#id");
} else {
    document.getElementById("id");
}
```

In this case, if there is no `querySelector` function in the `document` object, we can use the `document.getElementById()` as fallback.

## 7) Getting the last item in the array

The `Array.prototype.slice(begin, end)` has the power to cut arrays when you set the `begin` and `end` arguments. But if you don't set the `end` argument, this function will automatically set the max value for the array. I think that few people know that this function can accept negative values, and if you set a negative number as `begin` argument you will get the last elements from the array:

```
var array = [1, 2, 3, 4, 5, 6];
console.log(array.slice(-1)); // [6]
console.log(array.slice(-2)); // [5,6]
console.log(array.slice(-3)); // [4,5,6]
```

## 8) Array truncation

This technique can lock the array's size, this is very useful to delete some elements of the array based on the number of elements you want to set. For example, if you have an array with 10 elements, but you want to get only the first five elements, you can truncate the array, making it smaller by setting the `array.length = 5`. See this example:

```
var array = [1, 2, 3, 4, 5, 6];
console.log(array.length); // 6
array.length = 3;
console.log(array.length); // 3
console.log(array); // [1,2,3]
```

## 9) Replace all

The `String.replace()` function allows using String and Regex to replace strings, natively this function only replaces the first occurrence. But you can simulate a `replaceAll()` function by using the `/g` at the end of a Regex:

```
var string = "john john";
console.log(string.replace(/hn/, "ana")); // "joana john"
console.log(string.replace(/hn/g, "ana")); // "joana joana"
```

## 10) Merging arrays

If you need to merge two arrays you can use the `Array.concat()` function:

```
var array1 = [1, 2, 3];
var array2 = [4, 5, 6];
console.log(array1.concat(array2)); // [1,2,3,4,5,6];
```

However, this function is not the most suitable to merge large arrays because it will consume a lot of memory by creating a new array. In this case, you can use `Array.push.apply(arr1, arr2)` which instead creates a new array – it will merge the second array in the first one reducing the memory usage:

```
var array1 = [1, 2, 3];
var array2 = [4, 5, 6];
console.log(array1.push.apply(array1, array2)); // [1,2,3,4,5,6];
```

## 11) Converting NodeList to Arrays

If you run the `document.querySelectorAll("p")` function, it will probably return an array of DOM elements, the NodeList object. But this object doesn't have all array's functions, like: `sort()`, `reduce()`, `map()`, `filter()`. In order to enable these and many other native array's functions you need to convert NodeList into Arrays. To run this technique just use this function: `[].slice.call(elements)`:

```
var elements = document.querySelectorAll("p"); // NodeList
var arrayElements = [].slice.call(elements); // Now the NodeList is an array
var arrayElements = Array.from(elements); // This is another way of converting
NodeList to Array
```

## 12) Shuffling array's elements

To shuffle the array's elements without using any external library like *Lodash*, just run this magic trick:

```
var list = [1, 2, 3];
console.log(list.sort(function() {
    return Math.random() - 0.5
})); // [2,1,3]
```

## Conclusion

Now you learned some useful JS hacks which are largely used to minify JavaScript code and some of these tricks are used in many popular JS frameworks like *Lodash*, *Underscore.js*, *Strings.js*, among others. If you want to go even deeper and learn more about how you can minify your code even more and even protect it from prying eyes talk to [us](#). I hope you enjoyed this post and if you know other JS hacks, please let us know!

# 11 : 45 Useful JavaScript Tips, Tricks and Best Practices

<https://modernweb.com/45-useful-javascript-tips-tricks-and-best-practices/>

As you know, JavaScript is the number one programming language in the world, the language of the web, of mobile hybrid apps (like [PhoneGap](#) or [Appcelerator](#)), of the server side (like [NodeJS](#) or [Wakanda](#)) and has many other implementations. It's also the starting point for many new developers to the world of programming, as it can be used to display a simple alert in the web browser but also to control a robot (using [nodebot](#), or [nodruino](#)). The developers who master JavaScript and write organized and performant code have become the most sought after in the job market.

In this article, I'll share a set of JavaScript tips, tricks and best practices that should be known by all JavaScript developers regardless of their browser/engine or the SSJS (Server Side JavaScript) interpreter.

Note that the code snippets in this article have been tested in the latest Google Chrome version 30, which uses the V8 JavaScript Engine (V8 3.20.17.15).

## 1 – Don't forget `var` keyword when assigning a variable's value for the first time.

Assignment to an undeclared variable automatically results in a global variable being created. Avoid global variables.

## 2 – use `===` instead of `==`

The `==` (or `!=`) operator performs an automatic type conversion if needed. The `===` (or `!==`) operator will not perform any conversion. It compares the value and the type, which could be considered faster than `==`.

```
[10] === 10      // is false
[10] == 10       // is true
'10' == 10       // is true
'10' === 10      // is false
[] == 0          // is true
[] === 0         // is false
'' == false      // is true but true == "a" is false
'' === false     // is false
```

## 3 – `undefined`, `null`, `0`, `false`, `Nan`, `''` (empty string) are all falsy.

## 4 – Use Semicolons for line termination

The use of semi-colons for line termination is a good practice. You won't be warned if you forget it, because in most cases it will be inserted by the JavaScript parser. For more details about why you should use semi-colons, take a look to this article: <http://davidwalsh.name/javascript-semicolons>.

## 5 – Create an object constructor

```

function Person(firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
}

var Saad = new Person("Saad", "Mousliki");

```

## 6 – Be careful when using `typeof`, `instanceof` and `constructor`.

- `typeof`: a JavaScript unary operator used to return a string that represents the primitive type of a variable, don't forget that `typeof null` will return "object", and for the majority of object types (Array, Date, and others) will return also "object".
- `constructor`: is a property of the internal prototype property, which could be overridden by code.
- `instanceof`: is another JavaScript operator that check in all the prototypes chain the constructor it returns true if it's found and false if not.

```

var arr = ["a", "b", "c"];
typeof arr; // return "object"
arr instanceof Array // true
arr.constructor(); //[]

```

## 7 – Create a Self-calling Function

This is often called a Self-Invoked Anonymous Function or Immediately Invoked Function Expression (IIFE). It is a function that executes automatically when you create it, and has the following form:

```

(function(){
    // some private code that will be executed automatically
})();
(function(a,b){
    var result = a+b;
    return result;
})(10,20)

```

## 8 – Get a random item from an array

```

var items = [12, 548 , 'a' , 2 , 5478 , 'foo' , 8852, , 'Doe' , 2145 , 119];

var randomItem = items[Math.floor(Math.random() * items.length)];

```

## 9 – Get a random number in a specific range

This code snippet can be useful when trying to generate fake data for testing purposes, such as a salary between min and max.

```
var x = Math.floor(Math.random() * (max - min + 1)) + min;
```

## 10 – Generate an array of numbers with numbers from 0 to max

```
var numbersArray = [] , max = 100;

for( var i=1; numbersArray.push(i++) < max); // numbers = [1,2,3 ... 100]
```

## 11 – Generate a random set of alphanumeric characters

```
function generateRandomAlphaNum(len) {
    var rdmString = "";
    for( ; rdmString.length < len; rdmString += Math.random().toString(36).substr(2));
    return rdmString.substr(0, len);
}
```

## 12 – Shuffle an array of numbers

```
var numbers = [5, 458 , 120 , -215 , 228 , 400 , 122205, -85411];
numbers = numbers.sort(function(){ return Math.random() - 0.5});
/* the array numbers will be equal for example to [120, 5, 228, -215, 400, 458,
-85411, 122205] */
```

A better option could be to implement a random sort order by code (e.g. : Fisher-Yates shuffle), than using the native sort JavaScript function. For more details take a look to [this discussion](#).

## 13 – A string trim function

The classic trim function of Java, C#, PHP and many other language that remove whitespace from a string doesn't exist in JavaScript, so we could add it to the `String` object.

```
String.prototype.trim = function(){return this.replace(/\s+|s+$/.g, "")};
```

A native implementation of the trim() function is available in the recent JavaScript engines.

## 14 – Append an array to another array

```

var array1 = [12 , "foo" , {name "Joe"} , -2458];

var array2 = ["Doe" , 555 , 100];
Array.prototype.push.apply(array1, array2);
/* array1 will be equal to  [12 , "foo" , {name "Joe"} , -2458 , "Doe" , 555 , 100]
*/

```

## 15 – Transform the arguments object into an array

```
var argArray = Array.prototype.slice.call(arguments);
```

## 16 – Verify that a given argument is a number

```

function isNumber(n){
    return !isNaN(parseFloat(n)) && isFinite(n);
}

```

## 17 – Verify that a given argument is an array

```

function isArray(obj){
    return Object.prototype.toString.call(obj) === '[object Array]' ;
}

```

Note that if the `toString()` method is overridden, you will not get the expected result using this trick.

Or use...

```
Array.isArray(obj); // its a new Array method
```

You could also use `instanceof` if you are not working with multiple frames. However, if you have many contexts, you will get a wrong result.

```

var myFrame = document.createElement('iframe');
document.body.appendChild(myFrame);

var myArray = window.frames[window.frames.length-1].Array;
var arr = new myArray(a,b,10); // [a,b,10]

// instanceof will not work correctly, myArray loses his constructor

```

```
// constructor is not shared between frames
arr instanceof Array; // false
```

## 18 – Get the max or the min in an array of numbers

```
var numbers = [5, 458, 120, -215, 228, 400, 122205, -85411];
var maxInNumbers = Math.max.apply(Math, numbers);
var minInNumbers = Math.min.apply(Math, numbers);
```

## 19 – Empty an array

```
var myArray = [12, 222, 1000];
myArray.length = 0; // myArray will be equal to [].
```

## 20 – Don't use delete to remove an item from array

Use `splice` instead of using `delete` to delete an item from an array. Using `delete` replaces the item with `undefined` instead of removing it from the array.

Instead of...

```
var items = [12, 548, 'a', 2, 5478, 'foo', 8852, 'Doe', 2154, 119];
items.length; // return 11
delete items[3]; // return true
items.length; // return 11
/* items will be equal to [12, 548, "a", undefined × 1, 5478, "foo", 8852,
undefined × 1, "Doe", 2154, 119] */
```

Use...

```
var items = [12, 548, 'a', 2, 5478, 'foo', 8852, 'Doe', 2154, 119];
items.length; // return 11
items.splice(3,1);
items.length; // return 10
/* items will be equal to [12, 548, "a", 5478, "foo", 8852, undefined × 1, "Doe",
2154, 119] */
```

The `delete` method should be used to delete an object property.

## 21 – Truncate an array using length

Like the previous example of emptying an array, we truncate it using the `length` property.

```
var myArray = [12 , 222 , 1000 , 124 , 98 , 10 ];
myArray.length = 4; // myArray will be equal to [12 , 222 , 1000 , 124].
```

As a bonus, if you set the array length to a higher value, the length will be changed and new items will be added with `undefined` as a value. The array length is not a read only property.

```
myArray.length = 10; // the new array length is 10
myArray[myArray.length - 1] ; // undefined
```

## 22 – Use logical AND/ OR for conditions

```
var foo = 10;
foo == 10 && doSomething(); // is the same thing as if (foo == 10) doSomething();
foo == 5 || doSomething(); // is the same thing as if (foo != 5) doSomething();
```

The logical OR could also be used to set a default value for function argument.

```
function doSomething(arg1){
    arg1 = arg1 || 10; // arg1 will have 10 as a default value if it's not already
set
}
```

## 23 – Use the map() function method to loop through an array's items

```
var squares = [1,2,3,4].map(function (val) {
    return val * val;
});
// squares will be equal to [1, 4, 9, 16]
```

## 24 – Rounding number to N decimal place

```
var num = 2.443242342;
num = num.toFixed(4); // num will be equal to 2.4432
```

NOTE : the `toFixed()` function returns a string and not a number.

## 25 – Floating point problems

```
0.1 + 0.2 === 0.3 // is false
9007199254740992 + 1 // is equal to 9007199254740992
9007199254740992 + 2 // is equal to 9007199254740994
```

Why does this happen?  $0.1 + 0.2$  is equal to  $0.30000000000000004$ . What you need to know is that all JavaScript numbers are floating points represented internally in 64 bit binary according to the IEEE 754 standard. For more explanation, take a look to [this blog post](#).

You can use `toFixed()` and `toPrecision()` to resolve this problem.

## 26 – Check the properties of an object when using a for-in loop

This code snippet could be useful in order to avoid iterating through the properties from the object's prototype.

```
for (var name in object) {
    if (object.hasOwnProperty(name)) {
        // do something with name
    }
}
```

## 27 – Comma operator

```
var a = 0;
var b = ( a++, 99 );
console.log(a); // a will be equal to 1
console.log(b); // b is equal to 99
```

## 28 – Cache variables that need calculation or querying

In the case of a jQuery selector, we could cache the DOM element.

```
var navright = document.querySelector('#right');
var navleft = document.querySelector('#left');
var navup = document.querySelector('#up');
var navdown = document.querySelector('#down');
```

## 29 – Verify the argument before passing it to `isFinite()`

```
isFinite(0/0) ; // false
isFinite("foo"); // false
```

```
isFinite("10"); // true
isFinite(10); // true
isFinite(undefined); // false
isFinite(); // false
isFinite(null); // true !!!
```

### 30 – Avoid negative indexes in arrays

```
var numbersArray = [1,2,3,4,5];
var from = numbersArray.indexOf("foo") ; // from is equal to -1
numbersArray.splice(from,2); // will return [5]
```

Make sure that the arguments passed to `splice` are not negative.

### 31 – Serialization and deserialization (working with JSON)

```
var person = {name : 'Saad', age : 26, department : {ID : 15, name : "R&D"} };
var stringFromPerson = JSON.stringify(person);
/* stringFromPerson is equal to "{"name":"Saad","age":26,"department":
{"ID":15,"name":"R&D"} }" */
var personFromString = JSON.parse(stringFromPerson);
/* personFromString is equal to person object */
```

### 32 – Avoid the use of `eval()` or the `Function` constructor

Use of `eval` or the `Function` constructor are expensive operations as each time they are called script engine must convert source code to executable code.

```
var func1 = new Function(functionCode);
var func2 = eval(functionCode);
```

### 33 – Avoid using `with()` (The good part)

Using `with()` inserts a variable at the global scope. Thus, if another variable has the same name it could cause confusion and overwrite the value.

### 34 – Avoid using `for-in` loop for arrays

Instead of using...

```
var sum = 0;
for (var i in arrayNumbers) {
```

```
    sum += arrayNumbers[i];
}
```

...it's better to use...

```
var sum = 0;
for (var i = 0, len = arrayNumbers.length; i < len; i++) {
    sum += arrayNumbers[i];
}
```

As a bonus, the instantiation of `i` and `len` is executed once because it's in the first statement of the for loop. This is faster than using...

```
for (var i = 0; i < arrayNumbers.length; i++)
```

Why? The length of the array `arrayNumbers` is recalculated every time the loop iterates.

NOTE : the issue of recalculating the length in each iteration was fixed in the latest JavaScript engines.

### **35 – Pass functions, not strings, to `setTimeout()` and `setInterval()`**

If you pass a string into `setTimeout()` or `setInterval()`, the string will be evaluated the same way as with `eval`, which is slow. Instead of using...

```
setInterval('doSomethingPeriodically()', 1000);
setTimeout('doSomethingAfterFiveSeconds()', 5000);
```

...use...

```
setInterval(doSomethingPeriodically, 1000);
setTimeout(doSomethingAfterFiveSeconds, 5000);
```

### **36 – Use a switch/case statement instead of a series of if/else**

Using switch/case is faster when there are more than 2 cases, and it is more elegant (better organized code). Avoid using it when you have more than 10 cases.

### **37 – Use switch/case statement with numeric ranges**

Using a switch/case statement with numeric ranges is possible with this trick.

```

function getCategory(age) {
    var category = "";
    switch (true) {
        case isNaN(age):
            category = "not an age";
            break;
        case (age >= 50):
            category = "Old";
            break;
        case (age <= 20):
            category = "Baby";
            break;
        default:
            category = "Young";
            break;
    };
    return category;
}
getCategory(5); // will return "Baby"

```

### 38 – Create an object whose prototype is a given object

It's possible to write a function that creates an object whose prototype is the given argument like this...

```

function clone(object) {
    function OneShotConstructor(){}
    OneShotConstructor.prototype= object;
    return new OneShotConstructor();
}
clone(Array).prototype ; // []

```

### 39 – An HTML escaper function

```

function escapeHTML(text) {
    var replacements= {"<": "&lt;", ">": "&gt;", "&": "&amp;", """: "&quot;"};
    return text.replace(/[^<>&"]+/g, function(character) {
        return replacements[character];
    });
}

```

## 40 – Avoid using try-catch-finally inside a loop

The try-catch-finally construct creates a new variable in the current scope at runtime each time the catch clause is executed where the caught exception object is assigned to a variable.

Instead of using...

```
var object = ['foo', 'bar'], i;
for (i = 0, len = object.length; i < len; i++) {
    try {
        // do something that throws an exception
    }
    catch (e) {
        // handle exception
    }
}
```

...use...

```
var object = ['foo', 'bar'], i;
try {
    for (i = 0, len = object.length; i < len; i++) {
        // do something that throws an exception
    }
}
catch (e) {
    // handle exception
}
```

## 41 – Set timeouts to XMLHttpRequests

You could abort the connection if an XHR takes a long time (for example, due to a network issue), by using `setTimeout()` with the XHR call.

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function () {
    if (this.readyState == 4) {
        clearTimeout(timeout);
        // do something with response data
    }
}
var timeout = setTimeout(function () {
```

```

        xhr.abort(); // call error callback
    }, 60*1000 /* timeout after a minute */ );
xhr.open('GET', url, true);

xhr.send();

```

As a bonus, you should generally avoid synchronous XHR calls completely.

## 42 – Deal with WebSocket timeout

Generally when a WebSocket connection is established, a server could time out your connection after 30 seconds of inactivity. The firewall could also time out the connection after a period of inactivity.

To deal with the timeout issue you could send an empty message to the server periodically. To do this, add these two functions to your code: one to keep alive the connection and the other one to cancel the keep alive. Using this trick, you'll control the timeout.

Add a `timerID` ...

```

var timerID = 0;
function keepAlive() {
    var timeout = 15000;
    if (webSocket.readyState == webSocket.OPEN) {
        webSocket.send('');
    }
    timerId = setTimeout(keepAlive, timeout);
}
function cancelKeepAlive() {
    if (timerId) {
        clearTimeout(timerId);
    }
}

```

The `keepAlive()` function should be added at the end of the `onOpen()` method of the `webSocket` connection and the `cancelKeepAlive()` at the end of the `onClose()` method.

## 43 – Keep in mind that primitive operations can be faster than function calls. Use VanillaJS.

For example, instead of using...

```

var min = Math.min(a,b);
A.push(v);

```

...use...

```
var min = a < b ? a : b;  
A[A.length] = v;
```

**44 – Don't forget to use a code beautifier when coding. Use JSLint and minification (JSMin, for example) before going live.**

**45 – JavaScript is awesome: [Best Resources To Learn JavaScript](#)**

- Code Academy JavaScript tracks: <http://www.codecademy.com/tracks/javascript>
- Eloquent JavaScript by Marjin Haverbeke: <http://eloquentjavascript.net/>
- Advanced JavaScript by John Resig: <http://ejohn.org/apps/learn/>

## Conclusion

I know that there are many other tips, tricks and best practices, so if you have any ones to add or if you have any feedback or corrections to the ones that I have shared, please add a comment.

## 12 : JavaScript Architecture for the 23rd Century

<https://modernweb.com/javascript-architecture-23rd-century/>

Jonathan discusses JavaScript architecture patterns of the 23rd century and what the future of JavaScript holds for the 24th century."

JavaScript applications have grown in size and complexity for the last several years. More and more single page applications have hit the market, and demands for that type of experience have increased to the point where even Google finally decided to [render JavaScript](#) when it crawls pages.

This demand for SPA type applications has made JavaScript architecture increasingly important. JavaScript, being a dynamic language, you have to go the extra mile to ensure that the code is written in a maintainable fashion and to avoid spaghetti code.

For a long time it seemed like it was ok to just use a single file full of jQuery selectors and event handlers. This is just not a sustainable pattern. The Modern Web that we are entering here in the 23rd century demands a more thought out, and architected approach.

### Architecture Patterns

An architectural pattern is not something that you sit down and write. No one sits at their desk and thinks about how to write a new shiny pattern. A pattern is something that comes about as a discovery. Some problem is solved potentially multiple times and a pattern is extracted from the solution(s).

### Constructor

First of all let's take a step back into some native JavaScript functionality that has been there for years, the constructor pattern.

```
function Starship() {}
```

Every function you declare in JavaScript can be used as a **constructor** to create an instance. This allows you to create reusable functions.

```
var enterprise = new Starship(),
IKSBuruk = new Starship();
```

You can modify the original function to take arguments to help describe the instance better. You can assign the arguments to properties on `this` which will refer to the instance of the constructor.

```
function Starship(owner, operator, type) {
  this.owner = owner;
```

```
    this.operator = operator;
    this.type = type;
}

var enterprise = new Starship('Federation', 'Star Fleet', 'Class 1 Heavy Cruiser'),
birdOfprey = new Starship('Klingon Empire', 'Klingon Imperial Fleet', 'Klingon
Warship');
```

You can also utilize a built in feature of JavaScript constructors called the `prototype`. The `prototype` is simply an object. On it, you define properties and methods that will be added to every instance of a constructor.

```
function Starship(owner, operator, type, weapons) {
    /* ... */
    this.weapons = weapons;
}

Starship.prototype.fire = function(weapon) {
    this.weapons[weapon].launch();
};

function PhotonTorpedoSystem() {}
PhotonTorpedoSystem.prototype.launch = function() {
    console.log('launching torpedos');
};

var torpedos = new PhotonTorpedoSystem(),

var weaponSystem = {
    torpedos: torpedos;
};

var enterprise = new Starship(
    'Federation',
    'Star Fleet',
    'Class 1 Heavy Cruiser',
    weaponSystem
);
enterprise.fire('torpedos') // launching torpedos;
```

What's great about the `prototype` system in JavaScript is it enables you to create a system of inheritance. Inheritance in JavaScript is a little bit different than in other languages, but it's not too difficult with a bit of practice.

```
function ConstitutionClass(captain, firstOfficer, missionDuration) {
  this.captain = captain;
  this.firstOfficer = firstOfficer;
  this.missionDuration = missionDuration;

  Starship.apply(this, ['Federation', 'Star Fleet', 'Class 1 Heavy Cruiser',
  weaponSystem]);
}

ConstitutionClass.prototype = Object.create(Starship.prototype);
ConstitutionClass.prototype.constructor = ConstitutionClass;

ConstitutionClass.prototype.warp = function(speed) {
  console.log('warping at: ' + speed);
};

var enterprise = new ConstitutionClass('Kirk', 'Spock', 5);
enterprise.fire('torpedos'); // Launching Torpedos
enterprise.warp(14.1); // warping at: 14.1
```

Here we've created the `ConstitutionClass` constructor. This constructor takes a `captain`, `firstOfficer`, and `missionDuration` as arguments. It then goes on to call `Starship.apply` and passes in an array of arguments. This ensures that the parent constructor gets called.

Every function in JavaScript has a `call` and `apply` method that allows you to invoke a function by passing in a `context` and, in the case of `apply`, an array of arguments to use when invoking the function.

Next, to properly tell the `ConstitutionClass` to inherit from `Starship` we simply use `Object.create` to assign the prototype of `ConstitutionClass`.

What this does is tell `ConstitutionClass` that it's a type of `Starship` by adding all of the `Starship` prototype properties and methods to the `ConstitutionClass` prototype.

Check out how this looks when you run it and open up the console...

```

----- launching torpedos -----
warping at: 14.1
▼ ConstitutionClass {captain: "Kirk", firstOfficer: "Spock", missionDuration: 5, owner: "Federation", operator: "Starfleet"…} ⓘ
  captain: "Kirk"
  firstOfficer: "Spock"
  missionDuration: 5
  operator: "Starfleet"
  owner: "Federation"
  type: "Class 1 Heavy Cruiser"
▶ weapons: Object
▼ __proto__: ConstitutionClass
  ▶ constructor: function ConstitutionClass(captain, firstOfficer, missionDuration) {
  ▶ warp: function (speed) {
    ▶ __proto__: Starship
      ▶ constructor: function Starship(owner, operator, type, weapons) {
        ▶ fire: function (weapon) {
          ▶ __proto__: Object
            ▶ __defineGetter__: function __defineGetter__() { [native code] }
            ▶ __defineSetter__: function __defineSetter__() { [native code] }
            ▶ __lookupGetter__: function __lookupGetter__() { [native code] }
            ▶ __lookupSetter__: function __lookupSetter__() { [native code] }
            ▶ constructor: function Object() { [native code] }
            ▶ hasOwnProperty: function hasOwnProperty() { [native code] }
            ▶ isPrototypeOf: function isPrototypeOf() { [native code] }
            ▶ propertyIsEnumerable: function propertyIsEnumerable() { [native code] }
            ▶ toLocaleString: function toLocaleString() { [native code] }
            ▶ toString: function toString() { [native code] }
            ▶ valueOf: function valueOf() { [native code] }
            ▶ get __proto__: function __proto__() { [native code] }
            ▶ set __proto__: function __proto__() { [native code] }

```

You can easily see how the `enterprise` instance of `ConstitutionClass` has correctly been assigned its properties, as well as the properties of a `Starship`.

Notice also how the `ConstitutionClass` has a `warp` method as well as a `fire` method that it inherits from the `Starship`.

That's **prototypical inheritance**.

## IIFEs to prevent global leaks

An important thing to note about the code above is there are several global leaks occurring. When you simply declare a function or variable in a JavaScript file, it will automatically be added to the global namespace (the `window` in the case of a browser);

This code here will create several globals...

```

function PhotonTorpedoSystem() {}
PhotonTorpedoSystem.prototype.launch = function() {
  console.log('launching torpedos');
};

var torpedos = new PhotonTorpedoSystem(),

var weaponSystem = {
  torpedos: torpedos;
};

```

After running this code you will have created `window.PhotonTorpedoSystem`, `window.torpedos`, and `window.weaponSystem`. This is not an ideal scenario.

One great pattern for avoiding this issue is to utilize an Immediately Invoked Function Expression....

```
(function(global) {

    function PhotonTorpedoSystem() {}
    PhotonTorpedoSystem.prototype.launch = function() {
        console.log('launching torpedos');
    };

    var torpedos = new PhotonTorpedoSystem(),

    var weaponSystem = {
        torpedos: torpedos;
    };

    /* Create instance of enterprise etc */

    global.PhotonTorpedoSystem = PhotonTorpedoSystem; // Export just this to the window
}(window));
```

Here you can see the `(function(global) {` at the beginning and the `})(window));` at the end. This causes the anonymous function `function(global)` to fire right away, and sets `global` to the `window`. This all works because the IIFE creates a closure. Everything defined inside the IIFE is contained in the memory for that function alone unless it gets exported on the `global` we defined.

Now you can explicitly set things on the `window` for use in other JavaScript files.

This is a step in the right direction, but another even better pattern to utilize here would be the **Namespace Pattern**.

```
// weaponSystems.js
(function(global, NS) {
    NS.Weapons = NS.Weapons || {};

    function PhotonTorpedoSystem() {}
    PhotonTorpedoSystem.prototype.launch = function() {
        console.log('launching torpedos');
    };

    /* ... */

    NS.Weapons.PhotonTorpedoSystem = PhotonTorpedoSystem; // Export just this to the
```

```
NS.Weapons namespace
}(window, window.NS = NS || {});
```

Notice how we've changed the bottom of the IIFE to `}(window, window.NS = NS || {});`. This will pass the `window`, as well as an `NS` object. This slightly odd notation here says, "pass NS if it exists, or create a new object for it".

You can also see this syntax used again here `NS.Weapons = NS.Weapons || {};`.

It's effectively the same thing as using an if statement, but is more terse.

```
if (!NS.Weapons) {
  NS.Weapons = {};
}
```

You can add whatever objects and functions you want to on to the namespace. This will allow you to avoid adding too many globals to the window.

```
(function(global, NS) {
  NS.Ships = NS.Ships || {};

  function Starship(owner, operator, type, weapons) { /* ... */ }
  function ConstitutionClass(captain, firstOfficer, missionDuration) { /* ... */ }

  /* ... */

  NS.Ships.Starship = Starship;
  NS.Ships.ConstitutionClass = ConstitutionClass;

})(window, window.NS = NS || {});
```

## IIFE for the Revealing Module Pattern

Another pattern for creating functions is the "revealing module pattern".

```
NS.Owners.starFleet = (function() {
  var ships = [];

  var addShip = function(ship) {
    ships.push(ship);
  };

  return {
    addShip: addShip
  };
});
```

```

var removeShip = function(ship) {
  var index = ships.indexOf(ship);

  if (index > -1) {
    ships.splice(index, 1);
  }
};

var getTotal = function() {
  return ships.length;
};

return {
  addShip: addShip,
  removeShip: removeShip,
  totalShipsInFleet: getTotal
};
}());

```

This pattern allows you to define the public API for an object as well as have a few private members because of the IIFE. In this case, `ships` stays private because it's defined within the IIFE, whereas `addShip` and `removeShip` are exported via the object in the `return` statement.

```

(function() {
  var enterprise = new NS.Ships.ConstitutionClass('Kirk', 'Spock', 5);

  NS.Owners.starFleet.addShip(enterprise);

  NS.Owners.starFleet.getTotal(); // 1
})();

```

Now you can use `NS.Owners.starFleet` to add ships to Star Fleet.

## Organizing Files

Architectural patterns are very helpful for creating maintainable web applications. One further problem exists though in large scale applications, and that's file organization.

In the early days, having all of your code in a single Javascript file was still not a great idea, but it could work because there wasn't that much code. Today in the 23rd century, that's just not the case. There are

hundreds, if not thousands, of lines of code necessary for a web application. Having a single JavaScript file is just not feasible.

When you start adopting architectural patterns like constructor or module patterns, your code will have been broken down into smaller units. This is great for multiple reasons. For one, you'll now be able to move these small units in to their own files. And it makes unit testing much easier.

As a goal, try to keep each JavaScript file containing one logical “class” or module.

You may have something like this.

```
/js/  
/js/app.js  
/js/starship.js  
/js/constitutionClass.js  
/js/starFleet.js
```

This is much nicer. You're able to logically divide up each piece of functionality. After a while though, you may have too many JS files in the `/js` folder and may want to consider some subfolders.

There are only two hard things in Computer Science: cache invalidation and naming things.

— Phil Karlton

Naming things is hard, so when getting ready to divide your application up into subfolder, just pick some standards for you and your team and follow them. Consistency always wins!

```
/js/  
/js/app.js  
/js/classes/starship.js  
/js/classes/constitutionClass.js  
/js/core/starFleet.js  
/js/infrastructure/../js  
/js/templates/../js
```

Another common organizational strategy is to break up large pieces of functionality into their own folders.

```
/js/  
/js/app.js  
/js/ships/starship.js  
/js/ships/constitutionClass.js  
/js/core/starFleet.js  
/js/captainsLog/../js  
/js/weaponSystems/../js
```

No matter what you choose, the idea is to maintain consistency within your structure for your team. There's no wrong answer here, just pick something and go with it.

One hiccup you'll find breaking up your logic into many files though is dealing with loading them into your HTML pages.

With a structure like this you'll end up having to have many `script` includes that have to be in a very specific order to avoid errors.

```
<script src="https://x6ar5ez4ac-flywheel.netdna-ssl.com/js/app.js"></script>
<script src="https://x6ar5ez4ac-flywheel.netdna-ssl.com/js/core/starFleet.js">
</script>
<script src="https://x6ar5ez4ac-flywheel.netdna-
ssl.com/js/weaponSystems/photonTorpedos.js"></script>
<script src="https://x6ar5ez4ac-flywheel.netdna-ssl.com/js/starship.js"></script>
<script src="https://x6ar5ez4ac-flywheel.netdna-ssl.com/js/constitutionClass.js">
</script>
```

If any one of those files gets included in the wrong order, you'll run into problems. You also will have many synchronous script includes that slow your page render.

Enter AMD.

## AMD

Asynchronous Module Definition is a specification created by the CommonJS group to handle these types of issues. It defines a specification for creating and requiring modules into an application.

One of the best implementations of AMD is [require.js](#).

With require.js, you create many modules just like we have been working with so far, but there's a wrapper for each module. This wrapper not only hides globals from the window, but also helps define a dependency chain which allows you to have a single `script` tag on your page. There is also a great build tool called `r.js` that can compile all your modules to a single minified file.

To start, download require.js and include it in your page.

```
<script src="https://x6ar5ez4ac-flywheel.netdna-ssl.com/vendor/require.js" data-
main="/js/main"></script>
```

The `data-main` here points to a `main.js` file in our `/js` directory. This `main` file is responsible for some initial setup that tells `require` a bit about our application and kicks off the process of loading in files.

```

require.config({
  paths: {
    'foo': '/vendor/foo'
  }
});

require(['app'], function(app) {
  app.init();
});

```

Here is a very simple `main` file. The `require.config` can set up some aliases if you need them. Then you use the `require` method by passing in an array of module names. The module names correspond directly to their file names minus the `.js` extension. Here we kick off the application by loading `/js/app.js`.

`app.js` would look something like...

```

define(function(require) {
  var ConstitutionClass = require('ships/constitutionClass');
  starFleet = require('core/starFleet'),
  captainsLog = require('captainsLog/log');

  return {
    init: function() {
      var enterprise = new ConstitutionClass('Kirk', 'Spock', 5);

      starFleet.addShip(enterprise);
      captainsLog.record('Stardate 43125.8. Commence 5 year mission into deep
space');
    }
  };
});

```

This module is defined with the commonjs syntax. There are 2 ways you can export an API with require.js's commonjs syntax as well. Either via the `return` statement as above, or via the actual `commonjs` way of using `module.exports`.

```

module.exports = {
  init: function() {
    var enterprise = new ConstitutionClass('Kirk', 'Spock', 5);
  }
};

```

```

    starFleet.addShip(enterprise);

    captainsLog.record('Stardate 43125.8. Commence 5 year mission into deep
space');

}
};


```

Another quick note about require.js. All the examples below are going to use the commonjs syntax for creating modules, however you can either use the commonjs syntax or use a syntax like this...

```

define([
  'ships/constitutionClass',
  'core/starFleet',
  'captainsLog/log'
], function(ConstitutionClass, starFleet, captainsLog) {

  return {
    init: function() {
      var enterprise = new ConstitutionClass('Kirk', 'Spock', 5);

      starFleet.addShip(enterprise);
      captainsLog.record('Stardate 43125.8. Commence 5 year mission into deep
space');
    }
  }
});

```

Rather than a function that asks for the require function `function(require)`, you can pass in an array of dependencies. The modules are loaded in and are passed in as the arguments to the function.

`function(ConstitutionClass, starFleet, captainsLog) {}`. This is important to know because what require.js actually does is convert the commonjs syntax into this syntax for you.

Either syntax is perfectly valid, so choose which you prefer. The commonjs syntax is nice though because in theory you could share your modules with node, and/or browserify which we'll talk about later.

Then the `ships/constitutionClass.js` would look like...

```

define(function(require) {
  var Starship = require('ships/starship');

  function ConstitutionClass() { /* ... */}

```

```
    return ConstitutionClass;
});
```

This module returns a constructor function just like we created above, except now there's no need for any namespacing because you're working with a module that keeps everything private except what you return from the module.

You can also see how the dependency chain is beginning to form. `app.js` requires `ships/constitutionClass.js` which then requires `ships/starship.js`. This will continue on and on down the chain of dependencies.

A module like `core/starFleet.js` might look like this...

```
define(function() {
  var ships = [];

  var addShip = function(ship) {
    ships.push(ship);
  };

  var removeShip = function(ship) {
    var index = ships.indexOf(ship);

    if (index > -1) {
      ships.splice(index, 1);
    }
  };

  var getTotal = function() {
    return ships.length;
  };

  return {
    addShip: addShip,
    removeShip: removeShip,
    totalShipsInFleet: getTotal
  };
});
```

Effectively the same thing as the revealing module pattern gave us. Simply return an object literal with a public API.

## Browserify

Browserify is another method of creating JavaScript modules. With browserify, you basically write commonjs modules and run `browserify` with the global node.js module and it will bundle up your code for use in the browser.

The syntax of creating modules with require and browserify is very close if you choose to use the commonjs syntax and the `module.exports` when working with require.

The `app.js` above written as an actual commonjs module looks like...

```
var ConstitutionClass = require('ships/constitutionClass');
starFleet = require('core/starFleet'),
captainsLog = require('captainsLog/log');

module.exports = {
  init: function() {
    var enterprise = new ConstitutionClass('Kirk', 'Spock', 5);

    starFleet.addShip(enterprise);
    captainsLog.record('Stardate 43125.8. Commence 5 year mission into deep
space');
  }
};
```

The only difference is that you use `module.exports` to export your public API rather than a return statement.

To use browserify, install it with npm...

```
npm install -g browserifylanguage-shell
```

Then you run it...

```
browserify main.js > bundle.js
```

This will output a single `bundle.js` file for use in the browser. There are a lot of advantages to both browserify and require.js. Module loading helps break up your application and no matter which you choose, it will help you create a more maintainable application.

## ES6

One of the many awesome features coming in ES6 or ES.next is built-in classes with native JavaScript. ECMAScript is currently working on the spec for classes. You can view the [unofficial draft](#) of the spec.

The new spec for JavaScript defines a new way of creating classes in JavaScript. If we take our previous examples and write them using ES6, the `Starship` class would look like...

```
class Starship {
    constructor(owner, operator, type, weapons) {
        this.owner = owner;
        this.operator = operator;
        this.type = type;
        this.weapons = weapons;
    }

    fire(weapon) {
        this.weapons[weapon].launch();
    }
}
```

You can see immediate differences here. First is the `class` identifier. Then you can see how you define a constructor with `constructor` and methods by simply a `methodname(a, b, c)` signature. No need to use `function` anywhere here.

Inheritance is also much easier. The `ConstitutionClass` can easily inherit from the `Starship` using `extends`...

```
class ConstitutionClass extends Starship {
    constructor(captain, firstOfficer, missionDuration) {
        this.captain = captain;
        this.firstOfficer = firstOfficer;
        this.missionDuration = missionDuration;

        super('Federation', 'Star Fleet', 'Class 1 Heavy Cruiser', {
            torpedos: new PhotonTorpedoSystem()
        });
    }

    warp(speed) {
        console.log('warping at: ' + speed);
    }
}
```

You'll also notice `super` here which is how you call the parent method and takes the place of `Starship.apply(this, /* ... */)`.

This new `class` syntax is incredibly exciting for the future of JavaScript. Having built in classes is something JavaScript has needed for a while, and this new syntax is looking great.

## Traceur

There is no telling when this `class` syntax will make it into browsers, but fortunately there's a tool called [Traceur](#) that allows us to work with new features of JavaScript before they release.

You can install traceur with npm...

```
npm install -g traceur
```

And you can compile ES6 file down to ES5 for use today.

```
traceur --out build.js --script starship.js
```

Just note that if you plan on using it in the browser you must also include the runtime file above your includes...

```
<script src="bin/traceur-runtime.js"></script>
```

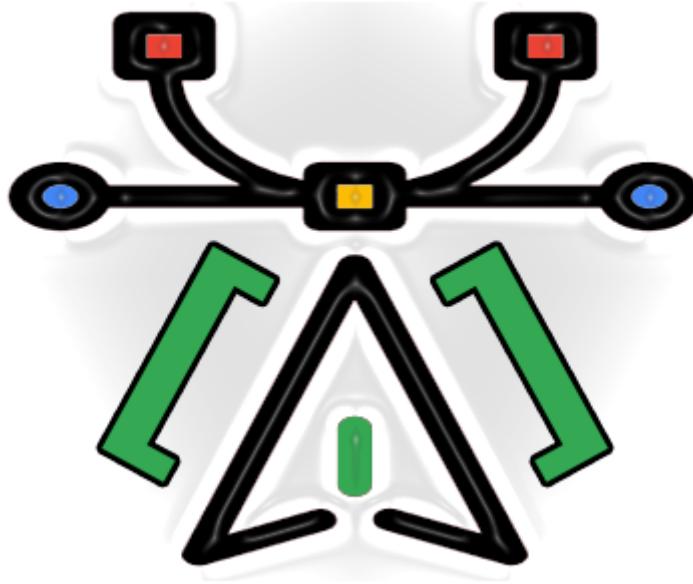
Hopefully ES6 classes is the future of how we'll be writing maintainable JavaScript in the future.

## Conclusion

JavaScript is an ever-changing language. As it grows there will be more and more ways to use it, and architect it. The architectural patterns above are a great way you can create JavaScript applications today, while the `class` syntax will be the way of things in the 24th century. Keep JavaScripting, and live long and prosper.

## 13 : 9 New Array Functions in ES6

<http://vegibit.com/new-array-functions-in-es6/>



There are many new extensions to the array object in ES6. In this journey of learning the foo of ES6, we will set our targets on mastering the ways of new functions like `Array.of()`, `Array.from()`, `Array.fill()`, `Array.find()`, `Array.findIndex()`, `Array.copyWithin()`, `Array.entries()`, `Array.keys()`, and `Array.values()`. These new functions make it easier to fill up arrays, or find data within them. In addition to that it is now easier to work with specific keys and values in the array itself. These newer functions are lessening the need for utility libraries so they are definitely welcome. Let's put the rubber to the road with all of the new array extensions in ES6 right now.

### 1. `Array.of()`

To learn about the new `Array.of()` function, let us first take a look at a strange little quirk in ES5 concerning using the `Array` constructor. We'll set up an array of `prices`, then take a look at the length of the array.

```
let prices = Array(5000);
console.log(prices.length);

// 5000
```

When the code runs, it tells us that we have an array with a length of 5000. What?! We only passed one value, `5000`, into the array. Well in ES5, if you pass just one value to the `array constructor` which is numeric, an array of that size will be created. That is kind of strange, don't you think? This is the purpose of `Array.of()`. Let have a redo with our new function.

```
let prices = Array.of(5000);
console.log(prices.length);

// 1
```

**Nice!** In this go round, we find the length of `prices` to be `1`. That seems to make much more sense.

`Array.of()` is a new way of creating an array which fixes this odd behavior in ES5. If you create an array with just one numeric value, the array is created with just that value and not the amount of that value.

## 2. `Array.from()`

Let's see how the new `Array.from()` function works. Below we have set up an array with three values of 500, 700, and 1000. On the second line, we make a call to `Array.from()` and pass in `prices` as the *first argument*, and an [arrow function](#) as the *second argument*. By running the code, we see that the prices are now taxed at 5 percent. `Array.from()` creates a brand new array based on `prices`, and for each element in that array the arrow function is called. So basically we take each price and multiply it by 1.05 denoting a tax rate of 5 percent.

```
let prices = [500, 700, 1000];
let taxed = Array.from(prices, price => price * 1.05);
console.log(taxed);

// [525, 735, 1050]
```

### Using `Array.from()` with three arguments

In the prior example, we passed two arguments to the `Array.from()` function. The first was the array we were working with, and the second was a function. In this example we will pass an array, a function, *and also an object*. Say you had a site that listed items for sale, but for each item sold you had to pay a listing fee of 5 dollars. Let's see how to calculate this.

```
let prices = [500, 700, 1000];
let totalprice = Array.from(prices, function (price) {
    return price + this.listingfee;
}, {listingfee: 5});

console.log(totalprice);

// [505, 705, 1005]
```

What is happening here is that the third argument to `Array.from()` is an object which becomes `this` in the function. That is why we are able to take the `price` and add `this.listingfee` to give us the total price. Note that when using this technique, we need to use a standard function as opposed to an arrow function for the second argument to `Array.from()`. This is because arrow functions do not allow meddling with the `this` value.

### 3. `Array.fill()`

ES6 now gives you an easy way to fill up an array using `Array.fill()`. Let's see a quick example.

```
let prices = [500, 700, 1000];
prices.fill(2000);
console.log(prices);

// Array [ 2000, 2000, 2000 ]
```

So it looks like this function will overwrite any existing values in all keys of the array with the provided value. Since we call `.fill()` on the `prices` array which has 3 elements in it, all elements in the array are now `2000`. There is also an option to pass a second argument to `Array.fill()` in order to start at a specific index. Let's see how to do that.

```
let prices = [500, 700, 1000];
prices.fill(2000, 2);
console.log(prices);

// Array [ 500, 700, 2000 ]
```

By passing the value of `2` as the second argument, we are telling the fill function to start filling the array at the 2nd index. Since arrays are 0 based as we know, it is the third value in our array that gets overwritten with the value of `2000`.

Now, why only pass two arguments when you can pass three?! Here we will pass another argument to `Array.fill()`. This will demonstrate that the second argument specifies what index to start at while the third argument specifies *where to stop*. Check it out now.

```
let prices = [500, 600, 700, 800, 900, 1000, 1500];
prices.fill(2000, 2, 4);
console.log(prices);

// Array [ 500, 600, 2000, 2000, 900, 1000, 1500 ]
```

We added a few more values to our original array so that it is a bit easier to see how this works. Notice that we begin filling with the value of `2000` at *index 2* and stop at *index 4*. Note that the filling stops before actually placing a value in the index. This is why you see only index 2 and 3 with the value of 2000.

## 4. Array.find()

**Array.find()** is another new function added to arrays in ES6. You can use it to easily find a value in an array that meets a given criteria. Let's see how it works.

```
let prices = [500, 600, 700, 800, 900, 1000, 1500];
let result = prices.find(price => price > 777);
console.log(result);

// 800
```

Notice that we pass an arrow function to the `.find()` function. That function is applied against every element in the array, and as soon as it finds a value that meets the criteria, that value is returned. It does not continue to return all values that meet the criteria. This is why we only get one result in this example. Let's run through this example just one more time to see it in action.

```
let prices = [500, 600, 700, 800, 900, 1000, 1500];
let result = prices.find(price => price < 777 && price > 600);
console.log(result);

// 700
```

So here, we use a compound expression in the arrow function to be more specific. We specify that we want a value that is less than 777 but also greater than 600. 700 is the first value to meet that criteria, so we get that result back.

## 5. Array.findIndex()

In addition to `Array.find()`, we now also have the **Array.findIndex()** function which works in a similar way but instead of returning the value, it returns the *index*. Let's see how it works.

```
let prices = [500, 600, 700, 800, 900, 1000, 1500];
let result = prices.findIndex(function (price) {
    return price == this;
}, 1000);
console.log(result);
```

```
// 5
```

Here we use a regular JavaScript function passed into the `.findIndex()` function. We simply return the result of price being equal to `this`. `this` is set to 1000, which is the second argument to `.findIndex()`. We can see that the value of 1000 lives at index 5 of the zero based array.

## 6. Array.copyWithin()

`Array.copyWithin()` is an interesting addition to the array function library in ES6. With it, you can copy values inside the array just like the name implies. It takes a value from one index, and places it in another. Here is an example.

```
let prices = [500, 600, 700, 800, 900, 1000, 1500];
prices.copyWithin(3, 1);
console.log(prices);

// Array [ 500, 600, 700, 600, 700, 800, 900 ]
```

The key to understanding `copyWithin()` is what the arguments mean. Argument 1 is the index which will be overwritten. It is where data will be copied to. The second argument is the data to copy from. So in our example, we are saying that we are going to copy the data at index 1 (600) and paste it into index 3 (800). After the function runs, we see that index 3 no longer holds 800, it now holds 600. It is working as expected. There is also the option to pass a third argument to `copyWithin()`. This third argument tells us how many items to copy. Let's see how it works.

```
let prices = [500, 600, 700, 800, 900, 1000, 1500];
prices.copyWithin(2, 0, 3);
console.log(prices);

// Array [ 500, 600, 500, 600, 700, 1000, 1500 ]
```

The destination index is 2, or the third value in the array. We start copying from index 0, or the first value in the array. We are going to copy three successive values starting at index 0. So the result correctly displays 500 in the 2nd index position, followed by two additional copied values of 600 and 700. Index 5 and 6 are unaffected and so they contain their original values of 1000 and 1500.

## 7. Array.entries()

The `Array.entries()` function is a really cool addition to the language. It takes an array, and creates a listing so to speak of each entry. Let's examine closely how it works.

```
let words = ['Lenovo', 'Tablet', 'Coffee'];
console.log(words.entries());

// Array Iterator { }
```

First up, we simply set up an array of words. Then we log out the value of calling `.entries()` on that array. Interesting! It results in an [array iterator](#). Hmm, well we learned that we can call the `.next()` function on iterators, so let's try that out!

```
let words = ['Lenovo', 'Tablet', 'Coffee'];
console.log(words.entries().next());

// Object { value: Array[2], done: false }
// The Array held in value contains 0: 0 and 1: Lenovo
```

Awesome! We can see that we get the first list so to speak of the array. The iterator object is not done, so it is set to `false`. However the `value` contains an array which holds two values. At index 0 of that array is the index of where `Lenovo` lives. That is index `0`. At index 1 is the value itself, which is `Lenovo`. Very neat.

Finally, since we know we can use the [spread operator](#) on iterators, let's go ahead and do that now.

```
let words = ['Lenovo', 'Tablet', 'Coffee'];
console.log(...words.entries());

// Array [ 0, "Lenovo" ]
// Array [ 1, "Tablet" ]
// Array [ 2, "Coffee" ]
```

The `.entries()` function gives us the index / value pair of the array.

## 8. Array.keys()

**Array.keys()** works in a similar way to `Array.entries()` except that it only provides the keys of the array. See it in action now.

```
let words = ['Lenovo', 'Tablet', 'Coffee'];
console.log(...words.keys());

// 0 1 2
```

## 9. Array.values()

Finally, we have our `Array.values()` function which is similar to the prior two examples but provides only the values at each key. Observe young Jedi.

```
let words = ['Lenovo', 'Tablet', 'Coffee'];
console.log(...words.values());

// Lenovo Tablet Coffee
```

## 9 New Array Functions in ES6 Summary

You are now wielding special kung foo powers in your handling of arrays in ES6. No longer are you confined by the burdens of working with arrays such as was done in ES5. On your path to enlightenment you have found new and easier ways to get your work done, and as such, we hope you have found this journey exciting.

## 14 : Implementing EventEmitter in ES6

<http://www.datchley.name/es6-eventemitter/>

So, since I've been swamped with work recently and haven't posted much, I'd like to get back into the swing of things and throw up a simple EventEmitter implementation I put together for my current [ngReflux](#) project I'm working on.

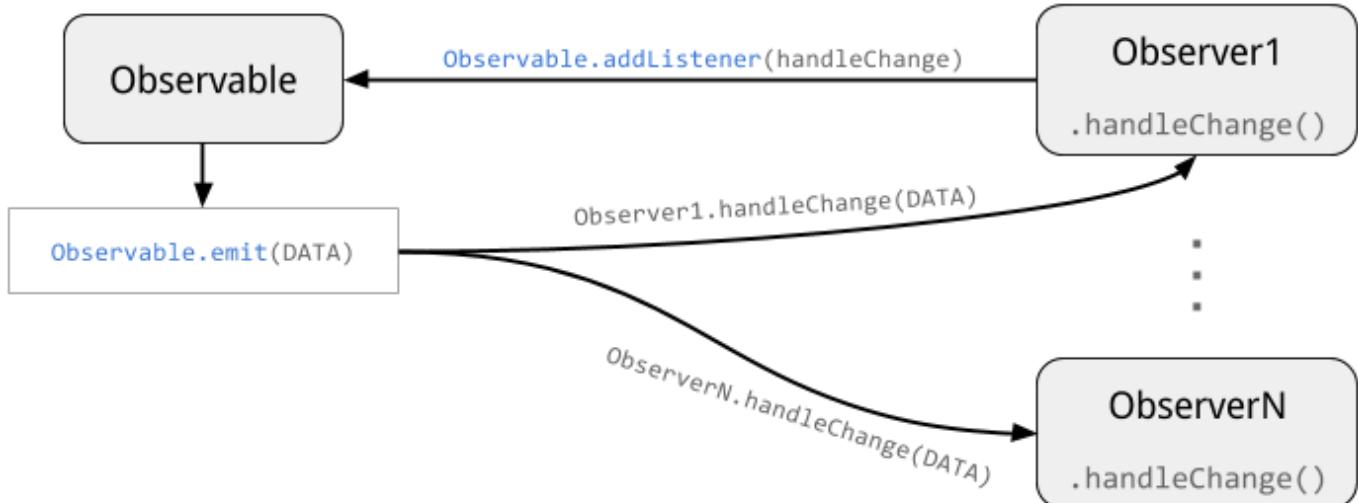
ngReflux is a Flux implementation for use with AngularJS applications and based on Mikael Brassman's [original gist](#) idea for RefluxJS. You can read more about the Flux architecture [elsewhere](#).

Since Reflux removes the Dispatcher from the Flux flow altogether, it makes both Actions and Stores observables by having them prototypically inherit from an EventEmitter. Reflux uses [EventEmitter3](#), which is a full implementation for Node and the browser; but I wanted to keep this small and simple, with few outside dependencies, so I rolled my own.

### The Observer Pattern

Before we look at the code, let's quickly review the [Observer Pattern](#), which is what EventEmitter is based on.

The Observer Pattern is fairly simple. An object (*typically called a "subject"*) allows other Objects called Observers to be notified when some state in the Observable changes. This is essentially the pub/sub architecture.



Making an object "observable" focuses primarily on providing at least three different behaviors:

1. `addListener(label, callback)` - Allow others to "listen" to the observer, providing it a callback to invoke when the observable's state changes.
2. `removeListener(label, callback)` - Allow those listening to remove themselves as listeners so they can stop receiving notifications of state changes.

3. `emit(label, ...)` - Allow the observable to notify all observers of a state change by invoking each observer's callback with any changed state.

In our `EventEmitter` implementation we also have labels, which allows an object acting as an observable to allow others to listen for changes on a specific "channel", which is just a string identifying the type or category of changes the observer wants to be notified about. Observables can have one or more channels depending on their needs.

## Implementing EventEmitter

Let's use some of the latest Javascript ES2015/ECMA Script 6 features and get this thing rolling.

First, we'll setup our `EventEmitter` as a `class`, which is really just syntactic sugar for creating a function that is intended to be invoked with `new`, combined with adding methods on its prototype property.

```
class EventEmitter {  
  constructor() {  
    this.listeners = new Map();  
  }  
  addListener(label, callback) {}  
  removeListener(label, callback) {}  
  emit(label, ...args) {}  
}
```

Our class has a constructor which sets up an initially empty map for keeping track of our listeners. Note, too, the shorter syntax of declaring prototype methods on the class. We've stubbed them out above and will show the implementation below.

### `addListener(label, callback)`

Adding a listener takes two parameters. We'll need the label that identifies the type of notifications the listener wants to receive; and the callback function we should invoke for the listener when we emit that event.

```
addListener(label, callback) {  
  this.listeners.has(label) || this.listeners.set(label, []);  
  this.listeners.get(label).push(callback);  
}
```

We just ensure we have a queue (*array of*) of listeners for the given label, creating it if it isn't there; and then push the callback function into that queue.

### `removeListener(label, callback)`

Removing a listener takes the same parameters as `addListener`; but we need to be able to find that callback in the appropriate list in order to remove it.

```

let isFunction = function(obj) {
    return typeof obj == 'function' || false;
};

removeListener(label, callback) {
    let listeners = this.listeners.get(label),
        index;

    if (listeners && listeners.length) {
        index = listeners.reduce((i, listener, index) => {
            return (isFunction(listener) && listener === callback) ?
                i = index :
                i;
        }, -1);

        if (index > -1) {
            listeners.splice(index, 1);
            this.listeners.set(label, listeners);
            return true;
        }
    }
    return false;
}

```

Outside of our helper function to determine if an object is a `function`, the implementation is straight forward. Loop through the listeners for that label, if there are any, and retain the index of the callback that matches the one passed in, using that to remove that item from the list.

If you aren't familiar with using Array functions like `reduce` and `map`, take a look at my [previous post](#) on the subject.

### `emit(label, ...args)`

Here's where we emit events to all our listeners passing on some data that we think might interest them. We just need the label to emit the event on, which tells us which set of listeners to notify; and the remaining arguments are passed to each listener callback directly.

```

emit(label, ...args) {
    let listeners = this.listeners.get(label);

```

```

    if (listeners && listeners.length) {
        listeners.forEach((listener) => {
            listener(...args);
        });
        return true;
    }
    return false;
}

```

We use ES6's spread/rest operator here to collect all our arguments for emit into the single variable `args`; as well as to pass them as individual arguments to each listener callback invocation. Every time we emit on a label, all the other data we pass in gets passed to *all* our listeners callbacks.

Handling the arguments for each of those actions above in ES5 would be analogous to the following:

```

function emit(label) {
    var args = [].slice.call(arguments, 1);
    // ...
    listeners.apply(null, args);
}

```

The spread/rest operator helps make for cleaner and clearer code in this case.

## Putting it all together

Now that we have our `EventEmitter` class let's create an observer and see how it works. Our simple `Observer` constructor takes a single argument of the specific `Observable` that we want to listen on. We then listen for "change" events and output any data received to the console.

```

class Observer {
    constructor(id, subject) {
        this.id = id;
        this.subject = subject;
        this.subject.addListener("change", (data) => this.onChange(data));
    }
    onChange(data) {
        console.log(`#${this.id} notified of change:`, data);
    }
}

```

We use ES6's new string templating interpolation to format our console output as well, which is pretty nice.

Now we can create an Observable using our EventEmitter and a couple of Observers and fire off some events and see it all work.

```
let observable = new EventEmitter();
let [observer1, observer2] = [
  new Observer(1, observable),
  new Observer(2, observable)
];

observable.emit("change", { a: 1 });
// => (1) notified of change: { a: 1 }
// => (2) notified of change: { a: 1 }
```

We also used ES6's destructuring for the assignment on our observers, too.

Hopefully this has helped give you some insight into both the Observer Pattern and how EventEmitters are implemented. You can find out more about ES2015 and ES6 through Babel's [learning resource](#). You can view the full code in [a gist](#); and feel free to ask questions and leave comments!

*Updated* As commenters Greg and Christian pointed out, the better way to compare functions in the `removeListener()` method is to use straight `==` comparison rather than relying on `.toString()` to compare a decompiled version of the function. Using `toString()`'s results is inconsistent across various browsers/devices, and will not work for bound functions, using `.bind()`.

# 15 : Getting Functional with Javascript (Part 1)

<http://www.datchley.name/getting-functional-with-javascript-part-1/>

After reading about *currying*, *partial application* and other functional programming techniques, some developers are left wondering when, exactly, they would ever use those methods; and why would they want to?

With this three part blog post series we'll try to tackle that issue and show how you could approach solving a problem in a functional fashion in the context of a small, realistic example.

## What is Functional Programming

Before we dive right in, let's take a brief moment to review some practical functional programming concepts.

Functional programming focuses on the "*function*" as the primary expression of reuse. By building small functions that focus on very specific tasks, functional programming uses composition to build up more complex functions - this is where techniques like *currying* and *partial application* come into play.

Functional Programming uses functions as the declarative expression of reuse, avoiding mutating state, eliminating side effects & using composition to build functionality

Functional programming is essentially programming with functions! The additional considerations like avoiding mutating state, pure functions with no side effects and elimination of loops in favor of recursion are part of a Pure Functional Programming approach, which is built in in languages like Haskell.

We'll focus on the practical parts of functional programming that we can immediately make use of in Javascript in this blog series.

**Higher Order Functions** - Javascript has *first-class functions*, which means we can pass functions as arguments to other functions; and return functions as values from functions.

**Decorators** - Because we can make higher order functions, we can create functions that augment other functions' behavior and/or arguments

**Composition** - we can also create functions that are composed of multiple functions, created chained processing of inputs.

We'll introduce the techniques we'll use to take advantage of these features as they are needed. This lets us introduce them in context and keep the concepts digestible and easy to understand.

## Let's Build Something, Already!

Ok, so what is it we are going to build?

Let's take the typical example of needing to process some data coming back from an asynchronous request. In this case, the response is in JSON format and contains a list of blog post summaries.

Here's our response data we'll be working with: [View full data in Gist](#) and an example record.

```
// An example record from our JSON response data
var records = [
  {
    "id": 1,
    "title": "Currying Things",
    "author": "Dave",
    "selfurl": "/posts/1",
    "published": 1437847125528,
    "tags": [
      "functional programming"
    ],
    "displayDate": "2015-07-25"
  },
  // ...
];
```

**Our Requirements:** Now, suppose that we want to build a display of recent posts (*no older than a month*), organized in groups by tags and sorted by publish date. Let's think for a moment about what we need to do:

- filter out posts older than a month (say, 30 days).
- group the posts by their tags (*this might mean posts show up in two groups if they have more than one tag.*)
- sort each tag listing by published date, descending.

We'll cover each requirement above in a single post in this series, starting with filtering in this post.

## Filtering the Records

Our first step is to filter out the records where the published date is older than the last 30 days. Since functional programming is all about *functions* as the primary expression for reuse, let's build a function to encapsulate the act of filtering a list.

```
function filter(list, fn) {
  return list.filter(fn);
}
```

I can hear you asking now..., "*Really? That's all!?*"

Well, yes, and no.

While this function encapsulates the concept of filtering an array (`list`) using a predicate function (`fn`), which could have easily been done by just calling `list.filter(fn)` directly. So, why not do it that way?

Because when we abstract the operation into a function, we can then use *currying* to build a more useful function.

**Currying** is the act of taking a function with  $N$  arguments and returning a nested series of  $N$  functions that each take 1 argument.

For more information and to brush up on this concept, read my [previous post](#) about currying and an implementation of *left->right* currying.

```
fn = function(a,b,c)
curry(fn) = function(a) {
    return function(b) {
        return function(c) { return fn(a,b,c); }
    }
}
```

```
fn = function(a,b,c)
rightCurry(fn) = function(c) {
    return function(b) {
        return function(a) { return fn(a,b,c); }
    }
}
```

In this case we'll use a function called `rightCurry()`, which curries a function's arguments from right to left. Typically, a plain `curry()` function would curry arguments left to right.

Here's our implementation, along with another utility function `flip()`, which it uses internally.

```
// Returns a function which reverses the order of the
// arguments to the passed in function when invoked.
function flip(fn) {
    return function() {
        var args = [].slice.call(arguments);
        return fn.apply(this, args.reverse());
    };
}
```

```
// Returns a new function that curries the original
// function's arguments from right to left.
function rightCurry(fn, n) {
```

```

var arity = n || fn.length,
    fn = flip(fn);
return function curried() {
  var args = [].slice.call(arguments),
      context = this;

  return args.length >= arity ?
    fn.apply(context, args.slice(0, arity)) :
    function () {
      var rest = [].slice.call(arguments);
      return curried.apply(context, args.concat(rest));
    };
}

```

With currying we can create functions that allow us to create new, partially applied functions we can reuse. In our case, we'll use it to create a function which partially applies a predicate to the operation of filtering a list.



```

// A function to filter a list with a given predicate
var filterWith = rightCurry(filter);

```

This is basically the same as manually currying our binary `filter(list, fn)` function like this.

```

function filterWith(fn) {
  return function(list) {
    return filter(list, fn);
  }
}

```

And we can use it as follows?

```

var list = [1,2,3,4,5,6,7,8,9,10];

// Create a partially applied filter to get even numbers from a list
var justEvens = filterWith(function(n) { return n%2 == 0; });

```

```
justEvens(list);
// [2,4,6,8,10]
```

Wow, seems like a lot of work initially; but what we got out of this approach is

- using *currying* to create a general, reusable function, `filterWith()`, which can be used in a number of situations to create more specific list filters
- the ability to execute this new filter lazily whenever we get some data. *We can't call `Array.prototype.filter` without having it act on a list of data immediately*
- a more declarative API that aids in readability and understanding

## About that Predicate Function

Our `filterWith()` function needs a predicate function, which returns *true* or *false* when given an item in the list, to determine if that item should be returned in the newly filtered list.

Let's start off by making a more generic comparison function that can tell us if a given number is greater than or equal to another number.

```
// Simple comparison for '>='
function greaterThanOrEqual(a, b) {
  return a >= b;
}
```

Given that our published dates are in numeric, timestamp format (*milliseconds since the Epoch*) this should work fine. But, the predicate function for filtering Arrays is only passed a single argument to check, not two.

So, how do we make our binary comparison function work in a situation that needs a unary function?

Currying to the rescue again! We'll use it to make a function that can create unary comparison functions.

```
var greaterThanOrEqualTo = rightCurry(greaterThanOrEqual);
```

We can now use this curried version to create a unary predicate function that will work with list filtering, such as:

```
var list = [5,3,6,2,8,1,9,4,7],
  // a unary comparison function to see if a value is >= 5
  fiveOrMore = greaterThanOrEqualTo(5);

filterWith(fiveOrMore)(list);
// [5,6,8,9,7]
```

Awesome! Now we can be more specific and create a predicate that solves our original problem of filtering dates greater than or equal to 30 days ago:

```
var thirtyDaysAgo = (new Date()).getTime() - (86400000 * 30),
    within30Days = greaterThanOrEqualTo(thirtyDaysAgo);

var dates = [
  (new Date('2015-07-29')).getTime(),
  (new Date('2015-05-01')).getTime()
];

filterWith(within30Days)(dates);
// [143812800000] - July 29th, 2015
```

So far, so good!

We've created a predicate for filtering that can be easily reused. Also, because we're using a functional approach, our code is much more declarative and easy to follow - it sort of reads exactly as it works. Readability and maintenance are important things to consider when writing any code!

## Type Problems...

*Uh-oh, we have another issue!* Our program needs to filter a list of objects, so our predicate function is going to need to access the `published` property on each item passed in.

Our current predicate, `within30Days()`, doesn't handle object argument types, only scalar values! Let's solve that with another function! (*are you seeing a pattern here?*)

We'd like to reuse our existing predicate function; but modify its arguments so that it can work with our specific object types. Here's a new utility function that let's us augment an existing function by modifying its arguments.

```
function useWith(fn /*, txfn, ... */) {
  var transforms = [].slice.call(arguments, 1),
      _transform = function(args) {
        return args.map(function(arg, i) {
          return transforms[i](arg);
        });
      };
  return function() {
    var args = [].slice.call(arguments),
        targs = args.slice(0, transforms.length),
        remaining = args.slice(transforms.length);
```

```

        return fn.apply(this, _transform(targs).concat(remaining));
    }
}

```

This is our most interesting functional utility so far, and nearly identical to the function of the same name in the [Ramda.js library](#).

`useWith()` returns a function which modifies the original function, `fn`, so that when it is invoked, it will pass each argument through a corresponding transform (`txnfn`) function. If there are more arguments when invoked than transform functions, the remaining arguments will be passed in "as is."

Let's help that definition out with a small example. Simply, `useWith()` lets us do the following:

```

function sum(a,b) { return a + b; }
function add1(v) { return v+1; }
var additiveSum = useWith(sum, add1, add1);

// Before sum receives 4 & 5, they are each passed through
// the 'add1()' function and transformed
additiveSum(4,5); // 11

```

When we call `additiveSum(4,5)` we essentially get the following call stack:

- `additiveSum(4,5)`
  - `add1(4) => 5`
  - `add1(5) => 6`
  - `sum(5, 6) => 11`

We can use `useWith()` to modify our existing predicate to operate on object types rather than scalar values. First, let's use *currying* again to create a function that lets us create partially applied functions that can access objects by property name.

```

// function to access a property on an object
function get(obj, prop) { return obj[prop]; }
// Curried version of `get()`
var getWith = rightCurry(get);

```

Now we can use `getWith()` as the transform function to grab the `.published` date from each object to pass to our unary predicate function used in the filter.

```
// Our modified predicate that can work on the `published`  
// property of our record objects.  
  
var within30Days = useWith(greaterThanOrEqualTo(thirtyDaysAgo),  
getWith('published'));
```

Let's give this a try with some test data:

```
// Array of sample object data  
  
var dates = [  
    { id: 1, published: (new Date('2015-07-29')).getTime() },  
    { id: 2, published: (new Date('2015-05-01')).getTime() }  
,  
    within30Days = useWith(greaterThanOrEqualTo(thirtyDaysAgo),  
getWith('published'));  
  
// Get any object with a published date in the last 30 days  
filterWith(within30Days)(dates);  
// { id: 1, published: 143812800000 }
```

## Ready to Filter!

Ok, given our first requirement to keep only post records within the last 30 days, let's give our full implementation a run with our response data.

```
filterWith(within30Days)(records);  
// [  
//     { id: 1, title: "Currying Things", displayDate: "2015-07-25", ... },  
//     { id: 2, title: "ES6 Promises", displayDate: "2015-07-26", ... },  
//     { id: 7, title: "Common Promise Idioms", displayDate: "2015-08-06", ... },  
//     { id: 9, title: "Default Function Parameters in ES6", displayDate: "2015-07-  
06", ... },  
//     { id: 10, title: "Use More Parenthesis!", displayDate: "2015-08-26", ... },  
// ]
```

We now have a new list of *just* posts within the last 30 days. Looks like we've met our first requirement and are off to a good start. As we go along, we'll put our functional utilities in a library that we can reuse.

**Get the Source:** You can see the [source code](#) for this post, which has all our functional utilities in a single `.js` file and our main application logic in its own file as well. We'll continue to add to these in each post in this series.

## Summary

We've discovered some key functional programming techniques like *currying* and *partial application* and the contexts in which we can use them. We also found out that focusing on building small, useful functions in conjunction with functional techniques allows us to compose higher order functions and enable better reuse. With these basics under our belt, the next two posts will seem much less daunting.

In the next post in this series we'll combine what we have so far for filtering with grouping the records by *tag name*, where we'll introduce more list related functions and more flexible function composition as well.

- [currying](#)
- [functional programming](#)

# 16 : Getting Functional with Javascript (Part 2)

<http://www.datchley.name/getting-functional-with-javascript-part-2/>

In our [previous post](#) on functional programming we began introducing some functional themes by working through requirements for processing a typical [JSON response](#).

Here's a recap of our requirements:

- filter out posts older than a month (say, 30 days).
- group the posts by their tags (this might mean posts show up in two groups if they have more than one tag.)
- sort each tag listing by published date, descending.

That first post focused on our first requirement above - filtering out posts older than 30 days.

We also started a [library](#) of useful functional utilities, which we'll continue to add to in this post. You can view [the gist](#) to see the full source code for this series.

In this post, we'll cover our second requirement which is grouping our records in the newly filtered list by their [tags](#).

## Grouping Things

In Javascript, we can group items in a list using `Array#reduce()`, which we covered in a [previous blog post](#). Definitely take a look at that now if you aren't familiar; but the basic idea is that `reduce()` allows us to iteratively build up a new value by doing something with each item in the array.

`list.reduce(fn, initVal) => value (any)`



The diagram illustrates the concept of reduction. On the left, there is a list of five geometric shapes: two circles and three triangles. A large yellow arrow points from this list towards a single blue diamond shape on the right, representing how the reduce function iterates over the list to produce a single result.

Usually, you think of taking some list of values and using `reduce()` to produce a single, new value, like:

```
[1,2,3,4].reduce(function(sum, n) { return sum += n; }, 0); // 10
```

It's this ability to iterate over values and build up or accumulate a new value that allows us to use `reduce()` to perform grouping operations. For instance:

```
var list = [
  { name: 'Dave', age: 40 },
  { name: 'Dan', age: 35 },
  { name: 'Kurt', age: 44 },
  { name: 'Josh', age: 33 }
```

```

];
list.reduce(function(acc, item) {
  var key = item.age < 40 ? 'under40' : 'over40';
  acc[key] = acc[key] || [];
  acc[key].push(item);
  return acc;
}, {});
// {
//   'over40': [
//     { name: 'Dave', age: 40 },
//     { name: 'Kurt', age: 44 }
//   ],
//   'under40': [
//     { name: 'Dan', age: 35 },
//     { name: 'Josh', age: 33 }
//   ]
// }

```

In the above snippet, we use `reduce()` to iterate over a list of objects. We use an empty object as the starting point and group the records based on their age. This allows us to treat an object like a map, assigning records to groups identified by property names on the resulting object.

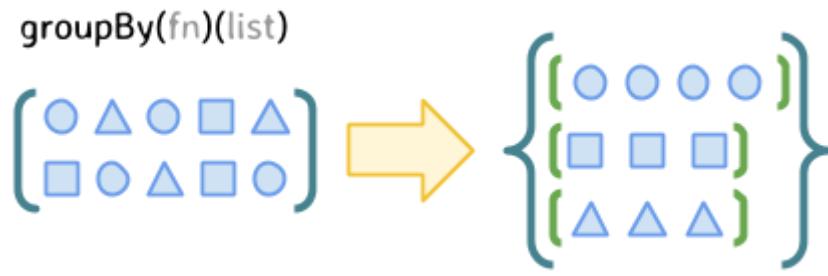
Let's use this ability via `reduce()` to create a `group()` function.

```

var toString = Object.prototype.toString;
var isFunction = function(o) { return toString.call(o) == '[object Function]'; };

function group(list, prop) {
  return list.reduce(function(grouped, item) {
    var key = isFunction(prop) ? prop.apply(this, [item]) : item[prop];
    grouped[key] = grouped[key] || [];
    grouped[key].push(item);
    return grouped;
  }, {});
}
// our right curried version of `group()`
var groupBy = rightCurry(group);

```



`group()` and `groupBy()` divide a list into sets, grouped by the property named `prop` on each object in the list. If `prop` is a function it will use the result of passing each value through `prop`. This is similar to the way `_.groupBy()` works in the lodash and underscore libraries.

Here's our previous example, now using `groupBy()`:

```
var getKey = function(item) { return item.age < 40 ? 'under40' : 'over40'; };
groupBy(getKey)(list);
// gives us the same results as previous example
```

In this case we passed a function as the `prop` argument that returned a string for grouping. But we can use the non-function use of `prop` for records like our JSON response, where we want to group by a given property on the record:

```
var list = [
  { value: 'A', tag: 'letter' },
  { value: 1, tag: 'number' },
  { value: 'B', tag: 'letter' },
  { value: 2, tag: 'number' },
];
groupBy('tag')(list);
// {
//   'letter': [
//     { value: 'A', tag: 'letter' },
//     { value: 'B', tag: 'letter' }
//   ],
//   'number': [
//     { value: 1, tag: 'number' },
//     { value: 2, tag: 'number' }
//   ]
// }
```

This looks like it should work well. However, the list of objects we were grouping above could only belong to one possible group: 'letter' or 'number'. The list of objects had a *many-to-one* relationship with the grouping

key.

In our JSON response, this isn't the case, as each post's `tag` property is an array of one or more tag names.

## Grouping Many-to-Many relationships?

So, did we go to all that trouble building a `groupBy()` function that won't work for our requirement?

Absolutely not! This is functional programming, after all, so we'll just use that function composed with other functions to build the one we want!

Let's take a step back and look at our JSON response again; but in a different light, like a typical database table:

JSON viewed as a table

id	title	author	tags	...
2	'Currying Things'	'Dave'	['functional programming']	...
3	'ES6 Promises'	'Kurt'	['es6', 'promises']	...
4	'Monads, Futures, Promises'	'Beth'	['promises', 'futures']	...
5	'Basic Destructuring in ES6'	'Angie'	['es6', 'destructuring']	...

We need a way to explode our list so that we output a list with one record for each tag-post combination.

That output list is very similar to a *linking* or *joining* table used in databases that have tables with a many-to-many relationship - *in this case tags to posts*.

Doing this will necessarily create duplicates on our output; but we need those since a post can show up in multiple groups given our requirements.

Our output list would resemble the following given our table example:



Given the above diagram, it's clear that what we are doing is outputting a combination. In this case, our output is the combination of the post record with each of its possible tags.

We know we'll need to map over our lists, so let's create a `map()` and `mapWith()` function we can use going forward:



```
// Returns a new list by applying the function `fn` to each item
// in `list`
function map(list, fn) {
  return list.map(fn);
}
var mapWith = rightCurry(map);
```

Now, let's create a `pair()` function that can combine the elements from two lists.

```
function isArray(o) { return toString.call(o) == '[object Array]'; }

function pair(list, listFn) {
  isArray(list) || (list = [list]);
  (isFunction(listFn) || isArray(listFn)) || (listFn = [listFn]);
  return mapWith(function(itemLeft){
    return mapWith(function(itemRight) {
      return [itemLeft, itemRight];
    })(isFunction(listFn) ? listFn.call(this, itemLeft) : listFn);
  })(list);
}
var pairWith = rightCurry(pair);
```

We basically take two lists, map over each item in the first list, and for each item, output the results of combining that item with each item in the second list using a nested map. We also allow a function that returns a list as the second parameter, which will be passed the item from the first list on each iteration.

Let's try this with our filtered records from the before. We'll use our right curried `getWith()` to pass a function as the second parameter that will return the array of tags on each post as the second set to combine against.

```
pair(filtered, getWith('tags'));
// [
//   [ [ /* ... */ ], 'functional programming' ] ],
```

```
//  [ [ { /* ... */ }, 'es6' ],
//    [ { /* ... */ }, 'promises' ]
//  ],
//  /* ... */
// ]
```

Interesting..., those are the right tag->post pairs, but they're nested in arrays within the array because we have nested `mapWith()` calls, each of which return an array.

We can, however, flatten that using another handy function called `flatten()`, which will flatten an array by one level, ie, transform `[[1,2],[3,4]]` into `[1,2,3,4]`.

```
function flatten(list) {
  return list.reduce(function(items, item) {
    return isArray(item) ? items.concat(item) : item;
  }, []);
}
```

We use `reduce()` here to build up the new array, concatenating any values that are arrays directly into the resulting array, removing nesting. Flattening our previous results gives us the following:

```
// [
//   [ { /* ... */ }, 'functional programming' ],
//   [ { /* ... */ }, 'es6' ],
//   [ { /* ... */ }, 'promises' ],
//   /* ... */
// ]
```

Now we have the data structure we want to start doing our grouping!

But, this operation is so common -- *the idea of flattening nested lists as we map across them* -- that it's usually combined into a single function, `flatMap(list, fn)`.

Let's create a `flatMap()` function and its right curried friend `flatMapWith()`.

`flatMapWith(fn)(list)`



```
function flatMap(list, fn) {
  return flatten(map(list, fn));
```

```

    }
var flatMapWith = rightCurry(flatMap);

```

We can use that inside our `pair()` function as well to ensure the right output:

`pair(listA, listB)`



```

function pair(list, listFn) {
  isArray(list) || (list = [list]);
  (isFunction(listFn) || isArray(listFn)) || (listFn = [listFn]);
  return flatMapWith(function(itemLeft){
    return mapWith(function(itemRight) {
      return [itemLeft, itemRight];
    })(isFunction(listFn) ? listFn.call(this, itemLeft) : listFn);
  })(list);
}

```

Now we can put our entire flow together, which consists of:

1. creating a many-to-many list of tag->post pairs using `pairWith()`
2. using that new list as input to `groupBy()` to group each record by its given tag (*the second item in each pair*).

```

var bytags = pairWith(getWith('tags'))(records); // #1
var groupedtags = groupBy(getWith(1), bytags); // #2
// {
//   'destructuring': [
//     [ { /* ... */ }, 'destructuring' ],
//     [ { /* ... */ }, 'destructuring' ]
//   ],
//   'es6': [
//     [ { /* ... */ }, 'es6' ],
//     [ { /* ... */ }, 'es6' ]
//   ],
//   /* ... */
// }

```

## Cleaning Up

So, after rebuilding the list as a many-to-many joined list and then grouping by the tags as the key, we end up with a structure that still isn't quite what we want - each post record is still nested in a list pair with its group key.

We need a way to map over our output object's properties and then map over each of those arrays and replace each list pair with just the post record.

We can map over arrays already using `map()` and its variants; but we can also do the same thing with objects if we think of the objects as a list where each item is a property and its value.

We'll call this `mapObject()` and it will return an object as well.



```
function mapObject(obj, fn) {
  return keys(obj).reduce(function(res, key) {
    res[key] = fn.apply(this, [key, obj[key]]);
    return res;
  }, {});
}

// A right curried version
var mapObjectWith = rightCurry(mapObject);
```

The function passed to `mapObject()` is passed not only the item but also the property name. Now, we can use our ability to map over an object to convert our structure:

```
// Remove our extraneous group key and pair, replacing with the post record
var finalgroups = mapObjectWith(function(group, set){
  return mapWith(getWith(0))(set);
})(groupedtags);

// {
//   'destructuring': [
//     { id: 2, title: 'ES6 Promises', ..., tags: ['es6', 'promises'] },
//     { id: 4, title: 'Basic Destructuring in ES6', ..., tags: ['es6', 'destructuring'] },
//   ],
//   'es6': [ /*...*/ ],
//   /*...*/
// }
```

## Being More Declarative

The operation we used above in which we want to pull out a specific property's value from a list of objects, `mapWith(getWith(prop))`, is a fairly common action. So much so, that this is commonly named `pluck()`, and you'll find it in numerous functional libraries.

```
// For each object in `list`, return the value of `prop`
function pluck(list, prop) {
  return mapWith(getWith(prop))(list);
}

// right curried version of `pluck`
var pluckWith = rightCurry(pluck);
```

That's a bit more declarative and gives us another higher order function we can reuse. But we'd like our resulting code to be a bit more descriptive of the actions it is actually performing - getting the post record from each nested pair.

Let's start by being explicit with the function we're passing to `mapObjectWith()`:

```
function getPostRecords(prop, pair) {
  return pluckWith(0)(pair);
}
```

Ah, that's a bit more descriptive. And when combined with our original solution, becomes much more declarative of the action we're actually performing.

```
var finalgroups = mapObjectWith(getPostRecords)(groupedtags);
```

## The full implementation

The final implementation for meeting our second requirement:

```
// Step 1: Build our many-to-many list
var bytags = pairWith(getWith('tags'))(records);

// Step 2: group by the tags (pair[1]):
var groupedtags = groupBy(getWith(1), bytags);

// Step 3: strip extra key in nested pairs:
function getPostRecords(prop, value) {
  return pluckWith(0)(value);
```

```
}
```

```
var finalgroups = mapObjectWith(getPostRecords)(groupedtags);
```

## Summary

In this post we've added a number of utility functions to our library. We also took a circuitous route through transforming our initial data due to its many-to-many relationship between posts and tags. We were then able to output a list of posts for each tag.

We also looked at a handful of common functional programming and combinative idioms like `pluck`, `map` and `mapObject`. Be sure and look over [the gist](#) of the full source for this second part of our blog series.

In the next and final blog post, we'll find out why we keep making right curried versions of all our functions as we discuss composition; and we'll finish our final requirement which is sorting each group of posts.

- [javascript](#)
- [functional programming](#)

# 17 : Understanding Prototypes, Delegation & Composition

<http://www.datchley.name/understanding-prototypes-delegation-composition/>

In the [last post](#) we covered the basics of creating and using Javascript objects, which included using `new` to invoke functions as constructors. In this post, we're going to focus on the following:

- how Javascript's prototype based objects work
- building up functionality through *pseudo-classical inheritance* via constructor functions and the prototype chain
- and a simpler, more idiomatic way to build up functionality using composition, delegation and mixins.

## What is this 'prototype' thing?

When most developers think of Object Oriented (OO) programming, they are usually coming from languages like Java and C++ where the concept of a *class* represents a kind of *blueprint* for a particular type of object's behavior. In those languages, a class is a separate, (*typically*) static representation of how an object should be created; and this is much different than an *instance* of an object created from that class.

Despite what some people might tell you, Javascript does not have classes. What some developers refer to as a "class", is actually just a function, with other functions assigned to its `.prototype` property; which, in turn, is called with `new` in order to treat the function like a constructor.

```
// Not a 'class', just a function
function Person(name) {
  this.name = name;
}

Person.prototype.hello = function() {
  console.log("Hello, my name is " + this.name);
};

// invoke our function with `new` for magical constructor behavior
var dave = new Person("Dave");
dave.hello();
// "Hello, my name is Dave"
```

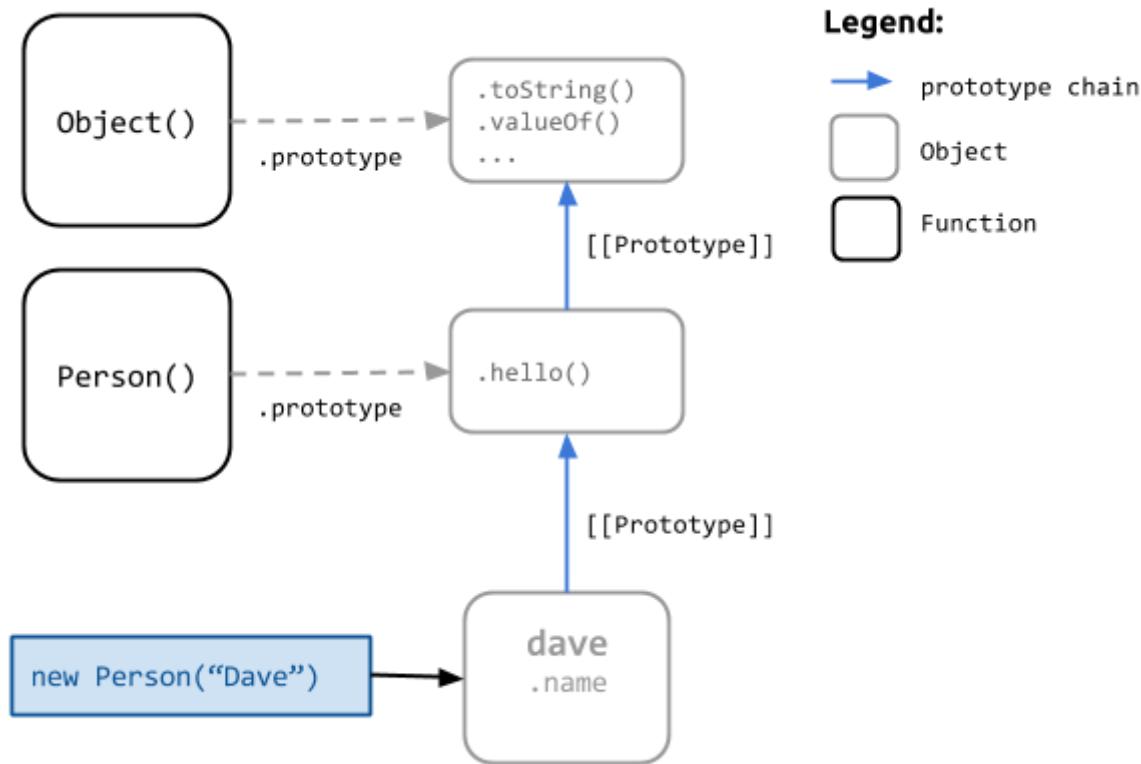
The `.prototype` property of `Person` above is an actual property on the `Person` function (*functions are objects*). This property is treated as an object and used to create the actual prototype to which our new object is linked.

**Note:** In some browsers you can access an objects prototype using `__proto__` property - typically referred to using the `[[Prototype]]` symbol.

The actual prototype of an object "*is just another object*" to which it is linked

Using the `.prototype` property of a function we can assign new properties so that when a new object is created from that constructor function, those properties are available on that object's actual prototype.

To wit, the above code snippet yields the following relationships:



These prototypes in Javascript are really just a way to compose functionality. They allow an object to *delegate* behavior to its *prototype*, which is simply just another object that it links to via an internal, single path chain.

This resembles "inheritance" in the sense that you can call a function on your object that isn't directly defined on that object, so long as it is defined on an object that your function links to in the prototype chain.

You may have heard this referred to as *prototypal inheritance*; but, in reality, this is simply Objects Linking to Other Objects (OLOO), as [Kyle Simpson](#) correctly termed it. The actual work is being accomplished via:

- **composition** - your object contains another object, called its prototype, and
- **delegation** - when you access a property on your object, if it's not directly defined on it, javascript will search the prototype chain and delegate that behavior to the first object it finds that defines that property.

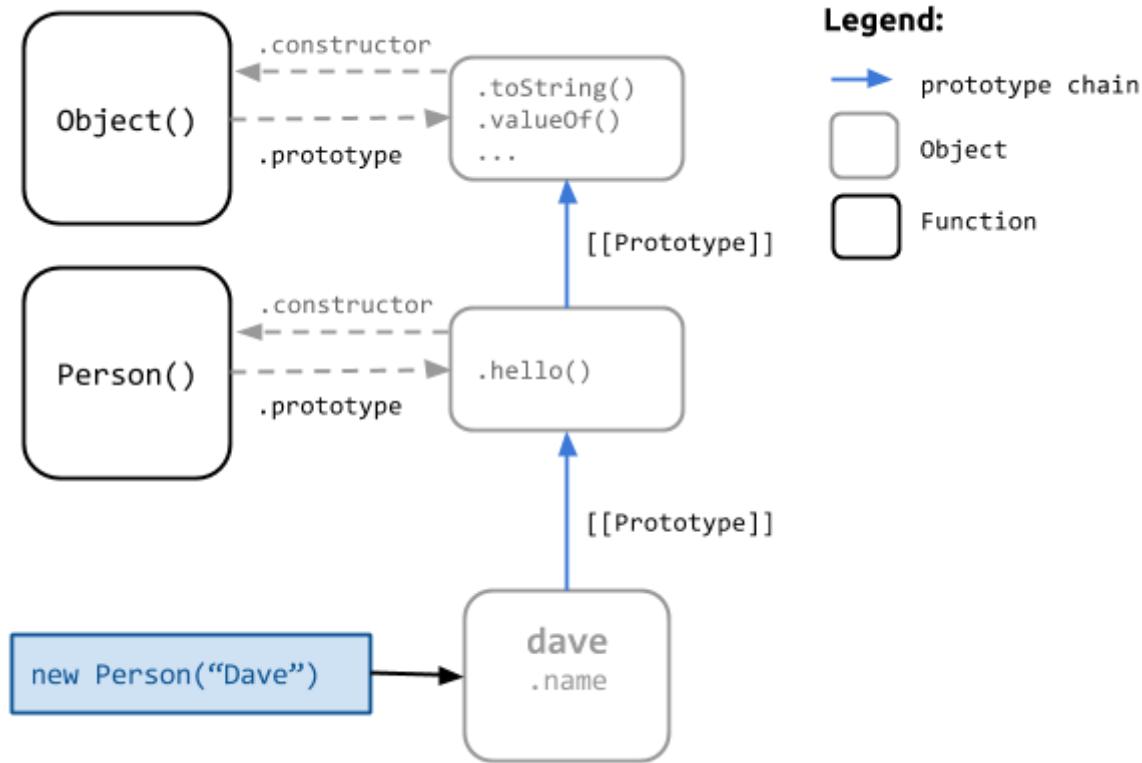
### .constructor

Just to make OO concepts even muddier in Javascript, we also have the `.constructor` property. This property will reference a function used to create one of the objects in a given object's prototype chain.

When you call a function with `new` the `.constructor` property is set for you on the returned object's prototype.

In the case of our example above, we can inspect the `.constructor` property and see that it holds a reference to the `Person()` function, which we used to construct the object.

```
dave.__proto__.constructor; // function Person()
dave.__proto__.__proto__.constructor; // function Object()
```



However, this direct relationship doesn't always hold true; especially when we start trying to implement *inheritance* using `new` and assigning prototypes.

```
function Foo(){}

function Bar(){
    // Call our parent constructor on our object
    Foo.call(this);

}

// "inherit" from Foo's prototype
Bar.prototype = Object.create(Foo.prototype);

var bar = new Bar();
```

```
console.log(bar.constructor);
// function Foo() !WOOPS, that's not our constructor
```

By re-assigning the prototype, we end up with the `.constructor` property of the prototype object referencing `Foo()`, not the constructor of our `bar` object, which we expected to be `Bar()`. To fix that, we have to manually reassign the constructor right after setting its prototype.

```
Bar.prototype = Object.create(Foo.prototype);
Bar.prototype.constructor = Bar;
```

## instanceof

When using `new` for constructor functions, Javascript allows us to use `instanceof` to check if an object is an instance of a particular constructor function. More precisely, `instanceof` checks if an object's prototype chain contains an instance of a given constructor's prototype.

For example:

```
function A(){}
function B(){}
B.prototype = Object.create(A.prototype);

var obj = new B();
console.log(obj instanceof B); // true
console.log(obj instanceof A); // true
console.log(obj instanceof Object); // true
console.log("%o", obj.constructor.prototype); // A {}
```

However, `instanceof` can be problematic in some circumstances and return incorrect results:

- the prototype of the object in question has changed
- you are comparing across frame/window execution contexts (*object from one frame, say an Array, checked in another frame will return false when doing obj instanceof Array*)
- you don't have a constructor function to use to introspect the object (*say, a third party library, module or closure*)

For instance:

```
// Related objects whose constructors are out of scope
var a = (function(){
    function A(){}
    return A;
}());
```

```

        return new A();
    })();

var b = (function(proto) {
    function B(){}
    B.prototype = proto;
    return new B();
})(a);

// Can we determine if b is related to a?
function F(){}
F.prototype = a;
console.log("Related? ", b instanceof F);
// Related? true

```

In this case, we have two objects whose constructor functions weren't available for comparison using `instanceof`; so we create a throw-away function and assign the base object `a` to its `.prototype` and then use `instanceof` to determine if `b` is an instance of the function `F()`. Rather, if the prototype of `F()` is anywhere in the prototype chain of `b`.

Clearly the object `b` was not constructed by the function `F`, but we still get a false positive because of how `instanceof` tests for ancestry using the constructor's prototype.

`obj instanceof F` is semantically misleading. It does not imply that `obj` was created by `F()` or is even remotely related to `F()`; only that the prototype `F` will use when invoked with `new` is somewhere in the prototype chain of `obj`.

## Implementing inheritance in Javascript

So far we've seen that Javascript provides us with a number of "features" that make it *seem* as if it supports thinking about and implementing objects using classical inheritance. But, we've also seen the caveats that surround those features like `instanceof`, `new` to invoke functions as constructors and the `.constructor` and `.prototype` properties.

To implement classical inheritance, we need to understand the details of how it's implemented behind the scenes to fully appreciate that Javascript isn't really using "classes" or inheritance at all.

Let's give it a go with a simple, albeit contrived, example of inheritance with a `Dog` which inherits from `Animal`.

```

// Our base "class" Animal
function Animal(name) {
    this.me = name;
}

```

```
// Some base "class" methods
Animal.prototype.who = function() {
    return "I am " + this.me;
};

Animal.prototype.speak = function() {
    // What should this do? I dunno...
    console.log("(Animal) speak!");
}

// A child "class" that "inherits" from Animal
function Dog(name) {
    Animal.call(this, name);
}

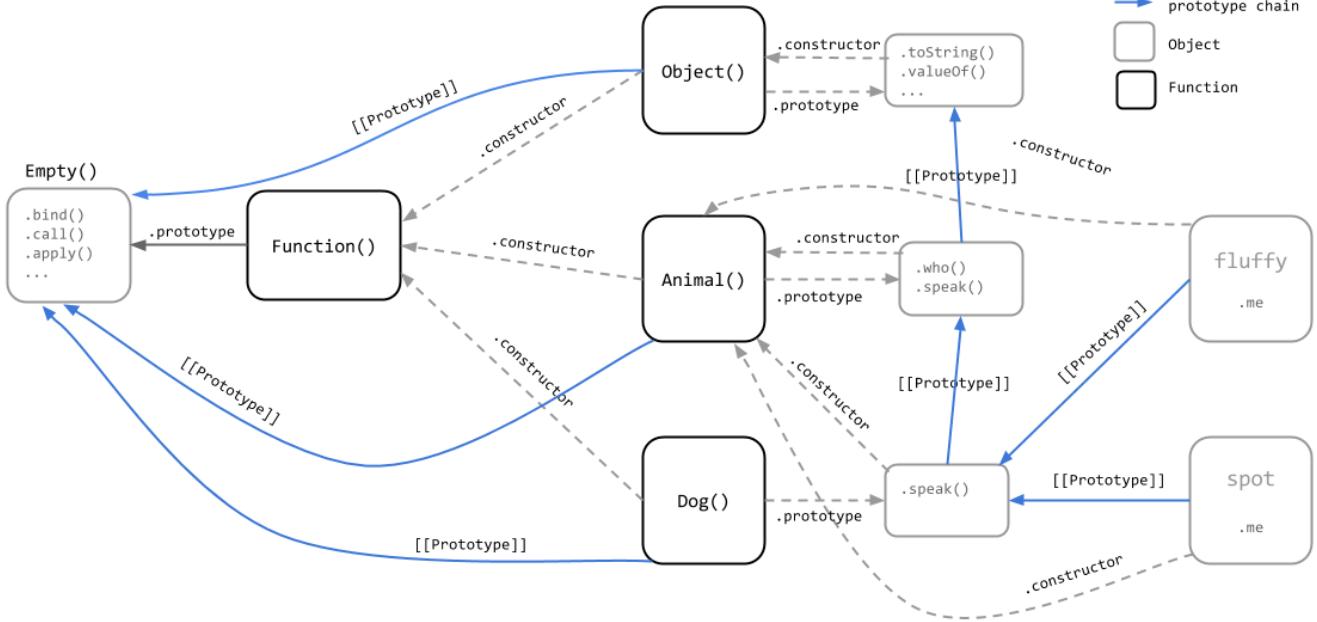
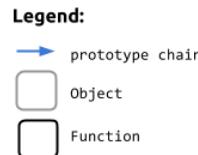
// Ensure we properly inherit from our base class
Dog.prototype = Object.create( Animal.prototype );
Dog.prototype.constructor = Dog;

// Add our own 'speak' method and call our base class 'speak' as well
Dog.prototype.speak = function() {
    Animal.prototype.speak.call(this);
    console.log("Hello, " + this.who() + ".");
};

// Puppies! Awwwww
var fluffy = new Dog( "Fluffy" );
var spot = new Dog( "Spot" );
```

So, yeah, we now have a couple of talking dogs. Nothing to see here, move along...

Get a cup of coffee (*or your favorite beverage*) and sit down for second. Take some deep breaths; because as straight forward (*not so much*) as the above code looks, the following diagram shows what's actually happening under the hood:



Seems like we're going to a lot of effort to treat Javascript like it has classes by

- using `new` to invoke functions as constructors
- mimicking methods by assigning them to the function's `.prototype` property
- ensuring that we create the proper relationships by setting the correct prototype for child classes and ensuring we call our parent class's constructor in the right context
- and re-wiring the `.constructor` property of our `.prototype` to ensure it points to our actual constructor function

In fact, this all boils down to jumping through hoops to basically ensure our objects properly compose behavior via creating the right linkage to another object via composition in the prototype chain.

I hear you asking, "*There must be a better, more idiomatic, way to do this in Javascript?*"

Yes, there is.

## Delegation using `Object.create()`

We used `Object.create()` above when creating the function prototypes. `Object.create()` does exactly what we really want - creating a linkage between objects by building an object based on another object.

So, maybe we can just use "plain objects", instead of constructor functions?

```

var foo = { a: 1 };

var bar = Object.create(foo);
bar.b = 2;

var baz = Object.create(bar);
  
```

```

baz.c = 3;

console.log("I can haz 'a' and 'b'? ", !(baz.a && baz.b));
// true

```

`Object.create()` properly sets up the prototype linkage and gives us a new object; which we can then link to another object. We don't have to worry about `instanceof` because we're not using functions as constructors; nor do we have to deal with setting a `.prototype` or `.constructor` property correctly.

So let's go back to our `Animal` and `Dog` example and see how we would compose that same functionality using delegation through `Object.create()` instead of constructors.

We'll use `Object.assign()` to initialize the objects in the our chain. Most browsers don't support `Object.assign()` yet, so you can use this simple polyfill to run the examples:

```

if (!Object.prototype.assign) {
    Object.prototype.assign = function() {
        var args = [].slice.call(arguments),
            target = args.shift();

        return args.reduce(function(base, obj) {
            Object.keys(obj).forEach(function(prop) {
                if (obj.hasOwnProperty(prop)) {
                    base[prop] = obj[prop];
                }
            });
            return base;
        }, target);
    }
}

```

The functionality we want to compose using delegation will be similar to our original; but delegation is a fundamentally different approach and pattern than inheritance.

- Delegation builds functionality by composing (*via links*) regular objects, not by using constructor functions.
- Delegation uses more specific method names on the delegator objects that are reflective of the actions they perform. Inheritance keeps method names fairly generic, as sub-classes tend to re-implement specific behavior that override the base classes method.
- State is typically maintained at the delegator level, not in the delegatee objects.

```

var Animal = {
    who: function() { return this.name; },
    speak: function(s) { console.log(this.who() + ": " + s); }
}

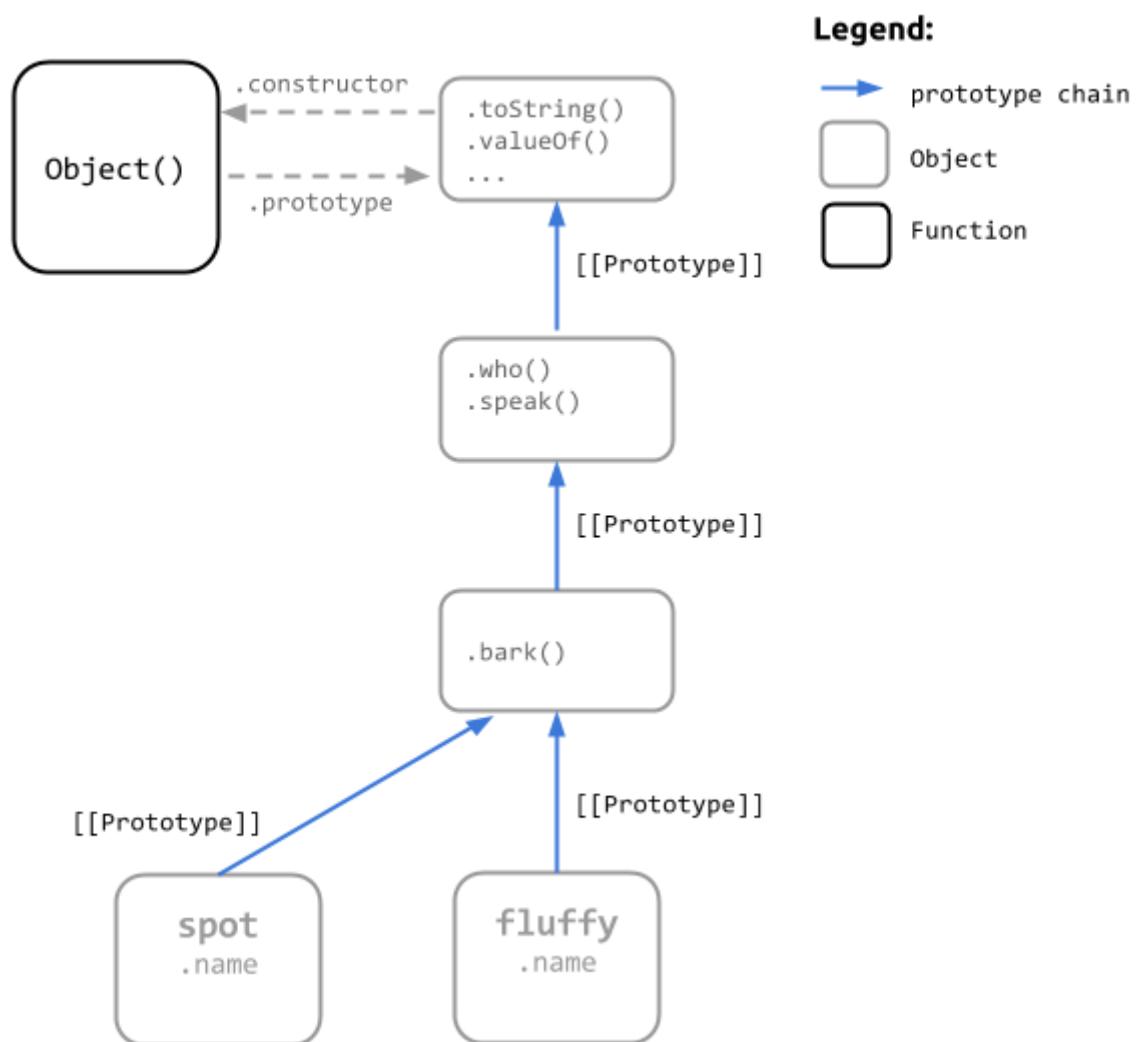
var Dog = Object.create(Animal, {
    bark: {
        value: function() { this.speak("woof!"); }
    }
});

var spot = Object.assign(Object.create(Dog), { name: "Spot" });
var fluffy = Object.assign(Object.create(Dog), { name: "Fluffy" });

spot.bark();
fluffy.bark();

```

That's much less code; and here's the new mental model you'll have to grok. Much easier!



We're composing functionality here using `Object.create()`. And the `bark()` method of our `Dog` objects delegates to `Animal.speak()`, available via its prototype chain.

The approach with the delegation pattern is to use an object as a base "type" that contains common behavior and have other objects link to that object to use that functionality.

In many respects, what we're trying to do is simply *compose behavior*, not create types, like inheritance does. The relationship between two objects is explicit via composition using `Object.create()`, rather than implicit via an internal inheritance mechanism.

So, if we're wanting to really just compose behavior; and those behaviors might be common to a number of different objects, there is actually another pattern we can use: **mixins**.

## Composing Behavior through Mixins

The concept of *mixins* is fairly wide spread and found in a number of languages. The idea is to factor out common, reusable behavior into their own objects and then integrate those objects into more specific objects that need to use them.

```
// A base object so we can create people
var Person = {
  who: function(){ return this.name; },
  init: function(name) {
    this.name = name;
  }
};

// Factor out common functionality into their own
// objects
var canSpeak = {
  speak: function(s) { console.log(this.who() + ": " + s); }
};

var canWalk = {
  walk: function() { console.log(this.who() + " is walking..."); }
};

var canBuild = {
  tools: ['hammer', 'pliers'],
  use: function(tool) { this.tools.push(tool); },
  build: function(thing) {
    var withTool = parseInt(Math.floor(Math.random() * this.tools.length));
    console.log(this.who() + " is building a " + thing + " using " +
this.tools[withTool]);
  }
};
```

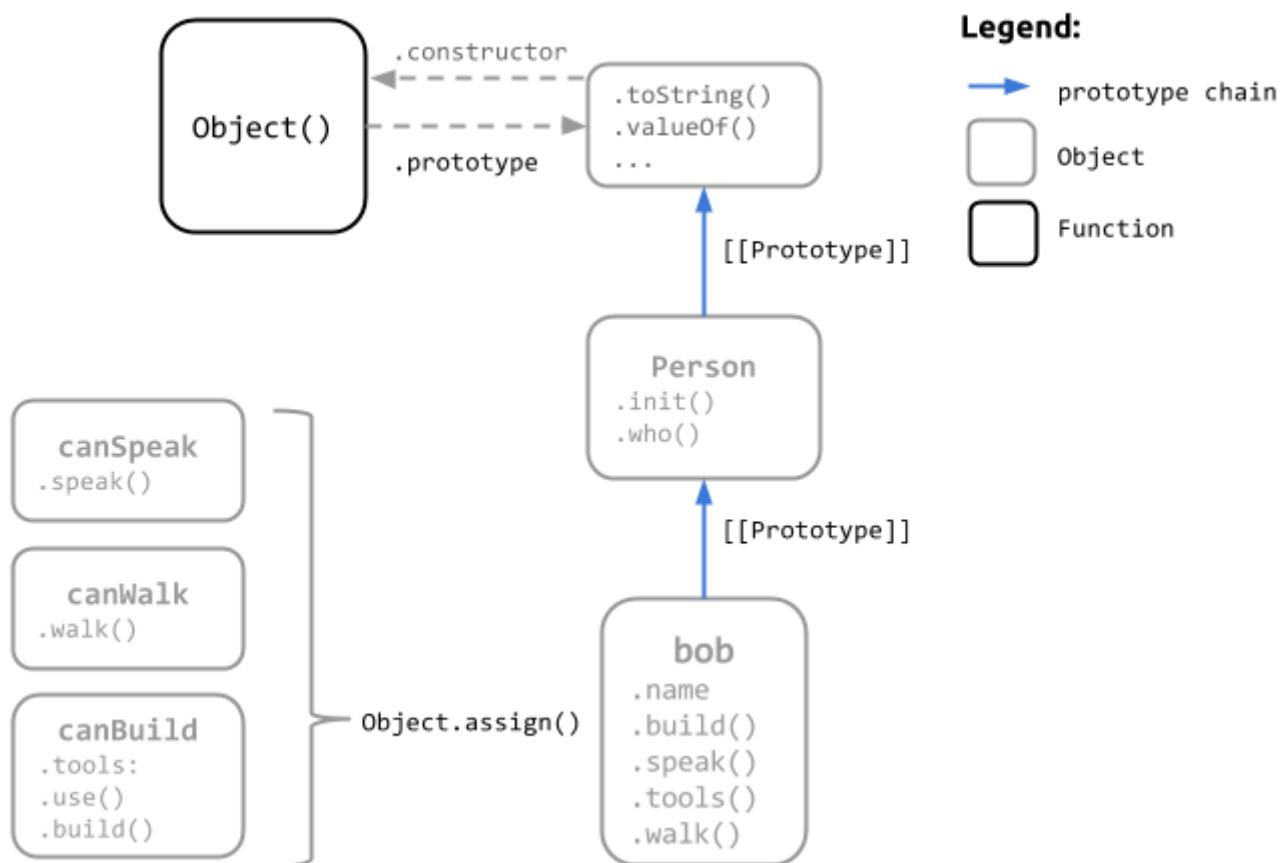
```
};
```

```
// Can we build it?...
var bob = Object.assign(Object.create(Person), canSpeak, canWalk, canBuild);
bob.init("Bob the Builder");
bob.speak("Hi there!");
bob.walk();
bob.use("stapler");
bob.build("web site");
```

This pattern improves things by

- further simplifying the mental model used to create objects,
- improves the readability of the code,
- makes it very easy to isolate functionality in specific objects for better reuse. (*DRY, Separation of Concerns*)

Here's the inside view of using mixins.



What an object can do is explicitly stated through the use of mixins. Plus, we can add as many behaviors/mixins as make sense to an object.

We're still using delegation here, via the prototype chain; but mixins give us the ability to inherit from objects in a different way. Mixins represent *concatenative inheritance*, meaning that instead of delegating behavior

to another object, we copy that behavior directly. You can see this in the above diagram, as all the mixins' properties now exist on our new object.

## Summary

There are many libraries available that wrap the implementation of *inheritance* making it easier to program in Javascript if you come from a strict OO background or "class" centric language. But using inheritance and trying to maintain that mental model in Javascript is complex and hard to manage due to various problems.

With the end goal of simply being able to compose functionality, we found that the more idiomatic way to do that in Javascript is through simple objects, the prototype chain, and using new patterns of thinking in our design like *delegation* and *mixins*.

Hopefully this post has been enlightening for at least some of you; as this is a topic that is amply covered in books and online by various people - each with their own take on the subject.

---

MDN can give you a good reference on [Object.create](#) and [Object.assign](#) if you're curious about the details.

Also, be sure and check out Kyle Simpson's [You Don't Know JS: this & Object Prototypes](#) and Eric Elliott's [Common Misconceptions about Inheritance in Javascript](#) for more on this subject and a much deeper insight.

## 18 : Some jQuery Functions And Their JavaScript Equivalents

<http://callmenick.com/post/jquery-functions-javascript-equivalents>

In light of my recent forays into the JavaScript DOM world, I'd decided to do a little research and write a snippet collection based on jQuery functions and their JavaScript equivalents. I think jQuery is a great tool, and it has done great things for me and many others as developers. But I'm a firm believer that nowadays, JavaScript is an indispensable tool to keep in your arsenal. Manipulating the DOM with JavaScript DOM methods is fun and easy, and the majority of times, plain old vanilla JS is all we need. It seems scary at first, but let's take a look at some core functions/methods that we're accustomed to, how they work, and how to execute them with just plain old JavaScript.

### Accessing Elements

There are many ways to access elements with JavaScript and jQuery. Let's look at a few important ones here.

#### By ID

In this above scenario, we're accessing by ID, so we'd expect that only one element is returned. With jQuery, we tend to access single elements by ID like this:

```
var el = $("#element");
```

With regular JS, we can achieve this two ways:

```
// traditional JS  
var el = document.getElementById("element");  
  
// new school JS  
var el = document.querySelector("#element");
```

From here on, we can use these variables and apply our manipulations whenever we want. We've successfully assigned an element to a variable, and can now use it wherever we want.

#### By Class & Tag

In jQuery, we also see that we can access all elements that have a certain class or tag. This is useful when we want to iterate over all of these elements and perform functions on them. We'd access them like this:

```
// access by tag name  
var els = $("p");
```

```
// access by class
var els = $(".class");
```

Here's the JavaScript equivalents:

```
// by tag name
var els = document.getElementsByTagName("p");           // OR
var els = document.querySelectorAll("p");

// by class name
var els = document.getElementsByClassName("class");    // OR
var els = document.querySelectorAll(".class");
```

## Useful Tips

The methods `getElementsByName`, `getElementsByClassName`, `querySelector`, and `querySelectorAll` can be applied to DOM elements to return collections of elements or node lists that are children of an element in question. We can, for example, do this:

```
var wrapper = document.getElementById("wrapper"),
    els = wrapper.querySelectorAll("p");
```

A jQuery equivalent would be this:

```
var els = $("#wrapper p");
```

## Caveats

Here are some things to note:

1. Using `querySelector` on an element that exists more than once (e.g.: `querySelector("p")` will return the first matching element).
2. Technically, `querySelectorAll` returns a non-live node list of elements. This is in contrast to the `HTMLCollection` returned by `getElementsByName` and `getElementsByClassName`. Be mindful of that!

## Getting & Setting the HTML of an Element

It's fairly common practice in simple interfaces that after a user commits to a call to action, the HTML of an element is replaced. This can be to display a success or error message after a form submission, for

example. Sometimes, we might want to fetch the HTML contents of an element and throw it into another element in our DOM. With jQuery, this is made possible by the simple `.html()` function. Here it is in action:

```
// getting the HTML
var content = $("#content").html();

// setting the HTML
$("#box").html(content);
```

With JavaScript, we're presented with the function `innerHTML` that serves the same purpose. Here's a replication of the above, but with pure JavaScript:

```
// getting the HTML
var content = document.getElementById("content").innerHTML;

// setting the HTML
document.getElementById("box").innerHTML = content;
```

Be mindful when using `innerHTML` though. According to the MDN:

If a `<div>`, `<span>`, or `<noembed>` node has a child text node that includes the characters (&), (<), or (>), `innerHTML` returns these characters as `&amp`, `&lt` and `&gt` respectively. Use `Node.textContent` to get a correct copy of these text nodes' contents.

## Attaching Event Handler Functions to Elements

A very important part of JavaScript programming is attaching event handler functions, or "listening" for events on elements in the DOM. These events range from click and hover, to blur. jQuery offers us a lot of direct attachment of event handlers, for example `$("#el").click();`, but we're going to look at a jQuery function that resembles pure JavaScript, and is very useful - the `.on()` function. Here it is in action, listening for a click on an element:

```
// listening for a click on #el
$("#el").on("click", function(){
    console.log("element clicked");
});
```

The above should output "element clicked" to the console. With JavaScript, the implementation is friendly and straightforward. In fact, the name itself is very self explanatory. Here's the implementation:

```
// listening for a click on #el
document.querySelector("#el").addEventListener("click", function(){
    console.log("element clicked");
});
```

## Multiple Event Handlers with jQuery

Here's something cool to note though. With jQuery, we can add multiple event handlers on the same element by comma-separating them before the function is called. Here's an example:

```
// listening for a click and hover on #el
$("#el").on("click", "hover", function(){
    console.log("click or hover");
});
```

## Appending and Prepending Content

Often, we want to append or prepend some additional information or message to an element after user interaction. jQuery provides us two neat functions to take care of this - `.append()` and `.prepend()`. The names give away their functionality, so let's look at some implementation:

```
// append "success" message to a log
$("#log").append("<p>Success!</p>");

// prepend "error" message to a log
$("#log").prepend("<p>Error...go back and fix.</p>");
```

For appending content, JavaScript offers us a self explanatory function too - `Node.appendChild`. In this case though, we are dealing with nodes, so we can't directly insert content. Instead, we must insert a node that contains content. Let's take a look at this function in action:

```
// create an element, give it some HTML, append it to #log
var p = document.createElement("p");
p.innerHTML = "Success!";
document.getElementById("log").appendChild(p);
```

Prepending isn't as straightforward, as there is no `Node.prependChild` function in JavaScript. However, using a combination of `insertBefore` and `firstChild`, we can insert our node before the first child of the element in question, i.e. prepending our node to `#log`. Let's take a look at this in action:

```
// create an element and get our #log div
var p = document.createElement("p"),
    log = document.getElementById("log");

// add some HTML to p
p.innerHTML = "Error...go back and fix./";

// prepend p to #log
log.insertBefore(p, log.firstChild);
```

## Altering the CSS of an Element

Users tend to feel more welcome and safe when they feel this sense of familiarity. It's not uncommon then to see different colours or styles of things changing based on user interaction. Form failed to submit? Change the background colour to red. Successful submission? Change it to green. Want to hide the form after successful submission? Set the property to `display: none`. These are attributes of DOM elements that can all be controlled via CSS. jQuery conveniently gives us the `.css()` function to tap into an elements CSS. Let's take a look how:

```
// this will output the element's background colour
var col = $("#el").css("background-color");
console.log(col);

// this will change the element's background colour to green
$("#el").css("background-color", "green");

// alternate syntax for passing array into .css
$("#el").css({
  "background-color": "green",
  "color": "white"
});
```

On the vanilla JS side of things, we're presented with the function `HTMLElement.style`. This allows us to set the style of an element. Here it is in action:

```
// change the colour of #el to green
document.getElementById("el").style.color = "green";
```

To actually get the style attribute of an element is a bit different though. According to the MDN:

The `style` property is not useful for learning about the element's style in general, since it represents only the CSS declarations set in the element's inline `style` attribute, not those that come from style rules elsewhere, such as style rules in the `<head>` section, or external style sheets. To get the values of all CSS properties for an element you should use `window.getComputedStyle()` instead.

In jQuery, a similar function goes down when getting the style. Here's a look at

`window.getComputedStyle()` in action:

```
// this will return the computed colour of "el"
var el = document.getElementById("el");
var elStyle = window.getComputedStyle(el, null);
console.log(elStyle.color);
```

## Iterating Over a Collection of Elements

The last comparison we'll go through is of significant importance as it entails looping. Looping is a very important part of any programming language, and JavaScript is no different. Often times, we need to iterate over a collection of elements and perform some functionality to each of those elements. This is what loops are for.

Let's target all elements in our DOM with a class of `.el`, loop over them, and change the colour to blue. With jQuery, the `.each()` function gives us a nice starting point. Combining what we've learnt above, here's the implementation:

```
$(".el").each(function(i, el){
    /**
     * i represents the integer value of where we are in the loop
     * el represents the element in question in the current loop
     * $(this) also represents the element in question in the current loop
     */
    // change the element colour to blue.
    $(this).css({
        "colour": "blue"
    });
});
```

With JavaScript, we can use a simple `for` loop to take care of this. The thought process behind it is the same - get a collection of HTML elements, loop over them, and edit them. Here's how:

```
// get all elements with class .el
var els = document.querySelectorAll(".el");

// loop over them while i less than number of elements
for (var i = 0, len = els.length; i < len; i++) {
    els[i].style.backgroundColor = "blue";
}
```

## Bonus

Here's some swanky alternative JavaScript syntax that resembles the `.each()` function a bit more. It's a `forEach` function, and we first create an empty array and then add each of the nodes to it. After that, we iterate using a `forEach` statement. Here it is in action:

```
[].slice.call(document.querySelectorAll(".el")).forEach(function(el,i){
    /**
     * i represents the integer value of where we are in the loop
     * el represents the element in question in the current loop
     */
    // change the element colour to blue.
    el.style.backgroundColor = "blue";
});
```

## Wrap Up

And that's a wrap! We've looked at some common and important jQuery/JavaScript comparisons, and seen that JavaScript isn't that scary after all. I'm not taking anything away from jQuery, I think it's amazing. But if you're using jQuery because you're scared to dive in to vanilla JS, then you shouldn't be. In 2014, JavaScript is a very important language and can take you far. I hope this collection of snippets helped clear the air for you a bit and got your feet a little wet with JS programming. If you have any comments, feedback, suggestions, or questions, feel free to leave them below.

## 19 : Is everything in JavaScript an Object?

<https://blog.simpleblend.net/is-everything-in-javascript-an-object/>

I have always found this statement confusing: “Everything in JavaScript is an Object”. What did they mean by this? How can a Function or an Array at the same time be an Object? Before we tackle this question, we need to understand how the different Data Types are categorized.

In JavaScript, there are two Data Types: **Primitives** and **Objects**. (Object Types are also sometimes referred to as Reference Types). 1

### Primitive    Object

Primitive	Object
Number	Function
String	Object
Boolean	Array
Symbol	
null	
undefined	

Based on this categorizing, the simple answer is no, not everything in JavaScript is an Object. Only values that belong to that Type are Objects. Another way of looking at it is, any Type that isn't a Primitive Type is an Object Type. But what is it that *differentiates* Primitives from Objects? And more importantly, what do people *really mean* when they say “everything” or “almost everything” is an Object”? There are two main distinctions: mutability, and comparison.

### Mutability

From my experience, what people *really mean* when they talk about values being “object like” is their mutability. More specifically, they’re talking about the ability to add and remove properties. For example, because Functions and Arrays belong to the Object Type you can add properties to them just like you would an object literal.

```
var func = function() {};
func.firstName = "Andrew";
func.firstName; // "Andrew"
```

```
var arry = [];
arry.age = 26;
arry.age; // 26
```

This opens the door to all sorts of fascinating use cases, and is really the key to understanding how Prototypes and Constructors work.

However because Primitive Types are immutable, we're unable to assign properties to them. 2 The parser will immediately discard them when attempting to read their value.

```
var me = "Andrew";
me.lastname = "Robbins";
me.lastname; // undefined

var num = 10;
num.prop = 11;
num.prop; // undefined
```

At this point, I think it would be useful to examine things at a more fundamental level. With regards to primitives, what does it really mean to say that their values cannot be changed? Consider the following code:

```
1 = 2; // ReferenceError
```

This might seem like a silly example, but I think it shines a much needed light on exactly what we're talking about when we talk about mutability. When you type the number 1 into the JavaScript console, the compiler assigns that piece of data to the Primitive data type. Therefore when you attempt to change the number 1 to the number 2, it fails and probably has a heartattack.

## Comparison and passing around

Besides mutability, another important distinction between Primitive Types and Object Types is the way they're compared and passed around within the program. Primitive Types are compared by value, while Object Types are compared by reference. What does this mean? Let's look at Primitives first. Consider the following code:

```
"a" === "a"; // true
```

This is true because the value "a" is equal to "a". Simple. However what happens when we introduce variables into the picture? Nothing has changed except that we're now storing our Primitive Types into variables.

```
var a = "a",
b = "a";

a === b; // true
```

Since Primitive Types are compared by value, the result will true. The value of the variable a is exactly equal to the value of the variable b. In other words, "a" equals "a". Aristotle would be proud. However look at this. If

we apply the same example to an Object Type, we get the opposite result.

```
var a = {name: "andrew"},  
        b = {name: "andrew"};  
  
a === b; // false
```

Why is this? Object Types must reference the same object in order for its comparison to be true. In the example above, we're creating a *new* object for the variable b. As David Flanagan puts it:

“...we say that objects are compared by reference: two object values are the same if and only if they refer to the same underlying object.” 3

Now, what happens when we pass these values around?

```
var a = {name: "andrew"},  
        b = a;  
  
b.name = "robbins";  
  
a === b; // true
```

This might seem strange at first, but look closer at what's happening. Because objects are part of the Object type, its values are compared and passed by reference. Reference to what? Reference to the same underlying object. In the above example, we're setting b equal to a. We didn't create a *new* object. We're simply creating a *reference* to another object. A different way of looking at it is that we're pointing the variable b to a. Therefore when we mutate the “name” property on b, we're at the same time mutating the “name” property on a.

Back to Primitives, how would the same example apply to them?

```
var a = "Andrew",  
        b = a;  
  
b = "Robbins";  
  
a === b; // false
```

Remembering that Primitives are compared and passed by value, when we set b equal to a, we're actually creating a new copy of a. Therefore when we change the value of b and then compare it to a, the value is not the same anymore.

## Wrapper Objects

Some of you may be wondering, “Ok, if Primitives aren’t Objects then why can I call methods on them?” The answer is Wrapper Objects.

When you attempt to call methods on a Primitive, JavaScript does a magic trick behind the scenes. It takes your Primitive value and converts it to a *temporary* Object using a constructor function.<sup>4</sup> The decision of which constructor function to use will depend on the Primitive value you’re attempting to change. For example, calling `.length` on a string will use the built in `String()` constructor to *temporarily* change the Primitive to an Object—allowing you to use the `length` method to mutate it. This temporary Object is called a Wrapper Object.<sup>5</sup>

Interestingly enough, the two Primitive values `null` and `undefined` do not behave this way. Trying to call methods on these values will result in a `TypeError`.

We can use the `typeof` keyword to show the difference.

```
typeof "s"; // "string"  
typeof new String(s); // "object"
```

As a sidenote, it’s useful to know that there’s a well-documented bug<sup>6</sup> in JS compilers which returns “object” when executing `typeof null`.

```
typeof null // "object"
```

Considering that JavaScript was written in 10 days<sup>7</sup>, I’m not going to lose any sleep over it =)

It’s also useful to know that properties on Primitives are read-only, and temporary.

```
var hello = "hello";  
hello.slice(1); // "ello" (Here we're actually calling slice not on hello, but of a  
copy of hello)  
hello; // "hello"
```

## Summary

JavaScript values can be categorized into two Types: Primitives and Objects. The Primitive Types are `String`, `Number`, `Boolean`, `Symbol`, `undefined` and `null`. The Object Types are `Function`, `Object` and `Array`.

The two distinctions between Primitives and Objects are their mutability and the way they’re compared and “passed around” within the program.

Primitives are immutable. Another way of saying this is that their values can’t be changed. On the other hand, Objects are mutable. Their values can be updated and changed.

Primitives are compared by value. When assigning one primitive to another using variables, a copy is made. Objects on the other hand are compared by reference. Reference to what? Reference to the underlying Object. When assigning one Object to another, a reference / pointer is created. At this stage, mutating a value on one Object will update the value on the other Object.

When attempting to call methods on Primitive values, JavaScript uses a Wrapper Object to temporarily coerce the Primitive. The resulting Object is read-only and garbage collected after execution.

In the next section, we'll go over how these Types fit into the bigger picture by analyzing Prototypes, Constructors, and Inheritance.

## Update: 11/8/14

Recently there was a discussion over at <https://javascriptkicks.com/stories/1669> regarding the performance of both Objects and Primitives. [@drewpcodes](#) wanted to know which approach would be faster: storing floating point numbers in Primitive form or Objects form? I was curious to know as-well so I wrote a small program to test it out. You can find the code I used here: <http://jsfiddle.net/pmqortov>

I created two arrays: one array stored the data as Objects, and another stored the data as Primitives. Based on [@drewpcodes](#) use-case, the data I used was 150 floating point numbers. I also decided to limit both arrays to 150 entries so it wouldn't freeze my computer.

I then iterated over the arrays and saved a timestamp before and after every iteration. Then I did a little math on each timestamp and found the average in ms. For the sake of accuracy, I looped 5000 times for each instance. I noticed increasing this number would freeze Chrome. =)

Surprisingly enough, the list containing the data stored as Objects was actually 2ms faster then stored as Primitives! Here were the actual numbers:

```
// List with Objects
// Average execution across 5000 repetitions: 16.8528 ms

// List with Primitives
// Average execution across 5000 repetitions: 18.2898 ms
```

At this point I don't have an opinion either way of which method is best for storing your data. It could be that at higher volumes the Object method would be advantagous, but I'd like to see more evidence first.

Cheers!

## 20 : Partial Application with Function#bind

---

<http://adripofjavascript.com/blog/drips/partial-application-with-function-bind.html>

## 21 : Partial Application with Function#bind

---

Originally published in the [A Drip of JavaScript newsletter](#).

In [last week's drip](#), we covered using `bind` to create bound functions. But `bind` is also a very convenient way of implementing partial application. First, let's take a look at exactly what partial application is.

According to [Wikipedia](#), partial application "refers to the process of fixing a number of arguments to a function, producing another function of smaller arity." Arity refers to the number of arguments that a function takes.

Here's an example:

```
function multiply (x, y) {
  return x * y;
}

function double (num) {
  return multiply(2, num);
}
```

That's a very rudimentary form of partial application, where inside `double` we permanently fix `multiply`'s first argument as `2`, producing a new function `double`.

But explicitly declaring the body of the new function isn't actually necessary if you use `bind`. Here's how we could rewrite that:

```
function multiply (x, y) {
  return x * y;
}

var double = multiply.bind(null, 2);
```

As we covered last time, the first argument to `bind` sets its internal `this` value. Since our function doesn't depend on `this`, we just pass in `null`. But any subsequent arguments that we supply will be used to permanently fix the arguments of the function we are binding. In this case, we are only fixing one argument, but we can potentially fix as many as we want.

```
function greet (salutation, person, delivery) {
  var message = ''' + salutation + ', ' + person + ',' ' +
    delivery + ' the greeter.';
```

```

        console.log(message);
    }

var hail = greet.bind(null, "Hail");

// Outputs: '"Hail, Lord Elrond," said the greeter.'
hail("Lord Elrond", "said");

var begone = greet.bind(null, "Begone", "Wormtongue", "commanded");

// Outputs: '"Begone, Wormtongue," commanded the greeter.'
begone();

```

One thing you may have noticed is that `bind` always fixes the arguments from left to right. This means that it isn't suitable for all forms of partial application, but it does cover the most common ones.

You may be wondering if partial application is actually all that useful in real code. To answer, let's revisit an example from our [discussion of dispatch tables](#):

```

var commandTable = {
    north:    function() { movePlayer("north"); },
    east:     function() { movePlayer("east"); },
    south:    function() { movePlayer("south"); },
    west:     function() { movePlayer("west"); },
    look:     describeLocation,
    backpack: showBackpack
};

function processUserInput(command) {
    commandTable[command]();
}

```

In this example we are taking user input from a text adventure game, and then calling the appropriate command from `commandTable`. As you can see, we're using the same rudimentary form of partial application that we saw in our first example.

Using `bind` we can clean this up a bit.

```

var commandTable = {
    north:    movePlayer.bind(null, "north"),
    east:     movePlayer.bind(null, "east"),

```

```
    south: movePlayer.bind(null, "south"),
    west: movePlayer.bind(null, "west"),
    look: describeLocation,
    backpack: showBackpack
};

function processUserInput(command) {
  commandTable[command]();
}
```

As I mentioned last time, `bind` isn't available in IE8 and older, so you may want to use a polyfill or library to achieve the same functionality. If you only want to implement partial application and don't need to create bound functions, you might want to consider the `partial` methods in [Underscore](#) and [Lo-Dash](#).

## 22 : Partial Application in JavaScript

<http://blakeembrey.com/articles/2014/01/partial-application-in-javascript/>

Partial application is the act of pre-filling arguments of a function and returning a new function of smaller arity. The returned function can be called with additional parameters and in JavaScript, the `this` context can also be changed when called. Using a partially applied function is extremely common in functional programming with JavaScript as it allows us to compose some really nifty utilities and avoid repeating ourselves in code.

In modern JavaScript engines, there is a [bind function](#) which can be used to achieve a similar result. The difference between `partial` and `bind` is that the a partial functions `this` context is set when the returned function is called, while a bound functions `this` context has already been defined and can't be changed.

```
var __slice = Array.prototype.slice;

/**
 * Wrap a function with default arguments for partial application.
 *
 * @param {Function} fn
 * @param {*} ...
 * @return {Function}
 */
var partial = function (fn /*, ...args */) {
    var args = __slice.call(arguments, 1);

    return function () {
        return fn.apply(this, args.concat(__slice.call(arguments)));
    };
};
```

From the function above, we can understand that `partial` accepts the function to be pre-filled and its default arguments. It then returns a new function which can be called with some more arguments. It's important to note that the context (`this`) is being defined when the returned function is called. But when would you even want to use this?

Normally I would be happy to give a simple example of transforming an `add` function into an `add5` by partially applying it - `partial(add, 5)`. This definitely demonstrates how we can use the utility, but doesn't really touch on why.

Consider writing a logging utility that accepts some different arguments that need to be logged - `var log = function (type, value) {}`. Fantastic, it looks like a really simple function to use. But now we want set

every log in our file to the `testing` type. We can do a couple of things to achieve this. One option would be to assign our type to a variable and reuse the variable - `var testType = 'Testing'` and `log(testType, value)`. This will get messy after we write it more than once. What if we just wrapped the `log` function automatically?

```
var testLog = function () {
  return log.apply(this, ['testing'].concat(__slice.call(arguments)));
};
```

Great, this looks familiar - we could have just used partial - `var testLog = partial(log, 'Testing')`. Now we have a function we can continue to reuse any number of times without fear of repeating ourselves.

## Bonus Points

If you've been reading any of my previous blog posts, you may have noticed me abusing the usefulness of [function arity](#) in anonymously returned functions. And in another article I wrote about a utility that can help us remove the [repetitive argument slicing](#). If you haven't checked out these utilities yet, take a quick look and I bet you'll see how we could use them here.

```
var partial = variadic(function (fn, args) {
  var remaining = Math.max(fn.length - args.length, 0);

  return arity(remaining, variadic(function (called) {
    return fn.apply(this, args.concat(called));
  }));
});
```

Now the returned partially applied function gives us the correct number of trailing arguments still to be filled using the `arity` utility. On top of that, we managed to get rid of slicing arguments constantly by using the `variadic` utility. In fact, I've been so interested in these reusable utilities that I published the [partial utility on Github](#) so I can reuse it later.

## 23 : Composing Functions in JavaScript

<http://blakeembrey.com/articles/2014/01/compose-functions-javascript/>

Composing multiple functions to create more complex ones is a common utility in any programming language. And the ability to construct functions in a way that is easily composable is a true talent, but it really shines with code maintenance and reuse. It's not uncommon to find huge applications composed of many, much smaller functions. Inspired by this pattern of extremely modular functions, I've been slowly migrating my programming style to allow for more composable and reusable functions.

To compose functions together, we will need to accept a list of functions for it to be made up from. Let's call the functions `a`, `b` and `c`. Now that we have the list of functions, we need to call each of them with the result of the next function. In JavaScript, we would do this with `a(b(c(x)))` - with `x` being the starting value. However, it would be much more useful to have something a little more reusable than this.

```
var compose = function () {
  var fns = arguments;

  return function (result) {
    for (var i = fns.length - 1; i > -1; i--) {
      result = fns[i].call(this, result);
    }

    return result;
  };
};
```

The above function iterates over the function list (our arguments) in reverse - the last function to pass in is executed first. Given a single value as the initial input, it'll chain that value between every function call and return the final result. This allows us to do some really cool things.

```
var number = compose(Math.round, parseFloat);

number('72.5'); //=> 73
```

### Sequence

Another utility I've seen about in some functional libraries is called [sequence](#). It's very similar to `compose`, except the arguments are executed in reverse. For example:

```
var sequence = function () {
    var fns = arguments;

    return function (result) {
        for (var i = 0; i < fns.length; i++) {
            result = fns[i].call(this, result);
        }

        return result;
    };
};
```

However, we should make a note of the almost identical function signature to `compose`. Usually, seeing something like this should trigger a warning in your head to find some way to reuse previous functionality, instead of replicating it. In this example, we can reuse the `compose` function to write the `sequence` implementation.

```
var __slice = Array.prototype.slice;

var sequence = function () {
    return compose.apply(this, __slice.call(arguments).reverse());
};
```

## 24 : A JavaScript Invoke Function

<http://blakeembrey.com/articles/2014/01/javascript-invoke-function/>

**Update:** Now available on [github](#) and [npm](#).

Under certain functional JavaScript toolbelts, we can find a utility that is used purely for invoking a method on a passed in object. The utility is a really simple snippet that can be used in a number of different circumstances.

```
var __slice = Array.prototype.slice;

var invoke = function (method /*, ...args */) {
    var args = __slice.call(arguments, 1);

    return function (obj /*, ..args */) {
        return obj[method].apply(obj, args.concat(__slice.call(arguments, 1)));
    };
};
```

The most useful situation for a utility such as this is in combination with other functional utilities and iterators. Consider the case where we have an array of objects with identical methods. Not uncommon in a complex MVC application where you may be tracking child views. To remove every child view, we need to iterate over an array of views and call `remove`.

```
var children = [/* ... */];

children.forEach(invoke('remove'));
```

## 25 : YOU MIGHT NOT NEED JQUERY

<http://youmightnotneedjquery.com/>

Your search didn't match any comparisons.

### AJAX

#### Alternatives:

- [reqwest](#)
- [then-request](#)
- [superagent](#)

### JSON

#### jQuery

```
$.getJSON('/my/url', function(data) {  
});
```

#### IE9+

```
var request = new XMLHttpRequest();  
request.open('GET', '/my/url', true);  
  
request.onload = function() {  
    if (request.status >= 200 && request.status < 400) {  
        // Success!  
        var data = JSON.parse(request.responseText);  
    } else {  
        // We reached our target server, but it returned an error  
  
    }  
};  
  
request.onerror = function() {  
    // There was a connection error of some sort  
};  
  
request.send();
```

## Post

### jQuery

```
$.ajax({  
    type: 'POST',  
    url: '/my/url',  
    data: data  
});
```

### IE8+

```
var request = new XMLHttpRequest();  
request.open('POST', '/my/url', true);  
request.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded;  
charset=UTF-8');  
request.send(data);
```

## Request

### jQuery

```
$.ajax({  
    type: 'GET',  
    url: '/my/url',  
    success: function(resp) {  
  
    },  
    error: function() {  
  
    }  
});
```

### IE9+

```
var request = new XMLHttpRequest();  
request.open('GET', '/my/url', true);
```

```

request.onload = function() {
    if (request.status >= 200 && request.status < 400) {
        // Success!
        var resp = request.responseText;
    } else {
        // We reached our target server, but it returned an error
    }
};

request.onerror = function() {
    // There was a connection error of some sort
};

request.send();

```

## Effects

### Alternatives:

- [animate.css](#)
- [move.js](#)

### Fade In

#### jQuery

```
$(el).fadeIn();
```

#### IE9+

```

function fadeIn(el) {
    el.style.opacity = 0;

    var last = +new Date();
    var tick = function() {
        el.style.opacity = +el.style.opacity + (new Date() - last) / 400;
        last = +new Date();

        if (+el.style.opacity < 1) {
            (window.requestAnimationFrame && requestAnimationFrame(tick)) ||

```

```
    setTimeout(tick, 16);
  }
};

tick();
}

fadeIn(el);
```

## Hide

### jQuery

```
$(el).hide();
```

### IE8+

```
el.style.display = 'none';
```

## Show

### jQuery

```
$(el).show();
```

### IE8+

```
el.style.display = '';
```

## Elements

### Alternatives:

- [bonzo](#)
- [\\$dom](#)

## Add Class

### jQuery

```
$(el).addClass(className);
```

IE8+

```
if (el.classList)
    el.classList.add(className);
else
    el.className += ' ' + className;
```

After

jQuery

```
$(el).after(htmlString);
```

IE8+

```
el.insertAdjacentHTML('afterend', htmlString);
```

Append

jQuery

```
$(parent).append(el);
```

IE8+

```
parent.appendChild(el);
```

Before

jQuery

```
$(el).before(htmlString);
```

IE8+

```
el.insertAdjacentHTML('beforebegin', htmlString);
```

## Children

### jQuery

```
$(el).children();
```

### IE9+

```
el.children
```

## Clone

### jQuery

```
$(el).clone();
```

### IE8+

```
el.cloneNode(true);
```

## Contains

### jQuery

```
$.contains(el, child);
```

### IE8+

```
el !== child && el.contains(child);
```

## Contains Selector

### jQuery

```
$(el).find(selector).length;
```

**IE8+**

```
el.querySelector(selector) !== null
```

## Each

**jQuery**

```
$(selector).each(function(i, el){  
});
```

**IE9+**

```
var elements = document.querySelectorAll(selector);  
Array.prototype.forEach.call(elements, function(el, i){  
});
```

## Empty

**jQuery**

```
$(el).empty();
```

**IE9+**

```
el.innerHTML = '';
```

## Filter

**jQuery**

```
$(selector).filter(filterFn);
```

**IE9+**

```
Array.prototype.filter.call(document.querySelectorAll(selector), filterFn);
```

## [Find Children](#)

### jQuery

```
$(el).find(selector);
```

### IE8+

```
el.querySelectorAll(selector);
```

## [Find Elements](#)

### Alternatives:

- [qwyry](#)
- [sizzle](#)

### jQuery

```
$('.my #awesome selector');
```

### IE8+

```
document.querySelectorAll('.my #awesome selector');
```

## [Get Attributes](#)

### jQuery

```
$(el).attr('tabindex');
```

### IE8+

```
el.getAttribute('tabindex');
```

## [Get Html](#)

## jQuery

```
$(el).html();
```

### IE8+

```
el.innerHTML
```

## [Get Outer Html](#)

## jQuery

```
$('
').append($(el).clone()).html();
```

### IE8+

```
el.outerHTML
```

## [Get Style](#)

## jQuery

```
$(el).css(ruleName);
```

### IE9+

```
getComputedStyle(el)[ruleName];
```

## [Get Text](#)

## jQuery

```
$(el).text();
```

### IE9+

```
el.textContent
```

## Has Class

### jQuery

```
$(el).hasClass(className);
```

### IE8+

```
if (el.classList)
    el.classList.contains(className);
else
    new RegExp('(^| )' + className + '( |$)', 'gi').test(el.className);
```

## Matches

### jQuery

```
$(el).is($(otherEl));
```

### IE8+

```
el === otherEl
```

## Matches Selector

### jQuery

```
$(el).is('.my-class');
```

### IE9+

```
var matches = function(el, selector) {
    return (el.matches || el.matchesSelector || el.msMatchesSelector ||
    el.mozMatchesSelector || el.webkitMatchesSelector || el.oMatchesSelector).call(el,
    selector);
```

```
};

matches(el, '.my-class');
```

## [Next](#)

### jQuery

```
$(el).next();
```

### IE9+

```
el.nextElementSibling
```

## [Offset](#)

### jQuery

```
$(el).offset();
```

### IE8+

```
var rect = el.getBoundingClientRect();

{
  top: rect.top + document.body.scrollTop,
  left: rect.left + document.body.scrollLeft
}
```

## [Offset Parent](#)

### jQuery

```
$(el).offsetParent();
```

### IE8+

```
el.offsetParent || el
```

## Outer Height

### jQuery

```
$(el).outerHeight();
```

### IE8+

```
el.offsetHeight
```

## Outer Height With Margin

### jQuery

```
$(el).outerHeight(true);
```

### IE9+

```
function outerHeight(el) {  
    var height = el.offsetHeight;  
    var style = getComputedStyle(el);  
  
    height += parseInt(style.marginTop) + parseInt(style.marginBottom);  
    return height;  
}  
  
outerHeight(el);
```

## Outer Width

### jQuery

```
$(el).outerWidth();
```

### IE8+

```
el.offsetWidth
```

## Outer Width With Margin

### jQuery

```
$(el).outerWidth(true);
```

### IE9+

```
function outerWidth(el) {  
    var width = el.offsetWidth;  
    var style = getComputedStyle(el);  
  
    width += parseInt(style.marginLeft) + parseInt(style.marginRight);  
    return width;  
}  
  
outerWidth(el);
```

## Parent

### jQuery

```
$(el).parent();
```

### IE8+

```
el.parentNode
```

## Position

### jQuery

```
$(el).position();
```

### IE8+

```
{left: el.offsetLeft, top: el.offsetTop}
```

## Position Relative To Viewport

### jQuery

```
var offset = el.offset();  
  
{  
    top: offset.top - document.body.scrollTop,  
    left: offset.left - document.body.scrollLeft  
}
```

### IE8+

```
el.getBoundingClientRect()
```

## Prepend

### jQuery

```
$(parent).prepend(el);
```

### IE8+

```
parent.insertBefore(el, parent.firstChild);
```

## Prev

### jQuery

```
$(el).prev();
```

### IE9+

```
el.previousElementSibling
```

## Remove

## jQuery

```
$(el).remove();
```

### IE8+

```
el.parentNode.removeChild(el);
```

## Remove Class

### jQuery

```
$(el).removeClass(className);
```

### IE8+

```
if (el.classList)
    el.classList.remove(className);
else
    el.className = el.className.replace(new RegExp('(^|\\b)' + className.split(' ')
        .join('|') + '(\\b|$)', 'gi'), ' ');
```

## Replace From Html

### jQuery

```
$(el).replaceWith(string);
```

### IE8+

```
el.outerHTML = string;
```

## Set Attributes

### jQuery

```
$(el).attr('tabindex', 3);
```

**IE8+**

```
el.setAttribute('tabindex', 3);
```

## [Set Html](#)

**jQuery**

```
$(el).html(string);
```

**IE8+**

```
el.innerHTML = string;
```

## [Set Style](#)

**jQuery**

```
$(el).css('border-width', '20px');
```

**IE8+**

```
// Use a class if possible  
el.style.borderWidth = '20px';
```

## [Set Text](#)

**jQuery**

```
$(el).text(string);
```

**IE9+**

```
el.textContent = string;
```

## [Siblings](#)

## jQuery

```
$(el).siblings();
```

### IE9+

```
Array.prototype.filter.call(el.parentNode.children, function(child){  
    return child !== el;  
});
```

## Toggle Class

### jQuery

```
$(el).toggleClass(className);
```

### IE9+

```
if (el.classList) {  
    el.classList.toggle(className);  
} else {  
    var classes = el.className.split(' ');  
    var existingIndex = classes.indexOf(className);  
  
    if (existingIndex >= 0)  
        classes.splice(existingIndex, 1);  
    else  
        classes.push(className);  
  
    el.className = classes.join(' ');\n}
```

## Events

### Off

### jQuery

```
$(el).off(eventName, eventHandler);
```

**IE9+**

```
el.removeEventListener(eventName, eventHandler);
```

**On****jQuery**

```
$(el).on(eventName, eventHandler);
```

**IE9+**

```
el.addEventListener(eventName, eventHandler);
```

**Ready****jQuery**

```
$(document).ready(function(){
});
```

**IE9+**

```
function ready(fn) {
  if (document.readyState != 'loading'){
    fn();
  } else {
    document.addEventListener('DOMContentLoaded', fn);
  }
}
```

**Trigger Custom****Alternatives:**

- [EventEmitter](#)

- [Vine](#)
- [microevent](#)

## jQuery

```
$(el).trigger('my-event', {some: 'data'});
```

## IE9+

```
if (window.CustomEvent) {
    var event = new CustomEvent('my-event', {detail: {some: 'data'}});
} else {
    var event = document.createEvent('CustomEvent');
    event.initCustomEvent('my-event', true, true, {some: 'data'});
}

el.dispatchEvent(event);
```

## [Trigger Native](#)

## jQuery

```
$(el).trigger('change');
```

## IE9+

```
// For a full list of event types: https://developer.mozilla.org/en-US/docs/Web/API/document.createEvent
var event = document.createEvent('HTMLEvents');
event.initEvent('change', true, false);
el.dispatchEvent(event);
```

## Utils

### [Bind](#)

## jQuery

```
$.proxy(fn, context);
```

**IE9+**

```
fn.bind(context);
```

## Array Each

**jQuery**

```
$.each(array, function(i, item){  
});
```

**IE9+**

```
array.forEach(function(item, i){  
});
```

## Deep Extend

**jQuery**

```
$.extend(true, {}, objA, objB);
```

**IE8+**

```
var deepExtend = function(out) {  
    out = out || {};  
  
    for (var i = 1; i < arguments.length; i++) {  
        var obj = arguments[i];  
  
        if (!obj)  
            continue;  
  
        for (var key in obj) {  
            if (obj.hasOwnProperty(key)) {  
                if (typeof obj[key] === 'object')
```

```
        out[key] = deepExtend(out[key], obj[key]);
    else
        out[key] = obj[key];
    }
}

return out;
};

deepExtend({}, objA, objB);
```

## [Extend](#)

### Alternatives:

- [lo-dash](#)
- [underscore](#)
- [ECMA6](#)

## jQuery

```
$.extend({}, objA, objB);
```

## IE8+

```
var extend = function(out) {
    out = out || {};
    for (var i = 1; i < arguments.length; i++) {
        if (!arguments[i])
            continue;
        for (var key in arguments[i]) {
            if (arguments[i].hasOwnProperty(key))
                out[key] = arguments[i][key];
        }
    }
    return out;
};
```

```
extend({}, objA, objB);
```

## [Index Of](#)

### **jQuery**

```
$.inArray(item, array);
```

### **IE9+**

```
array.indexOf(item);
```

## [Is Array](#)

### **jQuery**

```
$.isArray(arr);
```

### **IE9+**

```
Array.isArray(arr);
```

## [Map](#)

### **jQuery**

```
$.map(array, function(value, index){  
});
```

### **IE9+**

```
array.map(function(value, index){  
});
```

## Now

### jQuery

```
$.now();
```

### IE9+

```
Date.now();
```

## Parse Html

### jQuery

```
$.parseHTML(htmlString);
```

### IE9+

```
var parseHTML = function(str) {  
    var tmp = document.implementation.createHTMLDocument();  
    tmp.body.innerHTML = str;  
    return tmp.body.children;  
};  
  
parseHTML(htmlString);
```

## Parse Json

### jQuery

```
$.parseJSON(string);
```

### IE8+

```
JSON.parse(string);
```

## Trim

## jQuery

```
$.trim(string);
```

IE9+

```
string.trim();
```

## Type

### jQuery

```
$.type(obj);
```

IE8+

```
Object.prototype.toString.call(obj).replace(/^\[object (.+)\]\$/,
'$1').toLowerCase();
```

## 26 : The JavaScript Bind Function

<http://blakeembrey.com/articles/2013/12/javascript-bind-function/>

The JavaScript `bind` function is a common-place utility when working with many different frameworks and libraries. Its purpose is to bind the `this` value to a static object and is useful when passing functions around as callbacks, where maintaining the correct `this` value is required. A common convention to circumvent this utility is the `var that = this`, but this isn't very feasible everywhere.

In [modern JavaScript implementations](#) the function is built directly onto `Function.prototype`, giving us `bind` functionality on every function. For our implementation we'll be implementing a standalone functionality that works similar to the built-in `bind` function.

However, it's important to note that `bind` also comes with another handy feature. It accepts an unlimited number of arguments after the context to pass in as the function parameters, from left to right.

```
var __slice = Array.prototype.slice;

/** 
 * Bind a function to a certain context.
 *
 * @param {Function} fn
 * @param {Object} context
 * @param {*} ...
 * @return {Function}
 */
var bind = function (fn, context /*, ...args */) {
    var args = __slice.call(arguments, 2);

    return function () {
        return fn.apply(context, args.concat(__slice.call(arguments)));
    };
};
```

Bind allows us to keep the `this` context when passing the callback to another function. Imagine passing a function that uses `this` into `setTimeout` or someone else's library utility, where `this` could be unpredictable.

```
var greet = function (greeting) {
    return greeting + ' ' + this.user;
};
```

```
greet('Hello'); //=> "Hello undefined"

var boundGreet = bind(greet, { user: 'Bob' });

boundGreet('Hello'); //=> "Hello Bob"
```

We also have another useful feature built into `bind` - partial application. Partial application is essentially the act of pre-filling function arguments. Any future arguments are then appended to the arguments we have already defined.

```
var greet = function (user, greeting) {
  return greeting + ' ' + user;
};

var greetBlake = bind(greet, null, 'Blake');

greetBlake('Hi'); //=> "Hi Blake"
greetBlake('Hello'); //=> "Hello Blake"
```

## Bonus Implementation using Variadic

In my last post, I introduced the concept of a [variadic function](#). As this article demonstrates, `bind` is a perfect example of a variadic function, so let's reimplement `bind` with the variadic function.

```
var bind = variadic(function (fn, context, args) {
  return variadic(function (called) {
    return fn.apply(context, args.concat(called));
  });
});
```

## 27 : Forcing Function Arity in JavaScript

<http://blakeembrey.com/articles/2014/01/forcing-function-arity-in-javascript/>

Function arity is something in JavaScript that is usually overlooked. For the most part, that's perfectly understandable, it's just a number. Unfortunately, this number can be integral to many other functions working correctly. But first, what number am I talking about?

```
var fn = function (a, b) {};  
  
fn.length; //=> 2
```

As you can see, the length gives up the exact number of arguments the function is expecting to be passed in. This can be useful for other functions that might want to alter its behaviour based on this digit. For example, I found outlining this issue [with currying](#). Basically, the `curry` function implementation relies on using the arity information to know how many times the function needs to be curried.

To force the number of arity in our returned anonymous functions, we need to dynamically generate a function with the specified number of arguments. Why? Because the previous implementations of [wrapping functions](#), [bind](#), [variadic](#) and every other functional utility I have demonstrated don't proxy the number of arguments through the returned function.

This can be a problem in the case where we want to use this function somewhere that expects a function length to work correctly, like when currying. We could fix this at the source, a half a dozen times and any number of times more. Or we could write a little utility that will enforce a number of arguments for us.

```
var names    = 'abcdefghijklmnopqrstuvwxyz';  
var __slice = Array.prototype.slice;  
  
/**  
 * Make a function appear as though it accepts a certain number of arguments.  
 *  
 * @param {Number}    length  
 * @param {Function} fn  
 * @return {Function}  
 */  
  
var arity = function (length, fn) {  
    return eval(  
        '(function (' + __slice.call(names, 0, length).join(',') + ') {\n' +  
        'return fn.apply(this, arguments);\n' +  
        '})')
```

```
 );
}
```

The above function allows us to pass in an argument length and a function to proxy. It then returns to us an anonymous function with the correct number of arguments defined ( `.length` works!) and allows us to call the function and return the usual result. It doesn't do anything to the arguments in the interim, it just tells the world how many arguments we are accepting.

## The Other Arity Problem

So we've touched one of the arity problems, which is a expecting to read the correct arity from a function. The reverse arity problem is when a function is called with incorrect or overloaded arguments. Consider `parseInt`, which accepts two arguments - a string and the radix.

```
[1, 2, 3, 4, 5].map(parseInt); //=> [1, NaN, NaN, NaN, NaN]
```

Now we're having problems. To fix this we can make a utility function that limits the number of arguments passed through.

```
var __slice = Array.prototype.slice;

/**
 * Force a function to accept a specific number of arguments.
 *
 * @param {Number} length
 * @param {Function} fn
 * @return {Function}
 */
var nary = function (length, fn) {
    return function () {
        return fn.apply(this, __slice.call(arguments, 0, length));
    };
};
```

If you've been reading, you would have just noticed that we introduced the original bug we've been trying to avoid. That is, we're returning a new anonymous function without proxying the number of arguments through. Let's quickly correct that with the function we just wrote.

```
var __slice = Array.prototype.slice;

/**

```

```
* Force a function to accept a specific number of arguments.  
*  
* @param {Number} length  
* @param {Function} fn  
* @return {Function}  
*/  
  
var nary = function (length, fn) {  
    // Uses the previous function to proxy the number of arguments.  
    return arity(length, function () {  
        return fn.apply(this, __slice.call(arguments, 0, length));  
    });  
};
```

Now we can use this to fix our map error from earlier. We also have the added bonus of a correct argument representation.

```
nary(1, parseInt).length; //=> 1  
  
[1, 2, 3, 4, 5].map(nary(1, parseInt)); //=> [1, 2, 3, 4, 5]
```

## 28 : Wrapping JavaScript Functions

<http://blakeembrey.com/articles/2014/01/wrapping-javascript-functions/>

In the modern age of web applications and development, it seems we are constantly adding side effects to every part of our applications - everything from analytics to event triggering. Unfortunately in a lot of cases, we tend to cram this functionality into function with the useful stuff. As programmers, this causes numerous issues down the line - especially when it comes to refactoring and code comprehensibility.

A simple way to keep this functionality apart from the core code is create a helpful utility function to manage it for you. And to keep our code readability, we shouldn't allow anything advanced that can break our understanding of the original function. That means we don't want to be able to alter the original function, but we can still trigger any side effects we need to inline with the original function.

```
var before = function (before, fn) {
  return function () {
    before.apply(this, arguments);
    return fn.apply(this, arguments);
  };
};
```

To use the function, we can pass any function in as the first argument and the original function we want to wrap as the second argument. For example, we could do `before(logger, add)`. Even without seeing the `logger` or `add` functions, we can imagine what each do. And because we are passing all the arguments to the side effect function, we can do stuff with the information.

One thing I find myself doing is checking what arguments were passed to a certain function. To do this now, we can `before(console.log.bind(console), fn)`. Now, let's implement the reverse functionality.

```
var after = function (fn, after) {
  return function () {
    var result = fn.apply(this, arguments);
    after.call(this, result);
    return result;
  };
};
```

This is extremely similar to the first example. The main difference is that the first function passed in is the side effect, but now we have the side effect running after our wrapped function. Adapting the previous example, we can now do `after(add, logger)` and the logger will execute after the result is computed with the same arguments.

One cool thing we could actually do, is to run argument validation in the `before` function. Consider this.

```
var validAdd = before(function () {
  for (var i = 0; i < arguments.length; i++) {
    if (typeof arguments[i] !== 'number') {
      throw new TypeError('Expected a number');
    }
  }
}, add);
```

We can also put these two functions together and create a new utility. This one allows us to pass both a function before and after our core functionality. E.g. `around(logger, add, logger)`.

```
var around = function (over, fn, under) {
  return before(over, after(fn, under));
};
```

## Allow unlimited before and after functions

We can also adapt the functions to accept a variable number of arguments as the `before` and `after` functions. However, we can't do this to the `around` utility since we wouldn't know which argument is the core function.

```
var __slice = Array.prototype.slice;

var before = function /* ...before, fn */() {
  var fn      = arguments[arguments.length - 1];
  var before = __slice.call(arguments, 0, -1);

  return function () {
    for (var i = 0; i < before.length; i++) {
      before[i].apply(this, arguments);
    }

    return fn.apply(this, arguments);
  };
};

var after = function (fn /*, ...after */) {
  var after = __slice.call(arguments, 1);
```

```
return function () {
    var result = fn.apply(this, arguments);

    for (var i = 0; i < after.length; i++) {
        after[i].call(this, result);
    }

    return result;
};

};
```

## Advanced wrapping utility

So far we've seen some function wrapping utilities that are purely for side effects. They have no capability to alter the main function arguments or change the function result. For something more advanced than trigger side-effects, we might want to use something a little different.

```
var __slice = Array.prototype.slice;

var wrap = function (fn, wrap) {
    return function () {
        return wrap.apply(this, [fn].concat(__slice.call(arguments)));
    };
};
```

This is actually pretty similar to the `wrap` function used in Prototype.js. It allows us to call a custom wrapper function with the original function and all the arguments. But, how do we even use this?

```
var addAndMultiplyBy2 = wrap(add, function (originalFn, a, b) {
    return 2 * originalFn(a, b);
});
```

## 29 : Javascript : Basic Scope

<http://www.datchley.name/basic-scope/>

Part 1 of a [fundamentals series](#) covering Javascript basics that I routinely come across in mentoring junior developers.

It's common for Javascript developers at all levels to deal with scope and its associated implications. In this post we'll cover the basics of Javascript scope and some of the issues commonly encountered.

### Lexical Scope

Scope, in general, refers to how the browser's javascript engine looks up identifier names at run time to in order to set how they will be looked up during execution. That definition implies that there is a *lexing* phase of the engine which is done prior to executing.

Javascript has two lexical scopes: *global* and *function* level. With ES6 there is also *block* level scoping; but more on that later. These lexical scopes are also nested. For instance:

```
var a = 4;                                1 — global: a, foo

function foo(x) {
    var b = a * 4;                         2 — foo: x, b, bar

    function bar(y) {
        var c = y * b;
        return c;                           3 — bar: y, c
    }

    return bar(b);
}

console.log(foo(a));
// 256
```

Here we have three separate lexical scopes. The default global scope which contains declarations for `a` and `foo`, the lexical scope declared within the `foo()` function which contains `x`, `b` and `bar`, and the lexical scope declared within `bar()` which contains `y` and `c`.

Notice that outside of a function, the default scope of declared variables is the global scope (*window in the browser*). Also, that scopes can be nested. Nested scopes have access to the scope's they are declared in, which has two important implications:

- nested scopes can access variables declared in their parent scopes (*there is a scope lookup chain*)
- nested scopes can *shadow* variables declared in their parent scope.

## Closures

The first point allows *closures*, meaning we can define a function which will retain access to its enclosing scope, even outside of that enclosing scope.

```
var times = function(n) {
    return function(x) {
        return x * n;
    };
}
var times2 = times(2);
console.log(times2(4));
// => 8
```

Here, the function returned by `times` allows us to create functions that multiply their argument by a given value, `n`. The function returned retains a reference to `n`, even though the lexical scope of that function is not accessible outside of the actual function. `n`, in effect, becomes a *free variable* that is only accessible to the function defined and returned within the scope in which `n` was declared.

## Shadowing

Scope lookup during the lexical phase also stops once it finds the first match. This means you can *shadow* a variable further up the scope chain.

```
var a = 4;
function foo(x) {
    var a = x; // shadows parent 'a' declaration
    console.log(a);
}
console.log(foo(6)); // => 6
console.log(a); // => 4
```

Here, the locally declared `a` *shadows* the `a` declared in the parent, global scope. Without the `var` keyword, we would have been reassigning to the variable `a` in the parent scope.

```
var a = 4;
function foo(x) {
    a = x; // woops!
    console.log(a);
}
```

```
console.log(foo(6)); // => 6
console.log(a); // => 6
```

## Hoisting

Hoisting also plays a factor with Javascript's scoping. In Javascript, `var` and `function(){} declarations are hoisted to the top of the current scope; and hence, those identifiers are available to any code in that scope.`

```
(function(){
    console.log(inc(4)); // 5
    console.log("a =",a); // a = undefined?

    var a = 1;
    function inc(n){ return ++n; }

})();
```

Uh, oh. So we can clearly access `inc()` which was hoisted and it looks like we can use `a` without a Reference Error; but, `a` is undefined?

Javascript only hoists the declaration and not the initialized value. The value will still end up being assigned at the same spot you have the original declaration. The above code actually gets modified and works as follows:

```
(function(){
    var a;
    function inc(n){ return ++n;

        console.log(inc(4)); // 5
        console.log("a = ", a); // a = undefined

        a = 4; // Ah-ha!
    }());
});
```

This is the same for variables assigned function expressions as well. Because function expressions are just values in an assignment statement.

```
(function(){
    console.log(inc(4)); // 5
    console.log(dec(4)); // ReferenceError: dec undefined
});
```

```
var dec = function(n) { return --n; }
function inc(n){ return ++n; }
})();
```

## ES6 Scoping Additions

With the new ES6 specification, lexical scope has been altered in a handful of ways, primarily with the new `let` and `const` declarations and with the shorter `=>` function syntax.

In ES5, as discussed above, if you wanted to declare a lexically scoped block of code, you could do so by creating a closure, typically with an *immediately invoked function expression* or *IIFE*. ES5 was essentially lexically scoped to *functions*.

```
var greet = (function() {
  var greeting = "Hello!";
  return function(name) {
    console.log(greeting + ", " + name + "!");
  };
})();
greet("Cap'n Tight Pants");
// => "Hello, Cap'n Tight Pants!"
```

### let

With ES6's `let` declaration we can now have lexically scope *blocks* of code, without using functions or IIFEs.

```
var a = 1, b = 2, c = 3;
{
  let a = 4, b = 5, c = 6;
  console.log("a,b,c = ", a, b, c);
}
console.log("a,b,c = ", a, b, c);
// => a,b,c = 4 5 6
// => a,b,c = 1 2 3
```

Something you might encounter with `let` declarations also, is that according to the ES6 spec, `let` declarations don't *hoist* like `var` and `function` declarations.

With `let` declarations, you'll get a reference error if you try to access a `let` declared variable *before* its declaration in the code (*according to the ES6 spec*).

```

console.log("a = ", a);
console.log("b = ", b);
var a = 4;
let b = 6;
// => a = undefined
// => b = ReferenceError!

```

However, if you are using Traceur or Babel(6to5), this will not be the case, as transpiling this dead zone behavior would require much more work and nearly reimplementing the Javascript run-time to handle those cases.

## **const**

`const` is the other new, block scope declaration type, which, as you would suspect creates *constant* variables.

```

const a = 4;
console.log("a = ", a);
a = 3;    // TypeError! a is readonly

```

A `const` declared variable must be initialized with a value when it is declared. Declaring `const a;` and trying to assign a value to it later on will throw an error. Also, `const` works by limiting the assignment to a variable, not by freezing a variable's value. So, if you assign a reference type to it, you can still modify the underlying properties of that reference type:

```

const a = { name: 'Mal', ship: 'Serenity' };
a = 4;    // TypeError
a.name = 'Wash' // no problem!

const b = [1,2,3];
b.unshift(0); // b = [0,1,2,3]

```

`const` declared reference variables hold a constant reference to that variable, not a constant value - so the underlying reference assigned to a `const` can change.

## **Lexical this**

We've already encountered the idea that Javascript `var` and `function` declarations are lexically scoped to the enclosing function or scope they are contained in. But, what about the `this` keyword inside of functions? What does it reference?

Most familiar is the use of `this` in functions used as constructors with the `new` keyword. When using `new`, `this` refers to the eventual object that will be returned from the function.

```
function Tribble(color) {
  this.color = color;
}
var my_tribble = new Tribble("brown");
my_tribble.color; // "brown"
```

When calling a method on an object, `this` refers to the object that is the *context* that that function is being called with; in most cases, this is the object prototype or instance the function is assigned to.

```
var tardis = {
  where: "Gallifrey",
  go: function() {
    console.log("Off to " + this.where + ", Allons-y!");
  }
};
tardis.go(); // "Off to Gallifrey, Allons-y!"
```

When calling `tardis.go()`, the `this` references the current context of that function, which is the `tardis` object. But, even without a context object, `this` still refers to the functions context, which turns out to be `window` for browsers.

```
var where = "Gallifrey";
function go() {
  console.log("Off to " + this.where + ", Allons-y!");
}
go(); // "Off to Gallifrey, Allons-y!"
```

`this`, it seems, is fairly flexible and potentially dubious in Javascript depending on what you're trying to do.

## ES6 => functions

ES6 specifies a short-hand syntax for declarations for functions using the *fat arrow* (`=>`) syntax. However, `=>` functions bind `this` to their enclosing scope, unlike regular functions which create a new scope entirely.

```
var even = (x) => x % 2 == 0;
console.log(even(2)); // true
console.log(even(3)); // false
```

```
var a = 4,  
    list = [1,2,3].map((n) => n + a);  
// list = [5,6,7]
```

The left side of the `=>` declares the function's arguments and the right side declares the return value, or a block of code as the function body.

```
let evens = [1,2,3,4,5,6].filter((n) => {  
  if (even(n)) {  
    return n;  
  }  
});  
console.log(evens); // [2,4,6]
```

There is much more to Javascript scoping than this (*ha, get it?*); and I would suggest you take a look at Kyle Simpson's [You Don't Know JS: Scope & Closures](#) and the [ES6 spec](#) related to scoping of `let` and `const` as well for a more detailed coverage and understanding.

If you are using [Babel](#) or [Traceur](#), be sure and reference their documentation for any missing parts or inconsistencies from the ES6 specification as well, which will make it easier when you transition from transpilers to native ES6 in the browser.

---

In [Part 2](#) in the 'fundamentals' series I'll cover some of the basics of Javascript functions, notably `call`, `apply`, `bind` and more on the ES6 `=>` functions.

- [fundamentals](#)
- [scope](#)

## 30 : Part 2 : Functions

<http://www.datchley.name/functions/>

Part 2 of a series on Javascript Fundamentals. See [Part 1](#) on Scope in Javascript.

### Declaration, Expression, Invoking

I don't want to belabor the basics of function declaration and definition in Javascript; but there are primarily two different ways you'll see functions in Javascript:

- function declarations
- function expressions (named & anonymous)

Functions can be declared using the `function` keyword and a name, as in

```
function compare(a,b) {  
    return a == b ? 0 : (a < b) ? -1 : 1;  
}
```

This declares a function called `compare`, which is *hoisted* to the top of its enclosing scope and available to any code in that scope or its child scopes.

Functions can also be treated as values, meaning you can assign them to variables and pass and return them to and from functions as well. These are referred to as *first-class functions* and lead to *higher-order functions* when working in a more declarative, functional style of programming.

```
// function expression  
var even = function(n){ return n % 2 == 0; };  
  
// returning a function  
var log = function(base) { // returning a function  
    return function(n) {  
        return Math.log(n) / Math.log(base);  
    }  
};  
var log10 = log(10);  
log10(100); // 2
```

For a discussion of function `scope` in relation to *IIFEs* and lexical `this`, refer to [Part 1](#) of this series on Scope.

### Function Arguments & Arity

The arity of a function refers to the number of arguments a function expects. This is determined by the number of declared arguments in the function declaration and is available in the function's `.length` property.

```
function foo(a,b) { console.log(a, b); }
foo.length; // 2
```

A functions arity is, and can be, necessarily different than the arguments it actually receives when called. If we invoke a function with fewer arguments than it expects, the argument identifiers in the function for arguments not passed will be set to undefined.

```
foo(4, 5); // => 4, 5
foo(3); // => 3, undefined
foo(); // => undefined, undefined
```

Within a function we can access all the arguments passed, both those declared in the function declaration and any extras (*you can pass more than the declared arguments to a function*) using the available `arguments` variable.

```
function args(a,b) {
  console.log("a,b: ", a, b);
  console.log("all: ", arguments);
}
foo(1,2); // a,b: 1 2
           // all: [1,2]
foo(1,2,3,4); // a,b: 1 2
               // all: [1,2,3,4]
```

The `arguments` object is a local variable available within all functions and it is *array-like*. This means it is not an instance of the `Array` type and does not have many of its methods. `arguments` can be accessed by index, ie `arguments[0]`, `arguments[1]` and it has a `.length` property.

This feature allows Javascript functions to have variable arguments. To work with the arguments as a real array, you can simply convert the `arguments` object to an array.

```
function has() {
  let args = [].slice.call(arguments);
  args.forEach((arg) => console.log(arg));
}
has(1,2,3,4);
// 1
```

```
// 2
// 3
// 4
```

## Invoking functions

Invoking a function is done using the `()` operator on the function name or variable holding the function expression.

```
var foo(){ /* do foo */ }    // declaration
foo();    // invoke

(function baz() {           // IIFE
  console.log("baz!");
})();
// "baz!"

var bar = function(){ /* do bar */ }; // expression
bar();    // invoke
```

We can also assign a function to a property on an object and invoke it through the standard object property access method as well.

```
let princess = {
  name: 'Leia Organa',
  say: function(msg) { console.log(this.name + ": " + msg); }
};

princess.say("I love you!");
// => Leia Organa: I love you!
```

In this case, when invoking a function via an object's property it's assigned to, the context of `this` in that function is the object itself.

However, using Javascript's `.call()` and `.apply()`, we can change that context when invoking the function.

```
let name = 'Han Solo';
princess.say.call(this, "I know.");
// => Han Solo: I know.
```

What just happened there? Why would Han totally underplay that kind of declaration *and* how'd he steal her line? The `.call()` method's first argument is an object to use as the invoking context of the function being called. This allows us to override `princess.name` and use the global `name` variable by passing in `this`, which refers to the `window` object.

`.call()` also lets us pass in as many arguments for that function using the remaining parameters, which is how Han came across as so smug.

The only difference between `.call()` and `.apply()` is that `.apply()` only takes 2 parameters: a context object just like `call`, and an array of arguments (*instead of passing them as individual parameters*). Even though `.apply()` takes the arguments as an array, they are still passed normally to the function being invoked.

```
function add(a,b) { console.log(a + b); }
add.apply(null, [2,3]);
// 5
```

Why did we just pass `null` as the context object and what exactly did that do? According to the ECMAScript standard, passing in `null` or `undefined` will make the function's lexically scoped `this` point to the global scope.

In most cases with single functions, the lexical context of `this` is probably not a concern, as you probably aren't referencing `this` within the function. However, when dealing with functions assigned to object properties and invoked through them, understanding how the first argument of `.call()` and `.apply()` affect the function's `this` is important.

Why would we use `.apply()` and not just use `.call()` everywhere? Let's say you had a function called `after()` that would wrap an existing function and ensure some code was executed every time after that function was called. Using `.call()` would be nearly impossible given that you don't know the number of parameters that function might be called with - without resorting to something potentially dangerous like using `eval()`.

```
function lots(a,b,c,d) {
  console.log([a,b,c,d].join(','));
}

function after(fn) {
  var orig = fn;
  return function() {
    orig.call(null, /* how do you pass them? */);
  };
}

var lots = after(lots);
lots(1,2,3,4);
```

This is where `.apply()` and the `arguments` local variable come to the rescue:

```
function after(fn) {
  var orig = fn;
  return function() {
    var args = [].slice.call(arguments);
    return orig.apply(null, args);
  };
}
var lots = after(lots);
lots(1,2,3,4);
/// 1,2,3,4
```

## Using `.bind()`

So we know now that `.call()` and `.apply()` can be used to not only invoke the given function; but to also change its calling context and pass parameters as well. But, those methods directly invoke the function when used.

Javascript also gives us the `.bind()` method on functions to allow us to *bind* both a context and one or more parameters and *not* invoke the function; but instead return this newly bound function to be called later.

We can use the same example from above, but allow our `after` function to take a second parameter to specify the calling context the function should be executed with.

```
var doctor = {
  name: 'Matt Smith',
  who: function named() {
    console.log(this.name);
  }
};

function after(fn, context) {
  var orig = context ? fn.bind(context) : fn;
  return function() {
    var args = [].slice.call(arguments);
    return orig.apply(null, args);
  };
}

var tenth = { name: 'David Tennant' };
```

```
var thedoctor = after(doctor.who, tenth);
doctor.who(); // Matt Smith
thedoctor(); // David Tennant
```

.bind() also allows us to pre-bind one or more argument parameters to the function as well. For instance:

```
function add(a,b) { return a + b; }
var add2 = add.bind(null, 2);
add2(4); // 6
add2(3,6); // 5
```

Here, we create a new function by *partially applying* the first argument to `add()`. Passing any subsequent parameters makes no difference.

Keep in mind that `.call()`, `.apply()` and `.bind()` can not be used with ES6's `=>` functions to change the context of `this`, as `this` is explicitly bound to the enclosing scope where the function is declared. You can use `.apply` and `.call` to pass in argument parameters, but the first argument is ignored for changing context.

For more detailed information, the Mozilla Developer Network (MDN), a good reference for anything Javascript, has excellent coverage on [.call](#), [.apply](#) and [.bind](#).

*In Part 3 in the 'fundamentals' series I'll cover a number of the ES5/6 array methods and idioms, like `map`, `filter`, `reduce`, `from`, `of` and iterating arrays using `for..of`*

- [fundamentals](#)
- [functions](#)

# 31 : Part 3 : .map, .reduce & .filter, Oh My!

<http://www.datchley.name/working-with-collections/>

Part 3 of a series on Javascript Fundamentals. See the [full list of posts](#)

Making use of Javascript's `map()`, `reduce()` and `filter()` Array methods can help you write more declarative, fluent style code. Rather than building up `for` loops and nesting to process data in lists and collections, you can take advantage of these methods to better organize your logic into building blocks of functions, and then chain them to create more readable and understandable implementations. And ES6 gives us a few more nice array methods as well, like `.from`, `.find`, `.findIndex`, `.of` and `for..of` loops!

## Array.map

There are very common things we want to do to lists. The first one that comes to mind is cycling through the list and performing an operation on each item. You can do this with `Array.forEach`.

```
var numbers = [1,2,3,4,5,6,7,8,9,10];
numbers.forEach((n, index) => {
  numbers[index] = n + 1;
});
// => [2,3,4,5,6,7,8,9,10,11]
```

Here, we're looping through the array and adding one to each item. This approach also has the side effect that it mutates the original list of data. As good developers, we want to reduce side effects and be *transparent* and *idempotent* with our functions and processing.

So, let's use `map()` to perform the same operation and leave the original list in tact.

```
var plusone = numbers.map((n) => n+1);
// => numbers: [1,2,3,4,5,6,7,8,9,10]
// => plusone: [2,3,4,5,6,7,8,9,10,11]
```

Easy enough. No side-effects and we have a new list with exactly what we wanted.

Now, what if we then want to take just the *even* numbers from this result in a list?

## Array.filter

We can use `Array.filter()` to visit each item in a list, much like `map()`; however, the predicate function you pass to `filter()` should return either `true`, to allow that item in the list, or `false` to skip it. Also like

`map()`, `filter()` returns a new array with copies of the items that match the filter and does not modify the original.

```
var evens = plusone.filter((n) => n % 2 === 0);
// => evens: [2,4,6,8,10]
```

Even easier! Now that we've got the list we want to work with, lets get a count of the number of how many are evenly divisible by 4.

## Array.reduce

When we want to aggregate data in a list, or data related to a list, we can use `Array.reduce()`. `reduce()` applies a function to an accumulated value on each element in a list from left to right.

```
var byfour = evens.reduce((groups, n) => {
  let key = n % 4 == 0 ? 'yes' : 'no';
  (groups[key] = groups[key] || []).push(n);
  return groups;
}, {});
// => byfour: { 'yes': [4,8], 'no': [2,6,10] }
```

Unlike `map()` and `filter()`, however, `reduce()` doesn't return a new list, it returns the aggregate value directly. In the case of our number list, our accumulated value was the initial empty object passed as the last parameter to the `reduce()` call. We then used that to create a property based on the divisible-by-four-ness of each number that served as a bucket to put the numbers in.

## Being fluent-ish

A fluent API or interface is one that provides better readability through:

- chaining of method calls over some base context
- defining operations via the return value of each called method
- is self-referential, where the new context is equivalent to the last
- terminates via return of a void context

JQuery works like this, as well as lodash and underscore using the `_.chain()` method wrapper, allowing chaining of method calls on a base context that represents a DOM element tree.

Because `map()` and `filter()` return new arrays, we can take advantage of this and chain multiple array operations together.

```
[1,2,3,4,5,6,7,8,9,10]
.map((n) => n*2)
```

```
.filter((n) => 10 % n == 0)
  .reduce((sum, n) => (sum += n), 0);
// => 12
```

Now, this isn't a "real" fluent interface; but it does resemble one from a chaining perspective and gives us a more declarative approach to implementing operations on lists.

## (Some) Newer ES6 Array methods

ES6 gives us some new Array methods that make some things easier compared to what we previously had to do in ES5, as well as adding some extra functionality.

### Array.from()

The new `.from()` method is a static method of `Array` and allows us to create real arrays from "array-like" objects (such as `arguments` or `dom collections`); and it also allows us to pass a function to apply to items in those arrays, giving us some `.map()`-like behavior as well.

For instance, we can create a real array from a DOM collection, which isn't really an instance of `Array` but is array like in that it allows indexing and has a `length`.

```
var divs = document.querySelectorAll('div.pane');
var text = Array.from(divs, (d) => d.textContent);
console.log("div text:", text);
```

The above snippet takes the DOM collection returned from `querySelectorAll()` and uses `Array.from` to map across each item and return us a "real" array of the text content from those DOM elements. To use it with the `arguments` variable in functions is just as easy.

```
// Old, ES5 way to get array from arguments
function() {
  var args = [].slice.call(arguments);
  //...
}

// Using ES6 Array.from
function() {
  var args = Array.from(arguments);
  //...
}
```

You can also use `Array.from` for other cool things, like filling in holes in arrays, since `Array.from`'s map function is passed `undefined` for any empty indexes in the array-like value being operated on.

```
var filled = Array.from([1,,2,,3], (n) => n || 0);
console.log("filled:", filled);
// => [1,0,2,0,3]
```

## `Array.find()` and `Array.findIndex()`

ES5 gave us `Array.filter()` which we've seen and is fantastic for filtering out the elements in an array. But, to find an element by value in an array in ES5, we'd have had to resort to something like the following. To loop over an array and use more complex logic to find a value, we'd have to use a `for` loop.

```
function find(list, value) {
  var index = list.indexOf(value);
  return index == -1 ? undefined : list[index];
}

var arr = ['cat','dog','bat','badger','cow'];
console.log("dog? ", find(arr, 'dog'));
// dog? dog
console.log("weasle? ", find(arr, 'weasle'));
// weasle? undefined

var found;
// Find the first item longer than 3 characters
for (var i=0; i < arr.length; i++) {
  if (arr[i].length > 3) {
    found = arr[i];
    break;
  }
}
// found: 'badger'
```

But, ES6 allows us to do the same thing using `Array.find()` to get a value, if it exists in an array; and `Array.findIndex()` to get the index of something by value. However, instead of taking a value directly as a parameter, `find()` and `findIndex` take a predicate function that is applied to each element in the array, and once the predicate returns true, stops the search and returns the value at that position (*for .find*) or the index at that position (*for findIndex*).

```
// Array.find and Array.findIndex(fn)
var found = [1,4,-5,10].find((n) => n < 0);
```

```

console.log("found:", found);

var index = [1,4,-5,10].findIndex((n) => n < 0);
console.log("index:", index);

// found: -5
// index: 2

```

## Array.of()

`Array.of()` lets us create a new Array instance from a variable number of arguments, regardless of the type of those arguments.

```

var arr = Array.of(0,true, undefined, null);
console.log("arr:", arr);
// arr: [0, true, undefined, null]

```

This is essentially the same as the following ES5 function:

```

function ArrayOf(){
  return [].slice.call(arguments);
}

```

`Array.of()` works more consistently at creating instances of Arrays than using the `Array()` constructor. The `Array()` constructor can run into issues if a single argument is passed and it's a number.

```

console.log(new Array(3, -5, 10));
// [3, -5, 10] - an array with the arguments as entries
console.log(new Array(3));
// [,,] - an array with three empty holes
console.log(new Array(5.7));
// RangeError: invalid array length - woops!

```

## The `for..of` loop

The `for..of` loop creates a loop over any *iterable* object, including `Array`, `arguments`, `Set`, `Map` and custom iterable objects like `generators`. This is different than the `for..in` operator, as `for..in` iterates over an Array, you get indexes, not values. `for..in` can be used on Objects, but returns the property names, not values. `for..of` can not be used on `Objects`, as there is no default iterator defined for `Objects` in javascript.

```

var arr = [3,5,7,9],
    obj = { a: 1, b: 2, c: 3 };

// ES5 for..in over objects
for (var p in obj) {
    console.log(p);
}
// a  b  c

// ES5 for..in over arrays
for (var n in arr) {
    console.log(n);
}
// 0 1 2 3

// ES6 for..of over arrays
for (let n of arr) {
    console.log(n);
}
// 3 5 7 9

```

Using `for..of`, we can now actually iterate values on Arrays. Though we still can't use it on `Object`s, we can use a generator and `for..of` to loop over the keys and values in an object.

```

// using a generator function
function* entries(obj) {
    for (let key of Object.keys(obj)) {
        yield [key, obj[key]];
    }
}

for (let [key, value] of entries(obj)) {
    console.log(key, "->", value);
}
// a -> 1
// b -> 2
// c -> 3

```

Iterable objects and generators are, however, a topic for a different blog post!

If you're looking for more information on some of the ES6 Array methods, be sure and check out [Axel Rauschmayer's Post](#) and [Hemanth HM's post](#), and be sure and look at [MDN's entry](#) on `for..of`.

---

*In Part 4 in the 'fundamentals' series I'll cover some of the basics of ~~iterables and generators~~ (a bit early for this) Objects, prototypes, delegation and composition, including `Object.create()`.*

- [fundamentals](#)
- [javascript](#)
- [collections](#)

## 32 : Part 4 : Object-ively Javascript

<http://www.datchley.name/delegate-object-to-classical-inheritance/>

Part 4 of a series on Javascript Fundamentals. See the [full list of posts](#)

The goal of this post is to, at a high level, cover some of the basics of creating, using and implementing objects in Javascript. In a second part, we'll follow up with understanding Javascript object prototype, the prototype chain and making better use of composition and delegation over inheritance.

*I'll mention objects being "prototype linked" to other objects in this article quite often. If you're still confused by Javascript's prototype based object delegation scheme, just hold tight for the second part coming soon*

### Making Objects

The easiest way to create objects in Javascript, and the most efficient, is using object literals.

```
var actor = {
    name: "David Tenant",
    age: 44
};
```

Here, `actor` is an object with two properties, `name`, which contains a string, and `age` which contains a number. This is the same as using `var obj = new Object()` and then assigning properties. Both objects are prototype linked the standard `Object` prototype.

We can also create a function to make objects by calling it as a `constructor` function using the `new` operator.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
var actor = new Person("David Tenant", 44);
```

Using the `new` operator introduces some implicit operations, which we'll discuss shortly.

We also have `Object.create()` as well, though it works a bit differently.

`Object.create()` takes two parameters and returns a new object. This object is prototype linked to the first object passed as a parameter. The second parameter is an optional object containing property descriptors to assign directly on the newly returned object.

A property descriptor object is an object containing definitions for properties. The property names in this descriptor object are themselves objects which contain various attributes attributed to that property, such as:

`value`, `writable`, `configurable`, and `enumerable`.

- `value` (*any*) - is the value assigned to that property
- `writable` (*boolean*) - if false, means the value of that property can not be changed
- `enumerable` (*boolean*) - if true, means that property will be iterated over when a user does a `for..in` or similar operation.
- `configurable` (*boolean*) - if false, any attempts to delete the property or change its value will fail.

Try the following in Chrome DevTools to see how basic object properties are setup:

```
var obj = { a: 42 };
console.log("%o", Object.getOwnPropertyDescriptor(obj, 'a'));
// Object
//   configurable: true
//   enumerable: true
//   value: 42
//   writable: true
//   ...
```

Using `Object.create()` we can create objects using another object as a prototype (*an exemplar*); and an optional property descriptor object. Any properties on the exemplar are accessed via the object's prototype chain; and any properties in the descriptor object are *own properties* assigned directly to the new object returned.

```
// creates a new, empty object linked to Object.prototype
var obj = Object.create({});

// create a new, empty object not linked to Object.prototype
obj = Object.create(null);

var person_proto = {
    name: "", age: null
},
descriptor = {
    tardis: { value: "blue box" }
};

// creates a new object based on person_proto
var actor = Object.create(person_proto, descriptor)
actor.name = "David Tennant";
actor.age = 44;
// actor:
```

```
// name: "David Tennant"
// age: 44
// tardis: "blue box"
```

In ES6, we could use the following pattern in creating our `actor` instance instead:

```
var actor = Object.assign(Object.create(person_proto), {
  tardis: "blue box"
});
```

ES6's `Object.assign()` is similar to the many `extend()` style functions that libraries like jQuery and Underscore use to create/extend objects using one or more objects.

## Understanding what `new` does

There is much magic happening in a function invoked with the `new` operator. At its most basic, when using `new`, the function invocation is hijacked and given a new, empty object as context, which is assigned to the local variable `this` inside the function.

The expectation of functions called as constructors is that they do something with that context: assigning properties or otherwise manipulating it; and magically, that same context object is returned from the function.

Using `new` works similarly to the following function:

```
function _new(/* constructor, param, ... */){
  var args = [].slice.call(arguments),
    constructor = args.shift(),
    context = Object.create(constructor.prototype),
    res;

  res = constructor.apply(context, args);
  return (typeof res === 'object' && res !== null) ? res : context;
}

var actor = _new(Person, "David Tennant", 44);
```

This is very simplified, but it gives you an idea of what's going on.

But as [Eric Elliot](#), [Reginald Braithwaite](#) and [Kyle Simpson](#) point out using `new` and constructor functions can be problematic for a number of reasons.

“If a feature is sometimes dangerous, and there is a better option, then always use the better option.” -- Douglas Crockford

## Object Access

Objects have properties. Properties are named keys on the object that can hold values. Those values can be basic types like strings, numbers, booleans or even object types like arrays, functions or other objects stored as references. The easiest way to access properties on an object is using the `.` operator.

```
var obj = {
  foo: "bar",
  baz: 42,
  print: function(s){ console.log(s); }
};

obj.foo;
obj.print("hi");
```

You can also access object properties using the `[]` operator. The `[]` operator expects a string argument that identifies the property key you're trying to access. Any non-string value will be coerced to a string and then used.

```
var obj = {};
obj["name"] = "David Tennant";
obj["age"] = 44;
obj["33"] = "what?"; // can only access using [] operator
obj[true] = "true"; // coercion to property named "true"

obj.33; // syntax error!
obj["33"]; // "what?"
obj["true"]; // "true"
obj.true; // "true"
```

This makes it easy to dynamically access properties at runtime by storing or building the property names.

ES6 gives us a way to specify properties as shorthand, as well as computed property names. For example,

```
let age = 44;
let obj = {
  name: "David Tennant",
  // short for, who: function(){...}
  who() { console.log("the doctor"); },
  // short for age: age
  age,
```

```
// computed property names!
["hello_" + (() => "sweetie")()): "the wife"
};

obj.who(); // "the doctor"
console.log(obj.hello_sweetie); // "the wife"
```

I'll point out some more resources on the new ES6 object features at the end of this post. Outside of computed property names, most of the object literal shorthand notation is just syntactic sugar to make declaring and working with objects in Javascript easier.

## Iterating Objects

Now that our objects have all these properties, maybe we want to iterate through them. Objects themselves are not *iterable objects* like arrays - they don't have an iterator defined for them, as access isn't as straight forward as one might think - you might iterate over the property names or property values; and what about prototype linked objects?

You can iterate over objects using the special `for..in` loop, which does iterate over the object's property keys, not it's values. But, `for..in` will iterate over prototype linked, enumerable properties as well, so be sure and use a `.hasOwnProperty()` to help limit the scope and recursion into the prototype chain of the object you're iterating.

```
var obj = { a: 1, b: 2 },
  obj2 = Object.create(obj, {
    c: { value: 3, enumerable: true },
    d: { value: 4, enumerable: true }
  });

for (var prop in obj2) {
  console.log(prop);
}

// a b c d

for (var prop in obj2) {
  if (obj2.hasOwnProperty(prop)) {
    console.log(prop);
  }
}

// c d
```

You can get access to an object's property keys using `Object.keys()`. The benefit of which, is that it only returns enumerable properties directly on the object in question - not prototype linked, enumerable

properties.

```
Object.keys(obj2).forEach(function(p) {  
    console.log(p + " = " + obj2[p]);  
});  
// c = 3  
// d = 4
```

For information on ES6's new object shorthand properties and computed properties, check out [Strongloop's article](#) and [MDN](#).

Also take a look at Kyle Simpson's [You Don't Know JS: this & Object properties](#) for more details about Javascript objects, `new` and more.

*In the next article I'll cover specifically Javascript objects and prototypes and talk about why you should be using composition and delegation instead of trying to mimic "class" style inheritance with `new` and constructor functions.*

- [fundamentals](#)
- [javascript](#)

## 33 : Chapter 4: Currying

<https://drboolean.gitbooks.io/mostly-adequate-guide/content/ch4.html>

### Can't live if livin' is without you

My Dad once explained how there are certain things one can live without until one acquires them. A microwave is one such thing. Smart phones, another. The older folks among us will remember a fulfilling life sans internet. For me, currying is on this list.

The concept is simple: You can call a function with fewer arguments than it expects. It returns a function that takes the remaining arguments.

[Deepak Anand](#)

"call a function with fewer arguments than it expects"

Isnt that partial application?

[Derek Hannah](#)

nope. Currying always produces nested unary (1-ary) functions. The transformed function is still largely the same as the original. Partial application produces functions of arbitrary arity. The transformed function is different from the original – it needs less arguments.

[Derek Hannah](#)

<http://www.2ality.com/2011/09/currying-vs-part-eval.html>

[Sign in to comment](#)

You can choose to call it all at once or simply feed in each argument piecemeal.

```
var add = function(x) {  
    return function(y) {  
        return x + y;  
    };  
};  
  
var increment = add(1);  
var addTen = add(10);  
  
increment(2);  
// 3
```

```
addTen(2);
// 12
```

Here we've made a function `add` that takes one argument and returns a function. By calling it, the returned function remembers the first argument from then on via the closure. Calling it with both arguments all at once is a bit of a pain, however, so we can use a special helper function called `curry` to make defining and calling functions like this easier.

### [Boaz Blake](#)

By calling it, the returned function remembers the first argument from then on via the closure

[Sign in to comment](#)

Let's set up a few curried functions for our enjoyment.

```
var curry = require('lodash/curry');

var match = curry(function(what, str) {
  return str.match(what);
});

var replace = curry(function(what, replacement, str) {
  return str.replace(what, replacement);
});

var filter = curry(function(f, ary) {
  return ary.filter(f);
});

var map = curry(function(f, ary) {
  return ary.map(f);
});
```

The pattern I've followed is a simple, but important one. I've strategically positioned the data we're operating on (String, Array) as the last argument. It will become clear as to why upon use.

```
match(/\s+/g, 'hello world');
// [ ' ' ]

match(/\s+/g)('hello world');
// [ ' ' ]

var hasSpaces = match(/\s+/g);
```

```
// function(x) { return x.match(/\s+/g) }

hasSpaces('hello world');
// [ ' ' ]

hasSpaces('spaceless');
// null

filter(hasSpaces, ['tori_spelling', 'tori amos']);
// ['tori amos']

var findSpaces = filter(hasSpaces);
// function(xs) { return xs.filter(function(x) { return x.match(/\s+/g) }) }

findSpaces(['tori_spelling', 'tori amos']);
// ['tori amos']

var noVowels = replace(/[aeiouy]/ig);
// function(replacement, x) { return x.replace(/[aeiouy]/ig, replacement) }

var censored = noVowels("*");
// function(x) { return x.replace(/[aeiouy]/ig, '*') }

censored('Chocolate Rain');
// 'Ch*c*l*t* R**n'
```

What's demonstrated here is the ability to "pre-load" a function with an argument or two in order to receive a new function that remembers those arguments.

I encourage you to `npm install lodash`, copy the code above and have a go at it in the REPL. You can also do this in a browser where lodash or ramda is available.

## More than a pun / special sauce

Currying is useful for many things. We can make new functions just by giving our base functions some arguments as seen in `hasSpaces`, `findSpaces`, and `censored`.

We also have the ability to transform any function that works on single elements into a function that works on arrays simply by wrapping it with `map`:

```
var getChildren = function(x) {
  return x.childNodes;
};
```

```
var allTheChildren = map(getChildren);
```

Giving a function fewer arguments than it expects is typically called *partial application*. Partially applying a function can remove a lot of boiler plate code. Consider what the above `allTheChildren` function would be with the uncurried `map` from lodash (note the arguments are in a different order):

```
var allTheChildren = function(elements) {
  return _.map(elements, getChildren);
};
```

We typically don't define functions that work on arrays, because we can just call `map(getChildren)` inline. Same with `sort`, `filter`, and other higher order functions(Higher order function: A function that takes or returns a function).

When we spoke about *pure functions*, we said they take 1 input to 1 output. Currying does exactly this: each single argument returns a new function expecting the remaining arguments. That, old sport, is 1 input to 1 output.

No matter if the output is another function - it qualifies as pure. We do allow more than one argument at a time, but this is seen as merely removing the extra `()`'s for convenience.

## In summary

Currying is handy and I very much enjoy working with curried functions on a daily basis. It is a tool for the belt that makes functional programming less verbose and tedious.

We can make new, useful functions on the fly simply by passing in a few arguments and as a bonus, we've retained the mathematical function definition despite multiple arguments.

Let's acquire another essential tool called `compose`.

## [Chapter 5: Coding by Composing](#)

## Exercises

A quick word before we start. We'll use a library called [Ramda](#) which curries every function by default.

Alternatively you may choose to use [lodash/fp](#) which does the same and is written/maintained by the creator of lodash. Both will work just fine and it is a matter of preference.

There are [unit tests](#) to run against your exercises as you code them, or you can just copy-paste into a JavaScript REPL for the early exercises if you wish.

Answers are provided with the code in the [repository for this book](#). Best way to do the exercises is with an [immediate feedback loop](#).

```
var _ = require('ramda');

// Exercise 1
//=====
// Refactor to remove all arguments by partially applying the function.

var words = function(str) {
  return _.split(' ', str);
};

// Exercise 1a
//=====
// Use map to make a new words fn that works on an array of strings.

var sentences = undefined;

// Exercise 2
//=====
// Refactor to remove all arguments by partially applying the functions.

var filterQs = function(xs) {
  return _.filter(function(x) {
    return match(/q/i, x);
  }, xs);
};

// Exercise 3
//=====
// Use the helper function _keepHighest to refactor max to not reference any
// arguments.

// LEAVE BE:
var _keepHighest = function(x, y) {
  return x >= y ? x : y;
};

// REFACTOR THIS ONE:
var max = function(xs) {
```

```
return _.reduce(function(acc, x) {
    return _keepHighest(acc, x);
}, -Infinity, xs);
};

// Bonus 1:
// =====
// Wrap array's slice to be functional and curried.
// // [1, 2, 3].slice(0, 2)
var slice = undefined;

// Bonus 2:
// =====
// Use slice to define a function "take" that takes n elements from the beginning
// of the string. Make it curried.
// // Result for "Something" with n=4 should be "Some"
var take = undefined;
```

## 34 : Chapter 5: Coding by Composing

<https://drboolean.gitbooks.io/mostly-adequate-guide/content/ch5.html>

### Functional husbandry

Here's `compose`:

```
var compose = function(f, g) {
  return function(x) {
    return f(g(x));
  };
};
```

`f` and `g` are functions and `x` is the value being "piped" through them.

Composition feels like function husbandry. You, breeder of functions, select two with traits you'd like to combine and mash them together to spawn a brand new one. Usage is as follows:

```
var toUpperCase = function(x) {
  return x.toUpperCase();
};

var exclaim = function(x) {
  return x + '!';
};

var shout = compose(exclaim, toUpperCase);

shout("send in the clowns");
//=> "SEND IN THE CLOWNS!"
```

The composition of two functions returns a new function. This makes perfect sense: composing two units of some type (in this case function) should yield a new unit of that very type. You don't plug two legos together and get a lincoln log. There is a theory here, some underlying law that we will discover in due time.

In our definition of `compose`, the `g` will run before the `f`, creating a right to left flow of data. This is much more readable than nesting a bunch of function calls. Without `compose`, the above would read:

```
var shout = function(x) {
  return exclaim(toUpperCase(x));
};
```

Instead of inside to outside, we run right to left, which I suppose is a step in the left direction(boo). Let's look at an example where sequence matters:

```
var head = function(x) {
    return x[0];
};

var reverse = reduce(function(acc, x) {
    return [x].concat(acc);
}, []);

var last = compose(head, reverse);

last(['jumpkick', 'roundhouse', 'uppercut']);
//=> 'uppercut'
```

`reverse` will turn the list around while `head` grabs the initial item. This results in an effective, albeit inefficient, `last` function. The sequence of functions in the composition should be apparent here. We could define a left to right version, however, we mirror the mathematical version much more closely as it stands. That's right, composition is straight from the math books. In fact, perhaps it's time to look at a property that holds for any composition.

```
// associativity
var associative = compose(f, compose(g, h)) == compose(compose(f, g), h);
// true
```

Composition is associative, meaning it doesn't matter how you group two of them. So, should we choose to uppercase the string, we can write:

```
compose(toUpperCase, compose(head, reverse));

// or

compose(compose(toUpperCase, head), reverse);
```

Since it doesn't matter how we group our calls to `compose`, the result will be the same. That allows us to write a variadic compose and use it as follows:

```
// previously we'd have to write two composes, but since it's associative, we can
give compose as many fn's as we like and let it decide how to group them.

var lastUpper = compose(toUpperCase, head, reverse);

lastUpper(['jumpkick', 'roundhouse', 'uppercut']);
```

```
//=> 'UPPERCUT'

var loudLastUpper = compose(exclaim, toUpperCase, head, reverse);

loudLastUpper(['jumpkick', 'roundhouse', 'uppercut']);
//=> 'UPPERCUT!'
```

Applying the associative property gives us this flexibility and peace of mind that the result will be equivalent. The slightly more complicated variadic definition is included with the support libraries for this book and is the normal definition you'll find in libraries like [lodash](#), [underscore](#), and [ramda](#).

One pleasant benefit of associativity is that any group of functions can be extracted and bundled together in their very own composition. Let's play with refactoring our previous example:

```
var loudLastUpper = compose(exclaim, toUpperCase, head, reverse);

// or

var last = compose(head, reverse);
var loudLastUpper = compose(exclaim, toUpperCase, last);

// or

var last = compose(head, reverse);
var angry = compose(exclaim, toUpperCase);
var loudLastUpper = compose(angry, last);

// more variations...
```

There's no right or wrong answers - we're just plugging our legos together in whatever way we please.

Usually it's best to group things in a reusable way like `last` and `angry`. If familiar with Fowler's ["Refactoring"](#), one might recognize this process as "[extract method](#)"...except without all the object state to worry about.

## Pointfree

Pointfree style means never having to say your data. Excuse me. It means functions that never mention the data upon which they operate. First class functions, currying, and composition all play well together to create this style.

```
//not pointfree because we mention the data: word
var snakeCase = function(word) {
  return word.toLowerCase().replace(/\s+/ig, '_');
```

```

};

//pointfree
var snakeCase = compose(replace(/\s+/ig, '_'), toLowerCase);

```

See how we partially applied `replace`? What we're doing is piping our data through each function of 1 argument. Currying allows us to prepare each function to just take its data, operate on it, and pass it along. Something else to notice is how we don't need the data to construct our function in the pointfree version, whereas in the pointful one, we must have our `word` available before anything else.

Let's look at another example.

```

//not pointfree because we mention the data: name
var initials = function(name) {
  return name.split(' ').map(compose(toUpperCase, head)).join('. ');
};

//pointfree
var initials = compose(join('. '), map(compose(toUpperCase, head)), split(' '));

initials("hunter stockton thompson");
// 'H. S. T'

```

Pointfree code can again, help us remove needless names and keep us concise and generic. Pointfree is a good litmus test for functional code as it lets us know we've got small functions that take input to output. One can't compose a while loop, for instance. Be warned, however, pointfree is a double-edged sword and can sometimes obfuscate intention. Not all functional code is pointfree and that is O.K. We'll shoot for it where we can and stick with normal functions otherwise.

## Debugging

A common mistake is to compose something like `map`, a function of two arguments, without first partially applying it.

```

//wrong - we end up giving angry an array and we partially applied map with god
knows what.

var latin = compose(map, angry, reverse);

latin(['frog', 'eyes']);
// error

```

```
// right - each function expects 1 argument.
var latin = compose(map(angry), reverse);

latin(['frog', 'eyes']);
// ['EYES!', 'FROG!']
```

If you are having trouble debugging a composition, we can use this helpful, but impure trace function to see what's going on.

```
var trace = curry(function(tag, x) {
  console.log(tag, x);
  return x;
});

var dasherize = compose(join('-'), toLower, split(' '), replace(/\s{2,}/ig, ' '));

dasherize('The world is a vampire');
// TypeError: Cannot read property 'apply' of undefined
```

Something is wrong here, let's `trace`

```
var dasherize = compose(join('-'), toLower, trace('after split'), split(' '),
replace(/\s{2,}/ig, ' '));
// after split [ 'The', 'world', 'is', 'a', 'vampire' ]
```

Ah! We need to `map` this `toLower` since it's working on an array.

```
var dasherize = compose(join('-'), map(toLower), split(' '), replace(/\s{2,}/ig, ' '));

dasherize('The world is a vampire');

// 'the-world-is-a-vampire'
```

The `trace` function allows us to view the data at a certain point for debugging purposes. Languages like haskell and purescript have similar functions for ease of development.

Composition will be our tool for constructing programs and, as luck would have it, is backed by a powerful theory that ensures things will work out for us. Let's examine this theory.

## Category theory

Category theory is an abstract branch of mathematics that can formalize concepts from several different branches such as set theory, type theory, group theory, logic, and more. It primarily deals with objects, morphisms, and transformations, which mirrors programming quite closely. Here is a chart of the same concepts as viewed from each separate theory.



Sorry, I didn't mean to frighten you. I don't expect you to be intimately familiar with all these concepts. My point is to show you how much duplication we have so you can see why category theory aims to unify these things.

In category theory, we have something called... a category. It is defined as a collection with the following components:

- A collection of objects
- A collection of morphisms
- A notion of composition on the morphisms
- A distinguished morphism called identity

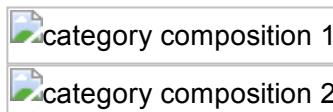
Category theory is abstract enough to model many things, but let's apply this to types and functions, which is what we care about at the moment.

**A collection of objects** The objects will be data types. For instance, `String`, `Boolean`, `Number`, `Object`, etc. We often view data types as sets of all the possible values. One could look at `Boolean` as the set of `[true, false]` and `Number` as the set of all possible numeric values. Treating types as sets is useful because we can use set theory to work with them.

**A collection of morphisms** The morphisms will be our standard every day pure functions.

**A notion of composition on the morphisms** This, as you may have guessed, is our brand new toy - `compose`. We've discussed that our `compose` function is associative which is no coincidence as it is a property that must hold for any composition in category theory.

Here is an image demonstrating composition:



Here is a concrete example in code:

```
var g = function(x) {  
    return x.length;  
};  
var f = function(x) {  
    return x === 4;
```

```

};

var isFourLetterWord = compose(f, g);

```

**A distinguished morphism called identity** Let's introduce a useful function called `id`. This function simply takes some input and spits it back at you. Take a look:

```

var id = function(x) {
  return x;
};

```

You might ask yourself "What in the bloody hell is that useful for?". We'll make extensive use of this function in the following chapters, but for now think of it as a function that can stand in for our value - a function masquerading as every day data.

`id` must play nicely with `compose`. Here is a property that always holds for every unary (unary: a one-argument function) function `f`:

```

// identity
compose(id, f) == compose(f, id) == f;
// true

```

Hey, it's just like the identity property on numbers! If that's not immediately clear, take some time with it. Understand the futility. We'll be seeing `id` used all over the place soon, but for now we see it's a function that acts as a stand in for a given value. This is quite useful when writing pointfree code.

So there you have it, a category of types and functions. If this is your first introduction, I imagine you're still a little fuzzy on what a category is and why it's useful. We will build upon this knowledge throughout the book. As of right now, in this chapter, on this line, you can at least see it as providing us with some wisdom regarding composition - namely, the associativity and identity properties.

What are some other categories, you ask? Well, we can define one for directed graphs with nodes being objects, edges being morphisms, and composition just being path concatenation. We can define with Numbers as objects and `>=` as morphisms (actually any partial or total order can be a category). There are heaps of categories, but for the purposes of this book, we'll only concern ourselves with the one defined above. We have sufficiently skimmed the surface and must move on.

## In Summary

Composition connects our functions together like a series of pipes. Data will flow through our application as it must - pure functions are input to output after all, so breaking this chain would disregard output, rendering our software useless.

We hold composition as a design principle above all others. This is because it keeps our app simple and reasonable. Category theory will play a big part in app architecture, modelling side effects, and ensuring correctness.

We are now at a point where it would serve us well to see some of this in practice. Let's make an example application.

## [Chapter 6: Example Application](#)

### Exercises

```
var _ = require('ramda');
var accounting = require('accounting');

// Example Data
var CARS = [
  {
    name: 'Ferrari FF',
    horsepower: 660,
    dollar_value: 700000,
    in_stock: true,
  },
  {
    name: 'Spyker C12 Zagato',
    horsepower: 650,
    dollar_value: 648000,
    in_stock: false,
  },
  {
    name: 'Jaguar XKR-S',
    horsepower: 550,
    dollar_value: 132000,
    in_stock: false,
  },
  {
    name: 'Audi R8',
    horsepower: 525,
    dollar_value: 114200,
    in_stock: false,
  },
  {
    name: 'Aston Martin One-77',
    horsepower: 750,
    dollar_value: 1850000,
    in_stock: true,
  },
  {
    name: 'Pagani Huayra',
  }
]
```

```
horsepower: 700,  
dollar_value: 1300000,  
in_stock: false,  
}];  
  
// Exercise 1:  
// =====  
// Use _.compose() to rewrite the function below. Hint: _.prop() is curried.  
var isLastInStock = function(cars) {  
  var last_car = _.last(cars);  
  return _.prop('in_stock', last_car);  
};  
  
// Exercise 2:  
// =====  
// Use _.compose(), _.prop() and _.head() to retrieve the name of the first car.  
var nameOfFirstCar = undefined;  
  
// Exercise 3:  
// =====  
// Use the helper function _average to refactor averageDollarValue as a  
composition.  
var _average = function(xs) {  
  return _.reduce(_.add, 0, xs) / xs.length;  
}; // <- leave be  
  
var averageDollarValue = function(cars) {  
  var dollar_values = _.map(function(c) {  
    return c.dollar_value;  
  }, cars);  
  return _average(dollar_values);  
};  
  
// Exercise 4:  
// =====  
// Write a function: sanitizeNames() using compose that returns a list of lowercase  
and underscored car's names: e.g: sanitizeNames([{name: 'Ferrari FF', horsepower:  
660, dollar_value: 700000, in_stock: true}]) //=> ['ferrari_ff'].
```

```
var _underscore = _.replace(/\W+/g, '_'); //--- leave this alone and use to
sanitize

var sanitizeNames = undefined;

// Bonus 1:
// =====
// Refactor availablePrices with compose.

var availablePrices = function(cars) {
  var available_cars = _.filter(_.prop('in_stock'), cars);
  return available_cars.map(function(x) {
    return accounting.formatMoney(x.dollar_value);
  }).join(', ');
};

// Bonus 2:
// =====
// Refactor to pointfree. Hint: you can use _.flip().

var fastestCar = function(cars) {
  var sorted = _.sortBy(function(car) {
    return car.horsepower;
  }, cars);
  var fastest = _.last(sorted);
  return fastest.name + ' is the fastest';
};
```

## 35 : Functional Mixins in ECMAScript 2015

<http://raganwald.com/2015/06/17/functional-mixins.html>

In [Prototypes are Objects](#), we saw that you can emulate “mixins” using `Object.assign` on the prototypes that underly JavaScript “classes.” We’ll revisit this subject now and spend more time looking at mixing functionality into classes.

First, a quick recap: In JavaScript, a “class” is implemented as a constructor function and its prototype, whether you write it directly, or use the `class` keyword. Instances of the class are created by calling the constructor with `new`. They “inherit” shared behaviour from the constructor’s `prototype` property.<sup>1</sup>

### the object mixin pattern

One way to share behaviour scattered across multiple classes, or to untangle behaviour by factoring it out of an overweight prototype, is to extend a prototype with a *mixin*.

Here's a class of todo items:

```
class Todo {
  constructor (name) {
    this.name = name || 'Untitled';
    this.done = false;
  }
  do () {
    this.done = true;
    return this;
  }
  undo () {
    this.done = false;
    return this;
  }
}
```

And a “mixin” that is responsible for colour-coding:

```
const Coloured = {
  setColourRGB ({r, g, b}) {
    this.colourCode = {r, g, b};
    return this;
  },
  getColourRGB () {
```

```

        return this.colourCode;
    }
};


```

Mixing colour coding into our Todo prototype is straightforward:

```

Object.assign(Todo.prototype, Coloured);

new Todo('test')
  .setColourRGB({r: 1, g: 2, b: 3})
//=> {"name":"test","done":false,"colourCode":{"r":1,"g":2,"b":3}}

```

We can “upgrade” it to have a [private property](#) if we wish:

```

const colourCode = Symbol("colourCode");

const Coloured = {
  setColourRGB ({r, g, b}) {
    this[colourCode] = {r, g, b};
    return this;
  },
  getColourRGB () {
    return this[colourCode];
  }
};

```

So far, very easy and very simple. This is a *pattern*, a recipe for solving a certain problem using a particular organization of code.



## functional mixins

The object mixin we have above works properly, but our little recipe had two distinct steps: Define the mixin and then extend the class prototype. Angus Croll pointed out that it's more elegant to define a mixin as a function rather than an object. He calls this a [functional mixin](#). Here's `Coloured` again, recast in functional form:

```

const Coloured = (target) =>
  Object.assign(target, {
    setColourRGB ({r, g, b}) {

```

```

        this.colourCode = {r, g, b};
        return this;
    },
    getColourRGB () {
        return this.colourCode;
    }
});

Coloured(Todo.prototype);

```

We can make ourselves a *factory function* that also names the pattern:

```

const FunctionalMixin = (behaviour) =>
    target => Object.assign(target, behaviour);

```

This allows us to define functional mixins neatly:

```

const Coloured = FunctionalMixin({
    setColourRGB ({r, g, b}) {
        this.colourCode = {r, g, b};
        return this;
    },
    getColourRGB () {
        return this.colourCode;
    }
});

```

## enumerability

If we look at the way `class` defines prototypes, we find that the methods defined are not enumerable by default. This works around a common error where programmers iterate over the keys of an instance and fail to test for `.hasOwnProperty`.

Our object mixin pattern does not work this way, the methods defined in a mixin *are* enumerable by default, and if we carefully defined them to be non-enumerable, `Object.assign` wouldn't mix them into the target prototype, because `Object.assign` only assigns enumerable properties.

And thus:

```
Coloured(Todo.prototype)
```

```
const urgent = new Todo("finish blog post");
urgent.setColourRGB({r: 256, g: 0, b: 0});

for (let property in urgent) console.log(property);
// =>
name
done
colourCode
setColourRGB
getColourRGB
```

As we can see, the `setColourRGB` and `getColourRGB` methods are enumerated, although the `do` and `undo` methods are not. This can be a problem with naïve code: we can't always rewrite all the *other* code to carefully use `.hasOwnProperty`.

One benefit of functional mixins is that we can solve this problem and transparently make mixins behave like `class`:

```
const FunctionalMixin = (behaviour) =>
function (target) {
  for (let property of Reflect.ownKeys(behaviour))
    Object.defineProperty(target, property, { value: behaviour[property] })
  return target;
}
```

Writing this out as a pattern would be tedious and error-prone. Encapsulating the behaviour into a function is a small win.

### Just Below the Surface

## mixin responsibilities

Like classes, mixins are metaobjects: They define behaviour for instances. In addition to defining behaviour in the form of methods, classes are also responsible for initializing instances. But sometimes, classes and metaobjects handle additional responsibilities.

For example, sometimes a particular concept is associated with some well-known constants. When using a class, can be handy to namespace such values in the class itself:

```
class Todo {
  constructor (name) {
    this.name = name || Todo.DEFAULT_NAME;
```

```

        this.done = false;
    }

    do () {
        this.done = true;
        return this;
    }

    undo () {
        this.done = false;
        return this;
    }
}

Todo.DEFAULT_NAME = 'Untitled';

// If we are sticklers for read-only constants, we could write:
// Object.defineProperty(Todo, 'DEFAULT_NAME', {value: 'Untitled'});

```

We can't really do the same thing with simple mixins, because all of the properties in a simple mixin end up being mixed into the prototype of instances we create by default. For example, let's say we want to define Coloured.RED, Coloured.GREEN, and Coloured.BLUE. But we don't want any specific coloured instance to define RED, GREEN, or BLUE.

Again, we can solve this problem by building a functional mixin. Our FunctionalMixin factory function will accept an optional dictionary of read-only mixin properties, provided they are associated with a special key:

```

const shared = Symbol("shared");

function FunctionalMixin (behaviour) {
    const instanceKeys = Reflect.ownKeys(behaviour)
        .filter(key => key !== shared);
    const sharedBehaviour = behaviour[shared] || {};
    const sharedKeys = Reflect.ownKeys(sharedBehaviour);

    function mixin (target) {
        for (let property of instanceKeys)
            Object.defineProperty(target, property, { value: behaviour[property] });
        return target;
    }
    for (let property of sharedKeys)
        Object.defineProperty(mixin, property, {
            value: sharedBehaviour[property],

```

```

        enumerable: sharedBehaviour.propertyIsEnumerable(property)
    });
    return mixin;
}

FunctionalMixin.shared = shared;

```

And now we can write:

```

const Coloured = FunctionalMixin({
    setColourRGB ({r, g, b}) {
        this.colourCode = {r, g, b};
        return this;
    },
    getColourRGB () {
        return this.colourCode;
    },
    [FunctionalMixin.shared]: {
        RED: { r: 255, g: 0, b: 0 },
        GREEN: { r: 0, g: 255, b: 0 },
        BLUE: { r: 0, g: 0, b: 255 },
    }
});

```

Coloured(Todo.prototype)

```

const urgent = new Todo("finish blog post");
urgent.setColourRGB(Coloured.RED);

urgent.getColourRGB()
//=> {"r":255,"g":0,"b":0}

```

## **mixin methods**

Such properties need not be values. Sometimes, classes have methods. And likewise, sometimes it makes sense for a mixin to have its own methods. One example concerns `instanceof`.

In earlier versions of ECMAScript, `instanceof` is an operator that checks to see whether the prototype of an instance matches the prototype of a constructor function. It works just fine with “classes,” but it does not work “out of the box” with mixins:

```

urgent instanceof Todo
//=> true

urgent instanceof Coloured
//=> false

```

To handle this and some other issues where programmers are creating their own notion of dynamic types, or managing prototypes directly with `Object.create` and `Object.setPrototypeOf`, ECMAScript 2015 provides a way to override the built-in `instanceof` behaviour: An object can define a method associated with a well-known symbol, `Symbol.hasInstance`.

We can test this quickly:<sup>2</sup>

```

Object.defineProperty(Coloured, Symbol.hasInstance, {value: (instance) => true});
urgent instanceof Coloured
//=> true
{} instanceof Coloured
//=> true

```

Of course, that is not semantically correct. But using this technique, we can write:

```

const shared = Symbol("shared");

function FunctionalMixin (behaviour) {
    const instanceKeys = Reflect.ownKeys(behaviour)
        .filter(key => key !== shared);
    const sharedBehaviour = behaviour[shared] || {};
    const sharedKeys = Reflect.ownKeys(sharedBehaviour);
    const typeTag = Symbol("isA");

    function mixin (target) {
        for (let property of instanceKeys)
            Object.defineProperty(target, property, { value: behaviour[property] });
        target[typeTag] = true;
        return target;
    }
    for (let property of sharedKeys)
        Object.defineProperty(mixin, property, {
            value: sharedBehaviour[property],
            enumerable: sharedBehaviour.propertyIsEnumerable(property)
        })
}

const Coloured = FunctionalMixin();

```

```

    });

Object.defineProperty(mixin, Symbol.hasInstance, {value: (instance) =>
!instance[typeTag]});

return mixin;

}

FunctionalMixin.shared = shared;

urgent instanceof Coloured
//=> true
{} instanceof Coloured
//=> false

```

Do you need to implement `instanceof`? Quite possibly not. “Rolling your own polymorphism” is usually a last resort. But it can be handy for writing test cases, and a few daring framework developers might be working on multiple dispatch and pattern-matching for functions.

## summary

The charm of the object mixin pattern is its simplicity: It really does not need an abstraction wrapped around an object literal and `Object.assign`.

However, behaviour defined with the mixin pattern is *slightly* different than behaviour defined with the `class` keyword. Two examples of these differences are enumerability and mixin properties (such as constants and mixin methods like `[Symbol.hasInstance]`).

Functional mixins provide an opportunity to implement such functionality, at the cost of some complexity in the `FunctionalMixin` function that creates functional mixins.

As a general rule, it's best to have things behave as similarly as possible in the domain code, and this sometimes does involve some extra complexity in the infrastructure code. But that is more of a guideline than a hard-and-fast rule, and for this reason there is a place for both the object mixin pattern *and* functional mixins in JavaScript.

(discuss on [hacker news](#) and [/r/javascript](#))

*follow-up:* [Purely Functional Composition](#)

more reading:

- [Prototypes are Objects \(and why that matters\)](#)
- [Classes are Expressions \(and why that matters\)](#)
- [Functional Mixins in ECMAScript 2015](#)
- [Using ES.later Decorators as Mixins](#)

- [Method Advice in Modern JavaScript](#)
- [super\(\) considered hmmm-fu!](#)
- [JavaScript Mixins, Subclass Factories, and Method Advice](#)
- [This is not an essay about 'Traits in Javascript'](#)

notes:

1. A much better way to put it is that objects with a prototype *delegate* behaviour to their prototype (and that may in turn delegate behaviour to its prototype if it has one, and so on). [←](#)
2. This may **not** work with various transpilers and other incomplete ECMAScript 2015 implementations. Check the documentation. For example, you must enable the “high compliancy” mode in [BabelJS](#). This is off by default to provide the highest possible performance for code bases that do not need to use features like this. [←](#)

## 36 : JavaScript Mixins, Subclass Factories, and Method Advice

<http://raganwald.com/2015/12/28/mixins-subclass-factories-and-method-advice.html>

*Mixins* solve a very common problem in class-centric OOP: For non-trivial applications, there is a messy many-to-many relationship between behaviour and classes, and it does not neatly decompose into a tree. In this essay, we only touch lightly over the benefits of using mixins with classes, and in their stead we will focus on some of the limitations of mixins and ways to not just overcome them, but create designs that are superior to those created with classes alone.

(For more on why mixins matter in the first place, you may want to review [Prototypes are Objects \(and why that matters\)](#), [Functional Mixins in ECMAScript 2015](#), and [Using ES.later Decorators as Mixins](#).)

### simple mixins

As noted above, for non-trivial applications, there is a messy many-to-many relationship between behaviour and classes. However, JavaScript's single-inheritance model forces us to organize behaviour in trees, which can only represent one-to-many relationships.

The mixin solution to this problem is to leave classes in a single inheritance hierarchy, and to mix additional behaviour into individual classes as needed. Here's a vastly simplified functional mixin for classes.<sup>1</sup>

```
function mixin (behaviour) {
  let instanceKeys = Reflect.ownKeys(behaviour);
  let typeTag = Symbol('isa');

  function _mixin (clazz) {
    for (let property of instanceKeys)
      Object.defineProperty(clazz.prototype, property, {
        value: behaviour[property],
        writable: true
      });
    Object.defineProperty(clazz.prototype, typeTag, { value: true });
    return clazz;
  }
  Object.defineProperty(_mixin, Symbol.hasInstance, {
    value: (i) => !!i[typeTag]
  });
  return _mixin;
}
```

This is more than enough to do a lot of very good work in JavaScript, but it's just the starting point. Here's how we put it to work:

```
let BookCollector = mixin({
  addToCollection (name) {
    this.collection().push(name);
    return this;
  },
  collection () {
    return this._collected_books || (this._collected_books = []);
  }
});

class Person {
  constructor (first, last) {
    this.rename(first, last);
  }
  fullName () {
    return this.firstName + " " + this.lastName;
  }
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
}

let Executive = BookCollector(
  class extends Person {
    constructor (title, first, last) {
      super(first, last);
      this.title = title;
    }

    fullName () {
      return `${this.title} ${super.fullName()}`;
    }
  }
);

let president = new Executive('President', 'Barak', 'Obama');
```

```

president

  .addToCollection("JavaScript Allongé")
  .addToCollection("Kestrels, Quirky Birds, and Hopeless Egocentricity");

president.collection()
//=> ["JavaScript Allongé", "Kestrels, Quirky Birds, and Hopeless Egocentricity"]

```

## multiple inheritance

If you want to mix behaviour into a class, mixins do the job very nicely. But sometimes, people want more. They want **multiple inheritance**. Meaning, what they really want is for class `Executive` to inherit from `Person` *and* from `BookCollector`.

What's the difference between `Executive` mixing `BookCollector` in and `Executive` inheriting from `BookCollector`?

1. If `Executive` mixes `BookCollector` in, the properties `addToCollection` and `collection` become own properties of `Executive`'s prototype. If `Executive` inherits from `BookCollector`, they don't.
2. If `Executive` mixes `BookCollector` in, `Executive` can't override methods of `BookCollector`. If `Executive` inherits from `BookCollector`, it can.
3. If `Executive` mixes `BookCollector` in, `Executive` can't override methods of `BookCollector`, and therefore it can't make a method that overrides a method of `BookCollector` and then uses `super` to call the original. If `Executive` inherits from `BookCollector`, it can.

If JavaScript had multiple inheritance, we could extend a class with more than one superclass:

```

class Todo {

  constructor (name) {
    this.name = name || 'Untitled';
    this.done = false;
  }

  do () {
    this.done = true;
    return this;
  }

  undo () {
    this.done = false;
  }
}

```

```
        return this;
    }

    toHTML () {
        return this.name; // highly insecure
    }
}

class Coloured {
    setColourRGB ({r, g, b}) {
        this.colourCode = {r, g, b};
        return this;
    }

    getColourRGB () {
        return this.colourCode;
    }
}

let yellow = {r: 'FF', g: 'FF', b: '00'},
    red    = {r: 'FF', g: '00', b: '00'},
    green  = {r: '00', g: 'FF', b: '00'},
    grey   = {r: '80', g: '80', b: '80'};

let oneDayInMilliseconds = 1000 * 60 * 60 * 24;

class TimeSensitiveTodo extends Todo, Coloured {
    constructor (name, deadline) {
        super(name);
        this.deadline = deadline;
    }

    getColourRGB () {
        let slack = this.deadline - Date.now();

        if (this.done) {
            return grey;
        }
        else if (slack <= 0) {
            return red;
        }
    }
}
```

```

    else if (slack <= oneDayInMilliseconds){
        return yellow;
    }
    else return green;
}

toHTML () {
    let rgb = this.getColourRGB();

    return `${super.toHTML()}`;
}
}

```

This hypothetical `TimeSensitiveTodo` extends both `Todo` and `Coloured`, and it overrides `toHTML` from `Todo` as well as overriding `getColourRGB` from `Coloured`.



## subclass factories

However, JavaScript does not have “true” multiple inheritance, and therefore this code does not work. But we can simulate multiple inheritance for cases like this. The way it works is to step back and ask ourselves, “What would we do if we didn’t have mixins or multiple inheritance?”

The answer is, we’d force a square multiple inheritance peg into a round single inheritance hole, like this:

```

class Todo {
    // ...
}

class ColouredTodo extends Todo {
    // ...
}

class TimeSensitiveTodo extends ColouredTodo {
    // ...
}

```

By making `ColouredTodo` extend `Todo`, `TimeSensitiveTodo` can extend `ColouredTodo` and override methods from both. This is exactly what most programmers do, and we know that it is an anti-pattern, as it leads to duplicated class behaviour and deep class hierarchies.

But.

What if, instead of manually creating this hierarchy, we use our simple mixins to do the work for us? We can take advantage of the fact that [classes are expressions](#), like this:

```
let Coloured = mixin({
  setColourRGB ({r, g, b}) {
    this.colourCode = {r, g, b};
    return this;
  },
  getColourRGB () {
    return this.colourCode;
  }
});

let ColouredTodo = Coloured(class extends Todo {});
```

Thus, we have a `ColouredTodo` that we can extend and override, but we also have our `Coloured` behaviour in a mixin we can use anywhere we like without duplicating its functionality in our code. The full solution looks like this:

```
class Todo {
  constructor (name) {
    this.name = name || 'Untitled';
    this.done = false;
  }

  do () {
    this.done = true;
    return this;
  }

  undo () {
    this.done = false;
    return this;
  }

  toHTML () {
    return this.name; // highly insecure
  }
}
```

```
}
```

```
let Coloured = mixin({  
    setColourRGB ({r, g, b}) {  
        this.colourCode = {r, g, b};  
        return this;  
    },  
  
    getColourRGB () {  
        return this.colourCode;  
    }  
});  
  
let ColouredTodo = Coloured(class extends Todo {});  
  
let yellow = {r: 'FF', g: 'FF', b: '00'},  
    red     = {r: 'FF', g: '00', b: '00'},  
    green   = {r: '00', g: 'FF', b: '00'},  
    grey    = {r: '80', g: '80', b: '80'};  
  
let oneDayInMilliseconds = 1000 * 60 * 60 * 24;  
  
class TimeSensitiveTodo extends ColouredTodo {  
    constructor (name, deadline) {  
        super(name);  
        this.deadline = deadline;  
    }  
  
    getColourRGB () {  
        let slack = this.deadline - Date.now();  
  
        if (this.done) {  
            return grey;  
        }  
        else if (slack <= 0) {  
            return red;  
        }  
        else if (slack <= oneDayInMilliseconds){  
            return yellow;  
        }  
        else return green;  
    }  
}
```

```

        }

    toHTML () {
        let rgb = this.getColourRGB();

        return `${super.toHTML()}`;
    }
}

let task = new TimeSensitiveTodo('Finish blog post', Date.now() +
oneDayInMilliseconds);

task.toHTML()
//=> Finish blog post

```

The key snippet is `let ColouredTodo = Coloured(class extends Todo {});`, it turns behaviour into a subclass that can be extended and overridden. We can turn this pattern into a function:

```

let subclassFactory = (behaviour) => {
    let mixBehaviourInto = mixin(behaviour);

    return (superclazz) => mixBehaviourInto(class extends superclazz {});
}

```

Using `subclassFactory`, we wrap the class we want to extend, instead of the class we are declaring. Like this:

```

let subclassFactory = (behaviour) => {
    let mixBehaviourInto = mixin(behaviour);

    return (superclazz) => mixBehaviourInto(class extends superclazz {});
}

let ColouredAsWellAs = subclassFactory({
    setColourRGB ({r, g, b}) {
        this.colourCode = {r, g, b};
        return this;
    },
    getColourRGB () {
        return this.colourCode;
    }
})

```

```

    }
});

class TimeSensitiveTodo extends ColouredAsWellAs(Todo) {
  constructor (name, deadline) {
    super(name);
    this.deadline = deadline;
  }

  getColourRGB () {
    let slack = this.deadline - Date.now();

    if (this.done) {
      return grey;
    }
    else if (slack <= 0) {
      return red;
    }
    else if (slack <= oneDayInMilliseconds){
      return yellow;
    }
    else return green;
  }

  toHTML () {
    let rgb = this.getColourRGB();

    return `${super.toHTML()}`;
  }
}

```

The syntax of `class TimeSensitiveTodo extends ColouredAsWellAs(Todo)` says exactly what we mean: We are extending our `Coloured` behaviour as well as extending `Todo`.<sup>2</sup>

## another way forward

The solution subclass factories offer is emulating inheritance from more than one superclass. That, in turn, makes it possible to override methods from our superclass as well as the behaviour we want to mix in. Which is fine, but we don't actually want multiple inheritance!

It's just that we're looking at an overriding/extending methods problem, but we're holding an inheritance-shaped hammer. So it looks like a multiple-inheritance nail. But what if we address the problem of overriding

and extending methods directly, rather than indirectly via multiple inheritance?



## simple overwriting with simple mixins

We start by noting that in the first pass of our `mixin` function, we blindly copied properties from the mixin into the class's prototype, whether the class defined those properties or not. So if we write:

```
let RED      = { r: 'FF', g: '00', b: '00' },
    WHITE    = { r: 'FF', g: 'FF', b: 'FF' },
    ROYAL_BLUE = { r: '41', g: '69', b: 'E1' },
    LIGHT_BLUE = { r: 'AD', g: 'D8', b: 'E6' };

let BritishRoundel = mixin({
  shape () {
    return 'round';
  },

  roundels () {
    return [RED, WHITE, ROYAL_BLUE];
  }
});

let CanadianAirForceRoundel = BritishRoundel(class {
  roundels () {
    return [RED, WHITE, LIGHT_BLUE];
  }
});

new CanadianAirForceRoundel().roundels()
//=> [
  {"r":"FF","g":"00","b":"00"},
  {"r":"FF","g":"FF","b":"FF"},
  {"r":"41","g":"69","b":"E1"}
]
```

Our `CanadianAirForceRoundel`'s third stripe winds up being regular blue instead of light blue, because the `roundels` method from the mixin `BritishRoundel` overwrites its own. (Yes, this is a ridiculous example, but it gets the point across.)

We can fix this by not overwriting a property if the class already defines it. That's not so hard:

```

function mixin (behaviour) {
  let instanceKeys = Reflect.ownKeys(behaviour);
  let typeTag = Symbol('isa');

  function _mixin (clazz) {
    for (let property of instanceKeys)
      if (!clazz.prototype.hasOwnProperty(property)) {
        Object.defineProperty(clazz.prototype, property, {
          value: behaviour[property],
          writable: true
        });
      }
    Object.defineProperty(clazz.prototype, typeTag, { value: true });
    return clazz;
  }
  Object.defineProperty(_mixin, Symbol.hasInstance, {
    value: (i) => !!i[typeTag]
  });
  return _mixin;
}

```

Now we can override `roundels` in `CanadianAirForceRoundel` while mixing `shape` in just fine:

```

new CanadianAirForceRoundel().roundels()
//=> [
  {"r":"FF","g":"00","b":"00"},
  {"r":"FF","g":"FF","b":"FF"},
  {"r":"AD","g":"D8","b":"E6"}
]

```

The method defined in the class is now the “definition of record,” just as we might expect. But it’s not enough in and of itself.

## combining advice with simple mixins

The above adjustment to ‘mixin’ is fine for simple overwriting, but what about when we wish to modify or extend a method’s behaviour while still invoking the original? Recall that our `TimeSensitiveTodo` example performed a simple override of `getColourRGB`, but its implementation of `toHTML` used `super` to invoke the method it was overriding.

Our adjustment will not allow a method in the class to invoke the body of a method in a mixin. So we can't use it to implement `TimeSensitiveTodo`. For that, we need a different tool, [method advice](#).

Method advice is a powerful tool in its own right: It allows us to compose method functionality in a declarative way. Here's a simple "override" function that decorates a class:

```
let override = (behaviour, ...overriddenMethodNames) =>
  (clazz) => {
    if (typeof behaviour === 'string') {
      behaviour = clazz.prototype[behaviour];
    }
    for (let overriddenMethodName of overriddenMethodNames) {
      let overriddenMethodFunction = clazz.prototype[overriddenMethodName];

      Object.defineProperty(clazz.prototype, overriddenMethodName, {
        value: function (...args) {
          return behaviour.call(this, overriddenMethodFunction.bind(this),
...args);
        },
        writable: true
      });
    }
    return clazz;
};
```

It takes behaviour in the form of a name of a method or a function, and one or more names of methods to override. It overrides each of the methods with the behaviour, which is invoked with the overridden method's function as the first argument.

This allows us to invoke the original without needing to use `super`. And although we don't show all the other use cases here, it is handy for far more than overriding mixin methods, it can be used to decompose methods into separate responsibilities.

Using `override`, we can decorate methods with any arbitrary functionality. We'd use it like this:

```
class Todo {
  constructor (name) {
    this.name = name || 'Untitled';
    this.done = false;
  }

  do () {
```

```
this.done = true;
return this;
}

undo () {
  this.done = false;
  return this;
}

toHTML () {
  return this.name; // highly insecure
}
}

let Coloured = mixin({
  setColourRGB ({r, g, b}) {
    this.colourCode = {r, g, b};
    return this;
  },
  getColourRGB () {
    return this.colourCode;
  }
});

let yellow = {r: 'FF', g: 'FF', b: '00'},
  red    = {r: 'FF', g: '00', b: '00'},
  green  = {r: '00', g: 'FF', b: '00'},
  grey   = {r: '80', g: '80', b: '80'};

let oneDayInMilliseconds = 1000 * 60 * 60 * 24;

let TimeSensitiveTodo = override('wrapWithColour', 'toHTML')(
  Coloured(
    class extends Todo {
      constructor (name, deadline) {
        super(name);
        this.deadline = deadline;
      }

      getColourRGB () {

```

```

let slack = this.deadline - Date.now();

if (this.done) {
    return grey;
}
else if (slack <= 0) {
    return red;
}
else if (slack <= oneDayInMilliseconds){
    return yellow;
}
else return green;
}

wrapWithColour (original) {
    let rgb = this.getColourRGB();

    return `${original()}`;
}
}

);

let task = new TimeSensitiveTodo('Finish blog post', Date.now() +
oneDayInMilliseconds);

task.toHTML()
//=> Finish blog post

```

With this solution, we've used our revamped mixin function to support `getColourRGB` overriding the mixin's definition, and we've used `override` to support wrapping functionality around the original `toHTML` method.

As a final bonus, if we are using a transpiler that supports ES.who-knows-when, we can use the proposed class decorator syntax:

```

@override('wrapWithColour', 'toHTML')
@Coloured
class TimeSensitiveTodo extends Todo {
    constructor (name, deadline) {
        super(name);
        this.deadline = deadline;
    }
}

```

```

getColourRGB () {
  let slack = this.deadline - Date.now();

  if (this.done) {
    return grey;
  }
  else if (slack <= 0) {
    return red;
  }
  else if (slack <= oneDayInMilliseconds){
    return yellow;
  }
  else return green;
}

wrapWithColour (original) {
  let rgb = this.getColourRGB();

  return `${original()}`;
}

```

This is extremely readable.



## method advice beyond extending mixin methods

`override` in and of itself is not spectacular. But most functionality that extends the behaviour of a method doesn't process the result of the original. Most extensions do some work *before* the method is invoked, or do some work *after* the method is invoked.

So in addition to `override`, or `toolbox` should include `before` and `after` method advice. `before` invokes the behaviour first, and if its return value is `undefined` or `truthy`, it invokes the decorated method:

```

let before = (behaviour, ...decoratedMethodNames) =>
  (clazz) => {
    if (typeof behaviour === 'string') {
      behaviour = clazz.prototype[behaviour];
    }
    for (let decoratedMethodName of decoratedMethodNames) {

```

```

let decoratedMethodFunction = clazz.prototype[decoratedMethodName];

Object.defineProperty(clazz.prototype, decoratedMethodName, {
  value: function (...args) {
    let behaviourValue = behaviour.apply(this, ...args);

    if (behaviourValue === undefined || !behaviourValue)
      return decoratedMethodFunction.apply(this, args);
  },
  writable: true
});
}

return clazz;
};

```

`before` should be used to decorate methods with setup or validation behaviour. Its “partner” is `after`, a decorator that runs behaviour after the decorated method is invoked:

```

let after = (behaviour, ...decoratedMethodNames) =>
  (clazz) => {
  if (typeof behaviour === 'string') {
    behaviour = clazz.prototype[behaviour];
  }
  for (let decoratedMethodName of decoratedMethodNames) {
    let decoratedMethodFunction = clazz.prototype[decoratedMethodName];

    Object.defineProperty(clazz.prototype, decoratedMethodName, {
      value: function (...args) {
        let decoratedMethodValue = decoratedMethodFunction.apply(this, args);

        behaviour.apply(this, ...args);
        return decoratedMethodValue;
      },
      writable: true
    });
  }
  return clazz;
};

```

With `before`, `after`, and `override` in hand, we have several advantages over traditional method overriding. First, `before` and `after` do a better job of declaring our intent when decomposing behaviour.

And second, method advice allows us to add behaviour to multiple methods at once, focusing responsibility for cross-cutting concerns, like this:

```

const mustBeLoggedIn = () => {
  if (currentUser() == null)
    throw new PermissionsException("Must be logged in!");
}

const mustBeMe = () => {
  if (currentUser() == null || !currentUser().person().equals(this))
    throw new PermissionsException("Must be me!");
}

@HasAge
@Before(mustBeMe, 'setName', 'setAge', 'age')
@Before(mustBeLoggedIn, 'fullName')
class Person {

  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }
};


```

Mixins allow us to have a many-to-many relationship between behaviour and classes. Method advice is similar: It makes a many-to-many relationship between behaviour and methods particularly easy to declare.

After using mixins and method advice on a regular basis, instead of using superclasses for shared behaviour, we use mixins and method advice instead. Superclasses are then relegated to those cases where we need to build behaviour into the constructor.

## wrapping up

A simple mixin can cover many cases, but when we wish to override or extend method behaviour, we need to either use the subclass factory pattern or incorporate method advice. Method advice offers benefits above and beyond overriding mixin methods, especially if we use `before` and `after` in addition to `override`.

That being said, subclass factories are most convenient of we are comfortable with hierarchies of superclasses and with using `super` to extend method behaviour. Subclass factories work best when we don't have a lot of behaviour that needs to be shared between different methods.

Method advice permits us to use a simpler approach to mixins, and makes it easy to have a many-to-many relationship between methods and behaviour, and it makes it easy to factor the responsibility for cross-cutting concerns out of each method.

No matter which approach we use, we find ourselves needing shallower and shallower class hierarchies when we use mixins to their fullest. Which demonstrates the power of working with simple constructs (like mixins and decorators) in JavaScript: We do not need nearly as much of the heavyweight OOP apparatus borrowed from 30 year-old languages, we just need to use the language we already have, in ways that cut with its grain.

(discuss on [hacker news](#))

---

more reading:

- [Prototypes are Objects \(and why that matters\)](#)
- [Classes are Expressions \(and why that matters\)](#)
- [Functional Mixins in ECMAScript 2015](#)
- [Using ES.later Decorators as Mixins](#)
- [Method Advice in Modern JavaScript](#)
- [super\(\) considered hmmm-fu](#)
- [JavaScript Mixins, Subclass Factories, and Method Advice](#)
- [This is not an essay about 'Traits in Javascript'](#)

notes:

1. A production-ready implementation would handle more than just methods. For example, it would allow you to mix getters and setters into a class, and it would allow us to attach properties or methods to the target class itself, and not just instances. But this simplified version handles methods, simple properties, “ mixin properties,” and `instanceof`, and that is enough for the purposes of investigating OO design questions. [←](#)
2. Justin Fagnani named this pattern “subclass factory” in his essay [“Real” Mixins with JavaScript Classes](#). It's well worth a read, and his implementation touches on other matters such as optimizing performance on modern JavaScript engines. [←](#)

## 37 : Compose me That: Function Composition in JavaScript

<https://www.linkedin.com/pulse/compose-me-function-composition-javascript-kevin-greene>

Source on Github: [Composition Tutorial](#)

Note: This article is the spiritual sibling of my previous post: [Curry me This: Partial Application in JavaScript](#). Both articles cover fundamental concepts for any functional programmer. If you're not familiar with currying I'd suggest reading that article first as we will be using curried functions here.

### Are We Painting a Picture Here?

Yes, we are painting a picture and functions are the lines we draw. Function composition is a formalized way of thinking about something that is largely considered a good practice in software development, keep your functions short, single purpose and referentially transparent. Doing this makes your functions easy to test, easy to reason about and easy to document for others who will be using your code.

Function composition is also directly a mathematical concept. In mathematics there is a composition operator, typically an empty circle ( $\circ$ ). It should be an empty mid dot, but probably shows up here as a degree symbol. The circle between two functions means take the result of the function on the right and pass it as the argument to the function on the left. The composition of two functions is a new function. If we have some function A such that  $A(3) = 15$  and some function B such that  $B(15) = 22$  then some function C that is equal to B compose A ( $C = B \circ A$ ) would be a function that given 3 as argument would return 22 ( $C(3) = 22$ ).

Short, single purpose functions don't really do much, that's their beauty. So the not so surprising secret as to how we make them do interesting things is that we compose them. Complex functions are compositions of simpler functions.

This function strips extra white space from a string. This function computes the distance between two points. This function finds the standard deviation of a list of numbers. It should be easy to describe what a function does. If you write a function and can't clearly describe what it is doing in a simple, declarative sentence you probably need to refactor that function into multiple functions, or step back and ask what it is you are actually trying to do. What you are trying to do can be described in a series of logical and declarative steps. We are dealing with computers here. At some point what you are trying to do needs to be translated into binary. Very often these steps will be your functions. Your application is the ultimate composition of these functions.

### Wouldn't This be Easier to Describe with Examples?

Let's start simple. Say we want to create a function that takes a string and capitalizes every word in that string, what do we need to do?

1. Find the words in the string.
2. Capitalize the first character of each word.
3. Return the new string.

Ignoring that we could do this fairly concisely with a regex (some of us, namely your humble narrator, suck at regexes), let's follow the steps and build this function. Something like this would do:

```

10  /**
11   * @name titleCase
12   * @param {String} str - The string to transform
13   * @returns {String} A new string with all words capitalized
14   */
15  function titleCase(str) {
16    // Break string into words
17    const parts = str.trim().split(' ');
18    // Remove extra white space
19    const trimmed = parts.filter((next) => next.length > 0);
20    // Capitalize the first letter of each words
21    const capitalized = trimmed.map((next) => {
22      return next[0].toUpperCase() + next.substring(1);
23    });
24    // Join words back into string and return
25    return capitalized.join(' ');
26  }
27

```

Nothing complicated. Each step follows logically from the previous. There is obviously some room for abstraction there though. There's a lot of implementation details in there that have nothing to do with the basic steps of what we want to do. The first thing we do is break the input string into an array of words. We have room here for a words function.

```

28 /**
29  * @name words
30  * @param {String} str - The string to break into words
31  * @returns {String[]} An array of individual words
32  */
33 function words(str) {
34   // Break string into words
35   const parts = str.trim().split(' ');
36   // Remove extra white space
37   return parts.filter((next) => next.length > 0);
38 }
39

```

Which turns our titleCase function into this:

```

10  /**
11  * @name titleCase
12  * @param {String} str - The string to transform
13  * @returns {String} A new string with all words capitalized
14  */
15  function titleCase(str) {
16
17  // Capitalize the first letter of each words
18  const capitalized = words(str).map((next) => {
19  return next[0].toUpperCase() + next.substring(1);
20 });
21
22 // Join words back into string and return
23 return capitalized.join(' ');
24 }

```

Does our titleCase function really care what it takes to capitalize a word? No. It only needs to know the words are capitalized. We could then break the capitalization of each word out into its own function.

```

10 /**
11 * @name capitalize
12 * @param {String} str - The string to capitalize
13 * @returns {String} A new string with the first letter capitalized
14 */
15 function capitalize(str) {
16 return str.charAt(0).toUpperCase() + next.substring(1);
17 }
18
19 /**
20 * @name titleCase
21 * @param {String} str - The string to transform
22 * @returns {String} A new string with all words capitalized
23 */
24 function titleCase(str) {
25 // Capitalize the first letter of each words
26 const capitalized = words(str).map(capitalize);
27 // Join words back into string and return
28 return capitalized.join(' ');
29 }
30

```

Then it would follow to create a function that capitalized each of a list of words. The ultimate idea is to strip the titleCase function down into only the steps described in our high-level overview, stripping away the implementation details of those steps and abstracting the implementation into other functions that can be logically described in terms of those implementation details.

## Hold on There Pokey, We're in Abstraction Hell

True, we haven't actually composed anything yet. I would argue though that our titleCase function should really be a one-liner, a composition of even simpler functions.

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name titleCase
5   * @param {String} str - A string to transform to title case
6   * @returns {String} The transformed string
7  */
8  const titleCase = compose(join(' '), capitalize, words);
9

```

Ok, so what's going on here? To start, let's look at this without the call to compose:

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name titleCase
5   * @param {String} str - A string to transform to title case
6   * @returns {String} The transformed string
7  */
8  const titleCase = function(str) {
9    return join(' ', capitalize(words(str)));
10 };
11

```

You've probably done something like this before, using a function invocation as the argument for another function. That is all function composition really is. We're just dealing with abstractions to make this idea more explicit.

We can't make our own operators or infix functions in JavaScript, so compose is a function that takes some number of functions and returns a new function that is the right to left composition of those functions. We'll look at an implementation shortly. In this case, the argument received by titleCase is some string that is given to the words function. Capitalize gets the result of the words function and join gets the result of capitalize. The return value of join is used as the return value of the composition.

Back when we originally described what this function should do we had three steps. Here we have three functions that correspond to each of those steps. Find the words, capitalize them, return the new string (by joining the capitalized words). The three functions compose together to create our titleCase function.

Alright, but what exactly do each of these three steps consist of? Our words function did a little more than just split on spaces:

1. Trim leading/trailing white space
2. Split string on spaces
3. Remove empty strings (extra spaces) from words list

Three more steps. Another composition:

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name words
5   * @param {String} str - A string to split into an array of words
6   * @returns {String[]} An array of individual words
7  */
8  const words = compose(removeEmpty, split(' '), trim);
9

```

The trim and split functions here are simple helpers that allow us to call String.prototype.trim and String.prototype.split as normal functions.

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name trim
5   * @param {String} str - The string to trim
6   * @returns {String} A string with leading and trailing whitespace removed
7  */
8  const trim = (str) => str.trim();
9

```

Split is a little more interesting in that it is curried. So split(' ') is a new function that splits on spaces:

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name split
5   * @param {String} separator - Character to split string on
6   * @param {String} str - The string to split
7   * @returns {String[]} An array of strings
8  */
9  const split = curry((separator, str) => str.split(separator));
10

```

It follows then that we could make a breakOnSpace function:

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name breakOnSpace
5   * @param {String} str - The string to break
6   * @returns {String[]} An array of strings
7  */
8  const breakOnSpace = split(' ');
9

```

The removeEmpty function handles the case when we split on a single space if there are extra spaces in our string we end up with empty strings in our words array. We remove these empty strings from our words array. Maybe this isn't desirable in all situations, but it's a wrinkle we'll add to our implementation.

For completeness, here is removeEmpty:

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name removeEmpty
5   * @param {Array} xs - List to filter
6   * @returns {Array} New array with empty elements removed
7  */
8  const removeEmpty = (xs) => xs.filter(notEmpty);
9

```

*Note: We'll skip the remaining two steps of titleCase (capitalize and join). Their implementations can be found in the linked source code.*

## TDD: It's Not All Puppy Dogs and Ice Cream

I write most of my JavaScript with heavy use of curry and compose. This allows and pushes me towards a few things. One is micro code reuse. The more of these small function I have, the more ways I can compose them to make more complex functions for use in whatever project I am working on. Two is that it changes the way I look at my code. I look at my code more and more with an eye of how I can abstract, simplify and reuse the code I write. Three is I am a big believer in TDD (Test Driven Development). Even if you don't strictly follow TDD at some point you need to unit test your JavaScript. These tiny functions are impossibly easy to unit test. Then, if my logic is sound, these tiny functions that are thoroughly unit tested are almost certain to work when I compose them together. If they don't work when I compose them together there is a higher level flaw in my reasoning about what a function should be doing, the steps it should be taking. Then once the composition is in place and unit tested I can then use that function in another composition that is more likely to work because its component parts have been correctly reasoned and unit tested.

Writing unit tests can be a real pain. The easier it is to do the more likely we are to do it. When the logic you are testing is simple and well-reasoned it will be much easier for you to write test. Building functions up through composition forces your functions to be well reasoned because they are simply compositions of the steps they need to perform. Even when I'm code sketching with a larger problem I'm not immediately sure how to tackle I always go back to building it up from smaller functions that are easy for me to test.

## So, How do we Make This Magic?

As I pointed out, function composition is probably something you've already done. You've probably used it a lot. Any time you've written something like this `g(f(arg))` you've used function composition. However, if we want the composition of `g` and `f` to be a reusable function we need to wrap it in some function that accepts the argument for us.

```

47
48  function composeGandF(arg) {
49    return g(f(arg));
50  }

```

There are certainly libraries that offer compose functions. Essentially the same libraries that offer curry functions. I'll list them again at the end. I at least like knowing how the magic works even if I'm going to use a library.

The examples we looked at above both had 3 steps. We can make a function that takes three functions and composes them.

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name compose3
5   * @param {Function} third - The third function to call
6   * @param {Function} second - The second function to call
7   * @param {Function} first - The first function to call
8   * @returns {*} The result of the second function
9  */
10 function compose3(third, second, first) {
11   return function composition(arg) {
12     return third(second(first(arg)));
13   };
14 }
15

```

Obviously not ideal, but gets the job done. We would need compose functions for all the different numbers of functions we would like to compose together (compose2, compose4...).

How do we make a generic compose function?

What exactly does a generic compose function need to do?

1. Take some n number of functions
2. Return a new function that takes an argument and returns the result of calling the composition of those functions on that argument

Well, we start like this:

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name compose
5   * @param {...Function} fns - The functions to compose
6   * @returns {Function} A new function that is the composition of fns
7  */
8  function compose(...fns) {
9    return function composition(arg) {
10      ...
11    };
12 }
13

```

Using rest parameters it's easy to get the functions we need to call as an array. We return a function that accepts the argument. How do we apply that to our list of functions? If we step back and don't think about function application, just what do we have? We have a list and need to return a result. Don't we have an array method that does something like that? Yep, sounds like a reduce:

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name compose
5   * @param {...Function} fns - The functions to compose
6   * @returns {Function} A new function that is the composition of fns
7  */
8
9  function compose(...fns) {
10    return function composition(arg) {
11      return fns.reduceRight((acc, next) => {
12        return next(acc);
13      }, arg);
14    };
15  }

```

The argument of the composition is the initial accumulator of reduce. The new accumulator is then the result of applying the next function to the accumulator.

Why do we use reduceRight? Why not apply the functions left to right? We could make it work either way. Keeping right to left composition keeps us closer to the mathematical roots of function composition. It also makes things read a little more clearly if we immediately invoke a composition:

```

55
56  compose(join(' '), capitalize, words)('this is a test');
57  // -> This Is A Test
58

```

It's a little more clear that 'this is a test' is being passed first as an argument to words. Right to left implies the flow of data through the composition.

## Library Implementations

1. [ramda.js](#)
2. [lodash.js](#) (calls their implementation flow/flowRight)
3. [underscore.js](#)

## 38 : Curry me This: Partial Application in JavaScript

<https://www.linkedin.com/pulse/curry-me-partial-application-javascript-kevin-greene>

Source on Github: [Curry Tutorial](#)

### Currying vs Partial Application

Currying and partial application are related concepts and the terms are often used interchangeably. It boils down to this, we have some function  $F(a,b,c)$  that has three parameters, but we don't want to apply all three arguments at the same time. Why on earth would we want to do that? Settle down, we'll get to that.

The prototypical example of this is an add (+) function:

```

1  /**
2   * @name add
3   * @param {Number} a Augend
4   * @param {Number} b Addend
5   * @returns {Number} The sum of a and b
6  */
7  function add(a, b) {
8    return a + b;
9  }
10

```

If we were to partially apply add, we could do something like this:

```

6
7  const increment = add(1);
8
9  increment(3); // -> 4
10

```

By partially applying the function we get back a new function with the 'a' parameter bound to the number 1. So we have a function that is just waiting for a 'b', a function that will add 1 to any number given to it. This gives us a great little semantic helper function we can use over and over again.

Partially applying? What about currying? As I said, these are similar concepts. To illustrate the difference lets look at a slightly different function:

```

1  /**
2  * @name threeSum
3  * @param {Number} a Augend
4  * @param {Number} b First addend
5  * @param {Number} c Second addend
6  * @returns {Number} The sum of a, b and c
7  */
8  function threeSum(a, b, c) {
9    return a + b + c;
10 }
11

```

Partial application generally refers to the ability to (surprise) partially apply a function. Just give it less arguments than it requires, as a result you are given back a new function that waits for the remaining arguments.

Curry can be thought of as a more specific case of this. If a function takes three arguments and is curried it is really three functions. Each function takes one argument and returns a new function that takes the next argument until all arguments are received then it returns the final result. Curried functions only ever take one argument.

```

19
20 // partial application
21 threeSum(2, 3)(4); // -> 9
22 // or
23 threeSum(2)(3)(4); // -> 9
24
25 // currying
26 threeSum(2, 3)(4); // -> ???? (Error or another Function)
27 // or
28 threeSum(2)(3)(4); // -> 9
29

```

Applying two arguments to a curried threeSum returns a result dependent on implementation. It may throw an error, or it may ignore the second argument. If it ignores the second argument in our example we are left with essentially this: threeSum(2)(4), a function waiting on a third argument to bind to 'c'.

The difference between partial application and currying may not be practically too important, but I make the distinction as a note that there is technically a distinction. In practice you will often see curry used to refer to partial application. Later, in these very typed words, I will slip into that myself.

Hey, but I know a little JavaScript. I know the functions we've written so far really only do something like this:

```

12
13 const increment = add(1);
14 // increment = NaN
15
16 increment(3); // -> Error 'NaN is not a function'
17

```

True. NaN is not a function, information for life.

## Using Function.prototype.bind

Native JavaScript already gives us a tool to perform partial application. `Function.prototype.bind` is a very useful bit of business. It allows us to bind a context and arguments to a function. The context bit can be cool, but we're going to concern ourselves only with binding arguments. Because of this, I will always use `null` as the first argument to bind.

*Note: If you are unfamiliar with `Function.prototype.bind` I suggest checking out the MDN documentation before continuing: [Function.prototype.bind](#).*

If we use `bind` to perform partial application, we end up with something like this:

```

39
40  const increment = add.bind(null, 1);
41
42  increment(3); // -> 4
43
44  const addFive = threeSum.bind(null, 2, 3);
45
46  addFive(4); // -> 9
47

```

The first argument (`null`) is bound to the context of the function. The following arguments are bound to the parameters of the function. If you've used React, you may have used `bind` to pass arguments to event handlers. Here we are using `bind` for partial application, to create new functions that are in essence more specific versions of the original function. This is the value of partial application, a subject we'll return to shortly.

`Function.prototype.bind` always returns a new function and always takes as many arguments as we care to pass to it. This would be perfectly fine:

```

50
51  const someJunk = add.bind(null, 0, 'henry', 9, 8, null, { foo : 'bar' });
52
53  someJunk('haha'); // -> '0henry'
54

```

In this case only the `0` and '`henry`' were bound to parameters. The `add` function only took two parameters so the first two arguments bound to `add` were bound to those parameters. Then on function invocation (`someJunk('haha')`) we were returned '`0henry`'. All the other arguments we gave to `add` were just thrown away.

This API is pretty clunky if our main want is to take advantage of partial application. Things were nicer in our theoretical API earlier where `add` just knew what to do based on the arguments it had so far.

## Going Our Own Way

How would we implement this ourselves? I want functions that allow me to partially apply them, always giving me back new functions until I have provided all the arguments, then I want a result.

Starting simply, if we were to just write our add function as a curriedAdd function:

```

1  /**
2  * @name curriedAdd
3  * @param {Number} a Augend
4  * @param {Number} b Addend
5  * @returns {Number} The sum of a and b
6  */
7  function curriedAdd(a) {
8    return function(b) {
9      return a + b;
10   };
11 }
12

```

This is a true curried function. It is a composition of two functions that each take one argument.

Binary functions are incredibly common. Say I want to take advantage of currying, but I don't want to go back and rewrite all the binary functions I've written before. I want something like bind that will allow me to transform those binary functions into curried functions. Write a function that takes as its argument a binary function and returns a curried version of that function.

It would look something like this:

```

1  /**
2  * @name binaryCurry
3  * @param {Function} fn A binary function to curry
4  * @returns {Function} A curried version of the supplied function
5  */
6  function binaryCurry(fn) {
7    return function(a) {
8      return function(b) {
9        return fn(a, b);
10     };
11   };
12 }
13

```

Yes, very similar to our curriedAdd function. It takes a function and returns a function that takes the first argument. When applied that function returns a function to take a second argument. When that function is applied we finally get a result. Now we can rewrite our curriedAdd function like this:

```

31
32  const curriedAdd = binaryCurry(add);
33  const increment = curriedAdd(1);
34
35  increment(4); // -> 5
36

```

Okay, so what do I do if I have a function that takes three arguments, or four? Do I really want to pass all the parameters separately every time? If we're going to take advantage of these patterns in our daily code we are probably more interested in partial application than true currying. We also want a generic version of our binaryCurry function that can turn any function into a function that can be partially applied.

So, let's write this function. An obvious requirement of this is we need to know how many arguments the function we are transforming expects to receive. I don't see this used often, but functions have a length

property which is the number of declared parameters. Like this:

```

39  function add(a, b) {
40    return a + b;
41  }
42
43
44  add.length === 2; // -> true
45

```

Note: This doesn't take into account arguments a function may use by accessing the arguments array-like thing we shouldn't really be using anyway.

Using ES6 rest parameters and the spread operator, writing a generic curry function is rather concise.

*Note: Rest parameters allow us to receive a variable number of arguments in our functions. Instead of using the arguments object we get an actual array of arguments that weren't otherwise bound to parameters. For a more detailed description of rest parameters MDN is again the spot: [Rest Parameters](#).*

*Note: The spread operator allows us to take an array and very literally spread it out, as in the case where we have an array and want to pass it to a function that takes multiple arguments. Hey, that's what we used to use apply for. Yes, that's what we used to use apply for. Again, here's MDN: [Spread Operator](#).*

```

1  /* jshint esversion: 6 */
2
3 /**
4  * @name curry
5  * @param {Function} fn A function to curry
6  * @returns {Function} A new function that is a curried version of fn
7 */
8 function curry(fn, ...args) {
9
10   // How many arguments do we expect to receive?
11   const arity = fn.length;
12
13   function curried(...args2) {
14
15     const locals = args.concat(args2);
16
17     // If we have all the arguments, apply the function and return result
18     if (locals.length >= arity) {
19       return fn(...locals);
20
21     // If we don't have all the arguments,
22     // return a new function that awaits remaining arguments
23     } else {
24       return curry(fn, ...locals);
25     }
26   }
27
28   // If we have all the arguments apply the function,
29   // else return a function to receive more arguments.
30   return ((args.length >= arity) ? curried() : curried);
31 }
32

```

What happens here? The first time this function is called it only expects one argument, a function to curry (to create a new function that can be partially applied). The args parameter will probably be an empty array on first invocation, unless you want to bind any arguments to this function from the beginning. We save the arity (number of arguments it expects) to a local variable. We then return a function we define inside the curry function. We call this function 'curried' to indicate this is the curried function. Yes, yes, the author is an idiot. Moving on. When this function is invoked we concat the new args2 array with the old args array and check if we have received all the arguments yet. If we have, we apply the original function and return the result. If we have not we recursively call the curry function, passing along all the new arguments which puts us back in the original position, returning the curried function to await more arguments. On this second call (and all future calls) to curry the args parameter will be all the arguments we have received so far. New arguments from the client will be bound to args2 and concatenated with args to form locals.

Play around, test it. Try to solve the problem for yourself. We now have a generic function we can use to transform any function into one that can be partially applied.

## Subclassing a Function?

Currying and partial application are tools like anything else. Their value comes in the form of being able to reuse more code. You can write functions in their most generic form and partially apply them to create functions suited for a specific situation. I've had the most success in explaining the concept to people new to it by describing it as subclassing a function. Most programmers are familiar with the idea of a generic base class that is subclassed for a given situation. Thinking about the abstraction in those terms will probably give you the best idea of what the value is.

## Can We Have Some Examples?

Sure, let's look at a couple situations where using partial application can help us write a little cleaner code. These examples will be simple, but should get you started.

First, let's write a function to help us test the length of given objects. Not a terribly complex task, but something that's fairly common.

```

1  /* jshint esversion: 6 */
2
3  /**
4   * @name hasLength
5   * @param {Number} len The length to test against
6   * @param {Object|Array} obj The object to test
7   * @returns {Boolean} Does the given object have a length >= to the given len?
8  */
9  const hasLength = curry((len, obj) => {
10    return (obj.length >= len);
11  });
12

```

Simple enough. It returns a boolean letting us know if a given object has a given length. Because this function is curried we can do something like this:

```
5  ...
6  const arr = [
7  ...'first item',
8  ...'another item',
9  ...'an item longer than the others'
10 ];
11 ...
12 const arr2 = arr.filter(hasLength(15))
13 ...
14 console.log(arr2); // -> ['an item longer than the others']
15
```

Because of partial application `hasLength(15)` returns a new function that waits on the object to test. The array filter method uses this new function as it iterates over the array, on each iteration passing the second argument to `hasLength`, which upon getting its second argument runs the test and returns a boolean. It makes filter a little more concise. It makes our code more semantic, thus making it a little easier to read. No giant gains, but we're just getting started. We could make this even a little more self-documenting by doing something like this:

```
5  ...
6  const arr = [
7  ...'first item',
8  ...'another item',
9  ...'an item longer than the others'
10 ];
11 ...
12 const isFifteenCharsLong = hasLength(15);
13 const arr2 = arr.filter(isFifteenCharsLong);
14 ...
15 console.log(arr2); // -> ['an item longer than the others']
16
```

This is even more code, but illustrates the kind of things you can start doing with utility or helper functions to make them a little easier to read and to make many functions that all use the same code by writing the original (`hasLength` in this case) in the most generic form, allowing us to create new functions for more specific situations.

Something else you may want to do is test if a given DOM element matches a given selector.

```

1  /* jshint esversion: 6 */
2
3  const isFunction = (obj) => typeof obj === 'function';
4
5  /**
6   * Tests whether an element matches a given CSS selector
7   *
8   * @name matches
9   * @param {String} selector
10  * @param {Object} element
11  * @returns {Boolean}
12  */
13 const matches = curry((selector, element) => {
14
15  if (isFunction(element.matches)) {
16    return element.matches(selector);
17  } else {
18    const elementList = document.querySelectorAll(selector);
19    let i = 0;
20
21    while (elementList[i] && elementList[i] !== element) {
22      i++;
23    }
24
25    return (elementList[i] ? true : false);
26  }
27});
28

```

You'll start to notice. If you are going to take advantage of partial application with your functions, the order in which the parameters are defined restricts how the function can be used. Here, I want the selector to be argument I partially apply to make my more specific function. I want to write these tester functions I can use on any element. If the element was first I could only use this function on one element, probably less useful, but it depends on your use case.

And to use this:

```

18
19
20  const isListItem = matches('.list-item');
21  const ul = document.getElementById('some-list');
22
23  ul.addEventListener('click', (evt) => {
24    if (isListItem(evt.target)) {
25      evt.target.classList.add('active');
26    }
27  });
28

```

## Implementations?

We've seen how to use bind to perform partial applications. We've seen how to build our own function for creating functions that can be partially applied. However, you really want someone else to deal with the

implementation for you. As mentioned in the comments there are libraries that provide curry implementations.

Some notable libraries:

1. [ramda.js](#)
2. [lodash.js](#)
3. [underscore.js](#) (which calls its implementation 'partial')

## Conclusion

Using partial application to reuse code probably won't drastically change things for you in your daily work. However, one of the things we are constantly doing as developers in finding ways to reuse our code and not write the same thing more than once. Something else we all do at times is write functions that are way too large and do way too much. We just get something working. Then, having a clearer picture of what we should do, we refactor that function into several functions, each performing one task. Often while doing this we find pieces of code that are generic, that are performing micro computations that aren't really tied to this specific use case. These smaller computations can often be handled by libraries, or can be moved into utility packages within our application, keeping our main application logic cleaner. All of these situations bring opportunities to take advantage of tools like partial application to build abstractions we might not have seen or considered without this tool in our chest. Like any tool, as you use it more you will become better at wielding it and will change the way you write code as you learn to see abstractions you didn't see previously.

## Further Reading:

1. [Currying on Wikipedia](#)
2. [Partial application on Wikipedia](#)

## 39 : Functional Data Structures in JavaScript: The Basics

<https://www.linkedin.com/pulse/function-data-structures-javascript-basics-kevin-greene>

Source on Github: [Functional Data Structures](#)

### Once Upon a Time

When I was in college I had no interest in thinking about the future, so I was an art major. I did however have an interest in computers and technology so for an elective I took an intro to computer science course. I honestly don't remember much about the course. It was years later that I returned to programming. You know, gave up on my hopes and dreams and got a real job. The thing that stuck with me from the course was recursion. It was early in the semester and the instructor was demonstrating loops by finding the nth Fibonacci number (cringe). He then showed the recursive solution.

```
function fibonacci(n) {  
  if (n < 2) {  
    return 1;  
  } else {  
    return fibonacci(n - 1) + fibonacci(n - 2);  
  }  
}
```

Nothing special, but nice. To my young art-student brain the elegance of recursion was very pleasing and this idea stuck with me. So, years later, when I returned to programming I was very interested in this idea that had stuck with me. That recursion thing was nice. I want to know more about what can be done with that. How do I break large problems down into small problems so I can solve them with recursion?

A lot of the beauty of recursion comes from how each function call maintains its own state, eliminating the need for temporary variables. If we look at an iterative solution to the same problem we find the need for multiple variable assignments.

```

function fibonacci(n) {
  var a = 0;
  var b = 1;
  var f = 1;
  while (n > 0) {
    f = a + b;
    a = b;
    b = f;
    n -= 1;
  }
  return f;
}

```

This could be reduced, but this illustrates my point. In order to maintain the state of our computation we have to create additional variables. The recursive solution just used the arguments passed into the function. This is the beginning of how we can persist data using only functions.

When we think about functional programming languages immutable data is one of the things we associate with them. What makes functional data structures immutable? Does the data have to be immutable? How are they different than the data structures we are used to using? A lot of these questions really don't matter practically when you're writing programs. It matters if data is mutable or not. It doesn't matter how it gets that way. In JavaScript we have `Object.freeze`. We can make immutable data if we want. All data in our programs is really just the location of some bits in memory. Those bits can always be changed. The lower level the programming language you're using, the more you have to be concerned with the specifics of managing those bits and their location. What we will be exploring here is storing data in functions.

This is a good subject for a book. Actually, the inspiration for this post was translating the Standard ML examples from Chris Okasaki's book [Purely Function Data Structures](#) to JavaScript. We'll see how far I want to take this post. I'll probably cover some basic structures, then next week or next month write another post about more complex structures. For now we'll cover numbers, booleans, tuples, lists and binary search trees.

## Church Numerals

A lot of the ideas that form the basis of functional programming languages are derived from lambda calculus. The lambda calculus is a model for computation. If you are interested in functional programming but unfamiliar with lambda calculus I suggest to check it out. I'll include some links to articles and books later. The key idea (specifically in untyped lambda calculus) is that the only data structure we have is a function. Actually, the only piece of syntax we have is a function and the argument bindings to that function. No numbers, strings, lists, conditionals, loops... etc. If you need these things you have to build them yourself with functions. We're going to loosen these restrictions a bit as we go, specifically we are going to use conditionals, but for now let's look at how we might do useful things with only functions.

How can we do things in an environment that doesn't even have numbers? When we talk about the existence of numbers what we are really talking about is the ability to count some occurrences of a thing. With our given limitations the thing we have to count are functions. Specifically what we are going to count is the number of applications of a function. If a given function isn't applied at all that is 0. If it is applied once that is one. For any number n the equivalent number to us is the nth composition of some function fn with itself.

```
function One(fn, x) {
  return fn(x);
}
```

This function takes a function "fn" and some argument "x". The function is applied to x once, thus One. Numbers expressed in this fashion are called Church numerals after Alonzo Church, who introduced the lambda calculus. If this is one what then is Zero?

```
function Zero(fn, x) {
  return x;
}
```

The same signature, but the function isn't applied. Moving along we come to Two.

```
function Two(fn, x) {
  return fn(fn(x));
}
```

Or we call the function fn one extra time on One:

```
function Two(fn, x) {
  return fn(One(fn, x));
}
```

This leads us to a function to increment any Church numeral:

```
// Find the successor of num
function succ(num) {
  return function(fn, x) {
    return fn(num(fn, x));
  };
}
```

Now we can write numbers in terms of the number that came before them.

```
const Zero = (fn, x) => x;
const One = succ(Zero);
const Two = succ(One);
const Three = succ(Two);
const Four = succ(Three);
```

We're placing artificial limitations on ourselves by using only functions to express data. It would be nice to at least have a way to check our work, maintain our sanity and see these things as the Arabic numerals we are used to.

```
function toNumber(num) {
  return num(function(acc) {
    return acc + 1;
  }, 0);
}
```

One of our numbers is just a function that takes two arguments. If we apply these functions correctly we can retrieve the corresponding Arabic numeral. By using 0 as the initial argument for the function we can keep track of how many times one of our numbers call its function. The function we then pass in as the first argument is an accumulator that just needs to increment the number it receives and pass it to the next function invocation.

If some number n is just a function representing the nth composition of some function fn, how do we express addition?

```
// Addition of two Church Numerals
function add(m, n) {
  return function(fn, x) {
    return m(fn, n(fn, x));
  };
}
```

When this function is applied n calls fn n times and m an additional m times resulting in the addition of m and n.

```
toNumber(add(Five, Four)); // -> 9
toNumber(add(Ten, Two)); // -> 12
```

We will return to numbers shortly and learn how to perform more operations on Church numerals. It will help to introduce a few more concepts first.

## Booleans

Booleans are where things start to get a little interesting to me. The patterns we start exploring with booleans are going to be the patterns we use to express the rest of the data structures we're going to look at. They also start representing patterns that could actually be used to build useful data structures.

We have two potential values for booleans. That's it. We know what they are ahead of time. We know logically what they represent. A boolean is really just a fork in the road. Either one thing or another.

```

function True(trueValue, _) {
  return trueValue;
}

function False(_, falseValue) {
  return falseValue;
}

```

An operation that was consuming one of these booleans would call the function with two values and would know whether the boolean was True or False based on the value returned.

```

function toBoolean(bool) {
  return bool(true, false);
}

toBoolean(True); // -> true
toBoolean(False); // -> false

```

In essence our boolean is its own if/else. If it's true the first argument is used, else the second argument.

We could perform a logical NOT, reverse the value, simply enough.

```

function not(bool) {
  return bool(False, True);
}

toBoolean(not(False)); // -> true
toBoolean(not(True)); // -> false

```

If this function is given a True it will return the left value of False and if given a False will return the right value of True. We can follow this up with logical AND.

```

function and(leftBool, rightBool) {
  return leftBool(rightBool, leftBool);
}

toBoolean(and(True, False)); // -> false
toBoolean(and(True, True)); // -> true

```

If the leftBool is a True the value of rightBool will be returned as the value of this function. It will evaluate to true only if both are true. If leftBool is False the value of leftBool will be returned as the value of this function.

Logical OR follows very quickly from this.

```

function or(leftBool, rightBool) {
  return leftBool(leftBool, rightBool);
}

toBoolean(or(True, False)); // -> true
toBoolean(or(False, False)); // -> false

```

If leftBool is True leftBool is returned, otherwise we return rightBool. This function returns True if either value is True.

## Pairs

Let's start persisting arbitrary data into functions. If I wanted to persist two values in a pair I would expect that to look something like this:

```
const pair = Pair(1, 2);
```

We know we are only persisting data as functions. If we pass two values to a function 'Pair' those values will be available to the scope of that function.

```

function Pair(left, right) {
  // The two values now belong to this scope
}

```

The two values 'left' and 'right' are now bound to the scope of this function invocation. How do we retrieve those values later? We return a function that gives us access to this scope.

```

function Pair(left, right) {
  return function(destructurePair) {
    return destructurePair(left, right);
  };
}

```

Now the value we receive from calling `Pair` is a function that gives us access to that inner scope. We invoke this returned function with a function that takes two arguments, the two values we gave to the `Pair` function.

```

const pair = Pair(1, 2);

pair(function(left, right) {
  // We can now retrieve the values.
});
```

We say that the function `destructures` the `Pair` because it pulls it apart into its component values. This is somewhat similar to how destructuring assignment works in JavaScript.

```

// Pull the object apart and assign its
// component values to the variables x, y, z.
const { x, y, z } = { x: 'x', y: 'y', z: 'z' };
```

We can use destructuring to create functions to retrieve the first and second values of the `Pair`.

```

function first(pair) {
  return pair(function(f, _) {
    return f;
  });
}

function second(pair) {
  return pair(function(_, s) {
    return s;
  });
}

first(Pair(3, 9)); // -> 3
second(Pair(1, 49)); // -> 49
```

## Lists

Lists follow naturally from Pairs. In our implementation of Pairs we knew exactly how many values our data structure would handle. It would hold two, no more, no less. With Lists we need to handle an arbitrary number of values. The data structure we are actually going to implement is a linked list where a node in our list holds two things the value at that position and some representation of the remainder of the list.

```
function List(head, tail) {  
  return function(destructureNode) {  
    return destructureNode(head, tail);  
  };  
}
```

Hmm, this is identical to our implementation of Pair. This isn't quite going to be good enough. What do we do when we do when we get to the end of a list? How do we represent there is no more list, or that we have an empty list? Do we just let the destructureNode function fail? No. We add a special value to represent the empty list. This is typically called Nil.

```
function List(head, tail) {  
  return function(destructureNode, _) {  
    return destructureNode(head, tail);  
  };  
}  
  
function Nil(_, destructureNil) {  
  return destructureNil();  
}
```

Now we can also represent the end of a List as Nil. If we're looping through a List we can make a recursive call in the destructureNode function and short-circuit the recursion when we hit the end of the List in our destructureNil function.

The key thing to remember here is head is a value our list is holding and tail is another List, specifically the remaining List after this node.

It's easy enough to build on this and write functions to retrieve the head and tail of our Lists. This is almost identical to the first and second functions for Pairs. The only difference is we need to handle the special case of Nil.

```

function head(list) {
  return list(function(h, _) {
    return h;
  }, function() {
    throw new Error('Empty List has no head');
  });
}

function tail(list) {
  return list(function(_, t) {
    return t;
  }, function() {
    throw new Error('Empty List has no tail');
  });
}

```

Before moving on, let's write a function to transform one of these Lists into a JavaScript array to make it easier to check our work.

```

function toArray(list) {
  return list(function(head, tail) {
    return [head].concat(toArray(tail));
  }, function() {
    return [];
  });
}

const list = List(1, List(2, List(3, Nil)));
toArray(list); // -> [1, 2, 3];

```

Let's do something more interesting. Let's concatenate two lists. Our data structures are immutable, so we won't be modifying either List in order to perform the concatenation. What we will do is copy the first List and use the second List as the tail of our copy. This will leave our second List in tact, but we will save ourselves from unnecessary copying if the two Lists just share those nodes.

```

function concat(first, second) {
  return first(function(head, tail) {
    return List(head, concat(tail, second));
  }, function() {
    return second;
  });
}

```

When our recursion reaches the end of the first List the destructureNil function is used which just returns the second List which is then used as the tail of the last node in our copy.

I'll take a moment to point out something obvious here. The List constructor is a prepend function.

```
const newList = List(5, Nil);
```

The prepend operation is performed in constant O(1) time. Our concat operation however must recurse through every node in the first List, meaning concatenation takes time proportional to the size of the first List O(n). In many functional languages this operation is referred to as 'cons'. There really is no List constructor. Every List is just created by prepending to the Nil value.

How about appending to a List? Appending to a List is going to be another O(n) operation. We are going to copy the List and just add one extra node to the end.

```
function append(val, list) {
  return list(function(head, tail) {
    return List(head, append(val, tail));
  }, function() {
    return List(val, Nil);
  });
}
```

To make a copy of a List we were using a prepend operation. Could we then use our new append operation to make a reverse copy?

```
function reverse(list) {
  return list(function(head, tail) {
    return append(head, reverse(tail));
  }, function() {
    return Nil;
  });
}

const list = List(1, List(2, List(3, List(4, Nil))));
const reversed = reverse(list);
toArray(list); // -> [1, 2, 3, 4]
toArray(reversed); // -> [4, 3, 2, 1]
```

What other operations do we commonly want to perform on lists? Getting the nth value of a List is pretty common.

```

function get(n, list) {
  return list(function(head, tail) {
    if (n === 0) {
      return head;
    } else {
      return get((n - 1), tail);
    }
  }, function() {
    throw new Error('Index requested is outside the bounds of this List');
  });
}

```

I told you, we would eventually get around to using conditionals. We've been able to do a lot with only functions. We could certainly do more, but I'm not that masochistic. Simple enough, we just keep decrementing n until it reaches 0 then we know we're at the correct node. If we get to an Empty node before n reaches 0 we know the requested index is out-of-bounds.

What follows quickly from this is updating the node at the nth index. Because we are again non-destructively updating our Lists, we are creating a new List with the given node updated. We accomplish this by copying every node until we reach the given index and then use any remaining nodes as the tail of our new List, similar to how we performed concatenation.

```

function update(list, i, val) {
  return list(function(head, tail) {
    if (i === 0) {
      return List(val, tail);
    } else {
      return List(head, update(tail, (i - 1), val));
    }
  }, function() {
    throw new Error('Index requested is outside the bounds of this List');
  });
}

```

We could obviously fill out all of the common functions for Lists. However, I think we get the idea. All the operations follow the same patterns. In the included source code I have implementations for map, filter, foldl and foldr if you want to take a look.

## Binary Search Trees

Binary Search Trees are going to be the last data structure we look at in this post. Following on from Pairs and Lists we are just going to be adding a little bit of complexity to what we have already seen. A node in a List is singly-linked to the rest of the List. A node in a binary tree is doubly-linked. All nodes in the left sub-

tree have values less than the current node and all values in the right sub-tree have values greater than the current node. For this implementation we are going to say all values in the left sub-tree are less than or equal to the value of the current node. As with Lists we are going to have a special type to represent an empty tree.

```

function Tree(val, left, right) {
  return function(destructureTree, _) {
    return destructureTree(val, left, right);
  };
}

function Empty(_, destructureEmpty) {
  return destructureEmpty();
}

```

As usual, we are going to make a function to turn our Tree into a more easily inspectable JavaScript object.

```

function toObject(tree) {
  return tree(function(val, left, right) {
    return {
      key : val,
      left : toObject(left),
      right : toObject(right)
    };
  }, function() {
    return 'Empty';
  });
}

```

The first thing we are going to want to do with our Tree is to insert values into it. Like Lists, Trees are built by inserting elements into the empty Tree. With binary search trees new elements are inserted as a new leaf. We just need to find the correct place to insert the new leaf. We compare the value of the new value to the value of the current node and move left or right depending on if it is smaller or larger. Once we find an Empty node we have a place to insert the new node.

```

function insert(toInsert, tree) {
  return tree(function(val, left, right) {
    if (toInsert <= val) {
      return Tree(val, insert(toInsert, left), right);
    } else {
      return Tree(val, left, insert(toInsert, right));
    }
  }, function() {
    return Tree(toInsert, Empty, Empty);
  });
}

const tree = insert(4, Empty);
toObject(tree); // -> { key : 4, left : 'Empty', right : 'Empty' }

```

You will notice this code copies nodes on the path to the new node. Nodes that are not on the insertion path are shared with the new Tree, returned as a result of this insertion, and the old Tree. Just look at the first conditional in the destructureTree branch. The value to insert is less than or equal to the current. The right sub-tree is going to go unaltered. It can be shared. The inverse is true if the value is greater.

The next thing we want to do with a binary search tree is to search. We will start with the simplest search. Just find the minimum (or maximum) value in our Tree. In the case of finding the minimum value we just keep moving left until we reach the end. We do so by recursively calling the min function on the left child node until we reach a left child node that has an empty left child. If you were to write this code in an object oriented language you would check for an empty left child node by checking a property on the node object. Something like `node.left === null`. Here we need to destructure the left node to see if it has an empty left child. If it does have an empty left node we return the current node we're on, else we recurse into the left node.

```

function min(tree) {
  return tree(function(val, left, right) {
    return left(function(_val, _left, _right) {
      return min(left);
    }, function() {
      return tree;
    });
  }, function() {
    throw new Error('No min node in empty tree');
  });
}

```

This code is easily adapted to finding the max by replacing the recursive call with the left node to a recursive call with the right node. I'll leave that out.

How about more generic searching? I want to know if a given value is already in the tree. This follows fairly directly from finding the min or max. Our code just needs a conditional to know to move left or right based on the search value relative to the current value. If we come to an Empty node we return false, otherwise we return true when we land on a node with a value equal to our search value.

```
function search(toFind, tree) {
  return tree(function(val, left, right) {
    if (toFind === val) {
      return true;
    } else if (toFind < val) {
      return search(toFind, left);
    } else {
      return search(toFind, right);
    }
  }, function() {
    return false;
  });
}
```

Deleting values is always where binary search trees get a little hairy to implement. However, in a functional data structure deleting values is much more straight forward. Deleting is much more straight forward because we don't have to worry about maintaining the mutated state of our data structure. We are copying all the pieces we are altering. We can be more declarative about saying, 'This is what this structure should look like now'. We will start with removing the minimum value in a Tree. We just need to move left until we reach a node with an empty left child. We then replace that node with its right child.

```
function removeMin(tree) {
  return tree(function(val, left, right) {
    return left(function(_val, _left, _right) {
      return Tree(val, removeMin(left), right);
    }, function() {
      return right;
    });
  }, function() {
    return Empty;
  });
}
```

If you look back at our code for finding the minimum value you'll see that this is almost identical. We are just constructing our new data structure as we recurse down to the minimum node. Once a node has no left child, which we know when the inner `destructureEmpty` is called, we just return the right node. If the initial Tree we pass in is an Empty tree we just return Empty.

Following on from this we can write a function to delete an arbitrary value from a Tree. This again looks very much like our generic search function, we just build up a new Tree as we recurse down, moving left or right depending on if the value to remove is less than or greater than the value of the current node. However, what happens when we find the node to remove? We need to fill in the hole left by removing a node.

To keep our remove function clean, we write a helper function to replace the removed node. We replace the removed node with the minimum node in the right sub-tree of the node we are removing. The minimum node in the right sub-tree is guaranteed to not break the correctness of our tree. It will be greater than all of the nodes in the left sub-tree of the node we are removing and it will be less than all of the nodes in what remains of the right sub-tree of the node we are removing. We take the value of the minimum node in the right sub-tree and create a new node to replace the node we are removing. This new node will use the left sub-tree of the node we are removing. Again, sharing this sub-tree with the tree we are deleting from as we are leaving it unaltered. Our new node will also use the right sub-tree of the node we are removing. However, for the right sub-tree we will be using a copy. We need to remove the minimum so we are not including the same node twice. Luckily enough we just made a `removeMin` function.

```
function replaceRemoved(tree, left, right) {-
  return tree(function(val, _left, _right) {-
    return Tree(val, left, removeMin(right));-
  }, function() {-
    return left;-
  });-
}

function remove(toRemove, tree) {-
  return tree(function(val, left, right) {-
    if (toRemove === val) {-
      return replaceRemoved(min(right), left, right);-
    } else if (toRemove < val) {-
      return Tree(val, remove(toRemove, left), right);-
    } else {-
      return Tree(val, left, remove(toRemove, right));-
    }-
  }, function() {-
    return tree;-
  });-
}
```

If you have written this in an object-oriented language, you will notice this is much less code. At each step we are able to be very declarative about what we want our tree to look like at that point.

In our functional implementation of Lists we noted we were actually implementing a linked list and many of our operations took time proportional to the size of the list. For random input functional binary search trees will still offer logarithmic inserts, deletes and searches. I'm going to follow this post up fairly soon with a look

at functional balanced binary search trees, specifically red/black trees. But this was our last data structure for today.

## A Return to Arithmetic

The first thing we covered were Church numerals. Almost certainly the least practical thing we looked at, but we're going to close for today and look at how we can do subtraction on Church numerals. This is an example of how far you can take computation with nothing more than functions. Once we can represent numbers and the operations on numbers we can support anything.

We are going to solve the problem of subtraction by creating a decrement function and then applying it some n number of times. If we define a number in Church numerals as the number of times a function is applied, how do we get the previous application? The function we are applying must maintain its previous state.

In our toNumber function we used the function application of our Church numerals to translate them into Arabic numerals. We could use this function application to perform operations between Church numerals.

We wrote our add function like this:

```
// Addition of two Church Numerals
function add(m, n) {
  return function(fn, x) {
    return m(fn, n(fn, x));
  };
}
```

The n numeral applies fn n times and then m applies it m more times. However, we also wrote an increment function we called succ.

```
// Find the successor of a Church Numeral
function succ(num) {
  return function(fn, x) {
    return fn(num(fn, x));
  };
}
```

This just applies fn once extra time to num. We could write an addition function in terms of the succ function. Increment the m numeral some n number of times.

```

function add(m, n) {
  return m(succ, n);
}

```

Instead of using another Church numeral as the argument for our function, what if we used a Pair? As we incremented the numeral the first value in the pair would be the previous value and the second value would be the current value. The succ function won't handle this for us, but we can write another function that could operate on Pairs.

```

function slide(pair) {
  return Pair(second(pair), succ(second(pair)));
}

```

If slide starts with a pair of zeros, `Pair(Zero, Zero)`, successive calls to slide, `slide(slide(Pair(Zero, Zero)))`, will work like the succ function, except the first value will always be one less. Therefore to implement a decrement function we apply slide some  $n$  number of times, starting with `Pair(Zero, Zero)`, and take the first value from the returned Pair.

```

// Find the preceding Church Numeral
function pred(num) {
  return first(num(slide, Pair(Zero, Zero)));
}

```

Subtraction can now be written in terms of pred the same way we wrote our second attempt at add in terms of succ.

```

// Subtraction of two Church Numerals ( $m - n$ )
function sub(m, n) {
  return n(pred, m);
}

```

Cool, that's it for this post. As I said I'll follow up sometime soon with an implementation of red/black trees and maybe some other stuff. We'll see.

## 40 : Model-View-Controller (MVC) with JavaScript

<https://alexatnet.com/articles/model-view-controller-mvc-javascript>

I like JavaScript because it is one of the most flexible languages in the world. It supports wide range of the programming styles and techniques, but such flexibility comes with danger - it is very easy for the JavaScript project to become a messy heap if the practices or design patterns are applied in a wrong way or inconsistently.

My goal for this article is to demonstrate how to apply the Model-View- Controller pattern while developing a simple JavaScript component. The component is a kind of the HTML ListBox (“select” HTML tag) control with an editable list of items: the user should be able to select and remove items and add new items into the list. The component will consist of three classes that corresponds to the parts of the [Model-View-Controller](#) design pattern.

I hope, this article will be a good reading for you, but it would be much better if you consider to run the examples and adapt them to you needs. I believe you have everything to create and run JavaScript programs: brains, hands, text editor, and an Internet Browser (Google Chrome, for example).

The Model-View-Controller pattern requires some description here. As you may know, the name of the pattern is based on the names of its main parts: Model, which stores an application data model; View, which renders Model for an appropriate representation; and Controller, which updates Model. Wikipedia defines typical components of the Model-View-Controller architecture as follows:

- **Model** - The domain-specific representation of the information on which the application operates. The model is another name for the domain layer. Domain logic adds meaning to raw data (e.g., calculating if today is the user’s birthday, or the totals, taxes and shipping charges for shopping cart items).
- **View** - Renders the model into a form suitable for interaction, typically a user interface element. MVC is often seen in web applications, where the view is the HTML page and the code which gathers dynamic data for the page.
- **Controller** - Processes and responds to events, typically user actions, and invokes changes on the model and perhaps the view.

The data of the component is just a list of items, in which one particular item can be selected and deleted. So, the model of the component is very simple - it consists of an array and a selected item index; and here it is:

```
/**  
 * The Model. Model stores items and notifies  
 * observers about changes.  
 */  
function ListModel(items) {  
    this._items = items;
```

```
this._selectedIndex = -1;

this.itemAdded = new Event(this);
this.itemRemoved = new Event(this);
this.selectedIndexChanged = new Event(this);

}

ListModel.prototype = {
    getItems : function () {
        return [].concat(this._items);
    },

    addItem : function (item) {
        this._items.push(item);
        this.itemAdded.notify({ item : item });
    },

    removeItemAt : function (index) {
        var item;

        item = this._items[index];
        this._items.splice(index, 1);
        this.itemRemoved.notify({ item : item });
        if (index === this._selectedIndex) {
            this.setSelectedIndex(-1);
        }
    },

    getSelectedIndex : function () {
        return this._selectedIndex;
    },

    setSelectedIndex : function (index) {
        var previousIndex;

        previousIndex = this._selectedIndex;
        this._selectedIndex = index;
        this.selectedIndexChanged.notify({ previous : previousIndex });
    }
};
```

Event is a simple class for implementing the Observer pattern:

```
function Event(sender) {
    this._sender = sender;
    this._listeners = [];
}

Event.prototype = {
    attach : function (listener) {
        this._listeners.push(listener);
    },
    notify : function (args) {
        var index;

        for (index = 0; index < this._listeners.length; index += 1) {
            this._listeners[index](this._sender, args);
        }
    }
};
```

The View class requires defining controls for interacting with. There are numerous alternatives of interface for the task, but I prefer a most simple one. I want my items to be in a Listbox control and two buttons below it: “plus” button for adding items and “minus” for removing selected item. The support for selecting an item is provided by Listbox’s native functionality. A View class is tightly bound with a Controller class, which “... handles the input event from the user interface, often via a registered handler or callback” (from [wikipedia.org](#)).

Here are the View and Controller classes:

```
/**
 * The View. View presents the model and provides
 * the UI events. The controller is attached to these
 * events to handle the user interaction.
 */

function ListView(model, elements) {
    this._model = model;
    this._elements = elements;

    this.listModified = new Event(this);
    this.addButtonClicked = new Event(this);
    this.delButtonClicked = new Event(this);
```

```
var _this = this;

// attach model listeners
this._model.itemAdded.attach(function () {
    _this.rebuildList();
});

this._model.itemRemoved.attach(function () {
    _this.rebuildList();
});

// attach listeners to HTML controls
this._elements.list.change(function (e) {
    _this.listModified.notify({ index : e.target.selectedIndex });
});

this._elements.addButton.click(function () {
    _this.addButtonClicked.notify();
});

this._elements.delButton.click(function () {
    _this.delButtonClicked.notify();
});

}

ListView.prototype = {
    show : function () {
        this.rebuildList();
    },
    rebuildList : function () {
        var list, items, key;

        list = this._elements.list;
        list.html('');

        items = this._model.getItems();
        for (key in items) {
            if (items.hasOwnProperty(key)) {
                list.append($('<option>' + items[key] + '</option>'));
            }
        }
        this._model.setSelectedIndex(-1);
    }
}
```

```
        }

    };

    /**
     * The Controller. Controller responds to user actions and
     * invokes changes on the model.
     */
function ListController(model, view) {
    this._model = model;
    this._view = view;

    var _this = this;

    this._view.listModified.attach(function (sender, args) {
        _this.updateSelected(args.index);
    });

    this._view.addButtonClicked.attach(function () {
        _this.addItem();
    });

    this._view.delButtonClicked.attach(function () {
        _this.delItem();
    });
}

ListController.prototype = {
    addItem : function () {
        var item = window.prompt('Add item:', '');
        if (item) {
            this._model.addItem(item);
        }
    },
    delItem : function () {
        var index;

        index = this._model.getSelectedIndex();
        if (index !== -1) {
            this._model.removeItemAt(this._model.getSelectedIndex());
        }
    }
}
```

```

    },
    updateSelected : function (index) {
        this._model.setSelectedIndex(index);
    }
};

```

And of course, the Model, View, and Controller classes should be instantiated. The sample, which you can see below, uses the following code to instantiate and configure the classes:

```

$(function () {
    var model = new ListModel(['PHP', 'JavaScript']),
        view = new ListView(model, {
            'list' : $('#list'),
            'addButton' : $('#plusBtn'),
            'delButton' : $('#minusBtn')
        }),
        controller = new ListController(model, view);

    view.show();
});

```

```

<select id="list" size="10"></select>
<button id="plusBtn"> + </button>
<button id="minusBtn"> - </button>

```

```

/*global $ */
/*jslint indent: 4, maxlen: 80, browser: true, nomen: true */

(function () {
    'use strict';

    function Event(sender) {
        this._sender = sender;
        this._listeners = [];
    }

    Event.prototype = {
        attach : function (listener) {
            this._listeners.push(listener);
        }
    }
});

```

```
        },

        notify : function (args) {
            var index;

            for (index = 0; index < this._listeners.length; index += 1) {
                this._listeners[index](this._sender, args);
            }
        }
    };

    /**
     * The Model. Model stores items and notifies
     * observers about changes.
     */
    function ListModel(items) {
        this._items = items;
        this._selectedIndex = -1;

        this.itemAdded = new Event(this);
        this.itemRemoved = new Event(this);
        this.selectedIndexChanged = new Event(this);
    }

    ListModel.prototype = {
        getItems : function () {
            return [].concat(this._items);
        },

        addItem : function (item) {
            this._items.push(item);
            this.itemAdded.notify({ item : item });
        },

        removeItemAt : function (index) {
            var item;

            item = this._items[index];
            this._items.splice(index, 1);
            this.itemRemoved.notify({ item : item });
            if (index === this._selectedIndex) {
                this.setSelectedIndex(-1);
            }
        }
    };
}
```

```
        }

    },

    selectedIndex : function () {
        return this._selectedIndex;
    },

    setSelectedIndex : function (index) {
        var previousIndex;

        previousIndex = this._selectedIndex;
        this._selectedIndex = index;
        this.selectedIndexChanged.notify({ previous : previousIndex });
    }
};

/***
 * The View. View presents the model and provides
 * the UI events. The controller is attached to these
 * events to handle the user interaction.
 */
function ListView(model, elements) {
    this._model = model;
    this._elements = elements;

    this.listModified = new Event(this);
    this.addButtonClicked = new Event(this);
    this.delButtonClicked = new Event(this);

    var _this = this;

    // attach model listeners
    this._model.itemAdded.attach(function () {
        _this.rebuildList();
    });
    this._model.itemRemoved.attach(function () {
        _this.rebuildList();
    });

    // attach listeners to HTML controls
    this._elements.list.change(function (e) {
```

```
        _this.listModified.notify({ index : e.target.selectedIndex });

    });

    this._elements.addButton.click(function () {
        _this.addButtonClicked.notify();
    });

    this._elements.delButton.click(function () {
        _this.delButtonClicked.notify();
    });
}

ListView.prototype = {

    show : function () {
        this.rebuildList();
    },

    rebuildList : function () {
        var list, items, key;

        list = this._elements.list;
        list.html('');

        items = this._model.getItems();
        for (key in items) {
            if (items.hasOwnProperty(key)) {
                list.append($('
' + items[key] + '
'));
            }
        }
        this._model.setSelectedIndex(-1);
    }
};

/**
 * The Controller. Controller responds to user actions and
 * invokes changes on the model.
 */
function ListController(model, view) {
    this._model = model;
    this._view = view;
```

```
var _this = this;

this._view.listModified.attach(function (sender, args) {
    _this.updateSelected(args.index);
});

this._view.addButtonClicked.attach(function () {
    _this.addItem();
});

this._view.delButtonClicked.attach(function () {
    _this.delItem();
});

ListController.prototype = {
    addItem : function () {
        var item = window.prompt('Add item:', '');
        if (item) {
            this._model.addItem(item);
        }
    },
    delItem : function () {
        var index;

        index = this._model.getSelectedIndex();
        if (index !== -1) {
            this._model.removeItemAt(this._model.getSelectedIndex());
        }
    },
    updateSelected : function (index) {
        this._model.setSelectedIndex(index);
    }
};

$(function () {
    var model = new ListModel(['PHP', 'JavaScript']),
        view = new ListView(model, {
            'list' : $('#list'),

```

```
' addButton' : $('#plusBtn'),  
    'delButton' : $('#minusBtn')  
}),  
controller = new ListController(model, view);  
  
view.show();  
});  
}());
```

# 41 : JavaScript Function Memoization

<http://inlehmsterms.net/2015/03/01/javascript-memoization/>

In this post I am going to walk through when you might want to use memoization for your JavaScript functions and how you can easily memoize any function. Before we can go much further, let's define what memoization is.

## What is memoization?

Memoization is an optimization technique where expensive function calls are cached such that the result can be immediately returned the next time the function is called with the same arguments. This method of optimization is not unique to JavaScript and is quite common in many programming languages. It is especially useful in recursive functions as calls are more likely to call with the same arguments while recursing. Take a recursive `factorial` function for example:

```
function factorial(num) {  
    if(num === 1) { return 1 };  
    return num * factorial(num - 1);  
}
```

If we call `factorial(3)`, the function calls `factorial(3)`, `factorial(2)`, and `factorial(1)` will be called. If we memoize this function, another call to `factorial(3)` will not need to recurse, it can simply return the result that it has cached. The real benefit is if we call `factorial(4)`, we will short circuit our recursion, because `factorial(3)` is already cached, so we do not need to recurse any further, we can just use that result.

## Sounds great, sign me up!

We can simply create a `memoize` function that takes another function and modifies it to memoize calls.

```
function memoize(func) {  
    var cache = {};  
    return function() {  
        var key = JSON.stringify(arguments);  
        if(cache[key]) {  
            return cache[key];  
        }  
        else {  
            var val = func.apply(this, arguments);  
            cache[key] = val;  
            return val;  
        }  
    };  
}
```

```
    }
};

}
```

Now we can easily memoize any pure function, like our `factorial` function.

```
var factorial = memoize(function(num) {
  console.log('working for factorial(' + num + ')');
  if(num === 1) { return 1 };
  return num * factorial(num - 1);
});

// first call
console.log(factorial(3));
//=> working for factorial(3)
//=> working for factorial(2)
//=> working for factorial(1)
//=> 6

// successive calls
console.log(factorial(3)); //=> 6
console.log(factorial(3)); //=> 6

// short circuit higher factorial calls
console.log(factorial(4));
//=> working for factorial(4)
//=> 24
```

## Advanced usage

Right now we have memoization working by simply wrapping a given function with our `memoization` function. The results are cached for calls with the same arguments. This is great, but what if the arguments are not our only dependencies. What if we are memoizing a method on an object and that method relies on both the arguments AND other properties on the object? How do we account for these other dependencies? If we do not do anything different, memoizing a function might actually cause it to produce incorrect values (if the other dependencies have changed). We need a way to invalidate the cache for these dependencies as well.

The good news is that we can easily take other dependencies into account. Earlier you might have been wondering why I am using `JSON.stringify` to create my cache keys, and soon you will see how this helps make it extremely easy to add any number of dependencies in addition to a function's arguments.

Let's say we have a `Person` model with a `firstName` and `lastName` as well as a method, `fullName`, that takes an optional argument, `title` and outputs the person's full name.

```
function memoize() { ... }

function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;

  this.fullName = memoize(function(title) {
    return title + ' ' + this.firstName + ' ' + this.lastName;
  });
}
```

All we need to do to memoize this function on the `Person` object, is to update the `memoize` function to take a second argument, `depsFunc`. `depsFunc` will be a function that returns an array of the dependencies. We can then use `depsFunc` as well as `func` to calculate the unique key in our hash.

```
function memoize(func, depsFunc) {
  var cache = {};
  return function() {
    var key = JSON.stringify([depsFunc(), arguments]);
    if(cache[key]) {
      return cache[key];
    }
    else {
      var val = func.apply(this, arguments);
      cache[key] = val;
      return val;
    }
  };
}

function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;

  this.fullName = memoize(
    // calculation
    function(title) {
      console.log('working... ');
    }
  );
}
```

```
        return title + ' ' + this.firstName + ' ' + this.lastName;
    },
    // dependencies
    function() {
        return [this.firstName, this.lastName];
    }.bind(this));
}

// create a new Person
var person = new Person('Jonathan', 'Lehman');

// first call to our memoized function does the work
console.log(person.fullName('Mr.'));
//=> working
//=> Mr. Jonathan Lehman

// successive calls
console.log(person.fullName('Mr.'));
//=> Mr. Jonathan Lehman

// work must be done if dependencies or arguments change

// change arguments
console.log(person.fullName('Mister'));
//=> work
//=> Mister Jonathan Lehman

// change deps
person.firstName = 'Jon';
console.log(person.fullName('Mr.'));
//=> work
//=> Mr. Jon Lehman
```

## Careful, memoization is not a magic bullet

Keep in mind that memoization does not make sense for all function calls. There is a higher memory overhead since we must store our cached results so that we can later recall them as well as an added complexity of using memoization, so it really only makes sense for functions that are computationally expensive.

Also, memoization does not work well for functions that are not pure, that is functions that have side effects. Memoizing only allows us to cache a result, so any other side effects get lost on successive calls. That said,

you can get around this constraint, by returning a function that includes your side effects that you will need to execute after getting the result.

## 42 : Javascript Memoization

<https://www.safaribooksonline.com/library/view/javascript-the-good/9780596517748/ch04s15.html>

Functions can use objects to remember the results of previous operations, making it possible to avoid unnecessary work. This optimization is called *memoization*. JavaScript's objects and arrays are very convenient for this.

Let's say we want a recursive function to compute Fibonacci numbers. A Fibonacci number is the sum of the two previous Fibonacci numbers. The first two are 0 and 1:

```
var fibonacci = function (n) {
    return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);
};

for (var i = 0; i <= 10; i += 1) {
    document.writeln('// ' + i + ': ' + fibonacci(i));
}

// 0: 0
// 1: 1
// 2: 1
// 3: 2
// 4: 3
// 5: 5
// 6: 8
// 7: 13
// 8: 21
// 9: 34
// 10: 55
```

This works, but it is doing a lot of unnecessary work. The `fibonacci` function is called 453 times. We call it 11 times, and it calls itself 442 times in computing values that were probably already recently computed. If we *memoize* the function, we can significantly reduce its workload.

We will keep our memoized results in a `memo` array that we can hide in a closure. When our function is called, it first looks to see if it already knows the result. If it does, it can immediately return it:

```
var fibonacci = (function ( ) {
    var memo = [0, 1];
    var fib = function (n) {
        var result = memo[n];
        if (result === undefined) {
            result = fib(n - 1) + fib(n - 2);
            memo[n] = result;
        }
        return result;
    };
    return fib;
})();
```

```

        if (typeof result !== 'number') {
            result = fib(n - 1) + fib(n - 2);
            memo[n] = result;
        }
        return result;
    };
    return fib;
}();

```

This function returns the same results, but it is called only 29 times. We called it 11 times. It called itself 18 times to obtain the previously memoized results.

We can generalize this by making a function that helps us make memoized functions. The `memoizer` function will take an initial `memo` array and the `formula` function. It returns a `recur` function that manages the memo store and that calls the `formula` function as needed. We pass the `recur` function and the function's parameters to the `formula` function:

```

var memoizer = function (memo, formula) {
    var recur = function (n) {
        var result = memo[n];
        if (typeof result !== 'number') {
            result = formula(recur, n);
            memo[n] = result;
        }
        return result;
    };
    return recur;
};

```

We can now define `fibonacci` with the `memoizer`, providing the initial `memo` array and `formula` function:

```

var fibonacci = memoizer([0, 1], function (recur, n) {
    return recur(n - 1) + recur(n - 2);
});

```

By devising functions that produce other functions, we can significantly reduce the amount of work we have to do. For example, to produce a memoizing factorial function, we only need to supply the basic factorial formula:

```

var factorial = memoizer([1, 1], function (recur, n) {
    return n * recur(n - 1);
});

```

```
});
```

## 43 : Hello World

A "Hello World"-like example of Javascript using the MVC pattern.

<http://sandbox.thewikies.com/javascript-mvc-hello-world/>

```
/*
 * =====
 * A "Hello World"-like example of Javascript using the MVC pattern.
 * ===== */
/* 
 * Model
 */
// a model is where the data object is created.
var ModelExample = function ( data ) {
    // the model instance has a property called "myProperty"
    // created from the data's "yourProperty".
    this.myProperty = data.yourProperty;

    // return the model instance
    return this;
};

// a model constructor might have a function that creates new model instances.
ModelExample.find = function ( id ) {
    // data used to create a new model may come from anywhere
    // but in this example data comes from this inline object.
    var ourData = {
        '123': {
            yourProperty: 'Hello World'
        }
    };
    // get a new model instance containing our data.
    var model = new ModelExample(ourData[id]);
    // return the model.
}
```

```
        return model;
    };

/*  

 * View  

 */

// a view is where the output is created.  

var ViewExample = function ( model ) {  

    this.model = model;  
  

    return this;  
};  
  

// a view might have a function that returns the rendered output.  

ViewExample.prototype.output = function () {  

    // data used to create a template may come from anywhere  

    // but in this example template comes from this inline string.  

    var ourData = '<h1><%= myProperty %></h1>';  
  

    // store this instance for reference in the replace function below  

    var instance = this;  
  

    // return the template using values from the model.  

    return ourData.replace(/<%= \s+(.*?)\s+%\>/g, function (m, m1) {  

        return instance.model[m1];  

    });
};  
  

// a view might have a function that renders the output.  

ViewExample.prototype.render = function () {  

    // this view renders to the element with the id of "output"  

    document.getElementById('output').innerHTML = this.output();
};

/*
 * Controller
*/
```

```
*/  
  
// a controller is where the model and the view are used together.  
var ControllerExample = function () {  
    return this;  
};  
  
// this function uses the Model and View together.  
ControllerExample.prototype.loadView = function ( id ) {  
    // get the model.  
    var model = ModelExample.find( id );  
  
    // get a new view.  
    var view = new ViewExample(model);  
  
    // run the view's "render" function  
    view.render();  
};  
  
/*  
 * Example  
 */  
  
function bootstrapper() {  
    var controller = new ControllerExample;  
    controller.loadView(123);  
}  
  
bootstrapper();  
  
/* ======  
 * A "Hello World"-like example of Javascript using the MVC pattern.  
 * This time with the addition of events.  
 * ====== */
```

```
/*
 * Model
 */

// a model is where the data object is created.
var _Model = function ( data ) {
    // the model instance has a property called "myProperty"
    // created from the data's "yourProperty".
    this.myProperty = data.yourProperty;

    // return the model instance
    return this;
};

// a model constructor might have a function that creates new model instances.
_Model.find = function ( id ) {
    // data used to create a new model may come from anywhere
    // but in this example data comes from this inline object.
    var ourData = {
        '123': {
            yourProperty: 'You clicked.'
        },
        '456': {
            yourProperty: 'You pressed a key.'
        }
    };

    // get a new model instance containing our data.
    var model = new _Model(ourData[id]);

    // return the model.
    return model;
};

/*
 * View
 */
```

```
// a view is where the output is created.  
var _View = function ( model ) {  
    this.model = model;  
  
    return this;  
};  
  
// a view might have a function that returns the rendered output.  
_View.prototype.output = function () {  
    // data used to create a template may come from anywhere  
    // but in this example template comes from this inline string.  
    var ourData = '<h1><%= myProperty %></h1>';  
  
    // store this instance for reference in the replace function below  
    var instance = this;  
  
    // return the template using values from the model.  
    return ourData.replace(/<%= \s+(.*?)\s+%>/g, function (m, m1) {  
        return instance.model[m1];  
    });  
};  
  
// a view might have a function that renders the output.  
_View.prototype.render = function () {  
    // this view renders to the element with the id of "output"  
    document.getElementById('output').innerHTML = this.output();  
};  
  
/*  
 * Controller  
 */  
  
// a controller is where the model and the view are used together.  
var _Controller = {};  
  
// this function uses the Model and View together.  
_Controller.loadView = function ( id ) {  
    // get the model.  
    var model = _Model.find( id );
```

```
// get a new view.  
var view = new _View(model);  
  
// run the view's "render" function  
view.render();  
};  
  
/*  
 * Event  
 */  
  
var Event123 = 1 << 0;  
var Event456 = 1 << 1;  
  
// an event is where something happening is captured.  
var _Event = function (flag) {  
    // check if Event123 was passed  
    if (flag & Event123) {  
        // run the controller's show function  
        _Controller.loadView(123);  
    }  
  
    // check if Event456 was passed  
    if (flag & Event456) {  
        // run the controller's show function  
        _Controller.loadView(456);  
    }  
};  
  
/*  
 * Example  
 */  
  
function bootstrapper() {  
    document.onclick = function () {  
        _Event(Event123);  
    };  
}
```

```
document.onkeydown = function () {
    _Event(Event456);
};

bootstrapper();
```