

JavaScript : Part 3

1 : Dealing with Design Patterns

2 : Seven JavaScript Quirks I Wish I'd Known About

3 : Named function expressions demystified

4 : "Real" Mixins with JavaScript Classes

5 : JavaScript Mixins

6 : Building a simple PubSub system in JavaScript

7 : Build A Simple Javascript App The MVC Way

8 : PubSub Design Pattern

9 : Glossary of Modern JavaScript Concepts: Part 1

10 : ES6 Features

11 : "What's the typeof null?", and other confusing JavaScript Types

12 : Asynchronous vs Deferred JavaScript

13 : An Overview of Client-Side Storage

14 : Back to JavaScript Basics: VARIABLES!

15 : Quick Tip: How to Sort an Array of Objects in JavaScript

16 : Effective Functional JavaScript: First-class and Higher Order Functions

17 : Ingredients of Effective Functional JavaScript: Closures, Partial Application and Currying

18 : Pub-Sub Event Manager

19 : Currying in JavaScript

20 : Currying the callback, or the essence of futures...

21 : Rethinking JavaScript: Eliminate the switch statement for better code

22 : Functional JavaScript: Function Composition For Every Day Use.

23 : Currying in JavaScript

24 : [object Object] Things You Didn't Know About valueOf

25 : Prototypal Inheritance

26 : How to impress me in an interview

27 : Rethinking JavaScript: Replace break by going functional

28 : An Introduction to Functional JavaScript

29 : Filtering and Chaining in Functional JavaScript

30 : A Beginner's Guide to Currying in Functional JavaScript

31 : Higher-Order Functions in JavaScript

32 : Recursion in Functional JavaScript

33 : How to Build and Structure a Node.js MVC Application

34 : A Beginner's Guide to JavaScript Variables and Datatypes

35 : 10 Node.js Best Practices: Enlightenment from the Node Gurus

36 : 10 Tips to Become a Better Node Developer in 2017

37 : Function Composition: Building Blocks for Maintainable Code

38 : Creating Unwritable Properties with Object.defineProperty

39 : Advanced objects in JavaScript

40 : JavaScript: Function Invocation Patterns

41 : Javascript: Object Prototypes

42 : Maps, Sets and Iterators in JavaScript

43 : ES6 Iterators and Generators in Practice

44 : ES6 Iterators and Generators -- 6 exercises and solutions

45 : What you should know about JavaScript regular expressions

46 : all this

47 : Everything you wanted to know about JavaScript scope

48 : Advanced Javascript: Objects, Arrays, and Array-Like objects

49 : Advanced Javascript: Logical Operators and truthy / falsy

50 : Design Patterns - Decorating in JavaScript

51 : Design Patterns - The Builder Pattern in JavaScript

52 : Better JavaScript with ES6, Pt. II: A Deep Dive into Classes

53 : Recursion in JavaScript with ES6, destructuring and rest/spread

54 : A Gentle Introduction to Composition in JavaScript

55 : Enhancing Mixins with Decorator Functions

56 : Currying - the Underrated Concept in Javascript

57 : An Introduction to Functional Programming in JavaScript

58 : Don't Be Scared Of Functional Programming

59 : Classes, Inheritance And Mixins In JavaScript

60 : JavaScript Mixins: Beyond Simple Object Extension

61 : Beautiful Javascript Mixins

62 : Classical Inheritance

63 : Prototypes

64 : Mixins

65 : JavaScript and Functional Programming

66 : Functional programming with Javascript

67 : Understanding JavaScript's async await

68 : Gettin' Freaky Functional w/Curried JavaScript

69 : Composing Synchronous and Asynchronous Functions in JavaScript

70 : Tidying Up a JavaScript Application with Higher-Order Functions

1 : Dealing with Design Patterns

Introduction

Last months I was dealing with Design Patterns, trying to understand how they work and how to implement in javascript, because of the Didgeridoo IDE development, and today I'll write about Module Pattern and Module Revealing Pattern. These are the ones I used for a part of the Didgeridoo project.

By the way, it's better if you know something about Design Patterns before. Creational Patterns, Structural Patterns and Behavioral Patterns. Those three categories you should keep in mind (There also exist Concurrency Patterns, but I will not write about these for now).

Brief Descriptions

Creational Patterns:

This kind of patterns define the way objects are created in a manner that they can be decoupled from the system. Some of those are:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Patterns:

This kind of patterns define the way classes and/or objects are structured. Some of those are:

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns:

This kind of patterns define the way classes and/or objects behave when used to manage algorithms, relationships and responsibilities between them. Some of those are:

- Chain of Responsibility
- Command
- Interpreter
- Iterator

- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

The Module Pattern

As of some programming languages provide mechanism for Module Pattern, in javascript we have to get a little bit tricky to simulate it.

```
var basketModule = (function() {  
    var basket = [];  
    function doSomethingPrivate(){  
        //...  
    }  
  
    function doSomethingElsePrivate(){  
        //...  
    }  
    return {  
        addItem: function(values) {  
            basket.push(values);  
        },  
        getItemCount: function() {  
            return basket.length;  
        },  
        doSomething: doSomethingPrivate(),  
        getTotal: function(){  
            var q = this.getItemCount(), p=0;  
            while(q--){  
                p+= basket[q].price;  
            }  
            return p;  
        }  
    }  
}());  
  
//basketModule is an object with properties which can also be methods  
basketModule.addItem({item:'bread',price:0.5});  
basketModule.addItem({item:'butter',price:0.3});
```

```

console.log(basketModule.getItemCount());
console.log(basketModule.getTotal());

//however, the following will not work:
console.log(basketModule.basket); // (undefined as not inside the returned object)
console.log(basket); // (only exists within the scope of the closure)

```

Example by Addy Osmani (<http://addyosmani.com/resources/essentialjsdesignpatterns/book/>).

At the beginning, the Module Pattern in javascript was thought as a way to simulate Classes with their own Execution Contexts and Scope in a DRY (Don't Repeat Yourself) manner. To understand how we can simulate modules in javascript, first we have to know about the concept of [Closures](#).

In the example above you can see that the public part is done by returning an object, which acts like an interface for the basketModule module. And there also exists a private part that has its own scope and maintain chainability with the global scope. That way, public parts of your code are able to touch the private parts, however the outside world is unable to touch the class's private parts (nice joke Addy Osmani and David Engfer).

Note that the Module Pattern itself belongs to Creational Patterns, due to this define the way you should create the objects/modules, and also belongs to Structural Patterns, due to the nature of organizing the code in modules is a way of code structuring. So, many people, consider the Module Pattern as a different kind of "category" apart from the others. I think it's just a pattern that define both aspects (creational and structural).

The Revealing Module Pattern

This pattern is an improved version of the Module Pattern by Christian Heilmann (Mozilla).

It consists in a better way of organizing the code, like not writing the code of the public methods in the returning part of the code, but pointing to them in the private part. In other words, define all your methods and variables in the private part and keep the public part clean, as follow:

```

var myRevealingModule = (function(){

    var name = 'John Smith';
    var age = 40;

    function updatePerson(){
        name = 'John Smith Updated';
    }

    function setPerson () {
        name = 'John Smith Set';
    }
}
```

```
function getPerson () {
    return name;
}

//Public interface
return {
    set: setPerson,
    get: getPerson
};
}();

// Sample usage:
myRevealingModule.get();
```

It seems like there's no difference but it will help you to maintain your code well organized and clean.

2 : Index

This post will go over the Javascript language from the point of view of a Java developer, focusing on the differences between the two languages and the frequent pain points. We will go over the following:

- Objects Only, No Classes
- Functions are just Values
- The 'this' Keyword
- Classic vs Prototypal Inheritance
- Constructors vs Constructor Functions
- Closures vs Lambdas
- Encapsulation and Modules
- Block Scope and Hoisting

Why Javascript in the Java World ?

A lot of Java frontend development work is done using Java/XML based frameworks like JSF or GWT. The framework developers themselves need to know Javascript, but in principle the application developers don't. However the reality is that:

- For doing custom component development in for example Primefaces (JSF), it's important to know Javascript and jQuery.
- In GWT, integrating at least some third-party Javascript widgets is common and cost effective.

The end result is that Javascript is usually needed to do at least the last 5 to 10% of frontend work, even using Java frameworks. Also it's starting to get more and more used for polyglot enterprise development, alongside [Angular](#) for example.

The good news is that, besides a few gotchas that we will get into, Javascript is a very learnable language for a Java developer.

Objects Only - No Classes

One of the most surprising things about Javascript is that although it's an object oriented language, there are no classes (although the [new Ecmascript 6 version](#) will have them).

Take for example this program, that initializes an empty object and set's two properties:

```
// create an empty object - no class was needed !!
var superhero = {};

superhero.name = 'Superman';
superhero.strength = 100;
```

Javascript objects are just like a Java HashMap of related properties, where the keys are Strings only. The following would be the 'equivalent' Java code:

```
Map<String, Object> superhero = new HashMap<>();

superhero.put("name", "Superman");
superhero.put("strength", 100);
```

This means that a Javascript object is just a multi-level 'hash map' of key/value pairs, with no class definition needed.

Functions Are Just Values

Functions in Javascript are just values of type `Function`, it's a simple as that! Take for example:

```
var flyFunction = function() {
    console.log('Flying like a bird!');
};

superhero.fly = flyFunction;
```

This creates a function (a value of type `Function`) and assigns it to a variable `flyFunction`. A new property named `fly` is then created in the `superhero` object, that can be invoked like this:

```
// prints 'Flying like a bird!' to the console
superhero.fly();
```

Java does **not** have the equivalent of the Javascript `Function` type, but almost. Take for example the `SuperHero` class that takes a `Power` function:

```
public interface Power {
    void use();
}

public class SuperHero {

    private Power flyPower;

    public void setFly(Power flyPower) {
        this.flyPower = flyPower;
    }
}
```

```

    }

    public void fly() {
        flyPower.use();
    }
}

```

This is how to pass `SuperHero` a function in Java 7 and 8:

```

// Java 7 equivalent
Power flyFunction = new Power() {
    @Override
    public void use() {
        System.out.println("Flying like a bird ...");
    }
};

// Java 8 equivalent
superman.setFly(
    ()->System.out.println("Flying like a bird ..."));

superman.fly();

```

So although a `Function` type does not exist in Java 8, this ends up not preventing a 'Javascript-like' functional programming style.

But if we pass functions around, what happens to the meaning of the `this` keyword?

The 'this' Keyword Usage

What Javascript allows to do with `this` is quite surprising compared to the Java world. Let's start with an example:

```

var superman = {

    heroName: 'Superman',

    sayHello: function() {
        console.log("Hello, I'm " + this.heroName );
    }
};

```

```
superman.sayHello();
```

This program creates an object `superman` with two properties: a String `heroName` and a Function named `sayHello`. Running this program outputs as expected `Hello, I'm Superman.`

What if we pass the function around?

By passing around `sayHello`, we can easily end up in a context where there is no `heroName` property:

```
var failThis = superman.sayHello;

failThis();
```

Running this snippet would give as output: `Hello, I'm undefined.`

Why does `this` not work anymore?

This is because the variable `failThis` belongs to the global scope, which contains no member variable named `heroName`. To solve this:

In Javascript the value of the `this` keyword is completely overridable to be anything that we want!

```
// overrides 'this' with superman
hello.call(superman);
```

The snippet above would print again `Hello, I'm Superman`. This means that the value of `this` depends on both the context on which the function is called, and on *how* the function is called.

Classic vs Prototypal Inheritance

In Javascript, there is no class inheritance, instead objects can inherit directly from other objects. The way this works is that each object has an implicit property that points to a 'parent' object.

That property is called `__proto__`, and the parent object is called the object's **prototype**, hence the name Prototypal Inheritance.

How does `prototype` work?

When looking up a property, Javascript will try to find the property in the object itself. If it does not find it then it tries in its prototype, and so on. For example:

```

var avengersHero = {
  editor: 'Marvel'
};

var ironMan = {};

ironMan.__proto__ = avengersHero;

console.log('Iron Man is copyrighted by ' + ironMan.editor);

```

This snippet will output Iron Man is copyrighted by Marvel.

As we can see, although the ironMan object is empty, its prototype does contain the property editor, which gets found.

How does this compare with Java inheritance?

Let's now say that the rights for the Avengers were bought by DC Comics:

```
avengersHero.editor = 'DC Comics';
```

If we call ironMan.editor again, we now get Iron Man is copyrighted by DC Comics. All the existing object instances with the avengersHero prototype now see DC Comics without having to be recreated.

This mechanism is very simple and very powerful. Anything that can be done with class inheritance can be done with prototypal inheritance. But what about constructors?

Constructors vs Constructor Functions

In Javascript an attempt was made to make object creation similar to languages like Java. Let's take for example:

```

function SuperHero(name, strength) {
  this.name = name;
  this.strength = strength;
}

```

Notice the capitalized name, indicating that it's a constructor function. Let's see how it can be used:

```

var superman = new SuperHero('Superman', 100);

console.log('Hello, my name is ' + superman.name);

```

This code snippet outputs Hello, my name is Superman.

You might think that this looks just like Java, and that is exactly the point! What this new syntax really does is to it creates a new empty object, and then calls the constructor function by forcing this to be the newly created object.

Why is this syntax not recommended then?

Let's say that we want to specify that all super heroes have a sayHello method. This could be done by putting the sayHello function in a common prototype object:

```
function SuperHero(name, strength) {
    this.name = name;
    this.strength = strength;
}

SuperHero.prototype.sayHello = function() {
    console.log('Hello, my name is ' + this.name);
}

var superman = new SuperHero('Superman', 100);
superman.sayHello();
```

This would output Hello, my name is Superman.

But the syntax SuperHero.prototype.sayHello looks anything but Java like! The new operator mechanism sort of half looks like Java but at the same time is completely different.

Is there a recommended alternative to new ?

The recommended way to go is to ignore the Javascript new operator altogether and use Object.create :

```
var superHeroPrototype = {
    sayHello: function() {
        console.log('Hello, my name is ' + this.name);
    }
};

var superman = Object.create(superHeroPrototype);
superman.name = 'Superman';
```

Unlike the new operator, one thing that Javascript absolutely got right where Closures.

Closures vs Lambdas

Javascript Closures are not that different from Java anonymous inner classes used in a certain way. take for example the `FlyingHero` class:

```
public interface FlyCommand {
    public void fly();
}

public class FlyingHero {

    private String name;

    public FlyingHero(String name) {
        this.name = name;
    }

    public void fly(FlyCommand flyCommand) {
        flyCommand.fly();
    }
}
```

We can pass it a fly command like this in Java 8:

```
String destination = "Mars";
superMan.fly(() -> System.out.println("Flying to " +
    destination));
```

The output of this snippet is `Flying to Mars`. Notice that the `FlyCommand` lambda had to 'remember' the variable `destination`, because it needs it for executing the `fly` method later.

This notion of a function that remembers about variables outside its block scope for later use is called a **Closure** in Javascript. For further details, have a look at this blog post [Really Understanding Javascript Closures](#).

What is the main difference between Lambdas and Closures?

In Javascript a closure looks like this:

```
var destination = 'Mars';

var fly = function() {
```

```

        console.log('Fly to ' + destination);
    }

fly();

```

The Javascript closure, unlike the Java Lambda does not have the constraint that the `destination` variable must be immutable (or effectively immutable since Java 8).

This seemingly innocuous difference is actually a 'killer' feature of Javascript closures, because it allows them to be used for creating encapsulated modules.

Modules and Encapsulation

There are no classes in Javascript and no `public` / `private` modifiers, but then again take a look at this:

```

function createHero(heroName) {

    var name = heroName;

    return {
        fly: function(destination) {
            console.log(name + ' flying to ' + destination);
        }
    };
}

```

Here a function `createHero` is being defined, which returns an object which has a function `fly`. The `fly` function 'remembers' `name` when needed.

How do Closures relate to Encapsulation?

When the `createHero` function returns, noone else will ever be able to directly access `name`, except via `fly`.

Let's try this out:

```

var superman = createHero('SuperMan');

superman.fly('The Moon');

```

The output of this snippet is `SuperMan flying to The Moon`. But happens if we try to access `name` directly ?

```
console.log('Hero name = ' + superman.name);
```

The result is `Hero name = undefined`. The function `createHero` is said to be a Javascript encapsulated **module**, with closed 'private' member variables and a 'public' interface returned as an object with functions.

Block Scope and Hoisting

Understanding block scope in Javascript is simple: there is no block scope! Take a look at this example:

```
function counterLoop() {

    console.log('counter before declaration = ' + i);

    for (var i = 0; i < 3 ; i++) {
        console.log('counter = ' + i);
    }

    console.log('counter after loop = ' + i);
}

counterLoop();
```

By looking at this coming from Java, you might expect:

- error at line 3: 'variable i does not exist'
- values 0, 1, 2 are printed
- error at line 9: 'variable i does not exist'

It turns out that only one of these three things is true, and the output is actually this:

```
counter before declaration = undefined
counter = 0
counter = 1
counter = 2
counter after loop = 3
```

Because there is no block scope, the loop variable `i` is visible for the **whole** function. This means:

- line 3 sees the variable declared but not initialized
- line 9 sees `i` after the loop has terminated

What might be the most puzzling is that line 3 actually sees the variable declared but undefined, instead of throwing `i is not defined`.

This is because the Javascript interpreter first scans the function for a list of variables, and then goes back to interpret the function code lines one by one.

The end result is that it's like the variable `i` was **hoisted** to the top, and this is what the Javascript runtime actually 'sees':

```
function counterLoop() {  
  
    var i; // i is 'seen' as if declared here!  
  
    console.log('counter before declaration = ' + i);  
  
    for (i = 0; i < 3 ; i++) {  
        console.log('counter = ' + i);  
    }  
  
    console.log('counter after loop: ' + i);  
}
```

To prevent surprises caused by hoisting and lack of block scoping, it's a recommended practice to declare variables always at the top of functions.

This makes hoisting explicit and visible by the developer, and helps to avoid bugs. The next version of Javascript (EcmaScript 6) will include a [new keyword 'let' to allow block scoping](#).

Conclusions

The Javascript language shares a lot of similarities with Java, but also some huge differences. Some of the differences like inheritance and constructor functions are important, but much less than one would expect for day to day programming.

Some of these features are needed mostly by library developers, and not necessarily for day to day application programming. This is unlike some of their Java counterparts which are needed every day.

So if you are hesitant to give it a try, don't let some of these features prevent you from going further into the language.

One thing is for sure, at least *some* Javascript is more or less inevitable when doing Java frontend development, so it's really worth to give it a try.

If you are interested in Type-Safe Frontend Development

If that is the case, you might want to have a look at our ongoing YouTube course on Angular 2 - Getting Started With Angular 2:

3 : Seven JavaScript Quirks I Wish I'd Known About

If you are new to JavaScript or it has only been a minor part of your development effort until recently, you may be feeling frustrated. All languages have their quirks -- but the paradigm shift from strongly typed server-side languages to JavaScript can feel especially confusing at times. I've been there! A few years ago, when I was thrust into full time JavaScript development, there were *many* things I wish I'd known going into it. In this article, I'll share a few of these quirks in hopes that I might spare you some of the headaches I endured. This isn't an exhaustive list -- just a sampling -- but hopefully it will shed some light on the language as well as show how powerful it can be once you get past these kinds of hurdles.

The Quirks We'll look at:

1. [Equality](#)
2. [Dot Notation vs Bracket Notation](#)
3. [Function Context](#)
4. [Function Declarations vs Function Expressions](#)
5. [Named vs Anonymous Functions](#)
6. [Immediately Invoked Function Expressions](#)
7. [typeof vs Object.prototype.toString](#)

1.) Equality

Coming from C#, I was well familiar with the `==` comparison operator. Value types (& strings) are either equal (have the same value) or they aren't. Reference types are either equal -- as in pointing to the same reference -- or NOT. (Let's just pretend you're not overloading the `==` operator, or implementing your own `Equals` and `GetHashCode` methods.) I was surprised to learn that JavaScript has *two* equality operators: `==` and `===`. Most of the code I'd initially seen used `==`, so I followed suit and wasn't aware of what JavaScript was doing for me as I ran code like this:

```
var x = 1;

if(x == "1") {
    console.log("YAY! They're equal!");
}
```

So what dark magic is *this*? How can the *integer* 1 equal the *string* "1"?

In JavaScript, there's equality (`==`) and then there's *strict* equality (`===`). The equality comparison operator will coerce the operands to the same type and *then* perform a strict equality comparison. So in the above example, the string "1" is being converted, behind the scenes, to the integer 1, and then compared to our variable `x`.

Strict equality doesn't coerce the types for you. If the operands aren't of the same type (as in the integer 1 and string "1"), then they *aren't* equal:

```

var x = 1;

// with strict equality, the types must be the *same* for it to be true
if(x === "1") {
    console.log("Sadly, I'll never write this to the console");
}

if(x === 1) {
    console.log("YES! Strict Equality FTW.")
}

```

You might already be thinking of the kinds of horrors that could occur when type coercion is done for you -- the sort of assumptions that you could make that misrepresent the true nature of the values in your app which lead to difficult-to-find-yet-hiding-in-plain-sight bugs. It's no surprise, then, that the general rule of thumb recommended by experienced JavaScript developers is to *always use strict equality*.

2.) Dot Notation vs Bracket Notation

Depending on what other languages you hail from, you probably weren't too surprised to see these forms of accessing a property on an object and accessing an element in an array in JavaScript:

```

// getting the "firstName" value from the person object:
var name = person.firstName;

// getting the 3rd element in an array:
var theOneWeWant = myArray[2]; // remember, 0-based index

```

However, did you know it's possible to use bracket notation to reference object members as well? For example:

```
var name = person["firstName"];
```

Why would this be useful? While you'll probably use dot notation the majority of the time, there are a few instances where bracket notation makes certain approaches possible that couldn't be done otherwise. For example, I will often refactor large `switch` statements into a [dispatch table](#), so that something like this:

```

var doSomething = function(dowhat) {
    switch(dowhat) {
        case "doThisThing":
            // more code...
            break;
        case "doThatThing":
            // more code...
            break;
    }
}

```

```

        case "doThisOtherThing":
            // more code....
            break;
        // additional cases here, etc.
        default:
            // default behavior
            break;
    }
}

```

Can be transformed into something like this:

```

var thingsWeCanDo = {
    doThisThing      : function() { /* behavior */ },
    doThatThing      : function() { /* behavior */ },
    doThisOtherThing : function() { /* behavior */ },
    default          : function() { /* behavior */ }
};

var doSomething = function(dowhat) {
    var thingToDo = thingsWeCanDo.hasOwnProperty(dowhat) ? dowhat : "default"
    thingsWeCanDo[thingToDo]();
}

```

There's nothing inherently wrong with using a `switch` (and in many cases, if you're iterating and performance is a huge concern, a `switch` may outperform the dispatch table). However, dispatch tables offer a nice way to organize and extend behavior, and bracket notation makes it possible by allowing you to reference a property dynamically at runtime.

3.) Function Context

There have been a number of great blog posts about properly understanding the "`this` context" in JavaScript (and I'll link to several at the bottom of this post), but it definitely makes my list of "things I wish I'd known" right away. It's really not difficult to look at code and confidently know what `this` is at any point -- you just have to learn a couple of rules. Unfortunately, many explanations I read about it early on just added to my confusion. As a result, I've tried to simplify the explanation for developers new to JavaScript.

First -- Start With a Global Assumption

By default, until there's a reason for the execution context to change, `this` refers to the *global object*. In the browser, that would be the `window` object (or `global` in node.js).

Second -- the Value of `this` in Methods

When you have an object with a member that is a function, invoking that method *from the parent object* makes `this` the parent object. For example:

```
var marty = {
  firstName: "Marty",
  lastName: "McFly",
  timeTravel: function(year) {
    console.log(this.firstName + " " + this.lastName + " is time traveling to " +
year);
  }
}

marty.timeTravel(1955);
// Marty McFly is time traveling to 1955
```

You might already know that you can take the `marty` object's `timeTravel` method and create a new reference to it from another object. This is actually quite a powerful feature of JavaScript -- enabling us to apply behavior (functions) to more than one target instance:

```
var doc = {
  firstName: "Emmett",
  lastName: "Brown",
}

doc.timeTravel = marty.timeTravel;
```

So -- what happens if we invoke `doc.timeTravel(1885)`?

```
doc.timeTravel(1885);
// Emmett Brown is time traveling to 1885
```

Again -- deep dark magic. Well, not really. Remember, when you're invoking a *method*, the `this` context is the parent object it's being invoked from. Hold on to your DeLoreans, though, cause it's about to get heavy.

(Note: I've updated this section for clarity and to make some corrections from the original post.)

What happens when we save a reference to the `marty.TimeTravel` method and invoke our saved reference? Let's look:

```
var getBackInTime = marty.timeTravel;
getBackInTime(2014);
// undefined undefined is time traveling to 2014
```

Why "undefined undefined"?! Why not "Marty McFly"?

Let's ask an important question: *What's the parent/owning object when we invoke our `getBackInTime` function?*

While the `getBackInTime` function will technically exist on the window, we're invoking it as a *function*, not a method of an object. When we invoke a function like this -- with no owning object -- the `this` context will be the global object. [David Shariff](#) has a great way of describing this:

Whenever a function is called, we must look at the immediate left side of the brackets / parentheses "()"". If on the left side of the parentheses we can see a reference, then the value of "this" passed to the function call is exactly of which that object belongs to, otherwise it is the global object.

Since `getBackInTime`'s `this` context is the `window` -- which does not have `firstName` and `lastName` properties -- that explains why we see "undefined undefined".

So we know that invoking a function directly -- no owning object -- results in the `this` context being the global object. But I also said that I knew our `getBackInTime` function would exist on the window. How do I know this? Well, unless I've wrapped the `getBackInTime` in a different scope (which we'll investigate when we discuss immediately-invoked-function-expressions), then any vars I declare get attached to the window. Here's proof from Chrome's console:

```
> window.getBackInTime
function (year) {
  console.log(this.firstName + " " + this.lastName + " is time traveling to " + year);
}
```

This is the perfect time to talk about one of the main areas that `this` trips up developers: subscribing event handlers.

Third (really just an extension of #2) -- The Value of `this` in Methods When Invoked Asynchronously

So, let's pretend we want to invoke our `marty.timeTravel` method when someone clicks a button:

```
var flux = document.getElementById("flux-capacitor");
flux.addEventListener("click", marty.timeTravel);
```

With the above code, when the user clicks the button, we'll see "undefined undefined is time traveling to [object MouseEvent]". WAT?! Well -- the first, and most obvious, issue is that we're not providing the `year` argument to our `timeTravel` method. Instead, we subscribed the method directly as an event handler, and the `MouseEvent` argument is being passed to the event handler as the first arg. That's easy to fix, but the real issue is that we're seeing "undefined undefined" again. Don't despair -- you already know why this is the case (even if you don't realize it yet). Let's make a change to our `timeTravel` function to log whatever `this` is to help give us some insight:

```
marty.timeTravel = function(year) {
  console.log(this.firstName + " " + this.lastName + " is time traveling to " + year);
  console.log(this);
};
```

Now -- when we click the button, we should see something similar to the following output in our browser console:

```
undefined undefined is time traveling to [object MouseEvent]
<button id="flux-capacitor"></button>
```

Our second `console.log` shows us the `this` context when the method was invoked -- and it's the actual button element which we subscribed to. Does this surprise you? Just like earlier -- when we assigned the `marty.timeTravel` reference to our `getBackInTime` var -- a reference to `marty.timeTravel` was saved as our event handler and is being invoked, but not from the "owning" `marty` object. In this case, it's being invoked asynchronously by the underlying [event emitting](#) implementation of the button element instance.

So -- is it possible make `this` what we want it to be? Absolutely! In this case, the solution is deceptively simple. Instead of subscribing `marty.timeTravel` directly as the event handler, we can use an anonymous function as our event handler, and call `marty.timeTravel` from within it. This also provides the opportunity to fix our issue with the missing `year` parameter.

```
flux.addEventListener("click", function(e) {
  marty.timeTravel(someYearValue);
});
```

Clicking the button now will result in something similar to this output on the console:

```
Marty McFly is time traveling to 1955
▶ Object {firstName: "Marty", lastName: "McFly", timeTravel: function}
```

Success! But why did this work? Think of how we're invoking the `timeTravel` method. In our first button click example, we subscribed the method reference *itself* as the event handler, so it was not being called from the parent object of `marty`. In the second example, it's our anonymous function that will have a `this` of the button element, and when we invoke `marty.timeTravel`, we're calling it from the `marty` parent object, and the `this` will be `marty`.

Fourth -- The Value of `this` Inside Constructor Functions.

When you use a constructor function to create a new instance of an object, the `this` value inside the function is the new object that's being created. For example:

```
var TimeTraveler = function(fName, lName) {
  this.firstName = fName;
  this.lastName = lName;
  // Constructor functions return the
  // newly created object for us unless
  // we specifically return something else
};

var marty = new TimeTraveler("Marty", "McFly");
```

```
console.log(marty.firstName + " " + marty.lastName);
// Marty McFly
```

Using Call, Apply & Bind

You might already suspect, given the above examples, that there might be some language-level features that would allow us to invoke a function and tell it at runtime what `this` should be. You'd be right. The `call` and `apply` methods that exist on the `Function` prototype both allow us to invoke a function and pass in a `this` value.

`call`'s method signature takes the `this` arg, followed by the parameters the function being invoked would take as separate arguments:

```
someFn.call(this, arg1, arg2, arg3);
```

'`apply`' takes '`this`' as the first arg, followed by an array of the remaining arguments:

```
someFn.apply(this, [arg1, arg2, arg3]);
```

Our '`doc`' and '`marty`' instances can time travel on their own, but [Einstein](#) needed their help to time travel. So let's add a method to our '`doc`' instance from earlier (the one we copied the '`timeTravel`' method to), so that '`doc`' can cause an '`einstein`' instance to travel in time:

```
doc.timeTravelFor = function(instance, year) {
  this.timeTravel.call(instance, year);
  // alternate syntax if you used apply would be
  // this.timeTravel.apply(instance, [year]);
};
```

Now it's possible to send Einstein on his way:

```
var einstein = {
  firstName: "Einstein",
  lastName: "(the dog)"
};
doc.timeTravelFor(einstein, 1985);
// Einstein (the dog) is time traveling to 1985
```

I know this is a contrived example, but it's enough to give you a glimpse of the power of applying functions to other objects.

There's still one other possibility we haven't explored. Let's give our `marty` instance a `goHome` method that's simply a shortcut to calling `this.timeTravel(1985)`:

```
marty.goHome = function() {
    this.timeTravel(1985);
}
```

However, we know that if we subscribe `marty.goHome` as the event handler to our button's click event, that `this` will be the button -- and unfortunately buttons don't have a `timeTravel` method. We could use our approach from earlier -- where an anonymous function was the event handler, and it invoked the method on the `marty` instance -- but we have another option, the `bind` function:

```
flux.addEventListener("click", marty.goHome.bind(marty));
```

The `bind` function actually returns a *new* function, with the `this` value of the new function set to what you provide as the argument. If you're supporting older browsers (less than IE9, for example), then you'll need to [shim the bind function](#) (or, if you're using jQuery, you can use `$.proxy`; both underscore & lodash provide `_.bind`).

One important thing to remember is that if you use `bind` on a *prototype* method, it creates an instance-level method, which bypasses the advantages of having methods on the prototype. It's not *wrong*, just something to be aware of. I write more about this particular issue [here](#).

4.) Function Expressions vs Function Declarations

There are two main ways you will typically see functions defined in JavaScript (though ES6 will introduce [another](#)): function declarations and function expressions.

Function Declarations do not require a `var` keyword. In fact, as [Angus Croll says](#): "It's helpful to think of them as *siblings of Variable Declarations*." For example:

```
function timeTravel(year) {
    console.log(this.firstName + " " + this.lastName + " is time traveling to " + year);
}
```

The function name `timeTravel` in the above example is visible not only inside the scope it was declared, but inside the function itself as well (this is especially useful for recursive function calls). Function declarations are, by nature, *named functions*. In other words, the above function's `name` property is `timeTravel`.

Function Expressions define a function and assign it to a variable. They typically look like this:

```
var someFn = function() {
    console.log("I like to express myself...");
};
```

It's possible to name function expressions as well -- though, unlike function declarations, a named function expression's name is only accessible *inside* its scope:

```

var someFn = function iHazName() {
    console.log("I like to express myself...");

    if(needsMoreExpressing) {
        iHazName(); // the function's name can be used here
    }
};

// you can call someFn() here, but not iHazName()
someFn();

```

Discussing function expressions and declarations wouldn't be complete without at least mentioning "hoisting" -- where function and variable declarations are moved to the top of the containing scope by the interpreter. We're not going to cover hoisting here, but be sure to read two great explanations by [Ben Cherry](#) and [Angus Croll](#).

5.) Named vs Anonymous Functions

Based on what we've just discussed, you might have guessed that an "anonymous" function is a function without a name. Most JavaScript developers would quickly recognize the second argument below as an anonymous function:

```

someElement.addEventListener("click", function(e) {
    // I'm anonymous!
});

```

However, it's also true that our `marty.timeTravel` method is anonymous as well:

```

var marty = {
    firstName: "Marty",
    lastName: "McFly",
    timeTravel: function(year) {
        console.log(this.firstName + " " + this.lastName + " is time traveling to " +
year);
    }
}

```

Since function declarations must have a name, only function *expressions* can be anonymous.

6.) Immediately Invoked Function Expressions

And since we're talking about function expressions, one thing I wish I'd known right away: the Immediately Invoked Function Expression (IIFE). There are a number of good posts on IIFEs (I'll list a few at the end), but in a nutshell, it's a function expression that is not assigned to a variable to be executed later, it's executed *immediately*. It might help to see this take shape in a browser console.

First -- let's start with just entering a function expression -- but not assigning it -- and see what happens:

```
> function() { return "oh, hai!"; }
✖ ➤ SyntaxError: Unexpected token (
```

That's not valid JavaScript syntax -- it's a function declaration missing its name. However, to make it an expression, we just need to surround it with parentheses:

```
> (function() { return "oh, hai!"; })();
"oh, hai!"
> (function() { return "oh, hai!"; })
function () { return "oh, hai!"; }
```

Making this an expression immediately returns our anonymous function to the console (remember, we're not assigning it, but since it's an expression we get its value back). So -- we have the "function expression" part of "immediately invoked function expression". To get the "immediately invoked" aspect, we call what the expression returns by adding another set of parentheses after the expression (just like we'd invoke any other function):

```
> (function() { return "oh, hai!"; })();
"oh, hai!"
```

"But wait, Jim! I think I've seen this before with the invocation parentheses *inside* the expression parentheses". Indeed you probably have -- it's perfectly legal syntax (and is well known to be Douglas Crockford's preferred syntax):

```
> (function() { return "oh, hai!"; })();
"oh, hai!"
```



Either approach on where to put the invocation parentheses is usable, however I highly recommend reading [one of the best explanations ever](#) on why & when it could matter.

Ok, great -- now we know *what* an IIFE *is* -- why is it useful?

It helps us control *scope* -- an essential part of any JavaScript endeavor! The `marty` instance we've looked at earlier was created in the *global* scope. This means that the window (assuming we're in a browser), will have a `marty` property. If we write all of our JavaScript code this way, we'll quickly end up with a metric ton of vars declared in the global scope, polluting the window with our app's code. Even in the best case scenarios, it's bad practice to leak that many details into the global scope, but what happens when someone names a variable the same name as an *already existing* property on the window? It gets overwritten!

For example, if your favorite "Amelia Earhart" fan site declares a `navigator` variable in the global scope, this is a before and after look at what happens:

```
> window.navigator
  ► Navigator {geolocation: Geolocation, webkitPersistentStorage: StorageQuota,
    webkitTemporaryStorage: StorageQuota, doNotTrack: null, onLine: true...}
> var navigator = "Fred Noonan";
undefined
> window.navigator
  "Fred Noonan"
```

OOPS!

Obviously -- polluting the global scope is **bad**. JavaScript utilizes *function scope* (not block scope, if you're coming from C# or Java, this is important!), so the way to keep our code from polluting the global scope is to create a *new* scope, and we can use an IIFE to do this since its contents will be inside its own function scope. In the example below, I'm showing you the `window.navigator` value in the console, and then I create an IIFE to wrap the behavior and data specific to Amelia Earhart. This IIFE happens to return an object that will serve as our "application namespace". Inside the IIFE I declare a `navigator` variable to show that it will *not* overwrite the `window.navigator` value.

```
> window.navigator
  ► Navigator {geolocation: Geolocation, webkitPersistentStorage: StorageQuota,
    webkitTemporaryStorage: StorageQuota, doNotTrack: null, onLine: true...}
> var amelia = (function() {
  var navigator = "Fred Noonan";
  return {
    getNavigator: function() {
      return navigator + " disappeared with Amelia Earhart on July 2, 1937."
    }
  };
})();
undefined
> window.amelia.getNavigator();
  "Fred Noonan disappeared with Amelia Earhart on July 2, 1937."
> window.navigator
  ► Navigator {geolocation: Geolocation, webkitPersistentStorage: StorageQuota,
    webkitTemporaryStorage: StorageQuota, doNotTrack: null, onLine: true...}
```

The IIFE

As an added bonus, the IIFE we created above is the beginnings of a module pattern in JavaScript. I'll include some links at the end so you can explore module patterns further.

7.) The `typeof` Operator and `Object.prototype.toString`

Eventually, you'll probably find yourself in a situation where you need to inspect the type of a value passed into a function, or something similar. The `typeof` operator might seem the obvious choice, however, it's not terribly helpful. For example, what happens when we call `typeof` on an object, an array, a string and a regular expression?

```
> typeof { foo: 'bar' };
"object"
> typeof ['foo', 'bar'];
"object"
> typeof "foobar";
"string"
> typeof /foo|bar/;
"object"
```

Well -- at least we can tell our strings apart from objects, arrays and regular expressions, right? Thankfully, we can take a different approach to get more accurate information about type we're inspecting. We'll use the `Object.prototype.toString` function and apply our knowledge of the `call` method from earlier:

```
> Object.prototype.toString.call({ foo: 'bar' });
"[object Object]"
> Object.prototype.toString.call(['foo', 'bar']);
"[object Array]"
> Object.prototype.toString.call("foobar");
"[object String]"
> Object.prototype.toString.call(/foo|bar/);
"[object RegExp]"
```

Why are we using the `toString` on `Object.prototype`? Because it's possible for 3rd party libraries, as well as your own code, to override the `toString` method with an instance method. By going to the `Object.prototype`, we can force the original `toString` behavior on an instance.

If you know what `typeof` will return and you don't need to check beyond what it will give you (for example, you just need to know if something is a string or not), then using `typeof` is perfectly fine. However, if you need to tell arrays from objects, regexes from objects, etc., use `Object.prototype.toString`.

Where to Go Next

I've benefitted tremendously from the insights of other JavaScript developers, so please check these links out and give these people a pat on the back for teaching the rest of us!

4 : Named function expressions demystified

1. [Introduction](#)
2. [Function expressions vs. Function declarations](#)
3. [Function Statements](#)
4. [Named function expressions](#)
5. [Function names in debuggers](#)
6. [JScript bugs](#)
7. [JScript memory management](#)
8. [Tests](#)
9. [Safari bug](#)
10. [SpiderMonkey peculiarity](#)
11. [Solution](#)
12. [Alternative solution](#)
13. [WebKit's displayName](#)
14. [Future considerations](#)
15. [Credits](#)

Introduction

Surprisingly, a topic of named function expressions doesn't seem to be covered well enough on the web. This is probably why there are so many misconceptions floating around. In this article, I'll try to summarize both — theoretical and practical aspects of these wonderful Javascript constructs; the good, bad and ugly parts of them.

In a nutshell, named function expressions are useful for one thing only — **descriptive function names in debuggers and profilers**. Well, there is also a possibility of using function names for recursion, but you will soon see that this is often impractical nowadays. If you don't care about debugging experience, you have nothing to worry about. Otherwise, read on to see some of the cross-browser glitches you would have to deal with and tips on how work around them.

I'll start with a general explanation of what function expressions are how modern debuggers handle them. Feel free to skip to a [final solution](#), which explains how to use these constructs safely.

Function expressions vs. Function declarations

One of the two most common ways to create a function object in ECMAScript is by means of either *Function Expression* or *Function Declaration*. The difference between two is **rather confusing**. At least it was to me. The only thing ECMA specs make clear is that *Function Declaration* must always have an *Identifier* (or a function name, if you prefer), and *Function Expression* may omit it:

FunctionDeclaration :

```
function Identifier ( FormalParameterList opt ){ FunctionBody }
```

FunctionExpression :

```
function Identifier opt ( FormalParameterList opt ){ FunctionBody }
```

We can see that when identifier is omitted, that “something” can only be an expression. But what if identifier is present? How can one tell whether it is a function declaration or a function expression — they look identical after all? It appears that ECMAScript differentiates between two based on a context. If a `function foo(){}` is part of, say, assignment expression, it is considered a function expression. If, on the other hand, `function foo(){}` is contained in a function body or in a (top level of) program itself — it is parsed as a function declaration.

```
function foo(){} // declaration, since it's part of a <em>Program</em>
var bar = function foo(){}; // expression, since it's part of an
<em>AssignmentExpression</em>

new function bar(){}; // expression, since it's part of a <em>NewExpression</em>

(function(){
  function bar(){} // declaration, since it's part of a <em>FunctionBody</em>
})();
```

A somewhat less obvious example of function expression is the one where function is wrapped with parenthesis — `(function foo(){})`. The reason it is an expression is again due to a context: "(" and ")" constitute a grouping operator and grouping operator can only contain an expression:

To demonstrate with examples:

```
function foo(){} // function declaration
(function foo(){}); // function expression: due to grouping operator

try {
  (var x = 5); // grouping operator can only contain expression, not a statement (which
`var` is)
} catch(err) {
  // SyntaxError
}
```

You might also recall that when evaluating JSON with `eval`, the string is usually wrapped with parenthesis — `eval('(' + json + ')')`. This is of course done for the same reason — grouping operator, which parenthesis are, forces JSON brackets to be parsed as expression rather than as a block:

```
try {
  { "x": 5 }; // "{" and "}" are parsed as a block
} catch(err) {
  // SyntaxError
}

({ "x": 5 }); // grouping operator forces "{" and "}" to be parsed as object literal
```

There's a subtle difference in behavior of declarations and expressions.

First of all, function declarations are parsed and evaluated before any other expressions are. Even if declaration is positioned last in a source, it will be evaluated **foremost any other expressions** contained in a scope. The following example demonstrates how `fn` function is already defined by the time `alert` is executed, even though it's being declared right after it:

```
alert(fn());  
  
function fn() {  
    return 'Hello world!';  
}
```

Another important trait of function declarations is that declaring them conditionally is non-standardized and varies across different environments. You should never rely on functions being declared conditionally and use function expressions instead.

```
// Never do this!  
// Some browsers will declare `foo` as the one returning 'first',  
// while others – returning 'second'
```

```
if (true) {  
    function foo() {  
        return 'first';  
    }  
}  
else {  
    function foo() {  
        return 'second';  
    }  
}  
foo();
```

// Instead, use function expressions:

```
var foo;  
if (true) {  
    foo = function() {  
        return 'first';  
    };  
}  
else {  
    foo = function() {  
        return 'second';  
    };  
}
```

```

}
foo();

```

If you're curious about actual production rules of function declarations, read on. Otherwise, feel free to skip the following excerpt.

FunctionDeclarations are only allowed to appear in *Program* or *FunctionBody*. Syntactically, they **can not appear in Block** (`{ ... }`) — such as that of `if`, `while` or `for` statements. This is because *Blocks* can only contain *Statements*, not *SourceElements*, which *FunctionDeclaration* is. If we look at production rules carefully, we can see that the only way *Expression* is allowed directly within *Block* is when it is part of *ExpressionStatement*. However, *ExpressionStatement* is explicitly defined **to not begin with "function" keyword**, and this is exactly why *FunctionDeclaration* cannot appear directly within a *Statement* or *Block* (note that *Block* is merely a list of *Statements*).

Because of these restrictions, whenever function appears directly in a block (such as in the previous example) it should actually be **considered a syntax error**, not function declaration or expression. The problem is that almost none of the implementations I've seen parse these functions strictly per rules (exceptions are [BESEN](#) and [DMDScript](#)). They interpret them in proprietary ways instead.

It's worth mentioning that as per specification, implementations are allowed to introduce **syntax extensions** (see section 16), yet still be fully conforming. This is exactly what happens in so many clients these days. Some of them interpret function declarations in blocks as any other function declarations — simply hoisting them to the top of the enclosing scope; Others — introduce different semantics and follow slightly more complex rules.

Function statements

One of such syntax extensions to ECMAScript is **Function Statements**, currently implemented in Gecko-based browsers (tested in Firefox 1-3.7a1pre on Mac OS X). Somehow, this extension doesn't seem to be widely known, either for good or bad ([MDN mentions them](#), but very briefly). Please remember, that we are discussing it here only for learning purposes and to satisfy our curiosity; unless you're writing scripts for specific Gecko-based environment, **I do not recommend relying on this extension**.

So, here are some of the traits of these non-standard constructs:

1. Function statements are allowed to be anywhere where plain *Statements* are allowed. This, of course, includes *Blocks*:

```

if (true) {
    function f(){ }
}
else {
    function f(){ }
}

```

2. Function statements are interpreted as any other statements, including conditional execution:

```

if (true) {
    function foo(){ return 1; }
}
else {
    function foo(){ return 2; }
}
foo(); // 1
// Note that other clients interpret `foo` as function declaration here,
// overwriting first `foo` with the second one, and producing "2", not "1" as a
result

```

3. Function statements are NOT declared during variable instantiation. They are declared at run time, just like function expressions. However, once declared, function statement's identifier **becomes available to the entire scope** of the function. This identifier availability is what makes function statements different from function expressions (you will see exact behavior of named function expressions in next chapter).

```

// at this point, `foo` is not yet declared
typeof foo; // "undefined"
if (true) {
    // once block is entered, `foo` becomes declared and available to the entire
    scope
    function foo(){ return 1; }
}
else {
    // this block is never entered, and `foo` is never redeclared
    function foo(){ return 2; }
}
typeof foo; // "function"

```

Generally, we can emulate function statements behavior from the previous example with this standards-compliant (and unfortunately, more verbose) code:

```

var foo;
if (true) {
    foo = function foo(){ return 1; };
}
else {
    foo = function foo() { return 2; };
}

```

4. String representation of functions statements is similar to that of function declarations or named function expressions (and includes identifier — "foo" in this example):

```

if (true) {
    function foo(){ return 1; }
}
String(foo); // function foo() { return 1; }

```

5. Finally, what appears to be a bug in earlier Gecko-based implementations (present in <= Firefox 3), is the way function statements overwrite function declarations. Earlier versions were somehow failing to overwrite function declarations with function statements:

```

// function declaration
function foo(){ return 1; }

if (true) {
    // overwritting with function statement
    function foo(){ return 2; }
}

foo(); // 1 in FF<= 3, 2 in FF3.5 and later

// however, this doesn't happen when overwriting function expression
var foo = function(){ return 1; };

if (true) {
    function foo(){ return 2; }
}

foo(); // 2 in all versions

```

Note that older Safari (at least 1.2.3, 2.0 - 2.0.4 and 3.0.4, and possibly earlier versions too) implement function statements **identically to SpiderMonkey**. All examples from this chapter, except the last "bug" one, produce same results in those versions of Safari as they do in, say, Firefox. Another browser that seems to follow same semantics is Blackberry one (at least 8230, 9000 and 9530 models). This diversity in behavior demonstrates once again what a bad idea it is to rely on these extensions.

Named function expressions

Function expressions can actually be seen quite often. A common pattern in web development is to "fork" function definitions based on some kind of a feature test, allowing for the best performance. Since such forking usually happens in the same scope, it is almost always necessary to use function expressions. After all, as we know by now, function declarations should not be executed conditionally:

```

// `contains` is part of "APE Javascript library" (http://dhtmlkitchen.com/ape/) by
Garrett Smith

var contains = (function() {
    var docEl = document.documentElement;

    if (typeof docEl.compareDocumentPosition != 'undefined') {

```

```

        return function(el, b) {
            return (el.compareDocumentPosition(b) & 16) !== 0;
        };
    }

    else if (typeof docEl.contains != 'undefined') {
        return function(el, b) {
            return el !== b && el.contains(b);
       };
    }

    return function(el, b) {
        if (el === b) return false;
        while (el != b && (b = b.parentNode) != null);
        return el === b;
   };
})();

```

Quite obviously, when a function expression has a name (technically — *Identifier*), it is called a **named function expression**. What you've seen in the very first example — `var bar = function foo(){};` — was exactly that — a named function expression with `foo` being a function name. An important detail to remember is that this name is **only available in the scope of a newly-defined function**; specs mandate that an identifier should not be available to an enclosing scope:

```

var f = function foo(){
    return typeof foo; // "foo" is available in this inner scope
};

// `foo` is never visible "outside"
typeof foo; // "undefined"
f(); // "function"

```

So what's so special about these named function expressions? Why would we want to give them names at all?

It appears that named functions make for a much more pleasant debugging experience. When debugging an application, having a call stack with descriptive items makes a huge difference.

Function names in debuggers

When a function has a corresponding identifier, debuggers show that identifier as a function name, when inspecting call stack. Some debuggers (e.g. Firebug) helpfully show names of even anonymous functions — making them identical to names of variables that functions are assigned to. Unfortunately, these debuggers usually rely on simple parsing rules; Such extraction is usually quite fragile and often produces false results.

Let's look at a simple example:

```

function foo(){
    return bar();
}
function bar(){
    return baz();
}
function baz(){
    debugger;
}
foo();

// Here, we used function declarations when defining all of 3 functions
// When debugger stops at the `debugger` statement,
// the call stack (in Firebug) looks quite descriptive:
baz
bar
foo
expr_test.html()

```

We can see that `foo` called `bar` which in its turn called `baz` (and that `foo` itself was called from the global scope of `expr_test.html` document). What's really nice, is that Firebug manages to parse the "name" of a function even when an anonymous expression is used:

```

function foo(){
    return bar();
}
var bar = function(){
    return baz();
}
function baz(){
    debugger;
}
foo();

// Call stack
baz
bar()
foo
expr_test.html()

```

What's not very nice, though, is that if a function expression gets any more complex (which, in real life, it almost always is) all of the debugger's efforts turn out to be pretty useless; we end up with a shiny question mark in place

of a function name:

```

function foo(){
    return bar();
}

var bar = (function(){
    if (window.addEventListener) {
        return function(){
            return baz();
        };
    }
    else if (window.attachEvent) {
        return function() {
            return baz();
        };
    }
})();
function baz(){
    debugger;
}
foo();

// Call stack
baz
(?]()
foo
expr_test.html()

```

Another confusion appears when a function is being assigned to more than one variable:

```

function foo(){
    return baz();
}

var bar = function(){
    debugger;
};

var baz = bar;
bar = function() {
    alert('spoofed');
};
foo();

// Call stack:

```

```
bar()
foo
expr_test.html()
```

You can see call stack showing that `foo` invoked `bar`. Clearly, that's not what has happened. The confusion is due to the fact that `baz` was “exchanged” references with another function — the one alerting “spoofed”. As you can see, such parsing — while great in simple cases — is often useless in any non-trivial script.

What it all boils down to is the fact that named **function expressions is the only way to get a truly robust stack inspection**. Let's rewrite our previous example with named functions in mind. Notice how both of the functions returning from self-executing wrapper, are named as `bar`:

```
function foo(){
    return bar();
}

var bar = (function(){
    if (window.addEventListener) {
        return function bar(){
            return baz();
        };
    }
    else if (window.attachEvent) {
        return function bar() {
            return baz();
        };
    }
})();
function baz(){
    debugger;
}
foo();

// And, once again, we have a descriptive call stack!
baz
bar
foo
expr_test.html()
```

Before we start dancing happily celebrating this holy grail finding, I'd like to bring a beloved JScript into the picture.

JScript bugs

Unfortunately, JScript (i.e. Internet Explorer's ECMAScript implementation) seriously messed up named function expressions. JScript is responsible for named function expressions **being recommended against** by many people

these days. It's also quite sad that even **last version of JScript — 5.8 — used in Internet Explorer 8, still exhibits every single quirk described below**

Let's look at what exactly is wrong with its broken implementation. Understanding all of its issues will allow us to work around them safely. Note that I broke these discrepancies into few examples — for clarity — even though all of them are most likely a consequence of one major bug.

Example #1: Function expression identifier leaks into an enclosing scope

```
var f = function g(){};
typeof g; // "function"
```

Remember how I mentioned that an identifier of named function expression is **not available in an enclosing scope**? Well, JScript doesn't agree with specs on this one — `g` in the above example resolves to a function object. This is a most widely observed discrepancy. It's dangerous in that it inadvertently pollutes an enclosing scope — a scope that might as well be a global one — with an extra identifier. Such pollution can, of course, be a source of hard-to-track bugs.

Example #2: Named function expression is treated as BOTH — function declaration AND function expression

```
typeof g; // "function"
var f = function g(){};
```

As I explained before, function declarations are parsed foremost any other expressions in a particular execution context. The above example demonstrates how **JScript actually treats named function expressions as function declarations**. You can see that it parses `g` before an “actual declaration” takes place.

This brings us to a next example:

Example #3: Named function expression creates TWO DISTINCT function objects!

```
var f = function g(){};
f === g; // false

f.expando = 'foo';
g.expando; // undefined
```

This is where things are getting interesting. Or rather — completely nuts. Here we are seeing the dangers of having to deal with two distinct objects — augmenting one of them obviously does not modify the other one; This could be quite troublesome if you decided to employ, say, caching mechanism and store something in a property of `f`, then tried accessing it as a property of `g`, thinking that it is the same object you're working with.

Let's look at something a bit more complex.

Example #4: Function declarations are parsed sequentially and are not affected by conditional blocks

```
var f = function g() {
    return 1;
};

if (false) {
    f = function g(){
        return 2;
    };
}

g(); // 2
```

An example like this could cause even harder to track bugs. What happens here is actually quite simple. First, `g` is being parsed as a function declaration, and since declarations in JScript are independent of conditional blocks, `g` is being declared as a function from the “dead” `if` branch — `function g(){ return 2 }`. Then all of the “regular” expressions are being evaluated and `f` is being assigned another, newly created function object to. “dead” `if` branch is never entered when evaluating expressions, so `f` keeps referencing first function — `function g(){ return 1 }`. It should be clear by now, that if you’re not careful enough, and call `g` from within `f`, you’ll end up calling a completely unrelated `g` function object.

You might be wondering how all this mess with different function objects compares to `arguments.callee`. Does `callee` reference `f` or `g`? Let’s take a look:

```
var f = function g(){
    return [
        arguments.callee == f,
        arguments.callee == g
    ];
};

f(); // [true, false]
g(); // [false, true]
```

As you can see, `arguments.callee` references whatever function is being invoked. This is actually good news, as you will see later on.

Another interesting example of “unexpected behavior” can be observed when using named **function expression in undeclared assignment**, but only when function is “named” the same way as identifier it’s being assigned to:

```
(function(){
    f = function f(){}
})()
```

As you might know, undeclared assignment (which is **not recommended** and is only used here for demonstration purposes) should result in creation of global `f` property. This is exactly what happens in conforming implementations. However, JScript bug makes things a bit more confusing. Since named function expression is parsed as function declaration (see [example #2](#)), what happens here is that `f` becomes declared as a local variable during the phase of variable declarations. Later on, when function execution begins, assignment is no longer undeclared, so `function f(){};` on the right hand side is simply assigned to this newly created **local f** variable. Global `f` is never created.

This demonstrates how failing to understand JScript peculiarities can lead to drastically different behavior in code.

Looking at JScript deficiencies, it becomes pretty clear what exactly we need to avoid. First, we need **to be aware of a leaking identifier** (so that it doesn't pollute enclosing scope). Second, we should **never reference identifier used as a function name**; A troublesome identifier is `g` from the previous examples. Notice how many ambiguities could have been avoided if we were to forget about `g`'s existence. Always referencing function via `f` or `argumentscallee` is the key here. If you use named expression, think of that name as something that's only being used for debugging purposes. And finally, a bonus point is to **always clean up an extraneous function** created erroneously during NFE declaration.

I think last point needs a bit of an explanation:

JScript memory management

Being familiar with JScript discrepancies, we can now see a potential problem with memory consumption when using these buggy constructs. Let's look at a simple example:

```
var f = (function(){
  if (true) {
    return function g(){};
  }
  return function g(){};
})();
```

We know that a function returned from within this anonymous invocation — the one that has `g` identifier — is being assigned to outer `f`. We also know that named function expressions produce superfluous function object, and that this object is not the same as returned function. The memory issue here is caused by this extraneous `g` function being literally “trapped” in a closure of returning function. This happens because inner function is declared in the same scope as that pesky `g` one. Unless we **explicitly break reference to g function** it will keep consuming memory.

```
var f = (function(){
  var f, g;
  if (true) {
    f = function g(){};
  }
  else {
```

```

f = function g(){};

}

// null `g` , so that it doesn't reference extraneous function any longer
g = null;
return f;
})();

```

Note that we explicitly declare `g` as well, so that `g = null` assignment wouldn't create a global `g` variable in conforming clients (i.e. non-JScript ones). By nulling reference to `g`, we allow garbage collector to wipe off this implicitly created function object that `g` refers to.

When taking care of JScript NFE memory leak, I decided to run a simple series of tests to confirm that nulling `g` actually does free memory.

Tests

The test was simple. It would simply create 10000 functions via named function expressions and store them in an array. I would then wait for about a minute and check how high the memory consumption is. After that I would null-out the reference and repeat the procedure again. Here's a test case I used:

```

function createFn(){
    return (function(){
        var f;
        if (true) {
            f = function F(){
                return 'standard';
            };
        }
        else if (false) {
            f = function F(){
                return 'alternative';
            };
        }
        else {
            f = function F(){
                return 'fallback';
            };
        }
        // var F = null;
        return f;
   })();
}
var arr = [ ];

```

```
for (var i=0; i<10000; i++) {
    arr[i] = createFn();
}
```

Results as seen in Process Explorer on Windows XP SP2 were:

IE6:

```
without `null`: 7.6K -> 20.3K
with `null`: 7.6K -> 18K
```

IE7:

```
without `null`: 14K -> 29.7K
with `null`: 14K -> 27K
```

The results somewhat confirmed my assumptions — explicitly nulling superfluous reference did free memory, but the difference in consumption was relatively insignificant. For 10000 function objects, there would be a ~3MB difference. This is definitely something that should be kept in mind when designing large-scale applications, applications that will run for either long time or on devices with limited memory (such as mobile devices). For any small script, the difference probably doesn't matter.

You might think that it's all finally over, but we are not just quite there yet :) There's a tiny little detail that I'd like to mention and that detail is Safari 2.x

Safari bug

Even less widely known bug with NFE is present in older versions of Safari; namely, Safari 2.x series. I've seen some [claims on the web](#) that Safari 2.x does not support NFE at all. This is not true. Safari does support it, but has bugs in its implementation which you will see shortly.

When encountering function expression in a certain context, Safari 2.x fails to parse the program entirely. It doesn't throw any errors (such as `SyntaxError` ones). It simply bails out:

```
(function f(){}())(); // <== NFE
alert(1); // this line is never reached, since previous expression fails the entire
program
```

After fiddling with various test cases, I came to conclusion that Safari 2.x **fails to parse named function expressions, if those are not part of assignment expressions**. Some examples of assignment expressions are:

```
// part of variable declaration
var f = 1;
```

```
// part of simple assignment
f = 2, g = 3;

// part of return statement
(function(){
    return (f = 2);
})();
```

This means that putting named function expression into an assignment makes Safari “happy”:

```
(function f(){}); // fails

var f = function f(){}; // works

(function(){
    return function f(){}; // fails
})();

(function(){
    return (f = function f(){}); // works
})();

setTimeout(function f(){ }, 100); // fails

Person.prototype = {
    say: function say() { ... } // fails
}

Person.prototype.say = function say(){ ... }; // works
```

It also means that we can't use such common pattern as returning named function expression without an assignment:

```
// Instead of this non-Safari-2x-compatible syntax:
(function(){
    if (featureTest) {
        return function f(){};
    }
    return function f(){};
})();

// we should use this slightly more verbose alternative:
```

```
(function(){
    var f;
    if (featureTest) {
        f = function f() {};
    }
    else {
        f = function f() {};
    }
    return f;
})();

// or another variation of it:
(function(){
    var f;
    if (featureTest) {
        return (f = function f() {});
    }
    return (f = function f() {});
})();

/*
Unfortunately, by doing so, we introduce an extra reference to a function
which gets trapped in a closure of returning function. To prevent extra memory usage,
we can assign all named function expressions to one single variable.
*/
var __temp;

(function(){
    if (featureTest) {
        return (__temp = function f() {});
    }
    return (__temp = function f() {});
})();

...
(function(){
    if (featureTest2) {
        return (__temp = function g() {});
    }
    return (__temp = function g() {});
})();
```

```
/*
 Note that subsequent assignments destroy previous references,
 preventing any excessive memory usage.
*/
```

If Safari 2.x compatibility is important, we need to make sure “**incompatible constructs do not even appear in the source**. This is of course quite irritating, but is definitely possible to achieve, especially when knowing the root of the problem.

It's also worth mentioning that declaring a function as NFE in Safari 2.x exhibits another minor glitch, where function representation does not contain function identifier:

```
var f = function g(){};

// Notice how function representation is lacking `g` identifier
String(f); // function () { }
```

This is not really a big deal. As I have already mentioned before, function decompilation is something that [should not be relied upon](#) anyway.

SpiderMonkey peculiarity

We know that identifier of named function expression is only available to the local scope of a function. But how does this "magic" scoping actually happen? It appears to be very simple. When named function expression is evaluated, a **special object is created**. The sole purpose of that object is to hold a property with the name corresponding to function identifier, and value corresponding to function itself. That object is then injected into the front of the current scope chain, and this "augmented" scope chain is then used to initialize a function.

The interesting part here, however, is the way ECMA-262 defines this "special" object — the one that holds function identifier. Spec says that an object is created "**as if by expression new Object()**" which, when interpreted literally, makes this object an instance of built-in `Object` constructor. However, only one implementation — SpiderMonkey — followed this specification requirement literally. In SpiderMonkey, it is possible to interfere with function local variables by augmenting `Object.prototype`:

```
Object.prototype.x = 'outer';

(function(){

    var x = 'inner';

    /*
     `foo` function here has a special object in its scope chain – to hold an identifier.
     That object is practically a –
     `{ foo: <function object> }`. When `x` is being resolved through the scope chain, it
```

```

is first searched for in
`foo`'s local context. When not found, it is searched in the next object from the
scope chain. That object turns out
to be the one that holds identifier - { foo: <function object> } and since it
inherits from `Object.prototype`,
`x` is found right here, and is the one that's `Object.prototype.x` (with value of
'outer'). Outer function's scope
(with x === 'inner') is never even reached.

*/
(function foo(){

    alert(x); // alerts `outer`

})();
})();

```

Note that later versions of SpiderMonkey actually **changed this behavior**, as it was probably considered a security hole. A "special" object no longer inherits from `Object.prototype`. You can, however, still see it in Firefox <=3.

Another environment implementing internal object as an instance of global `Object` is **Blackberry browser**. Only this time, it's *Activation Object* that inherits from `Object.prototype`. Note that specification actually doesn't codify *Activation Object* to be created "as if by expression new Object()" (as is the case with NFE's identifier holder object). It states that *Activation Object* is merely a specification mechanism.

So, let's see what happens in Blackberry browser:

```

Object.prototype.x = 'outer';

(function(){

    var x = 'inner';

    (function(){

        /*
        When `x` is being resolved against scope chain, this local function's Activation
        Object is searched first.

        There's no `x` in it, of course. However, since Activation Object inherits from
        `Object.prototype`, it is

        `Object.prototype` that's being searched for `x` next. `Object.prototype.x` does in
        fact exist and so `x`

        resolves to its value - 'outer'. As in the previous example, outer function's scope
    })
})()

```

```
(Activation Object)
  with its own x === 'inner' is never even reached.
  */

  alert(x); // alerts 'outer'

})();
})();
```

This might look bizarre, but what's really disturbing is that there's even more chance of conflict with already existing `Object.prototype` members:

```
(function(){

  var constructor = function(){ return 1; };

  (function(){

    constructor(); // evaluates to an object `{}`, not `1`

    constructor === Object.prototype.constructor; // true
    toString === Object.prototype.toString; // true

    // etc.

  })();
})();
```

Solution to this Blackberry discrepancy is obvious: avoid naming variables as `Object.prototype` properties — `toString`, `valueOf`, `hasOwnProperty`, and so on.

JScript solution

```
var fn = (function(){

  // declare a variable to assign function object to
  var f;

  // conditionally create a named function
  // and assign its reference to `f`
  if (true) {
    f = function F(){ };
  }
});
```

```

else if (false) {
    f = function F(){ };
}
else {
    f = function F(){ };
}

// Assign `null` to a variable corresponding to a function name
// This marks the function object (referred to by that identifier)
// available for garbage collection
var F = null;

// return a conditionally defined function
return f;
})();

```

Finally, here's how we would apply this "technique" in real life, when writing something like a cross-browser addEvent function:

```

// 1) enclose declaration with a separate scope
var addEvent = (function(){

    var docEl = document.documentElement;

    // 2) declare a variable to assign function to
    var fn;

    if (docEl.addEventListener) {

        // 3) make sure to give function a descriptive identifier
        fn = function addEvent(element, eventName, callback) {
            element.addEventListener(eventName, callback, false);
        };
    }
    else if (docEl.attachEvent) {
        fn = function addEvent(element, eventName, callback) {
            element.attachEvent('on' + eventName, callback);
        };
    }
    else {
        fn = function addEvent(element, eventName, callback) {
            element['on' + eventName] = callback;
        };
    }
});

```

```

}

// 4) clean up `addEvent` function created by JScript
//     make sure to either prepend assignment with `var`,
//     or declare `addEvent` at the top of the function
var addEvent = null;

// 5) finally return function referenced by `fn`
return fn;
})();

```

Alternative solution

It's worth mentioning that there actually exist alternative ways of having descriptive names in call stacks. Ways that don't require one to use named function expressions. First of all, it is often possible to define function via declaration, rather than via expression. This option is only viable when you don't need to create more than one function:

```

var hasClassName = (function(){

    // define some private variables
    var cache = { };

    // use function declaration
    function hasClassName(element, className) {
        var _className = '(?:^|\\s+)'+ className + '(?:\\s+|$)';
        var re = cache[_className] || (cache[_className] = new RegExp(_className));
        return re.test(element.className);
    }

    // return function
    return hasClassName;
})();

```

This obviously wouldn't work when forking function definitions. Nevertheless, there's an interesting pattern that I first seen used by [Tobie Langel](#). The way it works is by **defining all functions upfront using function declarations, but giving them slightly different identifiers**:

```

var addEvent = (function(){

    var docEl = document.documentElement;

    function addEventListener(){

```

```

    /* ... */
}

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

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

if (typeof docEl.addEventListener != 'undefined') {
    return addEventListener;
}
else if (typeof docEl.attachEvent != 'undefined') {
    return attachEvent;
}
return addEventAsProperty;
})();

```

While it's an elegant approach, it has its own drawbacks. First, by using different identifiers, you lose naming consistency. Whether it's good or bad thing is not very clear. Some might prefer to have identical names, while others wouldn't mind varying ones; after all, different names can often "speak" about implementation used. For example, seeing "attachEvent" in debugger, would let you know that it is an `attachEvent`-based implementation of `addEvent`. On the other hand, implementation-related name might not be meaningful at all. If you're providing an API and name "inner" functions in such way, the user of API could easily get lost in all of these implementation details.

A solution to this problem might be to employ different naming convention. Just be careful not to introduce extra verbosity. Some alternatives that come to mind are:

```

`addEvent`, `altAddEvent` and `fallbackAddEvent`
// or
`addEvent`, `addEvent2`, `addEvent3`
// or
`addEvent_addEventListener`, `addEvent_attachEvent`, `addEvent_asProperty`

```

Another minor issue with this pattern is increased memory consumption. By defining all of the function variations upfront, you implicitly create N-1 unused functions. As you can see, if `attachEvent` is found in `document.documentElement`, then neither `addEventListener` nor `addEventAsProperty` are ever really used. Yet, they already consume memory; memory which is never deallocated for the same reason as with JScript's buggy named expressions — both functions are "trapped" in a closure of returning one.

This increased consumption is of course hardly an issue. If a library such as Prototype.js was to use this pattern, there would be not more than 100-200 extra function objects created. As long as functions are not created in such way repeatedly (at runtime) but only once (at load time), you probably shouldn't worry about it.

WebKit's displayName

A somewhat different approach was taken by WebKit team. Frustrated with poor representation of functions — both, anonymous and named — WebKit introduced "special" `displayName` property (essentially a string) that when assigned to a function is displayed in debugger/profiler in place of that function's "name". [Francisco Tolmasky explains in details](#) the rationale and implementation of this solution.

Future considerations

Upcoming version of ECMAScript — [ECMA-262, 5th edition](#) — introduces so-called **strict mode**. The purpose of strict mode is to disallow certain parts of the language which are considered to be fragile, unreliable or dangerous. One of such parts is `argumentscallee`, "banned" presumably due to security concerns. When in strict mode, access to `argumentscallee` results in `TypeError` (see section 10.6). The reason I'm bringing up strict mode is because inability to use `argumentscallee` for recursion in 5th edition will most likely result in increased use of named function expressions. Understanding their semantics and bugs will become even more important.

```
// Before, you could use argumentscallee
(function(x) {
  if (x <= 1) return 1;
  return x * argumentscallee(x - 1);
})(10);

// In strict mode, an alternative solution is to use named function expression
(function factorial(x) {
  if (x <= 1) return 1;
  return x * factorial(x - 1);
})(10);

// or just fall back to slightly less flexible function declaration
function factorial(x) {
  if (x <= 1) return 1;
  return x * factorial(x - 1);
}
factorial(10);
```

Credits

Richard Cornford, for being one of the first people [to explain JScript bug with named function expressions](#).

Richard explains most of the bugs mentioned in this article. I highly recommend reading his explanation. I would also like to thank **Yann-Erwan Perio** and **Douglas Crockford** for [mentioning and discussing NFE issues in comp.lang.javascript](#) as far back as in 2003.

John-David Dalton, for giving useful suggestions about "final solution".

Tobie Langel, for ideas presented in "alternative solution".

Garrett Smith and **Dmitry A. Soshnikov** for various additions and corrections.

For an extensive explanation of functions in ECMAScript in Russian, see [this article by Dmitry A. Soshnikov](#).

5 : "Real" Mixins with JavaScript Classes

Mixins and Javascript: The Good, the Bad, and the Ugly.

Mixins and JavaScript are a like the classic Clint Eastwood movie.

The good is that composing objects out of small pieces of implementation is even possible due to JavaScript's flexible nature, and that mixins are fairly popular in certain circles.

The bad is a long list: there's no common idea of what the concept of a mixin even is in JavaScript; there's no common pattern for them; they require helper libraries to use; more advanced composition (like coordination between mixins and prototypes) is complicated and does not fall out naturally from the patterns; they're difficult to statically analyze and introspect; finally, most mixin libraries mutate objects or their prototypes, causing problems for VMs and some programmers to avoid them.

The ugly is that the result of all this is a balkanized ecosystem of mixin libraries and mixins, with often incompatible construction and semantics across libraries. As far as I can tell, no particular library is popular enough to even be called common. You're more likely to see a project implement its own mixin function than see a mixin library used.

JavaScript mixins so far have simply fallen well short of their potential, of how mixins are described in academic literature and even implemented in a few good languages.

For someone who loves mixins, and thinks they should be used as much as possible, this is terrible. Mixins solve a number of issues that single-inheritance languages have, and to me, most of the complaints that the prototypal crowd has against classes in JavaScript. I would argue all inheritance should be mixin inheritance - subclassing is just a degenerate form of mixin application.

Luckily, there's light at the end of the tunnel with JavaScript classes. Their arrival finally gives JavaScript very easy to use syntax for inheritance. JavaScript classes are more powerful than most people realize, and are a great foundation for building *real* mixins.

In this post I'll explore what mixins should do, what's wrong with current JavaScript mixins, and how simple it is to build a very capable mixin system in JavaScript that plays extremely well with classes.

What, Exactly, are Mixins?

To understand what a mixin implementation should do, let's first look at what mixins are:

A mixin is an abstract subclass; i.e. a subclass definition that may be applied to different superclasses to create a related family of modified classes.

Gilad Bracha and William Cook, [Mixin-based Inheritance](#)

This is the best definition of mixins I've been able to find. It clearly shows the difference between a mixin and a normal class, and strongly hints at how mixins can be implemented in JavaScript.

To dig deeper into the implications of this definition, let's add two terms to our mixin lexicon:

- *mixin definition*: The definition of a class that may be applied to different superclasses.

- *mixin application*: The application of a mixin definition to a specific superclass, producing a new subclass.

The mixin definition is really a *subclass factory*, parameterized by the superclass, which produces mixin applications. A mixin application sits in the inheritance hierarchy between the subclass and superclass.

The real, and only, difference between a mixin and normal subclass is that a normal subclass has a fixed superclass, while a mixin definition doesn't yet have a superclass. Only the *mixin applications* have their own superclasses. You can even look at normal subclass inheritance as a degenerate form of mixin inheritance where the superclass is known at class definition time, and there's only one application of it.

Examples

Here's an example of [mixins in Dart](#), which has a nice syntax for mixins while being similar to JavaScript:

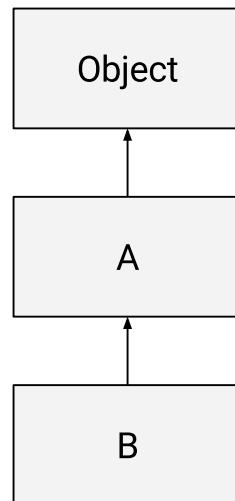
```
class B extends A with M {}
```

Here *A* is a base class, *B* the subclass, and *M* a mixin. The mixin application is the specific combination of *M* mixed into *A*, often called *A-with-M*. The superclass of *A-with-M* is *A*, and the actual superclass of *B* is not *A*, as you might expect, but *A-with-M*.

Some class declarations and diagrams might be helpful to show what's going on.

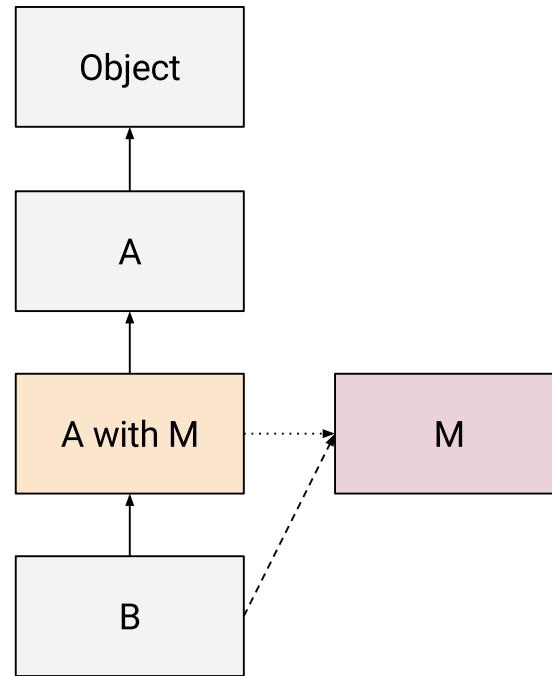
Let's start with a simple class hierarchy, with class *B* inheriting from class *A*:

```
class B extends A {}
```



Now let's add the mixin:

```
class B extends A with M {}
```



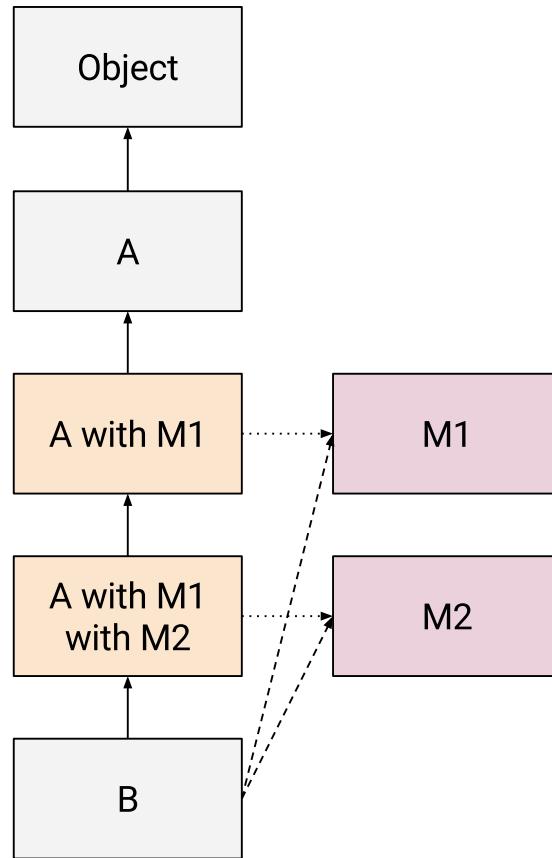
As you can see, the mixin application *A-with-M* is inserted into the hierarchy between the subclass and superclass.

Note: I'm using long-dashed line to represent the mixin declaration (*B* includes *M*), and the short-dashed line to represent the mixin application's definition.

Multiple Mixins

In Dart, multiple mixins are applied in left-to-right order, resulting in multiple mixin applications being added to the inheritance hierarchy:

```
class B extends A with M1, M2 {}  
class C extends A with M1, M2 {}
```



Traditional JavaScript Mixins

The ability to freely modify objects in JavaScript means that it's very easy to copy functions around to achieve code reuse without relying on inheritance.

Mixin libraries like [Cocktail](#), [traits.js](#), and patterns described in many blog posts (like one of the latest to hit Hacker News: [Using ES7 Decorators as Mixins](#)), generally work by modifying objects in place, copying in properties from mixin objects and overwriting existing properties.

This is often implemented via a function similar to this:

```

function mixin(source, target) {
  for (var prop in source) {
    if (source.hasOwnProperty(prop)) {
      target[prop] = source[prop];
    }
  }
}
  
```

A version of this has even made it into JavaScript as [Object.assign](#).

`mixin()` is usually then called on a prototype:

```

mixin(MyMixin, MyClass.prototype);
  
```

and now `MyClass` has all the properties defined in `MyMixin`.

What's So Bad About That?

Simply copying properties into a target object has a few issues. Some of this can be worked around with smart enough mixin functions, but the common examples usually have these problems:

Prototypes are modified in place.

When using mixin libraries against prototype objects, the prototypes are directly mutated. This is a problem if the prototype is used anywhere else that the mixed-in properties are not wanted. Mixins that add state can create slower objects in VMs that try to understand the shape of objects at allocation time. It also goes against the idea that a mixin application should create a new class by composing existing ones.

`super` doesn't work.

With JavaScript finally supporting `super`, so should mixins: A mixin's methods should be able to delegate to an overridden method up the prototype chain. Since `super` is essentially lexically bound, this won't work with copying functions.

Incorrect precedence.

This isn't necessarily always the case, but as often shown in examples, by overwriting properties, mixin methods take precedence over those in the subclass. They should only take precedence over methods in the superclass, allowing the subclass to override methods on the mixin.

Composition is compromised

Mixins often need to delegate to other mixins or objects on the prototype chain, but there's no natural way to do this with traditional mixins. Since functions are copied onto objects, naive implementations overwrite existing methods. More sophisticated libraries will remember the existing methods, and call multiple methods of the same name, but the library has to invent its own composition rules: What order the methods are called, what arguments are passed, etc.

References to functions are duplicated across all applications of a mixin, where in many cases they could be bundled in a shared prototype. By overwriting properties, the structure of prototypes and some of the dynamic nature of JavaScript is reduced: you can't easily introspect the mixins or remove or re-order mixins, because the mixin has been expanded directly into the target object.

If we actually use the prototype chain, all of this goes away with very little work.

Better Mixins Through Class Expressions

Now let's get to the good stuff: *Awesome Mixins™ 2015 Edition*.

Let's quickly list the features we'd like to enable, so we can judge our implementation against them:

1. Mixins are added to the prototype chain.

2. Mixins are applied without modifying existing objects.
3. Mixins do no magic, and don't define new semantics on top of the core language.
4. `super.foo` property access works within mixins and subclasses.
5. `super()` calls work in constructors.
6. Mixins are able to extend other mixins.
7. `instanceof` works.
8. Mixin definitions do not require library support - they can be written in a universal style.

Subclass Factories with this One Weird Trick

Above I referred to mixins as "subclass factories, parameterized by the superclass", and in this formulation they are literally just that.

We rely on two features of JavaScript classes:

1. `class` can be used as an expression as well as a statement. As an expression it returns a new class each time it's evaluated. (sort of like a factory!)
2. The `extends` clause accepts arbitrary expressions that return classes or constructors.

The key is that classes in JavaScript are first-class: they are values that can be passed to and returned from functions.

All we need to define a mixin is a function that accepts a superclass and creates a new subclass from it, like this:

```
let MyMixin = (superclass) => class extends superclass {
  foo() {
    console.log('foo from MyMixin');
  }
};
```

Then we can use it in an `extends` clause like this:

```
class MyClass extends MyMixin(MyBaseClass) {
  /* ... */
}
```

And `MyClass` now has a `foo` method via mixin inheritance:

```
let c = new MyClass();
c.foo(); // prints "foo from MyMixin"
```

Incredibly simple, and incredibly powerful! By just combining function application and class expressions we get a complete mixin solution, that generalizes well too.

Applying multiple mixins works as expected:

```
class MyClass extends Mixin1(Mixin2(MyBaseClass)) {
  /* ... */
}
```

Mixins can easily inherit from other mixins by passing the superclass along:

```
let Mixin2 = (superclass) => class extends Mixin1(superclass) {
  /* Add or override methods here */
}
```

And you can use normal function composition to compose mixins:

```
let CompoundMixin = (superclass) => Mixin2(Mixin3(superclass));
```

Benefits of Mixins as Subclass Factories

This approach gives us a very good implementation of mixins

Subclasses can override mixin methods.

As I mentioned before, many examples of JavaScript mixins get this wrong, and mixins override the subclass. With our approach subclasses correctly override mixin methods which override superclass methods.

`super` works.

One of the biggest benefits is that `super` works inside methods of the subclass and the mixins. Since we don't ever overwrite methods on classes or mixins, they're available for `super` to address.

`super` calls can be a little unintuitive for those new to mixins because the superclass isn't known at mixin definition, and sometimes developers expect `super` to point to the declared superclass (the parameter to the mixin), not the mixin application. Thinking about the final prototype chain helps here.

Composition is preserved.

This is really just a consequence of the other benefits, but two mixins can define the same method, and as long as they call `super`, both of them will be invoked then applied.

Sometimes a mixin will not know if a superclass even has a particular property or method, so it's best to guard the super call:

```
let Mixin1 = (superclass) => class extends superclass {
  foo() {
    console.log('foo from Mixin1');
    if (super.foo) super.foo();
  }
}
```

```

};

let Mixin2 = (superclass) => class extends superclass {
  foo() {
    console.log('foo from Mixin2');
    if (super.foo) super.foo();
  }
};

class S {
  foo() {
    console.log('foo from S');
  }
}

class C extends Mixin1(Mixin2(S)) {
  foo() {
    console.log('foo from C');
    super.foo();
  }
}

new C().foo();

```

prints:

```

foo from C
foo from Mixin1
foo from Mixin2
foo from S

```

Improving the Syntax

I find applying mixins as functions to be both elegantly simple - describing exactly what's going on, but a little bit ugly at the same time. My biggest concern is that this construction isn't optimized for readers unfamiliar with this pattern.

I'd like a syntax that was easier on the eyes and at least that gave new readers something to search for to explain what's going on, like the Dart syntax. I'd also like to add some more features like memoizing the mixin applications and automatically implementing `instanceof` support.

For that we can write a simple helper which applies a list of mixins to a superclass, in a fluent-like API:

```
class MyClass extends mix(MyBaseClass).with(Mixin1, Mixin2) {
  /* ... */
}
```

Here's the code:

```
let mix = (superclass) => new MixinBuilder(superclass);

class MixinBuilder {
  constructor(superclass) {
    this.superclass = superclass;
  }

  with(...mixins) {
    return mixins.reduce((c, mixin) => mixin(c), this.superclass);
  }
}
```

That's it! It's still extremely simple for the features and nice syntax it enables.

Constructors and Initialization

Constructors are a potential source of confusion with mixins. They essentially behave like methods, except that overriden methods tend to have the same signature, while constructors in a inheritance hierarchy often have different signatures.

Since a mixin likely does not know what superclass it's being applied to, and therefore its super-constructor signature, calling `super()` can be tricky. The best way to deal with this is to always pass along all constructor arguments to `super()`, either by not defining a constructor at all, or by using the spread operator: `super(...arguments)`.

This means that passing arguments specifically to a mixin's constructor is difficult. One easy workaround is to just have an explicit initialization method on the mixin if it requires arguments.

Further Exploration

This is just the beginning of many topics related to mixins in JavaScript. I'll post more about things like:

- [Enhancing Mixins with Decorator Functions](#) new post that covers:
 - Caching mixin applications so that the same mixin applied to the same superclass reuses a prototype.
 - Getting `instanceof` to work.
- How mixin inheritance can address the fear that ES6 classes and classical inheritance are bad for JavaScript.
- Using subclass-factory-style mixins in ES5.
- De-duplicating mixins so mixin composition is more usable.

The mixwith.js project

I started a project on GitHub to build a library around this pattern: <https://github.com/justinfagnani/mixwith.js> It can be found on npm: <https://www.npmjs.com/package/mixwith>

6 : JavaScript Mixins

Let's start by defining what a mixin is. I would explain it as *a way to share set of data (functions, methods, variables...) with objects.*

How does that look like? Let's say we have two functions `Clark` and `Bruce` and that we want to implement two methods `sayName` and `secretIdentity` to their prototypes.

This is how our functions would look like:

```
function Clark() {
    this.name = 'Clark Kent';
    this.alterego = 'Superman';
}
Clark.prototype.fly = function () {
    this.fly = true;
};

function Bruce() {
    this.name = 'Bruce Wayne';
    this.alterego = 'Batman';
}
Bruce.prototype.talk = function () {
    return "I'm Batman";
};
```

Most common ways of adding the methods these functions are:

1. Extending objects

Extending object doesn't really fit our description of mixins, since we are **extending** instead **mixing**, though idea is the same. At least that's the way I see it.

```
var Superhero = {
    sayName: function () {
        return this.name;
    },
    secretIdentity: function () {
        return this.alterego;
    }
};

// Here we extend function prototypes with object using jQuery.extend
```

```
$.extend(Clark.prototype, Superhero);
$.extend(Bruce.prototype, Superhero);
```

Biggest downside to this approach is that you'll either need `extend` function available from some library like *Underscore* or *jQuery* or you would have to write it.

2. Functional mixin

They don't require any additional library and they are faster compared to extending. Also I find this implementation frequently overlooked.

```
function Superhero () {
    this.sayName = function () {
        return this.name;
    };
    this.secretIdentity = function () {
        return this.alterego;
    };
}

// Here we extend function prototypes with using mixin function
Superhero.call(Clark.prototype);
Superhero.call(Bruce.prototype);
```

Which approach do you prefer?

7 : Building a simple PubSub system in JavaScript

Paul Kinlan December 8 2016

In a recent project building a [web push](#) service I wanted to have my UI respond to application level events (semantically if you will) because there were a couple of components that require information from the system but are not dependent with each other and I wanted them to be able to manage themselves independently of the 'business logic'.

I looked around at lots of different tools to help me, but because I frequently have a heavy case of NIH syndrome and the fact that I think people can implement their own infrastructural elements pretty quickly, I decided to quickly knock-up a simple client-side PubSub service — it worked pretty well for my needs.

I debated whether I should use a custom DOM `Event` and use the existing infrastructure that the DOM already provides to developers — the ability to events and consuming events using `addEventListener` — but the only problem was that you have to hang the event handler off a DOM Element or the window because you can't have a model that inherits or mixes in `EventTarget`.

Thought: having `EventTarget` as an object would help obviate the need for creating custom PubSub systems.

With this constraint in mind, a will to code something up, and a propensity for not minding bugs that I create myself, I sketched out a rough plan:

```
/* When a user is added, do something useful (like update UI) */
EventManager.subscribe('useradded', function(user) {
  console.log(user)
});

/* The UI submits the data, lets publish the event. */
form.onsubmit(function(e) {
  e.preventDefault();

  // do something with user fields

  EventManager.publish('useradded', user);
})
```

All of this isn't new. Redux and many other systems already do this and in many cases they also help you manage state. In my head, I don't really have state that needs a model that is separate to the state already in the browser.

The implementation is pretty simple to implement and the abstraction is quite useful for me at least.

```
var EventManager = new (function() {
  var events = {};
```

```
this.publish = function(name, data) {
    var handlers = events[name];
    if (!!handlers === false) return;
    handlers.forEach(function(handler) {
        handler.call(this, data);
    });
};

this.subscribe = function(name, handler) {
    var handlers = events[name];
    if (!!handlers === false) {
        handlers = events[name] = [];
    }
    handlers.push(handler);
};

this.unsubscribe = function(name, handler) {
    var handlers = events[name];
    if (!!handlers === false) return;

    var handlerIdx = handlers.indexOf(handler);
    handlers.splice(handlerIdx);
};

});
```

Edit: Removed the use of promise.

And there we are. A simple pubsub system that is likely full of bugs, but I like it. :) I've put it on [github](#) if you are interested in it.

8 : Build A Simple Javascript App The MVC Way

One of the best parts of JavaScript, can also be the worst. This is the undoubtedly simplistic ability to add an opening and closing script tag to the head of the HTML document and throw some spaghetti code in there. And voila! It works! Or does it? What is spaghetti code you ask? Spaghetti code is an unflattering term for code that is messy, has a tangled control structure, and is all over the place. It is nearly impossible to maintain and debug, usually has very poor structure, and is prone to errors. [Web design and development](#) is already challenging enough, don't make your job any harder than it has to be!

So how do you quit writing this kind of code? In my humble opinion, you only have two options. Again, this is just my opinion. One, you use any of the umpteen number of JavaScript frameworks available at your disposal. Or two, you learn how to write JavaScript code with some sort of pattern or structure. MVC, MVP, and MVVM are a few of the common patterns to help guide developers toward creating abstracted and decoupled solutions. The main difference between these patterns boils down to how the Data Layer, Presentation Layer, and Application Logic are handled. In this blog post, you will be focusing on the MVC or Model-View-Controller Pattern.

Model (Data Layer)- This is where the data is stored for your app. The model is decoupled from the views and controllers and has deliberate ignorance from the wider context. Whenever a model changes, it will notify its observers that a change has occurred using an Event Dispatcher. You will read about the Event Dispatcher shortly. In your To Do List App you will be building, the Model will hold the list of tasks and be responsible for any actions taken upon each task object.

View (Presentation Layer)- This part of your App has access to the DOM and is responsible for setting up Event Handlers such as: click, onmouseover, onmouseout, etc. The view is also responsible for the presentation of the HTML. In your To Do List App, the view will be responsible for displaying the list of tasks to the user. However, whenever a user enters in a new task through the input field, the view will use an Event Dispatcher to notify the controller, then the controller will update the model. This allows for swift decoupling of the view from the model.

Controller (Application Logic)- The controller is the glue between the model and the view. The controller processes and responds to events set off by either the model or view. It updates the model whenever the user manipulates the view, and can also be used to update the view whenever the model changes. In this tutorial, you will be updating the view directly from your model by dispatching an event. But, either way is completely acceptable. In your To Do list App, when the user clicks the add task button, the click is forwarded to the controller, and the controller modifies the model by adding the task to it. Any other decision making or logic can also be performed here, such as: saving the data using HTML local storage, making an asynchronous save to the database/server, and more.

Event Dispatcher - The Event Dispatcher is an object that allows you to attach an unlimited number of functions/methods to it. When you finally call the notify method on that Event object, every method you attached to that Event will be ran. You will see this happening a lot in the source code below.

Now that you have a basic understanding of why you should use a JavaScript Design Pattern, and what the MVC architecture is all about, now you can start building the app. Here is a quick overview of how this tutorial will work. First, I will present the code to you. This will give you the opportunity to examine and walk through it. Second, I will go into detail about a few of the core concepts that the below code base is using, and try to shed some light upon

any potential hazy or gray areas that might come across as confusing. In conclusion, I will show you a couple of screenshots of the final component. To get the most out of this tutorial, I suggest you go through this a couple of times until you feel comfortable creating this app on your own. This tutorial assumes prior HTML and JavaScript experience. If you are new to JavaScript, I recommend taking a quick look at <http://www.w3schools.com/Js/>. You can also find the project on GitHub at <https://github.com/joshcrawmer4/Javascript-MVC-App>. Go ahead and get started!

Start Writing Some Code

Create an **index.html** file with the below code in it:

```
<!doctype html>
<html>

<head>
  <title>Javascript MVC</title>
  <script src="https://code.jquery.com/jquery-2.2.3.min.js" integrity="sha256-a23g1Nt4dtEY0j7bR+vTu7+T8VP13humZFBJNlYoEJo=" crossorigin="anonymous"></script>

  <style>

  </style>

</head>

<body>

  <div class="js-container">

    <input type="text" class="js-task-textbox">
    <input type="button" class="js-add-task-button" value="Add Task">

    <div class="js-tasks-container"></div>
    <!-- end tasks -->
    <input type="button" class="js-complete-task-button" value="Complete Tasks">
    <input type="button" class="js-delete-task-button" value="Delete Tasks">

  </div>
  <!-- end js-container -->

<script src="EventDispatcher.js"></script>
```

```

<script src="TaskModel.js"></script>
<script src="TaskView.js"></script>
<script src="TaskController.js"></script>
<script src="App.js"></script>

</body>

</html>

```

Next create an **EventDispatcher.js** file with the below code:

```

var Event = function (sender) {
    this._sender = sender;
    this._listeners = [];
}

Event.prototype = {

    attach: function (listener) {
        this._listeners.push(listener);
    },

    notify: function (args) {
        for (var i = 0; i < this._listeners.length; i += 1) {
            this._listeners[i](this._sender, args);
        }
    }

};


```

Next Create a **TaskModel.js** file with the below code:

```

var TaskModel = function () {
    this.tasks = [];
    this.selectedTasks = [];
    this.addTaskEvent = new Event(this);
    this.removeTaskEvent = new Event(this);
    this.setTasksAsCompletedEvent = new Event(this);
    this.deleteTasksEvent = new Event(this);

};

TaskModel.prototype = {

```

```
addTask: function (task) {
    this.tasks.push({
        taskName: task,
        taskStatus: 'uncompleted'
    });
    this.addTaskEvent.notify();
},

getTasks: function () {
    return this.tasks;
},

setSelectedTask: function (taskIndex) {
    this.selectedTasks.push(taskIndex);
},

unselectTask: function (taskIndex) {
    this.selectedTasks.splice(taskIndex, 1);
},

setTasksAsCompleted: function () {
    var selectedTasks = this.selectedTasks;
    for (var index in selectedTasks) {
        this.tasks[selectedTasks[index]].taskStatus = 'completed';
    }
}

this.setTasksAsCompletedEvent.notify();

this.selectedTasks = [];

},

deleteTasks: function () {
    var selectedTasks = this.selectedTasks.sort();

    for (var i = selectedTasks.length - 1; i >= 0; i--) {
        this.tasks.splice(this.selectedTasks[i], 1);
    }

    // clear the selected tasks
    this.selectedTasks = [];
}
```

```
        this.deleteTasksEvent.notify();

    }

};

};
```

Next Create a **TaskView.js** file with the below code:

```
var TaskView = function (model) {
    this.model = model;
    this.addTaskEvent = new Event(this);
    this.selectTaskEvent = new Event(this);
    this.unselectTaskEvent = new Event(this);
    this.completeTaskEvent = new Event(this);
    this.deleteTaskEvent = new Event(this);

    this.init();
};

TaskView.prototype = {

    init: function () {
        this.createChildren()
            .setupHandlers()
            .enable();
    },

    createChildren: function () {
        // cache the document object
        this.$container = $('.js-container');
        this.$addTaskButton = this.$container.find('.js-add-task-button');
        this.$taskTextBox = this.$container.find('.js-task-textbox');
        this.$tasksContainer = this.$container.find('.js-tasks-container');

        return this;
    },

    setupHandlers: function () {

        this.addTaskButtonHandler = this.addTaskButton.bind(this);
        this.selectOrUnselectTaskHandler = this.selectOrUnselectTask.bind(this);
        this.completeTaskButtonHandler = this.completeTaskButton.bind(this);
    }
};
```

```
        this.deleteTaskButtonHandler = this.deleteTaskButton.bind(this);

        /**
         * Handlers from Event Dispatcher
         */
        this.addTaskHandler = this.addTask.bind(this);
        this.clearTaskTextBoxHandler = this.clearTaskTextBox.bind(this);
        this.setTasksAsCompletedHandler = this.setTasksAsCompleted.bind(this);
        this.deleteTasksHandler = this.deleteTasks.bind(this);

        return this;
    },

enable: function () {

    this.$addTaskButton.click(this.addTaskButtonHandler);
    this.$container.on('click', '.js-task', this.selectOrUnselectTaskHandler);
    this.$container.on('click', '.js-complete-task-button',
this.completeTaskButtonHandler);
    this.$container.on('click', '.js-delete-task-button',
this.deleteTaskButtonHandler);

    /**
     * Event Dispatcher
     */
    this.model.addTaskEvent.attach(this.addTaskHandler);
    this.model.addTaskEvent.attach(this.clearTaskTextBoxHandler);
    this.model.setTasksAsCompletedEvent.attach(this.setTasksAsCompletedHandler);
    this.model.deleteTasksEvent.attach(this.deleteTasksHandler);

    return this;
},

addTaskButton: function () {
    this.addTaskEvent.notify({
        task: this.$taskTextBox.val()
    });
},

completeTaskButton: function () {
    this.completeTaskEvent.notify();
},

deleteTaskButton: function () {
```

```
        this.deleteTaskEvent.notify();
    },

    selectOrUnselectTask: function () {

        var taskIndex = $(event.target).attr("data-index");

        if ($(event.target).attr('data-task-selected') == 'false') {
            $(event.target).attr('data-task-selected', true);
            this.selectTaskEvent.notify({
                taskIndex: taskIndex
            });
        } else {
            $(event.target).attr('data-task-selected', false);
            this.unselectTaskEvent.notify({
                taskIndex: taskIndex
            });
        }
    },

    show: function () {
        this.buildList();
    },
}

buildList: function () {
    var tasks = this.model.getTasks();
    var html = "";
    var $tasksContainer = this.$tasksContainer;

    $tasksContainer.html('');

    var index = 0;
    for (var task in tasks) {

        if (tasks[task].taskStatus == 'completed') {
            html += "<div style='color:green;'>";
        } else {
            html += "<div>";
        }

        $tasksContainer.append(html + "<label><input type='checkbox' class='js-task' data-index='" + index + "' data-task-selected='false'">" + tasks[task].taskName + "</label></div>");
    }
}
```

```
        index++;
    }

},

/* ----- Handlers From Event Dispatcher ----- */

clearTaskTextBox: function () {
    this.$taskTextBox.val('');
},

addTask: function () {
    this.show();
},

setTasksAsCompleted: function () {
    this.show();
},

deleteTasks: function () {
    this.show();
}

/* ----- End Handlers From Event Dispatcher ----- */

};

};
```

Next Create a **TaskController.js** file with the below code:

```
var TaskController = function (model, view) {
    this.model = model;
    this.view = view;
```

```
        this.init();
    };

TaskController.prototype = {

    init: function () {
        this.createChildren()
            .setupHandlers()
            .enable();
    },

    createChildren: function () {
        // no need to create children inside the controller
        // this is a job for the view
        // you could all as well leave this function out
        return this;
    },

    setupHandlers: function () {

        this.addTaskHandler = this.addTask.bind(this);
        this.selectTaskHandler = this.selectTask.bind(this);
        this.unselectTaskHandler = this.unselectTask.bind(this);
        this.completeTaskHandler = this.completeTask.bind(this);
        this.deleteTaskHandler = this.deleteTask.bind(this);
        return this;
    },

    enable: function () {

        this.view.addTaskEvent.attach(this.addTaskHandler);
        this.view.completeTaskEvent.attach(this.completeTaskHandler);
        this.view.deleteTaskEvent.attach(this.deleteTaskHandler);
        this.view.selectTaskEvent.attach(this.selectTaskHandler);
        this.view.unselectTaskEvent.attach(this.unselectTaskHandler);

        return this;
    },

    addTask: function (sender, args) {
        this.model.addTask(args.task);
    },
}
```

```

selectTask: function (sender, args) {
    this.model.setSelectedTask(args.taskIndex);
},
unselectTask: function (sender, args) {
    this.model.unselectTask(args.taskIndex);
},
completeTask: function () {
    this.model.setTasksAsCompleted();
},
deleteTask: function () {
    this.model.deleteTasks();
}
};


```

Next Create a **App.js** file with the below code:

```

$(function () {
    var model = new TaskModel(),
        view = new TaskView(model),
        controller = new TaskController(model, view);
});


```

Overview

Okay, so what is going on in each file!

Index.html - Not much new here. Including the jQuery CDN, setting up some basic HTML, and loading the JavaScript files that we will be using in this project. Wow! That was easy!

EventDispatcher.js - This is a class with two methods, **attach()** and **notify()**. The **attach()** method accepts a function as a parameter. You can call **attach()** as many times as you want, and the function you pass can contain whatever code inside you want. Once you call the **notify** method on that Event object, each function you attached to that Event will be ran.

TaskModel.js - This class has some basic methods for adding and deleting actual task objects from the **tasks** array. Setting up three Event objects inside the constructor function, allows the model to call the **notify()** method on each event object after a task has been added, marked as complete, or deleted. This, in turn, passes on the responsibility to the view to re-render the HTML to show the updated list of tasks. The main thing to recognize is that the Model passes off responsibility to the View. **Model -> View**.

TaskView.js - This is the largest file in this project and could have been abstracted into multiple views. But for simplicity's sake, I put everything into one class. The constructor function sets up five Event objects. This allows the view to call the **notify()** method on each Event object, thus passing the responsibility onto the controller. Next, you see that the constructor calls the **init()** method. This init method uses method chaining to set up the backbone of this class.

createChildren() - Caches the `$('.js-container')` DOM element in a `this.$container` variable, then refers to that variable for each element thereafter it needs to `find()`. This is merely a performance thing, and allows jQuery to pull any elements from the variable instead of requerying/crawling the DOM. Notice the use of `return this`. This allows for the method chaining inside the previous `init()` call.

setupHandlers() - This part can be a little tricky to wrap your head around for the first time. This method is setting up the event handlers and changing the scope of the `this` keyword inside that handler. Basically, whenever you run into a JavaScript event handler and plan to use the ever so famous `this` keyword inside that callback function, then this is going to reference the actual object or element the event took place on. This is not desirable in many cases, as in the MVC case when you want `this` to reference the actual class itself. Here, you are calling the `bind(this)` method on a JavaScript callback function. This changes the `this` keyword scope to point to that of the class instead of the object or element that initialized that event. Mozilla Foundation has a good tutorial explaining how to use [scope bound functions](#).

enable() - This method sets up any DOM events and attaches any functions to the Event Dispatcher that were created by the Model. Look at this line of code:

```
this.model.addTaskEvent.attach(this.addTaskHandler);
```

What is actually happening here? When the model calls `addTaskEvent.notify()` your view will run the `this.addTaskHandler()` method. Sweet! You're actually seeing how the EventDispatcher works! This allows your classes to talk to each other while also staying decoupled. The `addTaskHandler()` method then calls the `show()` method which in turn calls the `buildList()` method and re-renders the HTML list.

So what should you take from all of this? Basically, once the model passes responsibility to the view, the view then re-renders the HTML to show the most up-to-date task objects. Also, whenever a user manipulates the view through a DOM event the view then passes off responsibility to the controller. The view does not work directly with the model.

TaskController.js - This class sits between the view and the model and acts as the glue that binds them together. It allows for easy decoupling of your model and view. Whenever the view uses the EventDispatcher, the controller will be there to listen and then update the model. Besides that, all the method declarations inside this file should look relatively similar to the View and Model.

App.js - This file is responsible for kicking off the app. An instance of the Model-View-Controller gets created here.

	Add Task
<input type="checkbox"/> Cook	
<input type="checkbox"/> Clean	
<input checked="" type="checkbox"/> Study	
Complete Tasks	Delete Tasks

Conclusion

As a developer you must always be striving to stay current. Whether this is getting involved with projects on GitHub, dabbling in a new language, or learning design patterns and principles. But one thing is for certain, you must always be moving forward. Now that you have a basic overview of how to write JavaScript the MVC way, you see how it is possible to quit writing spaghetti code and how you can start writing cleaner and more maintainable code. Feel free to reach out to me on GitHub or post any questions in the comments section below.

9 : PubSub Design Pattern

Welcome to this humble article on the PubSub design pattern in Javascript. If you have ever written a relatively big Javascript app you'll know that it can get out of hand pretty quickly, for that very reason there are lots of solutions out there that you can use to aid you in this issue, **design patterns** are one of the most generic solutions and very often serve as a base for another more abstract solution, like an MV* framework such as Angular and Backbone.

A design pattern is simply a general reusable solution to a common problem, as you might expect the implementation is up to you, nevertheless a lot of production-ready implementations can be found out there, as we'll discuss later on this article.

The good thing about design patterns is that they are robust and generally accepted solutions, so you are in good hands if you decide to use a certain pattern. It's important to remember though, that when you have a hammer everything looks like a nail, you must know when to use it and when not to; in order to do this you need to clearly understand the advantages and disadvantages of the pattern you plan to use.

The PubSub design pattern stands for Publish/Subscribe, and might also be called Observer pattern. This is one of the most popular design patterns in modern Javascript, and for a good reason! The general idea of this pattern is to promote loose coupling, instead of objects calling other object's methods directly, they subscribe to an event and they get notified whenever that event occurs.

Let's get our feet wet with some code, as this is all quite abstract! A nice thing about PubSub is that it's not a complex design pattern at all, in fact, the idea is pretty simple as we just want to be notified when something happens, so if we had a pubsub implementation, our code would look something like this:

```
function myCallback() {  
    console.log('Event 1 triggered!');  
}  
pubsub.subscribe('event1', myCallback);  
pubsub.publish('event1');
```

As you can see the `pubsub` object exposes two methods, `subscribe` and `publish`, when we want to know whenever an event occurs we **subscribe** to that event, later on we can **publish** that event whenever in our code.

The pattern also exposes an `unsubscribe` method which simply removes a listener for a given event name.

Advantages of the PubSub Pattern

One of the most relevant advantages of this pattern is that it allows us to easily decouple our code, which inherently makes it more maintainable and easier to test. Another advantage is that it helps us to think in term of relationships between the different parts of our app, identifying which layers need to listen and which need to publish events.

We are also able to easily create dynamic relationships between the different entities of our app, making it quite flexible, as Addy Osmani says in his [Javascript Design Patterns](#) book:

Whilst it may not always be the best solution to every problem, it remains one of the best tools for designing decoupled systems and should be considered an important tool in any JavaScript developer's utility belt.

Disadvantages of the PubSub Pattern

The main disadvantage of this pattern is that as we highly decouple the different parts of our app, it can get hard to guarantee that a part of our app is working correctly.

Also, as the dependencies are dynamic they can get hard to track.

Implementing the PubSub Pattern

Now that you have an idea of the pattern, let's implement it! This is actually quite easy, one simplistic implementation could be as follows:

```
var subs = {};
var pubsub = {
    subscribe: function (evt, callback) {
        if(!subs[evt]) subs[evt] = [];
        subs[evt].push(callback);
        return callback;
    },
    publish: function (evt) {
        var args = Array.prototype.slice.call(arguments, 1);
        subs[evt].forEach(function (callback) {
            callback.apply(null, args);
        });
    },
    unsubscribe: function (evt, callback) {
        var idx = subs[evt].indexOf(callback);
        subs.splice(idx, 1);
    }
};
```

We can use this implementation as follows:

```
pubsub.subscribe('greet', function (name) {
    console.log('Hello ' + name + '!');
});
```

```
pubsub.publish('greet', 'Federico');
```

That would produce a message in the developer console `Hello Federico!`, as you can see it's pretty easy to implement and use.

Other PubSub Implementations

There are a bunch of PubSub implementations, the one I like the most is [Ben Alman's Gist](#), which is really solid and succinct. Other alternatives include [AmplifyJS](#) and [PubSubJS](#).

```
/* jQuery Tiny Pub/Sub - v0.7 - 10/27/2011
* http://benalman.com/
* Copyright (c) 2011 "Cowboy" Ben Alman; Licensed MIT, GPL */

(function($) {

    var o = ${};

    $.subscribe = function() {
        o.on.apply(o, arguments);
    };

    $.unsubscribe = function() {
        o.off.apply(o, arguments);
    };

    $.publish = function() {
        o.trigger.apply(o, arguments);
    };

})(jQuery);
```

Using PubSub with AJAX

A very popular use for PubSub is whenever we are working with AJAX and async code, in the following example I illustrate a case where an user wants to search for posts, whenever the user clicks the `#btn-search` button our app should:

- Send an AJAX request to fetch posts which correspond to a keyword
- Update the search history with the new keyword

We could just do everything in one function, but let's keep everything organized and decouple each task we need to perform:

```
// USING BEN ALMAN'S PUBSUB IMPLEMENTATION
// Subscribers
$.subscribe('posts/search', function (e, keyword) {
    $.ajax('/posts', { keyword: keyword }, function (data) {
        $.publish('/posts/all', data);
    });
})
$.subscribe('posts/search', drawHistory);
$.subscribe('posts/all', drawResults);
// Publishers
$('#btn-search').click(function () {
    var keyword = $('#search-input').val();
    $.publish('posts/search', keyword);
});
```

As you can see we don't need to wait for the AJAX request to finish to draw the search history, we can do it as soon as the button is clicked, on the other hand we do have to wait in order to draw the results. This somehow complex behaviour is easily and clearly implemented using the PubSub pattern. This scenario is pretty common in MV* frameworks such as [Backbone](#) which are getting increasingly popular, PubSub allows Backbone to easily decouple its different layers/actors from each other.

A Real World Example

Not too long ago I faced this issue and I was glad how easily PubSub solved it. Consider you are working in a quite big WordPress plugin which is divided into modules, each module must be independant of each other, meaning it's possible to swap those modules anytime and their tests must not have any hard dependency, to achieve this this modules must able to "talk" to each other indirectly. How can we do this? Well we know that PubSub excels at decoupling so let's try it out!

Out of all our modules, we are interested in two: `favs` and `images`. They both have a JS file: `favs.js` and `images.js` respectively.

We are given the task to implement "adding an image to favorites"; `favs.js` is in charge of favs, so without using pubsub a very bare-bones implementation of `images.js` might look similar to this:

```
(function ($) {
    'use strict';

    function onAdded() { /* display OK message */ }
    function onFailed() { /* display FAIL message */ }
    // add to favs
```

```

function addToFavs(id) {
    window.favs.addToFavs(id, onAdded, onFailed);
}
// When a #fav element is clicked save to favs
$('#fav').click(addToFavs);
})(jQuery);

```

There are some issues with that code, one is that we have to make some kind of global variable to be able to use the `favs` API from `favs.js`, we also assume that the `favs.js` file will be loaded **before** `images.js`; this isn't a very big deal as we could fix it by using something like `requirejs` but still, what if we want to test that code?

Testing the `addToFavs` function would depend on the `window.favs` object.

Using **PubSub** we can rewrite our code as follows:

```

(function ($) {
    'use strict';

    function onAdded() { /* display OK message */ }
    function onFailed() { /* display FAIL message */ }

    // When a #fav element is clicked save to favs
    $('#fav').click(function () {
        // favs.js is subscribed to this event
        pubsub.trigger('favs/add');
    });
    pubsub.subscribe('favs/added', onAdded);
    pubsub.subscribe('favs/failed', onFailed);
})(jQuery);

```

Where did `window.favs` go? We don't need it anymore! We just **publish** a `favs/add` event (note that it is a good practice to namespace your event names), the `favs.js` file will be listening and will react accordingly, triggering the `favs/added` and `favs/failed` events as needed. We have **decoupled** our little module, as a result we no longer need to use a global variable, have more flexibility on the loading order of the scripts, and more importantly we can now easily test our code by simply publishing events as needed; as a bonus this allows our tests to be more behaviour-driven rather than feature-driven, which is also a good practice :)

Promises: Another solution for async code

Promises are another tool we can use to deal with async code, they are gaining popularity and for a good reason! They can be quite helpful so it's important to know what they are and when they should be used, they also play nice with the PubSub design pattern.

The [Mozilla Developer Network](#) defines promises as follows:

The Promise interface represents a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers to an asynchronous action's eventual success or failure. This lets asynchronous methods return values like synchronous methods: instead of the final value, the asynchronous method returns a promise of having a value at some point in the future. A pending promise can become either fulfilled with a value, or rejected with a reason. When either of these happens, the associated handlers queued up by a promise's `then` method are called.

That definition might be a bit hard to swallow, but it's in fact pretty simple. Promises represent an entity which still doesn't have a value, but it's guaranteed that it will eventually either have a value (**succeed**) or **fail**.

```
// A simple promise
var promise = new MyPromise();
promise.then(onPromiseFinished, onPromiseFailed, onProgress);
```

A Promise **must** have a `then` function which accepts 3 callbacks, the first two are for success and failure respectively, the third one is to notify the consumer on progress, useful for very long operations but seldom used.

Something nice about promises is that they are **chainable**.

```
var promise = new MyPromise();
promise.then(function () {
    console.log('First callback');
}).then(function () {
    console.log('Second callback');
});
```

This code will print `First callback` and then `Second callback`. Something important to note is that if one callback fails, the following won't get executed.

```
var promise = new MyPromise();
promise.then(function () {
    console.log('I get executed');
    throw 'Error';
}).then(function () {
    console.log('I don't get executed');
});
```

This allows us to execute async code in a synchronous manner, as we are used to. Also, if one of your callbacks returns a promise, the subsequent `then` call will be evaluated on that promise you returned, this makes chaining Promises easy as pie.

Common Promise Scenario #1

So by now you've read quite a lot on abstract promises, let's get down to a real use case! I'll illustrate two, the first one is for caching AJAX requests. First let's write an implementation **without** promises:

```
var cache = {};
function loadSong(id, callback) {
    if(!cache[id] !== null) {
        callback(cache[id]);
        return;
    }
    if(cache[id] === null) {
        return;
    }
    cache[id] = null;
    $.post('/songs', { id: id }, function (data) {
        cache[id] = data;
        callback(cache[id]);
    });
}
```

The code above simply loads a song by `id` only once and then executes a `callback`. Not too bad, a lot of Javascript code looks like that, but we can do better! Let's use Promises! Because Promises only get resolved once, if `then` is called when a Promise is already resolved, the callback will be executed **immediately**, because of this and also taking advantage of jQuery's promises implementation we can easily cache AJAX requests as jQuery AJAX functions actually return promises:

```
var cache = {};
function loadSong(id, callback) {
    if(!cache[id]) {
        cache[id] = $.post('/songs', { 'id': id }).promise();
    }
    cache[id].done(callback);
}
```

Much better! You might notice the use of `.promise()`, this is because jQuery uses [Deferred](#), which is a superset of Promise. In this case we just want to return a Promise object which can be resolved only once, so we use `.promise()`. The code would also work if we returned a `Deferred` but it's a good jQuery practice to always return Promises.

Note on jQuery's Promises: There is a catch with jQuery Promises, they aren't fully compatible with the Promises specification as you can see in [Domenic Denicola's article](#) so there are some edge cases, but I find it convenient to use jQuery's implementation to demonstrate the concept as it's used by a wide variety of developers. If you want a more loyal implementation you can try other libraries such as [Q](#), [rsvp](#) or [when.js](#) as Domenic suggests.

Common Promise Scenario #2

This situation it's also quite common, it happens whenever we have to wait for several async actions to finish before proceeding, for example:

```
// Get latest comments based on user and latest posts
$.post('/user/get_current_user', function (user) {
    $.post('/posts/latest', function (posts) {
        $.post('/comments/latest', {
            user: user,
            posts: posts
        }, drawToDOM);
    });
});
```

Now **that's ugly**, callback spaghetti should always be avoided! Luckily a lot of libraries which implement Promises also provide method called `when` which resolves when all it's arguments (which are promises) are resolved, knowing this we can simply code:

```
$.when(
    $.post('/user/get_current_user'),
    $.post('/posts/latest')
).then(function (users, posts) {
    $.post('/comments/latest', drawToDom);
});
```

That's much better, now the first two AJAX requests are executed at the same time, so it will be considerably faster. Moreover, the code is much cleaner, but we can still improve! Remember that we can chain `then` calls? Let's try that:

```
$.when(
    $.post('/user/get_current_user'),
    $.post('/posts/latest')
).then(function (users, posts) {
    return $.post('/comments/latest');
```

```
}).then(drawToDom)
```

Looking good! Promises are awesome, don't be afraid to use them and try them out. If you want to know more about Promises I highly recommend watching [this great jQuery conference talk](#) by Alex McPherson. I also [wrote a brief article on the subject](#).

Conclusion

PubSub as well as all design patterns has known advantages and disadvantages which you should know very well. It's main strength is highly decoupling the different parts of your application, but it also helps us work with async code and is pretty easy to use and get started, so don't hesitate to try it out in your next project!

Promises can be used in conjunction with the PubSub pattern and are a very important tool we can use to deal with complex async events, as well as avoiding [callback hell](#).

References

The following are references I used for writing this article and are recommended reads on the subject.

10 : Glossary of Modern JavaScript Concepts: Part 1

Learn the fundamentals of functional programming, reactive programming, and functional reactive programming in JavaScript.

TL;DR: In the first part of the Glossary of Modern JS Concepts series, we'll gain an understanding of *functional programming*, *reactive programming*, and *functional reactive programming*. To do so, we'll learn about purity, statefulness and statelessness, immutability and mutability, imperative and declarative programming, higher-order functions, observables, and the FP, RP, and FRP paradigms.

Introduction

Modern JavaScript has experienced massive proliferation over recent years and shows no signs of slowing. Numerous concepts appearing in JS blogs and documentation are still unfamiliar to many front-end developers. In this post series, we'll learn intermediate and advanced concepts in the current front-end programming landscape and explore how they apply to modern JavaScript.

Concepts

In this article, we'll address concepts that are crucial to understanding **functional programming, reactive programming, and functional reactive programming and their use with JavaScript**.

You can jump straight into each concept here, or continue reading to learn about them in order.

- [Purity: Pure Functions, Impure Functions, Side Effects](#)
 - [State: Stateful and Stateless](#)
 - [Immutability and Mutability](#)
 - [Imperative and Declarative Programming](#)
 - [Higher-order Functions](#)
 - [Functional Programming](#)
 - [Observables: Hot and Cold](#)
 - [Reactive Programming](#)
 - [Functional Reactive Programming](#)
-

Purity: Pure Functions, Impure Functions, Side Effects

Pure Functions

A **pure function's return value** is determined only by its *input values* (arguments) with no side effects. When given the same argument, the result will always be the same. Here is an example:

```
function half(x) {  
    return x / 2;  
}
```

```
}
```

The `half(x)` function takes a number `x` and returns a value of half of `x`. If we pass an argument of `8` to this function, the function will always return `4`. When invoked, a pure function can be replaced by its result. For example, we could replace `half(8)` with `4` wherever used in our code with no change to the final outcome. This is called [referential transparency](#).

Pure functions only depend on what's passed to them. For example, a pure function cannot reference variables from a parent scope unless they are explicitly passed into the function as arguments. Even then, the function can *not modify* the parent scope.

```
var someNum = 8;

// this is NOT a pure function
function impureHalf() {
  return someNum / 2;
}
```

In summary:

- Pure functions must take arguments.
- The same input (arguments) will always produce the same output (return).
- Pure functions rely only on local state and do not mutate external state (*note: `console.log` changes global state*).
- Pure functions do not produce [side effects](#).
- Pure functions cannot call impure functions.

Impure Functions

An **impure function** mutates state outside its scope. Any function that has *side effects* (see below) is impure.

Procedural functions with no utilized return value are also impure.

Consider the following examples:

```
// impure function producing a side effect
function showAlert() {
  alert('This is a side effect!');
}
```

```
// impure function mutating external state
var globalVal = 1;
function incrementGlobalVal(x) {
  globalVal += x;
}
```

```
// impure function calling pure functions procedurally
function proceduralFn() {
    const result1 = pureFnFirst(1);
    const result2 = pureFnLast(2);
    console.log(`Done with ${result1} and ${result2}!`);
}

// impure function that resembles a pure function,
// but returns different results given the same inputs
function getRandomRange(min, max) {
    return Math.random() * (max - min) + min;
}
```

Side Effects in JavaScript

When a function or expression modifies state outside its own context, the result is a **side effect**. Examples of side effects include making a call to an API, manipulating the DOM, raising an alert dialog, writing to a database, etc. If a function produces side effects, it is considered *impure*. Functions that cause side effects are less predictable and harder to test since they result in changes outside their local scope.

Purity Takeaways

Plenty of quality code consists of *impure* functions that procedurally invoke *pure* functions. This still produces advantages for testing and immutability. Referential transparency also enables [memoization](#): caching and storing function call results and [reusing the cached results](#) when the same inputs are used again. It can be a challenge to determine when functions are truly pure.

To learn more about **purity**, check out the following resources:

- [Pure versus impure functions](#)
- [Master the JavaScript Interview: What is a Pure Function?](#)
- [Functional Programming: Pure Functions](#)

State

State refers to the information a program has access to and can operate on at a point in time. This includes data stored in memory as well as OS memory, input/output ports, database, etc. For example, the contents of variables in an application at any given instant are representative of the application's *state*.

Stateful

Stateful programs, apps, or components store data in memory about the current state. They can modify the state as well as access its history. The following example is *stateful*:

```
// stateful
var number = 1;
function increment() {
    return number++;
}
increment(); // global variable modified: number = 2
```

Stateless

Stateless functions or components perform tasks as though running them for the first time, every time. This means they do not reference or utilize any information from earlier in their execution. Statelessness enables *referential transparency*. Functions depend only on their arguments and do not access or need knowledge of anything outside their scope. [Pure functions](#) are stateless. See the following example:

```
// stateless
var number = 1;
function increment(n) {
    return n + 1;
}
increment(number); // global variable NOT modified: returns 2
```

Stateless applications *do* still manage state. However, they return their current state without *mutating* previous state. This is a tenet of [functional programming](#).

State Takeaways

State management is important for any complex application. Stateful functions or components modify state and store history, but are more difficult to test and debug. Stateless functions rely only on their inputs to produce outputs. A stateless program returns new state rather than *modifying* existing state.

To learn more about **state**, check out the following resources:

- [State](#)
- [Advantages of stateless programming](#)
- [Stateful and stateless components, the missing manual](#)
- [Redux: predictable state container for JavaScript apps](#)

Immutability and Mutability

The concepts of **immutability** and **mutability** are slightly more nebulous in JavaScript than in some other programming languages. However, you will hear a lot about immutability when reading about [functional programming](#) in JS. It's important to know what these terms mean classically and also how they are referenced and implemented in JavaScript. The definitions are simple enough:

Immutable

If an object is **immutable**, its value cannot be modified after creation.

Mutable

If an object is **mutable**, its value can be modified after creation.

By Design: Immutability and Mutability in JavaScript

In JavaScript, strings and number literals are *immutable by design*. This is easily understandable if we consider how we operate on them:

```
var str = 'Hello!';
var anotherStr = str.substring(2);
// result: str = 'Hello!' (unchanged)
// result: anotherStr = 'llo!' (new string)
```

Using the `.substring()` method on our `Hello!` string does *not* modify the original string. Instead, it creates a new string. We could reassign the `str` variable value to something else, but once we've created our `Hello!` string, it will always be `Hello!`.

Number literals are immutable as well. The following will always have the same result:

```
var three = 1 + 2;
// result: three = 3
```

Under no circumstances could `1 + 2` evaluate to anything other than `3`.

This demonstrates that immutability by design does exist in JavaScript. However, JS developers are aware that the language allows most things to be changed. For example, objects and arrays are *mutable by design*. Consider the following:

```
var arr = [1, 2, 3];
arr.push(4);
// result: arr = [1, 2, 3, 4]

var obj = { greeting: 'Hello' };
obj.name = 'Jon';
// result: obj = { greeting: 'Hello', name: 'Jon' }
```

In these examples, the *original* objects are mutated. New objects are not returned.

To learn more about mutability in other languages, check out [Mutable vs Immutable Objects](#).

In Practice: Immutability in JavaScript

[Functional programming](#) in JavaScript has gained a lot of momentum. But by design, JS is a very mutable, multi-paradigm language. Functional programming emphasizes *immutability*. Other functional languages will raise errors when a developer tries to mutate an immutable object. So how can we reconcile the innate mutability of JS when writing functional or functional reactive JS?

When we talk about functional programming in JS, the word "immutable" is used a lot, but it's the responsibility of the developer to write their code with immutability *in mind*. For example, [Redux relies on a single, immutable state tree](#). However, *JavaScript itself* is capable of mutating the state object. To implement an immutable state tree, we need to [return a new state object](#) each time the state changes.

JavaScript objects [can also be frozen](#) with `Object.freeze(obj)` [to make them immutable](#). Note that this is *shallow*, meaning object values within a frozen object can still be mutated. To further ensure immutability, [functions like Mozilla's deepFreeze\(\)](#) and [npm deep-freeze](#) can recursively freeze objects. Freezing is most practical when used in *tests* rather than in application JS. Tests will alert developers when mutations occur so they can be corrected or avoided in the actual build without `Object.freeze` cluttering up the core code.

There are also libraries available to support immutability in JS. [Mori](#) delivers persistent data structures based on Clojure. [Immutable.js](#) by Facebook also provides [immutable collections for JS](#). Utility libraries like [Underscore.js](#) and [lodash](#) provide methods and modules to promote a more immutable [functional programming style](#).

Immutability and Mutability Takeaways

Overall, JavaScript is a very mutable language. Some styles of JS coding *rely* on this innate mutability. However, when writing functional JS, implementing immutability requires mindfulness. JS will not natively throw errors when you modify something unintentionally. Testing and libraries can assist, but working with immutability in JS takes practice and methodology.

Immutability has advantages. It results in code that is simpler to reason about. It also enables [persistency](#), the ability to keep older versions of a data structure and copy only the parts that have changed.

The disadvantage of immutability is that many algorithms and operations cannot be implemented efficiently.

To learn more about **immutability and mutability**, check out the following resources:

- [Immutability in JavaScript](#)
- [Immutable Objects with Object Freeze](#)
- [Mutable vs Immutable Objects](#)
- [Using Immutable Data Structures in JavaScript](#)
- [Getting Started with Redux](#) (includes examples for addressing immutable state)

Imperative and Declarative Programming

While some languages were designed to be **imperative** (C, PHP) or **declarative** (SQL, HTML), JavaScript (and others like [Java](#) and [C#](#)) can support both programming paradigms.

Most developers familiar with even the most basic JavaScript have written imperative code: instructions informing the computer *how* to achieve a desired result. If you've written a `for` loop, you've written imperative JS.

Declarative code tells the computer *what* you want to achieve rather than how, and the computer takes care of how to achieve the end result without explicit description from the developer. If you've used `Array.map`, you've written declarative JS.

Imperative Programming

Imperative programming describes *how* a program's logic works in explicit commands with statements that modify the program state.

Consider a function that increments every number in an array of integers. An imperative JavaScript example of this might be:

```
function incrementArray(arr) {  
    let resultArr = [];  
    for (let i = 0; i < arr.length; i++) {  
        resultArr.push(arr[i] + 1);  
    }  
    return resultArr;  
}
```

This function shows exactly *how* the function's logic works: we iterate over the array and explicitly increase each number, pushing it to a new array. We then return the resulting array. This is a step-by-step description of the function's logic.

Declarative Programming

Declarative programming describes *what* a program's logic accomplishes *without* describing how.

A very straightforward example of declarative programming can be demonstrated with [SQL](#). We can query a database table (`People`) for people with the last name `Smith` like so:

```
SELECT * FROM People WHERE LastName = 'Smith'
```

This code is easy to read and describes *what* we want to accomplish. There is no description of *how* the result should be achieved. The computer takes care of that.

Now consider the `incrementArray()` function we implemented imperatively above. Let's implement this declaratively now:

```
function incrementArray(arr) {  
    return arr.map(item => item + 1);  
}
```

We show *what* we want to achieve, but not how it works. The [Array.map\(\)](#) method returns a new array with the results of running the callback on each item from the passed array. This approach does not modify existing values, nor does it include any sequential logic showing *how* it creates the new array.

Note: JavaScript's [map](#), [reduce](#), and [filter](#) are declarative, [functional](#) array methods. Utility libraries like [lodash](#) provide methods like [every](#), [sortBy](#), [uniq](#), and more in addition to [map](#), [reduce](#), and [filter](#).

Imperative and Declarative Programming Takeaways

As a language, JavaScript allows both **imperative and declarative programming** paradigms. Much of the JS code we read and write is imperative. However, with the rise of [functional programming](#) in JS, declarative approaches are becoming more common.

Declarative programming has obvious advantages with regard to brevity and readability, but at the same time it can feel magical. Many JavaScript beginners can benefit from gaining experience writing imperative JS before diving too deep into declarative programming.

To learn more about **imperative and declarative programming**, check out the following resources:

- [Imperative vs Declarative Programming](#)
- [What's the Difference Between Imperative, Procedural, and Structured Programming?](#)
- [Imperative and \(Functional\) Declarative JS In Practice](#)
- [JavaScript's Map, Reduce, and Filter](#)

Higher-order Functions

A **higher-order function** is a function that:

- accepts another function as an argument, or
- returns a function as a result.

In JavaScript, functions are [first-class objects](#). They can be stored and passed around as *values*: we can assign a function to a variable or pass a function to another function.

```
const double = function(x) {
  return x * 2;
}
const timesTwo = double;

timesTwo(4); // result: returns 8
```

One example of taking a function as an argument is a *callback*. Callbacks can be inline anonymous functions or named functions:

```

const myBtn = document.getElementById('myButton');

// anonymous callback function
myBtn.addEventListener('click', function(e) { console.log(`Click event: ${e}`); });

// named callback function
function btnHandler(e) {
  console.log(`Click event: ${e}`);
}

myBtn.addEventListener('click', btnHandler);

```

We can also pass a function as an argument to any other function we create and then execute that argument:

```

function sayHi() {
  alert('Hi!');
}

function greet(greeting) {
  greeting();
}

greet(sayHi); // alerts "Hi!"

```

Note: When *passing a named function as an argument*, as in the two examples above, we don't use parentheses `()`. This way we're passing the function as an object. Parentheses *execute* the function and pass the result instead of the function itself.

Higher-order functions can also return another function:

```

function whenMeetingJohn() {
  return function() {
    alert('Hi!');
  }
}

var atLunchToday = whenMeetingJohn();

atLunchToday(); // alerts "Hi!"

```

Higher-order Function Takeaways

The nature of JavaScript functions as first-class objects make them prime for facilitating [functional programming](#).

To learn more about **higher-order functions**, check out the following resources:

- [Functions are first class objects in JavaScript](#)

- [Higher-Order Functions in JavaScript](#)
 - [Higher-order functions - Part 1 of Functional Programming in JavaScript](#)
 - [Eloquent JavaScript - Higher-order Functions](#)
 - [Higher Order Functions](#)
-

Functional Programming

Now we've learned about purity, statelessness, immutability, declarative programming, and higher-order functions. These are all concepts that are important in understanding the functional programming paradigm.

In Practice: Functional Programming with JavaScript

Functional programming encompasses the above concepts in the following ways:

- Core functionality is implemented using pure functions without side effects.
- Data is immutable.
- Functional programs are stateless.
- Imperative container code manages side effects and executes declarative, pure core code.*

*If we tried to write a JavaScript web application composed of nothing but pure functions with no side effects, it couldn't interact with its environment and therefore wouldn't be particularly useful.

Let's explore an example. Say we have some text copy and we want to get its word count. We also want to find keywords that are longer than five characters. Using functional programming, our resulting code might look something like this:

```
const fpCopy = `Functional programming is powerful and enjoyable to write. It's very cool!`;

// remove punctuation from string
const stripPunctuation = (str) =>
  str.replace(/[.,\/#!\$%\^&\*;:{}=\-_`~()]/g, '');

// split passed string on spaces to create an array
const getArr = (str) =>
  str.split(' ');

// count items in the passed array
const getWordCount = (arr) =>
  arr.length;

// find items in the passed array longer than 5 characters
// make items lower case
const getKeywords = (arr) =>
  arr
```

```

.filter(item => item.length > 5)
.map(item => item.toLowerCase());

// process copy to prep the string, create an array, count words, and get keywords
function processCopy(str, prepFn, arrFn, countFn, kwFn) {
  const copyArray = arrFn(prepFn(str));

  console.log(`Word count: ${countFn(copyArray)} `);
  console.log(`Keywords: ${kwFn(copyArray)} `);
}

processCopy(fpCopy, stripPunctuation, getArr, getWordCount, getKeywords);
// result: Word count: 11
// result: Keywords: functional,programming,powerful,enjoyable

```

This code is available to run at this [JSFiddle: Functional Programming with JavaScript](#). It's broken into digestible, declarative functions with clear purpose. If we step through it and read the comments, no further explanation of the code should be necessary. Each *core* function is modular and relies only on its inputs ([pure](#)). The last function processes the core to generate the collective outputs. This function, `processCopy()`, is the impure container that executes the core and manages side effects. We've used a [higher-order function](#) that accepts the other functions as arguments to maintain the functional style.

Functional Programming Takeaways

Immutable data and statelessness mean that the program's existing state is not modified. Instead, new values are returned. Pure functions are used for core functionality. In order to implement the program and handle necessary side effects, impure functions can call pure functions imperatively.

Observables

Observables are similar to arrays, except instead of being stored in memory, items arrive asynchronously over time (also called *streams*). We can *subscribe* to observables and react to events emitted by them. JavaScript observables are an implementation of the [observer pattern](#). [Reactive Extensions](#) (commonly known as Rx*) provides an observables library for JS via [RxJS](#).

To demonstrate the concept of observables, let's consider a simple example: resizing the browser window. It's easy to understand observables in this context. Resizing the browser window emits a stream of events over a period of time as the window is dragged to its desired size. We can create an observable and subscribe to it to react to the stream of resize events:

```

// create window resize stream
// throttle resize events
const resize$ =
  Rx.Observable
    .fromEvent(window, 'resize')

```

```
.throttleTime(350);

// subscribe to the resize$ observable
// log window width x height
const subscription =
  resize$.subscribe((event) => {
    let t = event.target;
    console.log(` ${t.innerWidth}px x ${t.innerHeight}px`);
  });
});
```

The example code above shows that as the window size changes, we can throttle the observable stream and subscribe to the changes to respond to new values in the collection. This is an example of a *hot observable*.

Hot Observables

User interface events like button clicks, mouse movement, etc. are *hot*. **Hot observables** will always push even if we're not specifically reacting to them with a subscription. The window resize example above is a hot observable: the `resize$` observable fires whether or not `subscription` exists.

Cold Observables

A **cold observable** begins pushing *only* when we subscribe to it. If we subscribe again, it will start over.

Let's create an observable collection of numbers ranging from `1` to `5`:

```
// create source number stream
const source$ = Rx.Observable.range(1, 5);

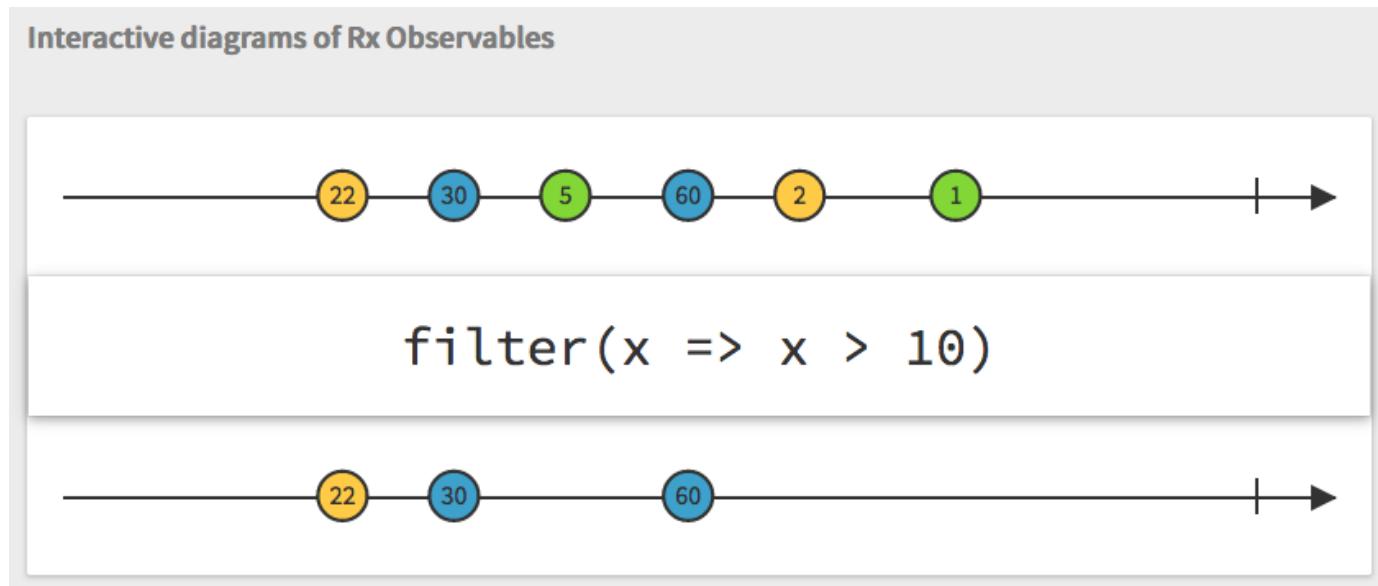
// subscribe to source$ observable
const subscription = source$.subscribe(
  (value) => { console.log(`Next: ${value}`); }, // onNext
  (event) => { console.log(`Error: ${event}`); }, // onError
  () => { console.log('Completed!'); } // onCompleted
);
```

We can [`subscribe\(\)`](#) to the `source$` observable we just created. Upon subscription, the values are sent in sequence to the observer. The `onNext` callback logs the values: `Next: 1`, `Next: 2`, etc. until completion: `Completed!`. The cold `source$` observable we created doesn't push unless we `subscribe` to it.

Observables Takeaways

Observables are streams. We can observe any stream: from resize events to existing arrays to API responses. We can create observables from almost anything. A *promise* is an observable with a single emitted value, but observables can return many values over time.

We can operate on observables in many ways. [RxJS utilizes numerous operator methods](#). Observables are often visualized using points on a line, as demonstrated on the [RxMarbles](#) site. Since the stream consists of asynchronous events over *time*, it's easy to conceptualize this in a linear fashion and use such visualizations to understand Rx* operators. For example, the following RxMarbles image illustrates the [filter operator](#):



To learn more about **observables**, check out the following resources:

- [Reactive Extensions: Observable](#)
- [Creating and Subscribing to Simple Observable Sequences](#)
- [The introduction to Reactive Programming you've been missing: Request and Response](#)
- [Introducing the Observable](#)
- [RxMarbles](#)
- [Rx Book - Observable](#)
- [Introducing the Observable](#)

Reactive Programming

Reactive programming is concerned with propagating and responding to incoming events over time, [declaratively](#) (describing *what* to do rather than *how*).

Reactive programming is often associated with [Reactive Extensions](#), an API for asynchronous programming with [observable streams](#). Reactive Extensions (abbreviated Rx*) [provides libraries for a variety of languages](#), including JavaScript ([RxJS](#)).

In Practice: Reactive Programming with JavaScript

Here is an example of reactive programming with observables. Let's say we have an input where the user can enter a six-character confirmation code and we want to print out the latest valid code attempt. Our HTML might look like this:

```
<!-- HTML -->
<input id="confirmation-code" type="text">
```

```
<p>
  <strong>Valid code attempt:</strong>
  <code id="attempted-code"></code>
</p>
```

We'll use RxJS and create a stream of input events to implement our functionality, like so:

```
// JS
const confCodeInput = document.getElementById('confirmation-code');
const attemptedCode = document.getElementById('attempted-code');

const confCodes$ =
  Rx.Observable
    .fromEvent(confCodeInput, 'input')
    .map(e => e.target.value)
    .filter(code => code.length === 6);

const subscription = confCodes$.subscribe(
  (value) => attemptedCode.innerText = value,
  (event) => { console.warn(`Error: ${event}`); },
  () => { console.info('Completed!'); }
);
```

This code can be run at this [JSFiddle: Reactive Programming with JavaScript](#). We'll observe [events from](#) the `confCodeInput` input element. Then we'll use the [map operator](#) to get the `value` from each input event. Next, we'll [filter](#) any results that are not six characters so they won't appear in the returned stream. Finally, we'll [subscribe](#) to our `confCodes$` observable and print out the latest valid confirmation code attempt. Note that this was done in response to events over time, declaratively: this is the crux of reactive programming.

Reactive Programming Takeaways

The reactive programming paradigm involves observing and reacting to events in asynchronous data streams. RxJS is used in [Angular](#) and is gaining popularity as a JavaScript solution for reactive programming.

To learn more about **reactive programming**, check out the following resources:

- [The introduction to Reactive Programming you've been missing](#)
- [Introduction to Rx](#)
- [The Reactive Manifesto](#)
- [Understanding Reactive Programming and RxJS](#)
- [Reactive Programming](#)
- [Modernization of Reactivity](#)
- [Reactive-Extensions RxJS API Core](#)

Functional Reactive Programming

In simple terms, functional reactive programming could be summarized as declaratively responding to events or behaviors over time. To understand the tenets of FRP in more depth, let's take a look at FRP's formulation. Then we'll examine its use in relation to JavaScript.

What is Functional Reactive Programming?

A [more complete definition](#) from [Conal Elliot, FRP's formulator](#), would be that **functional reactive programming** is "[denotative](#) and temporally continuous". Elliot mentions that he prefers to describe this programming paradigm as *denotative continuous-time programming* as opposed to "functional reactive programming".

Functional reactive programming, at its most basic, original definition, has two fundamental properties:

- **denotative**: the meaning of each function or type is precise, simple, and implementation-independent ("functional" references this)
- **continuous time**: [variables have a particular value for a very short time: between any two points are an infinite number of other points](#); provides transformation flexibility, efficiency, modularity, and accuracy ("reactive" references this)

Again, when we put it simply: [functional reactive programming is programming declaratively with time-varying values](#).

To understand *continuous time / temporal continuity*, consider an analogy using vector graphics. Vector graphics have an *infinite resolution*. Unlike bitmap graphics (discrete resolution), vector graphics scale indefinitely. They never pixellate or become indistinct when particularly large or small the way bitmap graphics do.

"FRP expressions describe entire evolutions of values over time, representing these evolutions directly as first-class values."

—Conal Elliot

Functional reactive programming should be:

- dynamic: can react over time *or* to input changes
- time-varying: reactive *behaviors* can change continually while reactive *values* change discretely
- efficient: minimize amount of processing necessary when inputs change
- historically aware: pure functions map state from a previous point in time to the next point in time; state changes concern the local element and not the global program state

Conal Elliot's slides on the [Essence and Origins of FRP can be viewed here](#). The programming language [Haskell](#) lends itself to true FRP due to its functional, pure, and lazy nature. Evan Czaplicki, the creator of [Elm](#), gives a great overview of FRP in his talk [Controlling Time and Space: Understanding the Many Formulations of FRP](#).

In fact, let's talk briefly about [Evan Czaplicki's Elm](#). Elm is a functional, typed language for building web applications. It compiles to JavaScript, CSS, and HTML. [The Elm Architecture](#) was the inspiration for the [Redux](#) state container for JS apps. [Elm was originally considered a true functional reactive programming language](#), but as of version 0.17, it implemented *subscriptions* instead of signals in the interest of making the language easier to learn and use. In doing so, Elm [bid farewell to FRP](#).

In Practice: Functional Reactive Programming and JavaScript

The traditional definition of FRP can be difficult to grasp, especially for developers who don't have experience with languages like Haskell or Elm. However, the term has come up more frequently in the front-end ecosystem, so let's shed some light on its application in JavaScript.

In order to reconcile what you may have read about FRP in JS, it's important to understand that [Rx*](#), [Bacon.js](#), [Angular](#), and others are *not* consistent with the two primary fundamentals of Conal Elliot's definition of FRP. [Elliot states that Rx* and Bacon.js are not FRP. Instead, they are "compositional event systems inspired by FRP".](#)

Functional reactive programming, *as it relates specifically to JavaScript implementations*, refers to programming in a [functional](#) style while creating and reacting to [streams](#). This is fairly far from Elliot's original formulation (which [specifically excludes streams as a component](#)), but is nevertheless inspired by traditional FRP.

It's also crucial to understand that JavaScript inherently interacts with the user and UI, the DOM, and often a backend. [Side effects](#) and [imperative](#) code are par for the course, even when taking a [functional](#) or functional reactive approach. *Without* imperative or impure code, a JS web application with a UI wouldn't be much use because it couldn't interact with its environment.

Let's take a look at an example to demonstrate the basic principles of *FRP-inspired* JavaScript. This sample uses RxJS and prints out mouse movements over a period of ten seconds:

```
// create a time observable that adds an item every 1 second
// map so resulting stream contains event values
const time$ =
  Rx.Observable
    .timer(0, 1000)
    .timeInterval()
    .map(e => e.value);

// create a mouse movement observable
// throttle to every 350ms
// map so resulting stream pushes objects with x and y coordinates
const move$ =
  Rx.Observable
    .fromEvent(document, 'mousemove')
    .throttleTime(350)
    .map(e => { return {x: e.clientX, y: e.clientY} });

// merge time + mouse movement streams
// complete after 10 seconds
const source$ =
  Rx.Observable
    .merge(time$, move$)
    .takeUntil(Rx.Observable.timer(10000));
```

```

// subscribe to merged source$ observable
// if value is a number, createTimeset()
// if value is a coordinates object, addPoint()
const subscription =
  source$.subscribe(
    // onNext
    (x) => {
      if (typeof x === 'number') {
        createTimeset(x);
      } else {
        addPoint(x);
      }
    },
    // onError
    (err) => { console.warn('Error:', err); },
    // onCompleted
    () => { console.info('Completed'); }
  );

// add element to DOM to list out points touched in a particular second
function createTimeset(n) {
  const elem = document.createElement('div');
  const num = n + 1;
  elem.id = 't' + num;
  elem.innerHTML = `<strong>${num}</strong>: `;
  document.body.appendChild(elem);
}

// add points touched to latest time in stream
function addPoint(pointObj) {
  // add point to last appended element
  const numberElem = document.getElementsByTagName('body')[0].lastChild;
  numberElem.innerHTML += ` (${pointObj.x}, ${pointObj.y}) `;
}

```

You can check out this code in action in this [JSFiddle: FRP-inspired JavaScript](#). Run the fiddle and move your mouse over the result area of the screen as it counts up to 10 seconds. You should see mouse coordinates appear along with the counter. This indicates where your mouse was during each 1-second time interval.

Let's briefly discuss this implementation step-by-step.

First, we'll create an [observable](#) called `time$`. This is a timer that adds a value to the collection every `1000ms` (every second). We need to `map` the timer event to extract its `value` and push it in the resulting stream.

Next, we'll create a `move$` observable from the `documentmousemove` event. Mouse movement is *continuous*. At any point in the sequence, there are an infinite number of points in between. We'll throttle this so the resulting stream is more manageable. Then we can `map` the event to return an object with `x` and `y` values to represent mouse coordinates.

Next we want to `merge` the `time$` and `move$` streams. This is a *combining operator*. This way we can plot which mouse movements occurred during each time interval. We'll call the resulting observable `source$`. We'll also limit the `source$` observable so that it completes after ten seconds (`10000ms`).

Now that we have our merged stream of time and movement, we'll create a `subscription` to the `source$` observable so we can react to it. In our `onNext` callback, we'll check to see if the value is a `number` or not. If it is, we want to call a function called `createTimeset()`. If it's a coordinates object, we'll call `addPoint()`. In the `onError` and `onCompleted` callbacks, we'll simply log some information.

Let's look at the `createTimeset(n)` function. We'll create a new `div` element for each second interval, label it, and append it to the DOM.

In the `addPoint(pointObj)` function, we'll print out the latest coordinates in the most recent timeset `div`. This will associate each set of coordinates with its corresponding time interval. We can now read where the mouse has been over time.

Note: These functions are [impure](#): they have no return value and they also produce side effects. The side effects are DOM manipulations. As mentioned earlier, the JavaScript we need to write for our apps frequently interacts with scope outside its functions.

Functional Reactive Programming Takeaways

FRP encodes actions that react to events using pure functions that map state from a previous point in time to the next point in time. FRP in JavaScript doesn't adhere to the two primary fundamentals of Conal Elliot's FRP, but there is certainly value in abstractions of the original concept. JavaScript relies heavily on side effects and imperative programming, but we can certainly take advantage of the power of FRP concepts to improve our JS.

Finally, consider this quote from the [first edition of Eloquent JavaScript](#) (the [second edition is available here](#)):

"Fu-Tzu had written a small program that was full of global state and dubious shortcuts. Reading it, a student asked 'You warned us against these techniques, yet I find them in your program. How can this be?'

Fu-Tzu said 'There is no need to fetch a water hose when the house is not on fire.' {This is not to be read as an encouragement of sloppy programming, but rather as a warning against neurotic adherence to rules of thumb.}"

—Marijn Haverbeke, [Eloquent JavaScript, 1st Edition, Chapter 6](#)

To learn more about **functional reactive programming (FRP)**, check out the following resources:

- [Functional Reactive Programming for Beginners](#)
- [The Functional Reactive Misconception](#)
- [What is Functional Reactive Programming?](#)

- [Haskell - Functional Reactive Programming](#)
 - [Composing Reactive Animations](#)
 - [Specification for a functional reactive programming language](#)
 - [A more elegant specification for FRP](#)
 - [Functional Reactive Programming for Beginners](#)
 - [Elm - A Farewell to FRP](#)
 - [Early inspirations and new directions in functional reactive programming](#)
 - [Breaking Down FRP](#)
 - [Rx* is not FRP](#)
-

Conclusion

We'll conclude with another excellent quote from the first edition of [Eloquent JavaScript](#):

"A student had been sitting motionless behind his computer for hours, frowning darkly. He was trying to write a beautiful solution to a difficult problem, but could not find the right approach. Fu-Tzu hit him on the back of his head and shouted '*Type something!*' The student started writing an ugly solution. After he had finished, he suddenly understood the beautiful solution."

—Marijn Haverbeke, [Eloquent JavaScript, 1st Edition, Chapter 6](#)

The concepts necessary for understanding [functional programming](#), [reactive programming](#), and [functional reactive programming](#) can be difficult to grasp, let alone *master*. Writing code that takes advantage of a paradigm's fundamentals is the initial step, even if it isn't entirely faithful at first. Practice illuminates the path ahead and also reveals potential revisions.

With this glossary as a starting point, you can begin taking advantage of these concepts and programming paradigms to increase your JavaScript expertise. If anything is still unclear regarding these topics, please consult the links in each section for additional resources. We'll cover more concepts in the next Modern JS Glossary post!

11 : ES6 Features

<https://codetower.github.io/es6-features/>

Arrow Functions

A short hand notation for `function()`, but it does not bind `this`.

```
no-eval

var odds = evens.map(v => v + 1); // no parentes and no brackets
var nums = evens.map((v, i) => v + i);
var pairs = evens.map(v => ({even: v, odd: v + 1}));

// Statement bodies
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});
```

How does `this` work?

```
var object = {
  name: "Name",
  arrowGetName: () => this.name,
  regularGetName: function() { return this.name },
  arrowGetThis: () => this,
  regularGetThis: function() { return this }

}

console.log(this.name)
console.log(object.arrowGetName());
console.log(object.arrowGetThis());
console.log(this)
console.log(object.regularGetName());
console.log(object.regularGetThis());
```

Classes

As we know them from "real" languages. Syntactic sugar on top of prototype-inheritance.

```
no-eval

class SkinnedMesh extends THREE.Mesh {
  constructor(geometry, materials) {
```

```

super(geometry, materials);

this.idMatrix = SkinnedMesh.defaultMatrix();
this.bones = [];
this.boneMatrices = [];
//...
}

update(camera) {
//...
super.update();
}

get boneCount() {
    return this.bones.length;
}

set matrixType(matrixType) {
    this.idMatrix = SkinnedMesh[matrixType]();
}

static defaultMatrix() {
    return new THREE.Matrix4();
}
}

```

Lebab.io

Enhanced Object Literals

```

var theProtoObj = {
    toString: function() {
        return "The ProtoObject To string"
    }
}

```

```
var handler = () => "handler"
```

```

var obj = {
    // __proto__
    __proto__: theProtoObj,
    // Shorthand for 'handler: handler'
    handler,
    // Methods
}

```

```

    toString() {

        // Super calls
        return "d " + super.toString();
    },

    // Computed (dynamic) property names
    [ "prop_" + (() => 42)(): 42 ]: 42
};

console.log(obj.handler)
console.log(obj.handler())
console.log(obj.toString())
console.log(obj.prop_42)

```

String interpolation

Nice syntax for string interpolation (but slightly worse performance, [Source](#))

```

var name = "Bob", time = "today";

var multiLine = `This
Line
Spans Multiple
Lines`


console.log(`Hello ${name}, how are you ${time}?`)
console.log(multiLine)

```

Destructuring

```

// list "matching"
var [a, , b] = [1,2,3];
console.log(a)
console.log(b)

```

Objects can be destructured as well.

```
nodes = () => { return {op: "a", lhs: "b", rhs: "c"}}
var { op: a, lhs: b , rhs: c } = nodes()
console.log(a)
console.log(b)
console.log(c)
```

Using Shorthand notation.

```
nodes = () => { return {lhs: "a", op: "b", rhs: "c"}}

// binds `op`, `lhs` and `rhs` in scope
var {op, lhs, rhs} = nodes()

console.log(op)
console.log(lhs)
console.log(rhs)
```

Can be used in parameter position

```
function g({name: x}) {
  return x
}

function m({name}) {
  return name
}

console.log(g({name: 5}))
console.log(m({name: 5}))
```

Fail-soft destructuring

```
var [a] = []
var [b = 1] = []
var c = [];
console.log(a)
console.log(b);
console.log(c);
```

Default

```
function f(x, y=12) {
  return x + y;
}

console.log(f(3))
```

Spread

In functions

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

// Pass each elem of array as argument
console.log(f(...[1,2,3]))
```

In arrays

```
var parts = ["shoulders", "knees"];
var lyrics = ["head", ...parts, "and", "toes"];

console.log(lyrics)
```

Spread + Object Literals

We can do cool stuff with this in object creations.

```
no-eval

let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
console.log(x); // 1
console.log(y); // 2
console.log(z); // { a: 3, b: 4 }

// Spread properties
let n = { x, y, ...z };
console.log(n); // { x: 1, y: 2, a: 3, b: 4 }
console.log(obj)
```

Sadly it is not support yet

```
npm install --save-dev babel-plugin-transform-object-rest-spread
```

Rest

```
function demo(part1, ...part2) {  
    return ({part1, part2})  
}  
  
console.log(demo(1,2,3,4,5,6))
```

Let

`Let` is the new `var`. As it has "sane" bindings.

```
{  
    var globalVar = "from demo1"  
}  
  
{  
    let globalLet = "from demo2";  
}  
  
console.log(globalVar)  
console.log(globalLet)
```

However, it does not assign anything to `window`

```
let me = "go"; // globally scoped  
var i = "able"; // globally scoped  
  
console.log(window.me);  
console.log(window.i);
```

It is not possible to redeclare a variable using `let`

```
let me = "foo";  
let me = "bar";  
console.log(me);
```

```
var me = "foo";  
var me = "bar";  
console.log(me)
```

Const

`Const` is for read only variables

```
const a = "b"  
a = "a"
```

For..of

New type of iterators with an alternative to the `for..in`. It returns the value instead of the `keys`.

```
let list = [4, 5, 6];  
  
console.log(list)  
  
for (let i in list) {  
    console.log(i);  
}
```

```
let list = [4, 5, 6];  
  
console.log(list)  
  
for (let i of list) {  
    console.log(i);  
}
```

Iterators

The iterator is a more dynamic type than arrays.

```
let infinite = {  
    [Symbol.iterator]() {  
        let c = 0;  
        return {  
            next() {  
                c++;  
                return { done: false, value: c }  
            }  
        }  
    }  
}
```

```
console.log("start");

for (var n of infinite) {
    // truncate the sequence at 1000
    if (n > 10)
        break;
    console.log(n);
}
```

Using Typescript interfaces we can see how it looks

```
no-eval
interface IteratorResult {
    done: boolean;
    value: any;
}
interface Iterator {
    next(): IteratorResult;
}
interface Iterable {
    [Symbol.iterator](): Iterator
}
```

Generators

Generators create iterators, and are more dynamic than iterators. They do not have to keep track of state in the same manner and does not support the concept of done.

```
var infinity = {
    [Symbol.iterator]: function*() {
        var c = 1;
        for (;;) {
            yield c++;
        }
    }
}

console.log("start")
for (var n of infinity) {
    // truncate the sequence at 1000
    if (n > 10)
        break;
```

```
    console.log(n);
}
```

Using typescript again to show the interfaces.

```
no-eval
interface Generator extends Iterator {
    next(value?: any): IteratorResult;
    throw(exception: any);
}
```

[function* Iterators and generator](#)

An example of yield*

```
function* anotherGenerator(i) {
    yield i + 1;
    yield i + 2;
    yield i + 3;
}
```

```
function* generator(i) {
    yield i;
    yield* anotherGenerator(i);
    yield i + 10;
}
```

```
var gen = generator(10);
```

```
console.log(gen.next().value);
console.log(gen.next().value);
console.log(gen.next().value);
console.log(gen.next().value);
console.log(gen.next().value);
```

Unicode

ES6 provides better support for Unicode.

```
var regex = new RegExp('\u{61}', 'u');

console.log(regex.unicode)
```

```
console.log("\uD842\uDFD7")
console.log("\uD842\uDFD7".codePointAt())
```

Modules & Module Loaders

Native support for modules.

```
no-eval
import defaultMember from "module-name";
import * as name from "module-name";
import { member } from "module-name";
import { member as alias } from "module-name";
import { member1 , member2 } from "module-name";
import { member1 , member2 as alias2 , [...] } from "module-name";
import defaultMember, { member [ , [...] ] } from "module-name";
import defaultMember, * as name from "module-name";
import "module-name";
```

```
no-eval
export { name1, name2, ... , nameN };
export { variable1 as name1, variable2 as name2, ... , nameN };
export let name1, name2, ... , nameN; // also var
export let name1 = ... , name2 = ... , ... , nameN; // also var, const

export expression;
export default expression;
export default function (...) { ... } // also class, function*
export default function name1(...) { ... } // also class, function*
export { name1 as default, ... };

export * from ...;
export { name1, name2, ... , nameN } from ...;
export { import1 as name1, import2 as name2, ... , nameN } from ...;
```

[Import](#)

[Export](#)

Set

Sets as in the mathematical counterpart where all items are unique.

```
var set = new Set();
set.add("Potato").add("Tomato").add("Tomato");
```

```
console.log(set.size)
console.log(set.has("Tomato"))

for(var item of set) {
    console.log(item)
}
```

Set

WeakSet

The `WeakSet` object lets you store weakly held objects in a collection. Objects without an reference will be garbage collected.

```
var item = { a:"Potato"}
var set = new WeakSet();
set.add({ a:"Potato"}).add(item).add({ a:"Tomato"}).add({ a:"Tomato"});
console.log(set.size)
console.log(set.has({a:"Tomato"}))
console.log(set.has(item))

for(let item of set) {
    console.log(item)
}
```

WeakSet

Map

Maps, also known as dictionary.

```
var map = new Map();
map.set("Potato", 12);
map.set("Tomato", 34);

console.log(map.get("Potato"))

for(let item of map) {
    console.log(item)
}

for(let item in map) {
```

```
    console.log(item)
}
```

Map

WeakMap

Uses objects for keys, and only keeps weak reference to the keys.

```
var wm = new WeakMap();
```

```
var o1 = {}
var o2 = {}
var o3 = {}
```

```
wm.set(o1, 1);
wm.set(o2, 2);
wm.set(o3, {a: "a"});
wm.set({}, 4);
```

```
console.log(wm.get(o2));
console.log(wm.has({}))
```

```
delete o2;
```

```
console.log(wm.get(o3));
```

```
for(let item in wm) {
    console.log(item)
}
```

```
for(let item of wm) {
    console.log(item)
}
```

WeakMap

Proxies

Proxies can be used to alter objects behavoir. It allows us to define traps.

```
var obj = function ProfanityGenerator() {
    return {
        words: "Horrible words"
    }
}()

var handler = function CensoringHandler() {
    return {
        get: function (target, key) {
            return target[key].replace("Horrible", "Nice");
        },
    }
}()

var proxy = new Proxy(obj, handler);

console.log(proxy.words);
```

The following traps are available

```
no-eval
var handler =
{
    get:....,
    set:....,
    has:....,
    deleteProperty:....,
    apply:....,
    construct:....,
    getOwnPropertyDescriptor:....,
    defineProperty:....,
    getPrototypeOf:....,
    setPrototypeOf:....,
    enumerate:....,
    ownKeys:....,
    preventExtensions:....,
    isExtensible:...
}
```

[Proxies](#)

Symbols

Symbols are a new type. Can be used to create anonymous properties.

```
var typeSymbol = Symbol("type");

class Pet {

    constructor(type) {

        this[typeSymbol] = type;

    }

    getType() {
        return this[typeSymbol];
    }

}

var a = new Pet("dog");
console.log(a.getType());
console.log(Object.getOwnPropertyNames(a))

console.log(Symbol("a") === Symbol("a"))
```

[More info](#)

Inheritable Built-ins

We can now inherit from native classes.

```
class CustomArray extends Array {

}

var a = new CustomArray();

a[0] = 2
console.log(a[0])
```

It is not possible to override the getter function without using Proxies of arrays.

New Library

Various new methods and constants.

```
console.log(Number.EPSILON)
console.log(Number.isInteger(Infinity))
console.log(Number.isNaN("NaN"))

console.log(Math.acosh(3))
console.log(Math.hypot(3, 4))
console.log(Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2))

console.log("abcde".includes("cd") )
console.log("abc".repeat(3) )

console.log(Array.of(1, 2, 3) )
console.log([0, 0, 0].fill(7, 1) )
console.log([1, 2, 3].find(x => x == 3) )
console.log([1, 2, 3].findIndex(x => x == 2))
console.log([1, 2, 3, 4, 5].copyWithin(3, 0))
console.log(["a", "b", "c"].entries() )
console.log(["a", "b", "c"].keys() )
console.log(["a", "b", "c"].values() )

console.log(Object.assign({}, { origin: new Point(0,0) }))
```

Documentation: [Number](#), [Math](#), [Array.from](#), [Array.of](#), [Array.prototype.copyWithin](#), [Object.assign](#)

Binary and Octal

Literals for binary and octal numbering.

```
console.log(0b11111)
console.log(0o2342)

console.log(0xff); // also in es5
```

Promises

The bread and butter for async programming.

```
var p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("1"), 101)
```

```
)  
var p2 = new Promise((resolve, reject) => {  
    setTimeout(() => resolve("2"), 100)  
})  
  
Promise.race([p1, p2]).then((res) => {  
    console.log(res)  
})  
  
Promise.all([p1, p2]).then((res) => {  
    console.log(res)  
})
```

Quick Promise

Need a quick always resolved promise?

```
var p1 = Promise.resolve("1")  
var p2 = Promise.reject("2")  
  
Promise.race([p1, p2]).then((res) => {  
    console.log(res)  
})
```

Fail fast

If a promise fails `all` and `race` will reject as well.

```
var p1 = new Promise((resolve, reject) => {  
    setTimeout(() => resolve("1"), 1001)  
})  
var p2 = new Promise((resolve, reject) => {  
    setTimeout(() => reject("2"), 1)  
})  
  
Promise.race([p1, p2]).then((res) => {  
    console.log("success" + res)  
}, res => {  
    console.log("error " + res)  
})  
  
Promise.all([p1, p2]).then((res) => {
```

```

        console.log("success" + res)
    }, res => {
        console.log("error " + res)
})

```

[More Info](#)

Reflect

New type of meta programming with new API for existing and also few new methods.

```

var z = {w: "Super Hello"};
var y = {x: "hello", __proto__: z};

console.log(Reflect.getOwnPropertyDescriptor(y, "x"));
console.log(Reflect.has(y, "w"));
console.log(Reflect.ownKeys(y, "w"));

console.log(Reflect.has(y, "x"));
console.log(Reflect.deleteProperty(y, "x"))
console.log(Reflect.has(y, "x"));

```

[More Info](#)

Tail Call Optimization

ES6 should fix ensure tail calls does not generate stack overflow. (Not all implementations work).

```

function factorial(n, acc = 1) {
    if (n <= 1) return acc;
    return factorial(n - 1, n * acc);
}
console.log(factorial(10))
console.log(factorial(100))
console.log(factorial(1000))
console.log(factorial(10000))
console.log(factorial(100000))
console.log(factorial(1000000))

```

12 : "What's the `typeof` null?", and other confusing JavaScript Types

The `typeof` operator in JavaScript evaluates and returns a string with the data type of an operand. For example, to find the type of `123`, we would write -

```
typeof 123
```

This will return a string with the type of `123`, which, in this case, will be "number". In addition to "number", the `typeof` operator can return one of 6 potential results -

```
typeof 123 // "number"  
typeof "abc" // "string"  
typeof true // "boolean"  
typeof {a: 1} // "object"  
typeof function foo() {} // "function"  
typeof undefined // "undefined"  
typeof Symbol('foo') // "symbol"
```

In the above examples, it's pretty straightforward what the type of the operands will be. However, there are a few cases where it is unclear or misunderstood what the type of the operand should be.

What's the `typeof typeof 123`?

```
typeof typeof 123 // "string"
```

What is the type of a `typeof` expression? Well, the `typeof` operator always returns a string with the type of the operand passed to it. If the resultant type of the expression is, for example, a number, what will be returned is "number". This means that, regardless of resultant type, the type of a `typeof [any operand]`, will always be a string.

What's the `typeof NaN`?

```
typeof NaN // "number"
```

The type of `Nan`, which stands for Not a Number is, surprisingly, a number. The reason for this is, in computing, `Nan` is actually technically a numeric data type. However, it is a numeric data type whose value cannot be represented using actual numbers. So, the name "Not a Number", doesn't mean that the value is not numeric. It instead means that the value cannot be expressed with numbers.

This also explains why not all `Nan` values are equal. For example -

```
const NaN1 = 2 * "abc";
const NaN2 = 2 * "abc";

NaN1 === NaN2 // false
```

Those two `Nan` values are not equal because they are not necessarily the same unrepresentable number.

What's the `typeof [1,2,3]`?

```
typeof [1,2,3] // "object"
```

The `typeof` an array is an object. In JavaScript, arrays are technically objects; just with special behaviours and abilities. For example, arrays have a `Array.prototype.length` property, which will return the number of elements in the array. Arrays also have special methods, e.g. `Array.prototype.push()` or `Array.prototype.unshift()` (See [JavaScript Array Methods - Mutator Methods](#)).

To differentiate an Array object from an Object object, we can use the `Array.isArray()` method.

```
Array.isArray( [1,2,3] ) // true
Array.isArray( { a: 1 } ) // false
```

What's the `typeof null`?

```
typeof null // "object"
```

The `null` value is technically a primitive, the way "object" or "number" are primitives. This would typically mean that the type of null should also be "null". However, this is not the case because of a peculiarity with the way JavaScript was first defined.

In the first implementation of JavaScript, values were represented in two parts - a type tag and the actual value. There were 5 type tags that could be used, and the tag for referencing an object was `0`. The `null` value, however, was represented as the `NULL` pointer, which was `0x00` for most platforms. As a result of this similarity, null has the `0` type tag, which corresponds to an object.

What's the `typeof class Foo {}`?

```
typeof class Foo {} // "function"
```

Finally, we have Classes. Classes were introduced in ES2015 (ES6) as a better syntax for prototype-based inheritance. Before Classes, to make an inheritable object, we would have to use a function.

```
function Dog() { }
Dog.prototype.bark = function() {
```

```
    alert("woof!");  
}  
  
const snoopy = new Dog();  
snoopy.bark() // alert("woof!")
```

With Classes, we can create the same object this way -

```
class Dog {  
  bark() {  
    alert("woof!");  
  }  
}  
  
const snoopy = new Dog();  
snoopy.bark() // alert("woof!")
```

However, Classes have always just been a syntactical wrapper around the function method. The same function is actually being created, but just with the author writing it in a different, cleaner way. This is why the `typeof` a Class, is still just a Function.

13 : Asynchronous vs Deferred JavaScript

In my article on [Understanding the Critical Rendering Path](#), I wrote about the effect JavaScript files have on the Critical Rendering Path.

JavaScript is considered a "parser blocking resource". This means that the parsing of the HTML document itself is blocked by JavaScript. When the parser reaches a `<script>` tag, whether that be internal or external, it stops to fetch (if it is external) and run it.

This behaviour can be problematic if we are loading several JavaScript files on a page, as this will interfere with the time to first paint even if the document is not actually dependent on those files.

Fortunately, the `<script>` element has two attributes, `async` and `defer`, that can give us more control over how and when external files are fetched and executed.

Normal Execution

Before looking into the effect of the two attributes, we must first look at what occurs in their absence. By default, as mentioned above, JavaScript files will interrupt the parsing of the HTML document in order for them to be fetched (if not inline) and executed.

Take, for example, this script element located somewhere in the middle of the page -

```
<html>
<head> ... </head>
<body>
  ...
  <script src="script.js">
  ....
</body>
</html>
```

As the document parser goes through the page, this is what occurs -



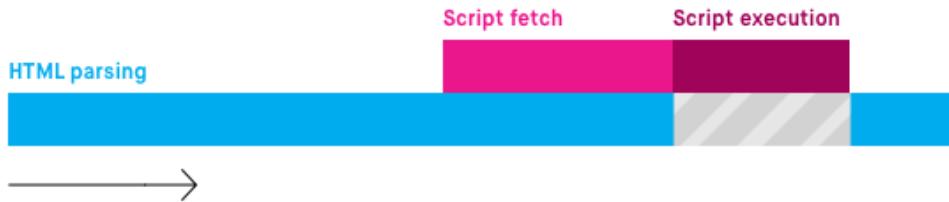
The HTML parsing is paused for the script to be fetched and executed, thereby extending the amount of time it takes to get to first paint.

The `async` Attribute

The `async` attribute is used to indicate to the browser that the script file *can* be executed asynchronously. The HTML parser does not need to pause at the point it reaches the script tag to fetch and execute, the execution can happen whenever the script becomes ready after being fetched in parallel with the document parsing.

```
<script async src="script.js">
```

This attribute is only available for externally located script files. When an external script has this attribute, the file can be downloaded while the HTML document is still parsing. Once it has been downloaded, the parsing is paused for the script to be executed.

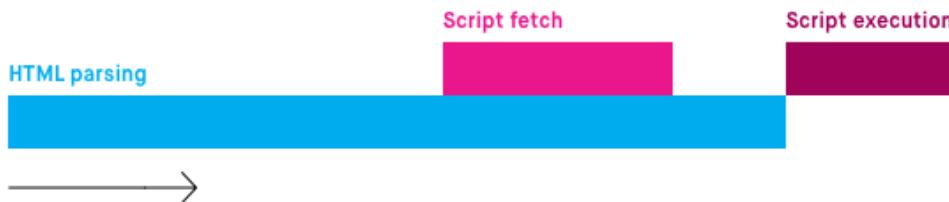


The `defer` Attribute

The `defer` attribute tells the browser to only execute the script file once the HTML document has been fully parsed.

```
<script defer src="script.js">
```

Like an asynchronously loaded script, the file can be downloaded while the HTML document is still parsing. However, even if the file is fully downloaded long before the document is finished parsing, the script is not executed until the parsing is complete.



Asynchronous, Deferred or Normal Execution?

So, when should we use asynchronous, deferred, or normal JavaScript execution? As always, it depends on the situation, and there are a few questions to consider.

Where is the `<script>` element located?

Asynchronous and deferred execution of scripts are more important when the `<script>` element is not located at the very end of the document. HTML documents are parsed in order, from the first opening `<html>` element to its close. If an externally sourced JavaScript file is placed right before the closing `</body>` element, it becomes much

less pertinent to use an `async` or `defer` attribute. Since the parser will have finished the vast majority of the document by that point, JavaScript files don't have much parsing left to block.

Is the script self-contained?

For script files that are not dependent on other files and/or do not have any dependencies themselves, the `async` attribute is particularly useful. Since we do not care exactly at which point the file is executed, asynchronous loading is the most suitable option.

Does the script rely on a fully parsed DOM?

In many cases, the script file contains functionality that requires interaction with the DOM. Or, it may have a dependency on another file included on the page. In these cases, the DOM must be fully parsed before the script should be executed. Typically, such a file will be placed at the bottom of the page to ensure everything before it has been parsed. However, in situation where, for whatever reason, the file in question needs to be placed elsewhere, the `defer` attribute can be used.

Is the script a (small) dependency?

Finally, if the script is relatively small, and/or is depended on by other files, it may be more useful to have it defined inline. Although having it inline will block the parsing of the HTML document, it should not be a significant interference if it's a small size. Additionally, if it is depended on by other files, the minor blocking may be necessary.

Support and Modern Browser Engines

It is worth noting that the behaviour of these attributes may be slightly different across different JavaScript engines. For example, in V8 (used in Chromium), an attempt is made to parse all scripts, regardless of their attributes, on a separate dedicated thread for script execution. This way, the "parser blocking" nature of JavaScript files should be minimised as a default.

14 : An Overview of Client-Side Storage

Storing data directly within the browser has a lot of benefits, the main one being quick and network-independent access to a "database". There are currently four active methods (plus one deprecated), for storing data on the client side -

1. Cookies
2. Local Storage
3. Session Storage
4. IndexedDB
5. WebSQL (deprecated)

Cookies

Cookies are the classic way of storing simple string data within a document. Typically, cookies are sent from the server to the client, which can then store it, and send it back to the server on subsequent requests. This can be used for things like managing account sessions and tracking user information.

Additionally, cookies can be used for storing data purely on the client-side. Because of this, they have also been used for storing general data, such as user preferences.

Basic CRUD with Cookies

We can create, read, update, and delete cookies using the following syntax -

```
// Create
document.cookie = "user_name=Ire Aderinokun";
document.cookie = "user_age=25;max-age=31536000;secure";

// Read (All)
console.log( document.cookie );

// Update
document.cookie = "user_age=24;max-age=31536000;secure";

// Delete
document.cookie = "user_name=Ire Aderinokun;expires=Thu, 01 Jan 1970 00:00:01 GMT";
```

Advantages of Cookies

- They can be used for communication with the server
- We can set when we want the cookie to expire automatically, instead of having to manually delete

Disadvantages of Cookies

- They add to the page load of the document
- They can only store a small amount of data
- They can only store Strings
- Potential [security issues](#)
- It is not the recommended method for client-side storage anymore since the introduction of the [Web Storage API](#) (Local and Session Storage)

Support

Cookies have basic support in all major browsers.

Local Storage

Local Storage is one type of the [Web Storage API](#), which is an API for storing key-value pairs of data within the browser. It arose as a solution to the issues with Cookies, by offering a more intuitive and secure API for storing simple data within the browser.

Although technically we can only store strings in Local Storage, this can be worked around by storing stringified JSON. This allows us to store a bit more complex data in Local Storage than we can with Cookies.

Basic CRUD with Local Storage

We can create, read, update, and delete data to Local Storage using the following syntax -

```
// Create
const user = { name: 'Ire Aderinokun', age: 25 }
localStorage.setItem('user', JSON.stringify(user));

// Read (Single)
console.log( JSON.parse(localStorage.getItem('user')) )

// Update
const updatedUser = { name: 'Ire Aderinokun', age: 24 }
localStorage.setItem('user', JSON.stringify(updatedUser));

// Delete
localStorage.removeItem('user');
```

Advantages of Local Storage

- Offers a more simple intuitive interface to storing data (than Cookies)
- More secure for client-side storage (than Cookies)
- Allows for the storage of more data (than Cookies)

Disadvantages of Local Storage

- Only allows the storage of Strings

Support

Session Storage

Session Storage is the second type of the [Web Storage API](#). It is exactly the same as Local Storage, except that the data is only stored for the browser tab session. Once the user closes that browser tab, the data is cleared.

Basic CRUD with Session Storage

We can create, read, update, and delete data to Session Storage using the following syntax -

```
// Create
const user = { name: 'Ire Aderinokun', age: 25 }
sessionStorage.setItem('user', JSON.stringify(user));

// Read (Single)
console.log( JSON.parse(sessionStorage.getItem('user')) )

// Update
const updatedUser = { name: 'Ire Aderinokun', age: 24 }
sessionStorage.setItem('user', JSON.stringify(updatedUser));

// Delete
sessionStorage.removeItem('user');
```

Advantages, Disadvantages, and Support for Session Storage

Same as for Local Storage.

IndexedDB

IndexedDB is a much more complex and well-rounded solution for storing data in the browser. It is a "low-level API for client-side storage of significant amounts of structured data" (Mozilla). It is a JavaScript-based, object-oriented, database that allows us to easily store and retrieve data that has been indexed with a key.

In my article on [Building a Progressive Web Application](#), I went over in more detail how you can use IndexedDB to create an offline-first application.

Basic CRUD with IndexedDB

Note: In all my examples, I use Jake Archibald's [IndexedDB Promised library](#) which offers a Promise-ified version of the IndexedDB methods.

Using IndexedDB is more complicated than the other browser storage methods. Before we can create/read/update/delete any data, we need to first open up the database, creating any stores (which are like

tables in a database) we need.

```
function OpenIDB() {
    return idb.open('SampleDB', 1, function(upgradeDb) {
        const users = upgradeDb.createObjectStore('users', {
            keyPath: 'name'
        });
    });
}
```

To create (or update) data within a store, we need to go through the following steps -

```
// 1. Open up the database
OpenIDB().then((db) => {
    const dbStore = 'users';

    // 2. Open a new read/write transaction with the store within the database
    const transaction = db.transaction(dbStore, 'readwrite');
    const store = transaction.objectStore(dbStore);

    // 3. Add the data to the store
    store.put({
        name: 'Ire Aderinokun',
        age: 25
    });

    // 4. Complete the transaction
    return transaction.complete;
});
```

To retrieve data, we need to go through the following -

```
// 1. Open up the database
OpenIDB().then((db) => {
    const dbStore = 'users';

    // 2. Open a new read-only transaction with the store within the database
    const transaction = db.transaction(dbStore);
    const store = transaction.objectStore(dbStore);

    // 3. Return the data
    return store.get('Ire Aderinokun');
}).then((item) => {
```

```
    console.log(item);
})
```

Finally, to delete data, we need to do the following -

```
// 1. Open up the database
OpenIDB().then((db) => {
  const dbStore = 'users';

  // 2. Open a new read/write transaction with the store within the database
  const transaction = db.transaction(dbStore, 'readwrite');
  const store = transaction.objectStore(dbStore);

  // 3. Delete the data corresponding to the passed key
  store.delete('Ire Aderinokun');

  // 4. Complete the transaction
  return transaction.complete;
})
```

If you're interested in learning more about how to use IndexedDB, you can read my [article](#) showing how I used it in a Progressive Web Application.

Advantages of IndexedDB

- Can handle more complex, structured, data
- Can have multiple "databases" and "tables" within each "database"
- More allowance for storage
- More control over how we interact with it

Disadvantages of IndexedDB

- More complex to use than the Web Storage API

Support

WebSQL

WebSQL is an API for a relational database on the client, similar to SQLite. Since 2010, the W3C Web Applications Working Group has ceased working on the specification. **It is no longer a part of HTML specification, and should not be used.**

A Comparison

Feature	Cookies	Local Storage	Session Storage	IndexedDB
---------	---------	---------------	-----------------	-----------

Storage Limit	<u>~4KB</u>	<u>~5MB</u>	<u>~5MB</u>	<u>Up to half of hard drive</u>
Persistent Data?	Yes	Yes	No	Yes
Data Value Type	String	String	String	Any structured data
Indexable?	No	No	No	Yes

15 : Back to JavaScript Basics: VARIABLES!

So, let's talk about variables. First, a definition:

Variable: A label that points to a value that is stored in memory. Once a variable is assigned, you can refer to it later in your code.

OK, that's nice, you might think. But what's the big? There are TWO big reasons why we care about this property of variables that I'll go over real quick.

Building efficient programs

```
var storedNum = (1+1)*2;
console.log(result); // 4
console.log(result); // 4
```

In this example, the variable `storedNum` is computed only once, and the computed result is logged twice. If you ran `console.log((1+1)*2)`; each time you needed to access the stored number, the computer would have to compute the result each time. Maybe that's fine for this super simple example, but you can imagine how this would be meaningfully inefficient if you had a complex computation that required a non-trivial amount of memory. What else should you know about variables? You can use them to point to the values stored in other variables!

```
var storedNum = (1+1)*2;
console.log(storedNum); // 4
var other = storedNum;
storedNum = 5;
console.log(other); // 4
```

If you're wondering why `other` still evaluates to 4, even after `storedNum` is set to 5, it might help to think of variables as physical labels, or nametags.

Let's say you have a "cookie" nametag on a box of Oreos, and then you stick a "milk" nametag on the same box. Now imagine you peel off the "cookie" nametag and put it on a box of Teddy Grahams. What happens to the "milk" nametag? It's still on the box of Oreos. Just because you moved the "cookie" nametag to the Teddy Grahams box, it doesn't mean that the "milk" nametag moves to the Teddy Grahams. The "milk" nametag stays on the Oreos until it's moved and affixed to something else.

Man. Now I really want Oreos.

Here's another way to think about variables pointing to other variables:

```
var storedNum = (1+1)*2;
console.log(storedNum); // 4
var other = storedNum + 0;
```

```
storedNum = 5;  
console.log(other); // 4
```

One helpful trick is to assign `other` a value of `storedNum + 0`. This is the exact same thing as before, but makes it clearer that `other` is independent of the `storedNum` variable.

** But, seriously, why do we care? **

Accessing certain values

The second major reason that we care about variables is because sometimes it's not possible to type out a literal that represents a value in your program. For example, let's say you need a random number:

```
var randomNum = Math.random();  
console.log(randomNum);
```

Since you don't know what the random number will be, you have to store the value in a variable if you want to access that random number value later.

16 : Quick Tip: How to Sort an Array of Objects in JavaScript

If you have an array of objects that you need to sort into a certain order, the temptation might be to reach for a JavaScript library. Before you do however, remember that you can do some pretty neat sorting with the native [Array.sort](#) function. In this article I'll show you how to sort an array of objects in JavaScript with no fuss or bother.

To follow along with this article, you will need a knowledge of basic JavaScript concepts, such as declaring variables, writing functions, and conditional statements. I'll also be using ES6 syntax. You can get a refresher on that here: <https://www.sitepoint.com/tag/es6/>

Basic Array Sorting

By default, the JavaScript `Array.sort` function converts each element in the array to be sorted, into a string, and compares them in [Unicode code point](#) order.

```
const foo = [9, 2, 3, 'random', 'panda'];
foo.sort(); // returns [ 2, 3, 9, 'panda', 'random' ]  
  
const bar = [4, 19, 30, function(){}, {key: 'value'}];
bar.sort(); // returns [ 19, 30, 4, { key: 'value' }, [Function] ]
```

You may be wondering why 30 comes before 4... not logical huh? Well, actually it is. This happens because each element in the array is first converted to a string, and "30" comes before "4" in Unicode order.

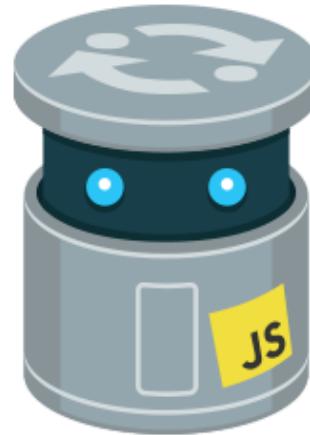
It is also worth noting that unlike many other JavaScript array functions, `Array.sort` actually changes, or mutates the array it sorts.

```
const baz = ['hello world', 31, 5, 9, 12];
baz.sort(); // baz array is modified
console.log(baz); // shows [12, 31, 5, 9, "hello world"]
```

To avoid this, you can create a new instance of the array to be sorted and modify that instead.

```
const baz = ['hello world', 31, 5, 9, 12];
const newBaz = baz.slice().sort(); // new instance of baz array is created and sorted
console.log(baz); // "hello world", 31, 5, 9, 12]
console.log(newBaz); // [12, 31, 5, 9, "hello world"]
```

Try it out



Using `Array.sort` alone would not be very useful for sorting an array of objects, thankfully the function takes an optional `compareFunction` parameter which causes the array elements to be sorted according to the return value of the compare function.

Using Compare Functions to Sort

Lets say that `a` and `b` are the two elements being compared by the compare function. If the return value of the compare function is:

1. less than 0 — `a` comes before `b`
2. greater than 0 — `b` comes before `a`
3. equal to 0 — `a` and `b` are left unchanged with respect to each other

Let's look at a simple example with an array of numbers:

```
const arr = [1,2,30,4];

function compare(a, b){
    let comparison = 0;

    if (a > b) {
        comparison = 1;
    } else if (b > a) {
        comparison = -1;
    }

    return comparison;
}

arr.sort(compare);
// => 1, 2, 4, 30
```

This can be refactored somewhat to obtain the return value by subtracting `a` from `b`:

```
function compare(a, b){
    return a - b;
}
```

Which now makes a good candidate for an arrow function:

```
arr.sort((a, b) => a - b);
```

Sort an Array of Objects in JavaScript

Now let's look at sorting an array of objects. Let's take an array of band objects:

```
const bands = [
    { genre: 'Rap', band: 'Migos', albums: 2},
    { genre: 'Pop', band: 'Coldplay', albums: 4},
    { genre: 'Rock', band: 'Breaking Benjamins', albums: 1}
];
```

We can use the following compare function to sort this array of objects according to genre:

```
function compare(a, b) {
    // Use toUpperCase() to ignore character casing
    const genreA = a.genre.toUpperCase();
    const genreB = b.genre.toUpperCase();

    let comparison = 0;
    if (genreA > genreB) {
        comparison = 1;
    } else if (genreA < genreB) {
        comparison = -1;
    }
    return comparison;
}

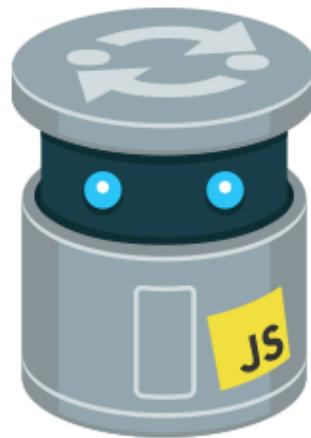
bands.sort(compare);

/* returns [
{ genre: 'Pop', band: 'Coldplay', albums: 4 },
{ genre: 'Rap', band: 'Migos', albums: 2 },
{ genre: 'Rock', band: 'Breaking Benjamins', albums: 1 }
] */
```

To reverse the sorting order, you can simply invert the return value of the compare function:

```
function compare(a, b) {  
  ...  
  
  //invert return value by multiplying by -1  
  return comparison * -1;  
}
```

Try it out



Creating a Dynamic Sorting Function

Let's finish up by making this more dynamic. Let's create a sorting function, which you can use to sort an array of objects, whose values are either strings or numbers. This function has two parameters — the key we want to sort by and the order of the results (i.e. ascending or descending).

```
const bands = [  
  { genre: 'Rap', band: 'Migos', albums: 2},  
  { genre: 'Pop', band: 'Coldplay', albums: 4, awards: 13},  
  { genre: 'Rock', band: 'Breaking Benjamins', albums: 1}  
];  
  
// function for dynamic sorting
```

```
function compareValues(key, order='asc') {
  return function(a, b) {
    if(!a.hasOwnProperty(key) || !b.hasOwnProperty(key)) {
      // property doesn't exist on either object
      return 0;
    }

    const varA = (typeof a[key] === 'string') ?
      a[key].toUpperCase() : a[key];
    const varB = (typeof b[key] === 'string') ?
      b[key].toUpperCase() : b[key];

    let comparison = 0;
    if (varA > varB) {
      comparison = 1;
    } else if (varA < varB) {
      comparison = -1;
    }
    return (
      (order === 'desc') ? (comparison * -1) : comparison
    );
  };
}
```

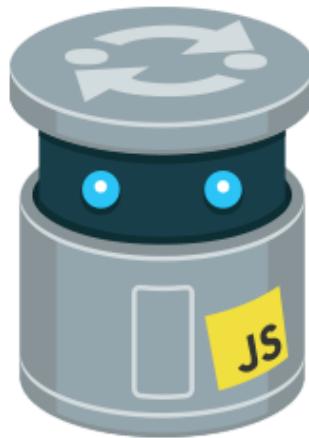
And this is how you'd use it:

```
// array is sorted by band, in ascending order by default
bands.sort(compareValues('band'));

// array is sorted by band in descending order
bands.sort(compareValues('band', 'desc'));

// array is sorted by albums in ascending order
bands.sort(compareValues('albums'));
```

Try it out



In the code above, the [hasOwnProperty method](#) is used to check if the specified property is defined on each object and has not been [inherited via the prototype chain](#). If it is not defined on the objects, the function returns `0`, which causes the sort order to remain as is (i.e the objects remain unchanged with respect to each other).

The [typeof operator](#) is also used to check the data type of the value of the property. This allows the function to determine the proper way to sort the array. For example, if the value of the specified property is a `string`, a `toUpperCase` method is used to convert all its characters to uppercase, so character casing is ignored when sorting.

You can adjust the above function to accommodate other data types, and any other peculiarity your script needs.

Conclusion

So there you have it — a short introduction to sorting an array of objects. Although many JavaScript libraries offer this kind of dynamic sorting ability (e.g. [Underscore.js](#), [Lodash](#) and [Sugar](#)), as demonstrated, it's not all that hard to implement this kind of functionality yourself.

If you have any questions or comments, I'd be glad to hear them in the comments.

This article was peer reviewed by [Moritz Kröger](#), [Giulio Mainardi](#), [Vildan Softic](#) and [Rob Simpson](#). Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

17 : Effective Functional JavaScript: First-class and Higher Order Functions

<https://hackernoon.com/effective-functional-javascript-first-class-and-higher-order-functions-713fde8df50a>

Functions: the killer JavaScript feature we never talk about.

JavaScript is a very in-demand language today. It runs in a lot of places from the browser to embedded systems and it brings a non-blocking I/O model that is faster than others for some types of applications. What really sets JavaScript apart from the rest of scripting languages is its highly functional nature.

JavaScript has more in common with functional languages like [Lisp or Scheme](#) than with C or Java.—

Douglas Crockford in [JavaScript: the World's Most Misunderstood Language](#)

It's a widely publicised fact that Brendan Eich, when creating the first version of JavaScript, was actually told he would be able to build a [Lisp, a list processing language like Scheme or common Lisp](#). It turns out that Sun Microsystems had other ideas and this programming language needed to wear a coat of procedural syntax inherited from a C or Java-like language. Despite all the semicolons, at its heart JavaScript remains a functional language with features such as first-class functions and closures.

This post will focus on first-class functions and higher order functions. To read more about closures and how to leverage them to wield JavaScript's functional powers, I've written the [following post](#):

[Ingredients of Effective Functional JavaScript: Closures, Partial Application and Currying](#)

To use JavaScript to its full potential you have to embrace its strong functional programming

base.hackernoon.com

First-class functions

A language with first-class functions means that it treats functions like expressions of any other type. Functions are like any other object.

You can pass them into other functions as parameters:

```
function runFunction(fn, data) {
  return fn(data);
}
```

You can assign a function as a value to a variable:

```
var myFunc = function() {
  // do something
};
```

```
const myNewGenFunc = someParam => someParam;
```

You can return a function:

```
function takeFirst(f, g) {
  return f;
}
```

First-class functions in practice

Cheap dependency injection

In languages without first-class functions to pass a dependency in means passing an object. Here we can just pass functions around, which is great.

For example in Java to run a custom sort on an ArrayList, we have to use `ArrayList#sort` which expects a `Comparator` object as a parameter (see [the Java API docs here](#)). In JavaScript we use `Array#sort` ([MDN reference here](#)) which expects a function. This is a bit less verbose since in this case we're just going to be implementing one method of the `Comparator` interface

It gets even better with ES6 default parameter syntax.

```
// in fetch.js
import axios from 'axios';

export function fetchSomething(fetch = axios) {
  return fetch('/get/some/resource');
}
```

```
// in someModule.js
import axios from 'axios';
```

```
import { fetchSomething } from './fetch';
```

```
const fetcherWithHeaders = axios({
  // pass some custom configs
});
```

```
fetchSomething(fetcherWithHeaders)
  .then(/* do something */)
```

```
.catch(/* handle error */);

// in someOtherModule.js

import { fetchSomething } from './fetch';

fetchSomething()
.then(/* do something */)
.catch(/* handle error */);
```

Callbacks and non-blocking IO

JavaScript's default behaviour on IO is non-blocking. This means we tend to pass a lot of callbacks (until Promises came along at least). Being able to pass a function as a callback instead of an object on which we will run a method (like we would in say Java) means we can have terseness in callback-based code.

For example in Node:

```
const fs = require('fs');

fs.readFile('./myFile.txt', 'utf-8', function(err, data) {
  // this is a callback, it gets executed
  // once the file has been read
});
```

More functional programming primitives that require first-class functions

Being able to return a function and closures means we have access to things like partial application and currying.

[Read more about those FP superpowers here.](#)

It also means we can start creating higher order functions.

Higher order functions

A function is a higher order function if it takes a function as a parameter, or returns a function as its result. Both of these requirements rely on functions being first-class objects in a language.

`map`, `filter` and `reduce / reduceRight` are the functions present in JavaScript that map to classic higher order functions in other functional languages.

In other languages:

- `map` tends to be called `map` or `transform`

- filter is called select in some languages
- reduce and reduceRight are the fold left and right functions (also called accumulate, aggregate, compress or inject in different languages)

In functional programming languages, there are no loops. When you need to do an operation like traversing a list or a tree, there are two predominant styles: recursion and higher order functions.

Recursion relies on a function calling itself, usually on a different part of the parameter (list being traversed for example). That's a topic for [another post](#).

Higher order functions are usually provided by the language. In ES6 JavaScript these functions are defined on the Array prototype.

Array#map

`map` is used if we want to perform the same change on each member of the array. It takes the function that should be applied to each element of the array as a parameter. That function is passed `(element, index, wholeArray)` as parameters.

```
const myArr = [ 'some text', 'more text', 'final text' ];

const mappedArr = myArr.map( function(str) {
  return str.split(' ');
});

console.log(mappedArr);
// [ [ 'some', 'text' ], [ 'more', 'text' ], [ 'final', 'text' ] ]
```

Array#filter

`filter` allows us to pick which elements of the array should remain in the transformed list by passing a filtering function that returns a Boolean value (true/false). As for `map` this function is passed `(element, index, wholeArray)`.

```
const myArr = [ 5, 10, 4, 50, 3 ];

const multiplesOfFive = myArr.filter( function(num) {
  return num % 5 === 0;
});

console.log(multiplesOfFive);
// [ 5, 10, 50 ]
```

Array#reduce

`reduce` is used to change the shape of the array. We provide more than 1 parameter to this function (in addition to the array we're reducing). We pass a reducing function and optionally the initial value of to reduce with. The function is passed `(prev, curr, index, wholeArray)`. `prev` is the value returned by the previous reduction, for the first iteration that means it's either the initial value of the first element of the array. `curr` is the value in the array at which we're at.

The classic example is summing or concatenating.

```
const myNumbers = [ 1, 2, 5 ];
const myWords = [ 'These', 'all', 'form', 'a', 'sentence' ];
```

```
const sum = myNumbers.reduce( function(prev, curr) {
  return prev + curr;
}, 0);
```

```
console.log(sum); // 8
```

```
const sentence = myWords.reduce( (prev, curr) => {
  return prev + ' ' + curr;
}); // the initial value is optional
```

```
console.log(sentence);
// 'These all form a sentence'
```

If you want to learn more about the internal of `map`, `filter` and `reduce` or recursion, I've reimplemented them in a recursive style using ES6 destructuring in the following post:

[Recursion in JavaScript with ES6, destructuring and rest/spread](#)

The latest ECMA standard for JavaScript (ECMAScript 6) makes JavaScript more readable by encouraging a more declarative...hackernoon.com

Higher order functions in practice

No more loops

Higher order functions allow us to get rid of the imperative loops that seem to be spread everywhere.

```
var newArr = [];
var myArr = [ 1, 2, 3 ];
```

```

for(var i = 0; i < myArr.length; i++) {
  newArr.push(myArr[i] * 2);
}

```

```
console.log(newArr); // [ 2, 4, 6 ]
```

```

// nicer with `map`
const doubled = myArr.map( x => x * 2 );
console.log(doubled); // [ 2, 4, 6 ]

```

The intent is just clearer with map. We can also extract the `double` function so we're making our code more readable and modular.

```

const double = x => x * 2;
const doubled = arr.map(double);

```

It reads like a book and that's important because we write for humans not machines.

Side-effect free programming

`Array` higher order functions do not mutate the variable they are called on. This is good, because the loop-based approach using `.push` and `.pop` changes it. It means if you pass a variable as a parameter, it's not suddenly going to get changed by a function down the call stack.

```

// some random module
// obviously no one actually does this
function doesBadSideEffect(arr) {
  return arr.pop();
}

```

```

// somewhere quite important
var arr = [ 'a', 'b', 'c' ];

```

```
var joinedLetters = '';
```

```

for(var i = 0; i < arr.length; i++) {
  joinedLetters += arr[i];
}

```

```
  doesBadSideEffect(arr)
}
```

```
console.log(joinedLetters);
// whoops 'ab'
// even though the naming makes us
// expect 'abc'
```

Declarative Code Optimisations

In some languages functions like `map` are parallelised. That's because we don't actually need to know what's in the rest of the array to compute this particular element's new value. If we're doing complex things in the mapping function then this sort of optimisation could be very useful.

Effective Functional JavaScript

Use first-class and higher order functions to write nicer code more easily. It's nice and declarative instead of imperative, say what you want done not how to do it

This will enable you to compose your functions and write code that is extremely terse without losing readability.

Remember to give this post some  if you liked it. Follow me [Hugo Di Francesco](#) or [@hugo_df](#) for more JavaScript content :).

Edit: 24/01/2017, rephrased "Callbacks and non-blocking IO" following [Anton Alexandrenok](#)'s comments

18 : Ingredients of Effective Functional JavaScript: Closures, Partial Application and Currying

19 : Ingredients of Effective Functional JavaScript: Closures, Partial Application and Currying

To use JavaScript to its full potential you have to embrace its strong functional programming base. We're going to explore some crucial and powerful functional constructs: closures, partial application and currying that make JavaScript terse yet understandable.

The basics

[Functional programming](#) is a programming paradigm that follows a more mathematical computation model. Let's go through some basics to make your JavaScript more functional.

Declarative programming

Functional programs tend to be declarative (as opposed to imperative), that's a case of telling the compiler *what* you want instead of *how* you want it.

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

```
// imperative style
const imperative = list[0];
```

```
// declarative style
const declarative = firstElement(list);
```

In the above code snippet we're trying to get the first element of `list`, the declarative example says what we want, `firstElement` can do whatever it likes as long as it returns the first element of the passed parameter. Whereas in the imperative style, we say I want index 0 of `list` explicitly. In JavaScript and at this program size, this doesn't make a massive difference.

To build functional programs, we should prefer the declarative style and avoid mutation.

Recursion and higher order functions

There are no loops in functional programming, just recursion and higher order functions.

Mechanisms such as [pattern matching](#) allow for easier recursive function declarations. In ECMAScript 6 (the 2015 edition of the standard JavaScript is based on) we've added destructuring to the toolbox, which is a basic pattern matching that works for lists. You can read more about it [here](#).

Recursion in JavaScript with ES6, destructuring and rest/spread

The latest ECMA standard for JavaScript (ECMAScript 6) makes JavaScript more readable by encouraging a more declarative...hackernoon.com

Higher order functions allow you to traverse iterable collections (Arrays). In JavaScript we have Array#map, Array#filter and Array#reduce. Each of these takes a function as an argument. This is possible because we have first-class functions in JavaScript, which means you can pass them around like any other type of variable :).

Lambdas (anonymous functions)

In JavaScript we can declare lambdas (anonymous functions), which is quite handy considering the API of a lot of libraries expects a function as a parameter. We can just declare the function inline. It might mean a bit of a indentation/bracketing problem but inlining until you can generalise or refactor is actually great.

```
[ 1, 2, 3 ].map(function(el) {
    return el * 2;
});
```

Closures

Here are some >140 character explanations of closures, thanks to [Mateusz Zatorski](#) for asking and his esteemed followers for answering :).

We can use closures to put state inside an outer function while having access to that state in an inner function. That state is not global but still accessible to child functions.

```
function outerFunction() {
    const someState = { count: 0 };
    return {
        increment: function() {
            someState.count++;
        },
        getCount: function() {
            return someState.count;
        }
    }
}
```

```
const counter = outerFunction();
```

```
counter.increment();
counter.increment();
counter.getCount(); // 2
counter.increment();
counter.increment();
counter.getCount(); // 4
```

```
someState; // ReferenceError: someState is not defined
```

The state (`someState`) isn't global since the last statement returns an error. It is however available to the functions it returned, because they can "see" `someState`, it's in their lexical scope.

Function application

Function application is the first "hardcore" functional programming concept we're going to introduce today. It's a concept that comes from the mathematics world.

A function application, in JavaScript, can look like a function call but doesn't have to be.

```
function someFunc(arg1, arg2) {
  return arg1 + arg2;
}
```

```
// function call
someFunc(1, 2); // 3
```

```
// different ways to apply a function
someFunc(1, 2); // 3
someFunc.apply(this, [ 1, 2 ]); // 3
someFunc.call(this, 1, 2); // 3
```

A function call is an imperative construct whereas a function application belongs to the realm of functional programming and mathematics.

In JavaScript you can even use `apply` and `call` to define what `this` will be set to during the application.

Partial application

Partial application is when you apply some of the required parameters of a function and return a function that takes the rest of the parameters.

We're going to flip the parameters to the `map` function. Instead of taking parameters `(list, fn)` it's going to take `(fn, list)`. This is to illustrate the value of partial application.

```
function functionalMap(fn, list) {
  return list.map(fn);
}
```

```
function partialFunctionalMap(fn) {
  return function(list) {
    return functionalMap(fn, list);
  }
}
```

```
// Example 1
// Let's apply all the arguments at once
functionalMap(x => x * 2, [ 1, 2, 3 ]);
functionalMap(x => x * 2, [ 2, 3, 5 ]);
```

```
// Example 2
// Let's apply them one at a time
const doubleListItems = partialFunctionalMap(x => x * 2);
doubleListItems([ 1, 2, 3 ]);
doubleListItems([ 2, 3, 5 ]);
```

What the code does in example 1 is less obvious than in example 2. You have to read what the lambda does instead of being told by the variable function name.

This is something we can use in places like React event handlers.

Now if we went the Object-oriented route, we would use `.bind` `partialHandleClick` function to the component instance (`this`) and to be able to access `this.props.activeType` from inside `partialHandleClick`. We're trying to leverage functional programming, so no accessing `this` from all the way inside an event handler. We get to store some information that we can get at `.map` time (which type is this handler for). When the event triggers, we get the final parameter we need `e` (the event object) and the handler can finish applying.

Currying

A curried function is a function that you apply 1 parameter at a time.

```
function partialFunctionalMap(fn) {
  return function(list) {
    return functionalMap(fn, list);
  }
}
```

`partialFunctionalMap` is curried. In the event handler example `partialHandleLinkClick` isn't, since the first application provided 2 parameters.

We could rewrite it though.

```

function curriedHandleLinkClick(type){
  return function(activeType) {
    return function(e) {
      const hasKeyboardModifier = e.ctrlKey || e.shiftKey || e.altKey || e.metaKey;
      updateType(type, activeType, hasKeyboardModifier);
    };
  };
}

```

And we would use `this.curriedHandleLinkClick(type)(this.props.activeType)` instead of `this.partialHandleLinkClick(type, this.props.activeType)`.

This isn't as pretty in JavaScript as in other languages since we're replacing `(arg1, arg2, arg3)` with `(arg1)(arg2)(arg3)`.

Currying and partial application

Currying is strict: a curried function **always** applied 1 parameter at a time. Partial application is not this strict.

A curried function tends to be partially applied but a partially applied function does **not** have to be curried.

This means we can automate the currying process.

In JavaScript we can use libraries to curry functions with multiple arguments. [Lodash has a curry function](#) and [so does Ramda](#). They take a function and when applied with a parameter either returns if all required arguments are present or returns a curried function that accepts the rest of the arguments.

You can also write your own by accessing the `arguments` object of the function and using `Function#apply`. Here are a couple of tutorials that take you through this process.

[Currying in JavaScript](#)

[A technique using partial evaluation](#)medium.com

[Currying in JavaScript](#)

[I've been thinking a lot lately about functional programming, and I thought it might be kind of fun to walk through the...](#)kevvin.in

Some languages like Haskell are auto-curried. This means that if the function application does not provide the required number of parameters, it will return a function which will accept the rest of the parameters one at a time, just like the [Lodash](#) and [Ramda](#) curry functions do. Another cool thing in Haskell is that partial application looks like non-partial application and curried function calls aren't ugly, since the separator for parameters is a space.

```
times a b = a * b
```

```
times 1 2
```

```
double = times 2
```

Effective Functional JavaScript Recipe

Use and abuse closures, partial application and currying.

This will enable you to compose your functions and write code that is extremely terse without losing readability.

Remember to give this post some  if you liked it. Follow me [Hugo Di Francesco](#) or [@hugo_df](#) for more JavaScript content :).

20 : Pub-Sub Event Manager

```
/*
 * MicroEvent - to make any js object an event emitter (server or browser)
 *
 * - pure javascript - server compatible, browser compatible
 * - dont rely on the browser doms
 * - super simple - you get it immediatly, no mistery, no magic involved
 *
 * - create a MicroEventDebug with goodies to debug
 *   - make it safer to use
 */

var MicroEvent = function(){};
MicroEvent.prototype = {
    bind : function(event, fct){
        this._events = this._events || {};
        this._events[event] = this._events[event] || [];
        this._events[event].push(fct);
    },
    unbind : function(event, fct){
        this._events = this._events || {};
        if( event in this._events === false ) return;
        this._events[event].splice(this._events[event].indexOf(fct), 1);
    },
    trigger : function(event /* , args... */){
        this._events = this._events || {};
        if( event in this._events === false ) return;
        for(var i = 0; i < this._events[event].length; i++){
            this._events[event][i].apply(this,
                Array.prototype.slice.call(arguments, 1));
        }
    }
};

/** 
 * mixin will delegate all MicroEvent.js function in the destination object
 *
 * - require('MicroEvent').mixin(Foobar) will make Foobar able to use MicroEvent
 *
 * @param {Object} the object which will support MicroEvent
*/

```

```
MicroEvent.mixin      = function(destObject){  
    var props      = ['bind', 'unbind', 'trigger'];  
    for(var i = 0; i < props.length; i ++){  
        if( typeof destObject === 'function' ){  
            destObject.prototype[props[i]]  =  
                MicroEvent.prototype[props[i]];  
        }else{  
            destObject[props[i]] = MicroEvent.prototype[props[i]];  
        }  
    }  
    return destObject;  
}  
  
// export in common js  
if( typeof module !== "undefined" && ('exports' in module)){  
    module.exports  = MicroEvent;  
}
```

21 : Currying in JavaScript

A technique using partial evaluation

Currying refers to the process of transforming a function with multiple arity into the same function with less arity.

The curried effect is achieved by binding some of the arguments to the first function invoke, so that those values are fixed for the next invocation. Here's an example of what a curried function looks like:

```
var babyAnimals = function(a) {
  return function(b) {
    var result = 'i love '.concat(a).concat(' and ').concat(b);

    return result;
  };
};

var babyKoala = babyAnimals('koalas');

babyKoala('elephants'); // 'i love koalas and elephants'
babyKoala('flamingos'); // 'i love koalas and flamingos'
babyKoala('axolotls'); // 'i love koalas and axolotls'
```

babyAnimals is a curried function; it is designed for the first argument to be 'prefilled' before the function itself is fully executed. With this pattern, 'koala' can be bound to babyAnimals, and my love for animals other than elephants can easily be expressed.

Currying can be integrated with callbacks to create higher-order 'factory' functions. This pattern is extremely useful in event handling, and can also replace the callback pattern that is utilized in node.js (Brun Jouhie has an excellent [blog post](#) on this pattern). Here is an example of combining currying with node.js to process a file:

```
var read = curriedReadFile(path);

read(function(err, data) {
  if (err) {
    throw err;
  }

  // Do something with data
});
```

At first it may not seem like much happened here, but this pattern is actually quite powerful. Integrating node with curried functions allows for the read data to be passed around to other bits of code as the file is getting processed. We can defer invoking the read function's callback until the result is needed. Currying node.js functions can allow for sequential and parallel I/O processing of multiple files, much like the `async` library for node.js.

```
var readA = curriedReadFile(pathA);
var readB = curriedReadFile(pathB);

readA(function(err, dataA) {
  readB(function(err, dataB) {
    if (err) {
      throw err;
    }

    // Do something with datas
  });
});
```

Currying is not a pattern that is native to javascript, so it is often handy to write a (currier) utility function that can transform any given function into a curried version of itself.

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

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

Now we can apply currying to any function by passing the function as the first argument to currier.

```
var sequence = function(start, end) {
  var result = [];

  for (var i = start; i <= end; i++) {
    result.push(i);
  }

  return result;
};

var seq5 = currier(sequence, 1);
seq5(5); // [1, 2, 3, 4, 5];
```

And that's it for currying. I hope you have learned something new about this powerful technique! Do you have an example of currying that you'd like to share? If so, please leave it as a gist in the comments.

```
var byebye = function(str1, str2) {
  return str1.concat(' ').concat(str2);
};

var buhbye = currier(byebye, 'That\'s all folks!');

buhbye('Until next time! :)); // 'That's all folks! Until next time :)'
```

22 : Currying the callback, or the essence of futures...

Wow, this one is not for dummies!!!

I decided to play it pedantic this time because I posted a layman explanation of this thing on the node.js forum and nobody reacted. Maybe it will get more attention if I use powerful words like *currying* in the title. We'll see...

Currying

I'll start with a quick explanation of what *currying* means so that Javascript programmers who don't know it can catch up. After all, it is a bit of a scary word for something rather simple, and many Javascript programmers have probably already eaten the curry some day without knowing what it was.

The idea behind currying is to take a function like

```
function multiply(x, y) { return x * y; }
```

and derive the following function from it:

```
function curriedMultiply(x) {
  return function(y) { return x * y; }
}
```

This function does something simple: it returns specialized multiplier functions. For example, `curriedMultiply(3)` is nothing else than a function which multiplies by 3:

```
function(y) {
  return 3 * y;
}
```

Attention: `curriedMultiply` does not multiply because it does not return numbers. Instead, it returns functions that multiply.

It is also interesting to note that `multiply(x, y)` is equivalent to `curriedMultiply(x)(y)`.

Currying the callback

Now, what happens if we apply this *currying* principle to node APIs, to single out the callback parameter?

For example, by applying it to node's `fs.readFile(path, encoding, callback)` function, we obtain a function like the following:

```
fs.curriedReadFile(path, encoding)
```

The same way our `curriedMultiply` gave us specialized *multiplier* functions, `curriedReadFile` gives us specialized *reader* functions. For example, if we write:

```
var reader = fs.curriedReadFile("hello.txt", "utf8");
```

we get a specialized reader function that only knows how to read `hello.txt`. This function is an asynchronous function with a single callback parameter. You would call it as follows to obtain the contents of the file:

```
reader(function(err, data) {
  // do something with data
});
```

Of course, we have the same equivalence as we did before with `multiply`: `fs.readFile(path, encoding, callback)` and `fs.curriedReadFile(path, encoding)(callback)` are equivalent.

This may sound silly, and you may actually think that this whole currying business is just pointless intellectual masturbation. But it is not! The interesting part is that if we are smart, we can implement `curriedReadFile` so that it starts the asynchronous read operation. And we are not forced to use the reader right away. We can keep it around, pass it to other functions and have our program do other things while the I/O operation progresses. When we need the result, we will call the reader with a callback.

By currying, we have separated the initiation of the asynchronous operation from the retrieval of the result. This is very powerful because now we can initiate several operations in a close sequence, let them do their I/O in parallel, and retrieve their results afterwards. Here is an example:

```
var reader1 = curriedReadFile(path1, "utf8");
var reader2 = curriedReadFile(path2, "utf8");
// I/O is parallelized and we can do other things while it runs

// further down the line:
reader1(function(err, data1) {
  reader2(function(err, data2) {
    // do something with data1 and data2
  });
});
```

Futures

Futures is a powerful programming abstraction that does more or less what I just described: it encapsulates an asynchronous operation and allows you to obtain the result later. Futures usually come with some API around them and a bit of runtime to support them.

My claim here is that we can probably capture the essence of futures with the simple *currying* principle that I just described. The `reader1` and `reader2` of the previous example are just *futures*, in their simplest form.

Implementation

This looks good but how hard is it to implement?

Fortunately, all it takes is a few lines of Javascript. Here is our curried `readFile` function:

```
function curriedReadFile(path, encoding) {
  var done, err, result;
  var cb = function(e, r) { done = true; err = e, result = r; };
  fs.readFile(path, encoding, function(e, r) { cb(e, r); });
  return function(_) { if (done) _(err, result); else cb = _; };
}
```

I won't go into a detailed explanation of how it works.

Going one step further

Now, we can go one step further and create a generic utility that will help us *curry* any asynchronous function.

Here is a simplified version of this utility (the complete source is on [streamline's GitHub site](#)):

```
function future(fn, args, i) {
  var done, err, result;
  var cb = function(e, r) { done = true; err = e, result = r; };
  args = Array.prototype.slice.call(args);
  args[i] = function(e, r) { cb(e, r); };
  fn.apply(this, args);
  return function(_) { if (done) _(err, result); else cb = _; };
}
```

With this utility we can rewrite `curriedReadFile` as:

```
function curriedReadFile(path, encoding) {
  return future(fs.readFile, arguments, 2);
}
```

And then, we could even get one step further, and tweak the code of the original `fs.readFile` in the following way:

```
var original = fs.readFile;
fs.readFile = function(path, encoding, callback) {
  // note: assumes always called with encoding arg
  if (!callback) return future(readFile, arguments, 2);
  // delegate implementation to original function
```

```
    original.apply(this, arguments);
}
```

With this tweak we obtain a handy API that can be used in two ways:

- directly, as a normal asynchronous call:

```
fs.readFile("hello.txt", "utf8", function(err, data) { ... })
```

- indirectly, as a synchronous call that returns a future:

```
// somewhere:
var reader = fs.readFile("hello.txt", "utf8");

// elsewhere:
reader(function(err, data) { ... });
```

Blending it with streamline.js

I have integrated this into streamline.js. The transformation engine adds the little `if (!callback) return future(...)` test in every function it generates. Then, every *streamlined* function can be used either directly, as an asynchronous call, or indirectly, to obtain a future.

Moreover, obtaining a value from a future does not require any hairy callback code any more because the streamline engine generates the callbacks for you. Everything falls down very nicely into place as the following example demonstrates:

```
function countLines(path, _) {
  return fs.readFile(path, "utf8", _).split('\n').length;
}

function compareLineCounts(path1, path2, _) {
  // parallelize the two countLines operations
  // with two futures.
  var n1 = countLines(path1);
  var n2 = countLines(path2);
  // get the results and combine them
  return n1(_) - n2(_);
}
```

Wrapping it up

I was looking for an elegant way to implement futures in streamline.js and I'm rather happy with this design. I don't know if it will get wider adoption but I think that it would be nice if node's API could behave this way and return

futures when called without callback. Performance should not be an issue because all that is required is a simple test upon function entry.

I also find it cute that a future would just be the curried version of an asynchronous function.

23 : Rethinking JavaScript: Eliminate the switch statement for better code

In my last 3 articles I convinced you to [remove the if statement](#), [kill the for loop](#), and [never use break](#). Now I'm going try to convince you to eliminate the switch statement.

While the switch statement is useful, it doesn't quite fit in with the rest of our functional code. It's not Immutable, it can't be composed with other functions, and it's a little side effecty.

Switch also uses `break`, which is also anti-functional. (Find out why `break` is so horrible in the article below)

[Rethinking JavaScript: Replace break by going functional](#)

In my last article, Death of the for Loop, I tried to convince you to abandon the for loop for a more functional...hackernoon.com

What I really want is something immutable and free of side effects. Something I can pass one value in and get one value out. You know, something functional!

Let's dig right in and take a look at an example straight from the [redux website](#).

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
```

I tried playing with this a bit and ended up converting it to a nested ternary.

```
const counter = (state = 0, action) =>
  action.type === 'INCREMENT' ? state + 1 :
  action.type === 'DECREMENT' ? state -1 :
  state
```

(If you find this syntax confusing, then check out this article, it will explain `if`'s and nested ternary.)

[Rethinking JavaScript: The if statement](#)

Thinking functionally has opened my mind about programming.medium.com

This is better, but one thing I didn't like was repetition of `action.type ===`. So then I thought I could probably use an object literal. I could also create a function to wrap the object literal and at the same time add support for a default case.

```
function switchcase (cases, defaultCase, key) => {
  if (key in cases) {
    return cases[key]
  } else {
    return defaultCase
  }
}
```

Now let's curry this bad boy, convert the `if` to a ternary and ES6 it up a bit!

```
const switchcase = cases => defaultCase => key =>
  key in cases ? cases[key] : defaultCase
```

Sexy!

Now our reducer would look like this...

```
const counter (state = 0, action) =>
  switchcase({
    'INCREMENT': state + 1,
    'DECREMENT': state -1
  })(state)(action.type)
```

The problem with this early version of `switchcase` is the entire object literal is evaluated before being passed to the `switchcase` function. This means `state +1` and `state -1` are both evaluated. This can be very dangerous... No bueno.

Hmm... if the values in the object literal were functions, then they could be executed only when there is a matching case.

So let's create a companion function called `switchcaseF` (F is for function), which is just like `switchcase` except its values are functions.

```
const switchcaseF = cases => defaultCase => key =>
  switchcase(cases)(defaultCase)(key)()
```

Now the reducer's values can be changed to functions and only the matching case will be evaluated.

```
const counter (state = 0, action) =>
  switchcase({
    'INCREMENT': () => state + 1,
    'DECREMENT': () => state -1
  })((state)(action.type))
```

...but I kinda want these functions to be *optional*. So let's try adding this...

```
const executeIfFunction = f =>
  f instanceof Function ? f() : f

const switchcaseF = cases => defaultCase => key =>
  executeIfFunction(switchcase(cases))(defaultCase)(key)
```

Now our redux reducer can look like this (notice the default value)...

```
const counter (state = 0, action) =>
  switchcase({
    'INCREMENT': () => state + 1,
    'DECREMENT': () => state -1
  })(state)(action.type)
```

OK, not vastly different, but we could do stuff like this. (added a RESET)

```
const counter (state = 0, action) =>
  switchcase({
    'RESET': 0,
    'INCREMENT': () => state + 1,
    'DECREMENT': () => state -1
  })(state)(action.type)
```

Another Example

This example is taken from http://www.w3schools.com/js/js_switch.asp

```
var day;

switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
  case 1:
    day = "Monday";
  case 2:
    day = "Tuesday";
  case 3:
    day = "Wednesday";
  case 4:
    day = "Thursday";
  case 5:
    day = "Friday";
  case 6:
    day = "Saturday";
}
```

```

        break;

    case 1:
        day = "Monday";
        break;

    case 2:
        day = "Tuesday";
        break;

    case 3:
        day = "Wednesday";
        break;

    case 4:
        day = "Thursday";
        break;

    case 5:
        day = "Friday";
        break;

    case 6:
        day = "Saturday";
}

}

```

Because we have properly curried `switchcase` like good boys and girls we can decompose this switch to multiple reusable methods, and we all know...

making our code reusable should always be one of our goals.

```

const getDay = switchcase({
    0: 'Sunday',
    1: 'Monday',
    2: 'Tuesday',
    3: 'Wednesday',
    4: 'Thursday',
    5: 'Friday',
    6: 'Saturday'
})('Unknown')

```

```

const getCurrentDay = () =>
    getDay(new Date().getDay())

```

```

const day = getCurrentDay()

```

Here we have separated out all the logic from our impure `getCurrentDay` function into a pure `getDay` function. This will make testing and **function composition** much easier.

Functional JavaScript: Function Composition For Every Day Use.

Function composition has got to be my favorite part of functional programming. I hope to provide you with a good real...hackernoon.com

Extra Credit

Now we can do cool things like create a `switchcase` programatically!

```
const days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',  
'Saturday']
```

```
const getDay = switchcase(  
  days.reduce((acc, value) =>  
    (acc[value] = `The day is ${value}.`, acc), {})  
  )("I don't know what day it is.")
```

```
console.log(getDay('Monday'))
```

```
// 'The day is Monday.'
```

This isn't the greatest example. There are obvious and better ways to generate this sentence, I'm just trying to demo what the `switchcase` can do, so cut me some slack!

The Code

Here's our final, pure, immutable and functional `switchcase` code.

3rd Party Libraries

For those interested in digging deeper into the functional world, there are some great libraries out there I would like to recommend.

Ramda—Ramda is a functional library similar to lodash and underscore. Check out Ramda's `cond` function. The functionality if similar to this `switchcase` and a great alternative!

lodash/fp—lodash/fp is the functional version of lodash. It's worth taking a look.

Fantasy Land—One of the more popular functional libraries for JavaScript.

Summary

The switch statement doesn't fit into our functional paradigm, but we can create a function that has similar characteristics. Doing so let's us decompose the switch into smaller and reusable pieces. We can even generate one programatically!

I know it's a small thing, but it makes my day when I get those follow notifications on Medium and Twitter ([@joelnet](#)). Or if you think I'm full of shit, tell me in the comments below.

Cheers!

note: before you flip out about `switchcase` not supporting 'fall-through', this is because I didn't have a need for it in my project. It's not meant to be an exact replica of `switch`. [YAGNI](#) or something.

24 : Functional JavaScript: Function Composition For Every Day Use.

<https://hackernoon.com/javascript-functional-composition-for-every-day-use-22421ef65a10#.v2ffc1h30>

Function composition has got to be my favorite part of functional programming. I hope to provide you with a good real world example so not only do you understand function composition, but so you can begin using today! In this article will learn how to write as well as *organize your files*, so you can write short, clean and **functional** code like this...

```
import { listGroupPanel } from './lib/html'
import { setInnerHTML } from './lib/dom'

const content = document.getElementById('content')
const main = e => compose(setInnerHTML(e), listGroupPanel)

const list = [
  'Cras justo odio',
  'Dapibus ac facilisis in',
  'Morbi leo risus',
  'Porta ac consectetur ac',
  'Vestibulum at eros'

]

main(content)(list)
```

...that will produce output like this...

```
<div class="panel panel-default">
  <div class="panel-body">
    <ul class="list-group">
      <li class="list-group-item">Cras justo odio</li>
      <li class="list-group-item">Dapibus ac facilisis in</li>
      <li class="list-group-item">Morbi leo risus</li>
      <li class="list-group-item">Porta ac consectetur ac</li>
      <li class="list-group-item">Vestibulum at eros</li>
    </ul>
  </div>
</div>
```

Head straight to the [codepen](#) if you want to tinker. Continue reading to start learning :)

The Basics

We have to walk before we can run, so let's start with the boring stuff that you need to know.

Function composition is a mathematical concept that allows you to combine two or more functions into a new function.

You have probably stumbled across this example when googling function composition. If you haven't yet, I assure you, you will.

```
const add = x => y => x + y
const multiply = x => y => x * y
const add2Multiply3 = compose(multiply(3), add(2))
```

I myself am guilty of using this example and what I failed to realize is that the student is not yet able to see how this can be practically applied in their codebase *today*. Instead they are comparing that example with something like this:

```
const value = (x + 2) * 3
```

It is hard to see why you would prefer the functional example.

A teacher's failure to properly provide good real world examples will result in a student's failure to understand why.

Hopefully I can do a better job of demonstrating the power of function composition.

Back to Basics

A key to function composition is having functions that are composable. A composable function should have 1 input argument and 1 output value.

You can turn any function into a composable function by currying the function. I'll expand on currying in another article, but you should still be able to follow along without knowing what currying is.

You might do html stuff, so that's a good point to start. Let's make a `tag`. (I'm going to be working with strings here, but you could also do this with React).

```
const tag = t => contents => `<${t}>${contents}</${t}>`
```

```
tag('b')('this is bold!')
```

```
> <b>this is bold!</b>
```

I also want a function to handle tags with attributes like `<div class="title">...</div>`. So I'll add another function to handle this use case.

```
const encodeAttribute = (x = '') =>
  x.replace(/"/g, '')

const toAttributeString = (x = {}) =>
  Object.keys(x)
    .map(attr => `${encodeAttribute(attr)}=${encodeAttribute(x[attr])}`)
    .join(' ')

const tagAttributes = x => c =>
  `<${x.tag}${x.attr?' ':''}${toAttributeString(x.attr)}>${c}</${x.tag}>`
```

Sprinkle in a little refactoring to combine it all into these four functions...

```
const encodeAttribute = (x = '') =>
  x.replace(/"/g, '')

const toAttributeString = (x = {}) =>
  Object.keys(x)
    .map(attr => `${encodeAttribute(attr)}=${encodeAttribute(x[attr])}`)
    .join(' ')

const tagAttributes = x => (c = '') =>
  `<${x.tag}${x.attr?' ':''}${toAttributeString(x.attr)}>${c}</${x.tag}>`

const tag = x =>
  typeof x === 'string'
    ? tagAttributes({ tag: x })
    : tagAttributes(x)
```

Now we can call `tag` with a `string` or an `object`.

```
const bold = tag('b')

bold('this is bold!')
// <b>this is bold!</b>

tag('b')('this is bold!')
// <b>this is bold!</b>
```

```
tag({ tag: 'div', attr: { 'class': 'title' }})('this is a title!')
// <div class="title">this is a title!</div>
```

Making Something Real

Now that you've made it through that boring stuff, you deserve some fun code.

Let's use that fancy new `tag` function to create something tangible. We can use something easy and something familiar, [bootstrap's list group](#).

```
<ul class="list-group">
  <li class="list-group-item">Cras justo odio</li>
  <li class="list-group-item">Dapibus ac facilisis in</li>
  <li class="list-group-item">Morbi leo risus</li>
  <li class="list-group-item">Porta ac consectetur ac</li>
  <li class="list-group-item">Vestibulum at eros</li>
</ul>
```

First, let's create a function for each of these tags and `listGroupItems` to support multiple `listGroupItems`s.

```
const listGroup = tag({ tag: 'ul', attr: { class: 'list-group' }})
const listGroupItem = tag({ tag: 'li', attr: { class: 'list-group-item' }})
const listGroupItems = items =>
  items.map(listGroupItem)
    .join('')

listGroup()
// <ul class="list-group"></ul>

listGroupItem('Cras justo')
// <li class="list-group-item">Cras justo</li>

listGroupItems(['Cras justo', 'Dapibus ac'])
// <li class='list-group-item'>Cras justo</li>
// <li class='list-group-item'>Dapibus ac</li>

listGroup(listGroupItems(['Cras justo', 'Dapibus ac']))
// <ul class='list-group'>
//   <li class='list-group-item'>Cras justo</li>
//   <li class='list-group-item'>Dapibus ac</li>
// </ul>
```

If we look at the structure of the list-group html we can see that there is one outer element that contains multiple children. Since it will *always* be created this way, it seems a little verbose to call

```
listGroup(listGroupItems(['Cras justo', 'Dapibus ac']))
```

to render the list every time.

I should just be able to call `listGroup(['Cras justo', 'Dapibus ac'])`. The function should know what I want to do.

To do this I'll start by renaming `listGroup` to `listGroupTag`. That way I can create a new `listGroup` function that will encapsulate the call to `listGroupTag(listGroupItems([]))`.

```
const listGroupTag = tag({ tag: 'ul', attr: { class: 'list-group' } })
const listGroup = items =>
  listGroupTag(listGroupItems(items))
```

Function Composition

For those of you that skipped the whole article and scrolled all the way down to this section, you might be disappointed. Composing the functions is actually the easiest part of the whole process. After you have created your functions to be composable, they just kind of snap together.

Take a look at the code below. Any time you recognize this pattern, the functions can be easily composed.

```
const listGroup = items =>
  listGroupTag(listGroupItems(items))
```

When composed together, the result will look similar to the original, `listGroupTag` on the left, followed by `listGroupItems`, and then `items` on the right.

```
const listGroup = items =>
  compose(listGroupTag, listGroupItems)(items)
```

Let's look at them side-by-side so we can see the similarities and differences.

```
listGroupTag (listGroupItems (items))
compose(listGroupTag, listGroupItems)(items)
```

When functions are composed together, they are read from right to left just like regular functions.

Because `compose` returns a function that takes a `list` and our `listGroup` function also takes a `list`, we can simplify `listGroup` to equal our composition and remove `list`.

```
const listGroup =
  compose(listGroupTag, listGroupItems)
```

By now your mind is probably not blown and I understand. Out of all this code we have written, function composition helped us simplify only a single line of code.

The power of function composition is realized as your codebase grows, allowing you to create numerous compositions.

So let's add a bootstrap panel.

```
const panelTag = tag({ tag: 'div', attr: { class: 'panel panel-default' }})
const panelBody = tag({ tag: 'div', attr: { class: 'panel-body' }})
const basicPanel =
  compose(panelTag, panelBody)
```

Now if we want to create an element that is a `list-group` inside of a `panel`, all we have to do is this:

```
const listGroupPanel =
  compose(basicPanel, listGroup)
```

The (above) function is also equivalent to this (below). You can compose any amount of functions!

```
const listGroupPanel =
  compose(basicPanel, listGroupTag, listGroupItems)
```

Organizing Your Code

Organizing your code is also very important. This involves separating your functions into multiple files.

I like to create a file called `functional.js` and this is where I put `compose` and related functional functions.

All of the functions we created above I would put in `html.js`.

I am also creating a `dom.js` for DOM manipulation (you will see in the codepen.)

Breaking our code out into multiple library files allows us to reuse these functions in other projects.

Now when we write the main program in `main.js`, there will be very little code.

The Example

I can't break up the codepen example into multiple files, so I have separated them using comments so you can see how to properly layout your files.

I put the final app up on codepen , so you can tinker with it.

<https://codepen.io/joelnet/pen/QdVpwB>

The Math

Originally I had some stuff here, but you don't need to know gravity's equation to know objects attract.

Compose and Pipe

It is also worth mentioning, `compose` has a companion function `pipe`. `pipe` also composes functions, but in reverse order. In some situations, it can be easier to understand when it's written left to write.

```
// pseudo code
```

```
const login = pipe(  
    validateInput,  
    getCustomer,  
    getToken  
    loginResponse)
```

Here are the functions, I would recommend putting these in a `functional.js` library that you can include in all your projects.

```
const compose = (...functions) => data =>  
    functions.reduceRight((value, func) => func(value), data)  
  
const pipe = (...functions) => data =>  
    functions.reduce((value, func) => func(value), data)
```

Summary

Function composition requires you to write your functions in a composable way. This means your functions must have 1 input and 1 output. Functions with multiple inputs must be curried.

Composing functions is not only easy but fun too.

You will achieve the highest level of code reuse with function composition.

making our code reusable should always be one of our goals.

My next article will be about composing **asynchronous functions**. I thought I could fit it all into a single article, but this page got long quickly. So let's just stop it here. Subscribe so you do not miss out on Part 2!

I know it's a small thing, but it makes my day when I get those follow notifications on Medium and Twitter ([@joelnet](#)). Or if you think I'm full of shit, tell me in the comments below.

Cheers!

25 : Currying in JavaScript

I've been thinking a lot lately about functional programming, and I thought it might be kind of fun to walk through the process of writing a `curry` function.

For the uninitiated, currying refers to the process of taking a function with `n` arguments and transforming it into `n` functions that each take a single argument. It essentially creates a chain of partially applied functions that eventually resolves with a value.

Here's a basic example of how you'd use it:

```
function volume( l, w, h ) {  
    return l * w * h;  
}  
  
var curried = curry( volume );  
  
curried( 1 )( 2 )( 3 ); // 6
```

Disclaimer

This post assumes basic familiarity with closures and higher-order functions, as well as stuff like `Function#apply()`. If you're not comfortable with those concepts, you might want to brush up before reading further.

Writing our curry function

The first thing you'll notice is that `curry` expects a function as its argument, so we'll start there.

```
function curry( fn ) {  
}  
}
```

Next, we need to know how many arguments our function expects (called its "arity"). Otherwise, we won't know when to stop returning new functions and give back a value instead.

We can tell how many arguments a function expects by accessing its `length` property.

```
function curry( fn ) {  
    var arity = fn.length;
```

```
}
```

From there, things get a little bit trickier.

Essentially, every time a curried function is called, we add any new arguments to an array that's saved in a closure. If the number of arguments in that array is equal to the number of arguments that our original function expects, then we call it. Otherwise, we return a new function.

To do that, we need (1) a closure that can retain that list of arguments and (2) a function that can check the total number of arguments and either return another partially applied function or the return value of the original function with all of the arguments applied.

I usually do this with an immediately invoked function called `resolver`.

```
function curry( fn ) {  
  var arity = fn.length;  
  
  return (function resolver() {  
  
    }());  
}
```

Now, the first thing we need to do in `resolver` is make a copy of any arguments it received. We'll do that by creating a variable called `memory` that uses `Array#slice` to make a copy of the `arguments` object.

```
function curry( fn ) {  
  var arity = fn.length;  
  
  return (function resolver() {  
    var memory = Array.prototype.slice.call( arguments );  
    }());  
}
```

Next, `resolver` needs to return a function. This is what the outside world sees when it calls a curried function.

```
function curry( fn ) {  
  var arity = fn.length;  
  
  return (function resolver() {  
    var memory = Array.prototype.slice.call( arguments );  
    return function() {  
  
    };  
  };
```

```
})();  
}
```

Since this internal function is the one that ends up actually being called, it needs to accept arguments. But it also needs to *add* those to any arguments that might be stored in `memory`. So first, we'll make a copy of `memory` by calling `slice()` on it.

```
function curry( fn ) {  
  var arity = fn.length;  
  
  return (function resolver() {  
    var memory = Array.prototype.slice.call( arguments );  
    return function() {  
      var local = memory.slice();  
    };  
  }());  
}
```

Now, lets add our new arguments by using `Array#push`.

```
function curry( fn ) {  
  var arity = fn.length;  
  
  return (function resolver() {  
    var memory = Array.prototype.slice.call( arguments );  
    return function() {  
      var local = memory.slice();  
      Array.prototype.push.apply( local, arguments );  
    };  
  }());  
}
```

Good. Now we have a new array containing all the arguments we've received so far in this chain of partially applied functions.

The last thing to do is to compare the length of arguments we've received with the arity of our curried function. If the lengths match, we'll call the original function. If not, we'll use `resolver` to return yet another function that has all of our current arguments stored in `memory`.

```
function curry( fn ) {  
  var arity = fn.length;  
  
  return (function resolver() {
```

```

var memory = Array.prototype.slice.call( arguments );
return function() {
    var local = memory.slice(), next;
    Array.prototype.push.apply( local, arguments );
    next = local.length >= arity ? fn : resolver;
    return next.apply( null, local );
};
}();
}

```

This can be a little bit difficult to wrap your head around, so let's take it step by step in an example.

```

function volume( l, w, h ) {
    return l * w * h;
}

var curried = curry( volume );

```

Okay, so `curried` is the result of passing `volume` into our `curry` function.

If you look back, what's happening here is:

1. We store the arity of `volume`, which is `3`.
2. We immediately invoke `resolver` with no arguments, which means that for now, its `memory` array is empty.
3. `resolver` returns an anonymous function.

Still with me? Now let's call our `curried` function and pass in a length.

```

function volume( l, w, h ) {
    return l * w * h;
}

var curried = curry( volume );
var length = curried( 2 );

```

Again, here are the steps:

1. What we actually called here was the anonymous function being returned by `resolver`.
2. We made a copy of `memory` (which was empty) and called it `local`.
3. We added our argument (`2`) to the `local` array.
4. Since the length of `local` is less than the arity of `volume`, we call `resolver` again with the list of arguments we have so far. That creates a new closure with a new `memory` array, which contains our first argument of `2`.
5. Finally, `resolver` returns a new function that has access to an outer closure with our new `memory` array.

So what we get back is that inner anonymous function again. But this time, it has access to a `memory` array that isn't empty. It has our first argument (`a[2]`) inside of it.

If we call our `length` function, the process repeats.

```
function volume( l, w, h ) {
  return l * w * h;
}

var curried = curry( volume );
var length = curried( 2 );
var lengthAndWidth = length( 3 );
```

1. Again, what we actually called was the anonymous function being returned by `resolver`.
2. This time, `resolver` had been primed with some previous arguments. So we make a copy of that array [`2`].
3. We add our new argument, `3`, to the `local` array.
4. Since the length of `local` is still less than the arity of `volume`, we call `resolver` again with the list of arguments we have so far -- and that returns a new function.

Now it's time to call our `lengthAndWidth` function and get back a value.

```
function volume( l, w, h ) {
  return l * w * h;
}

var curried = curry( volume );
var length = curried( 2 );
var lengthAndWidth = length( 3 );

console.log( lengthAndWidth( 4 ) ); // 24
```

This time, the steps are a little bit different at the end:

1. Once again, what we actually called was the anonymous function being returned by `resolver`.
2. This time, `resolver` had been primed with two previous arguments. So we make a copy of that array [`2, 3`].
3. We add our new argument, `4`, to the `local` array.
4. Now the length of `local` is `3`, which is the arity of `volume`. So instead of returning a new function, we return the result of calling `volume` with all of the arguments we've been saving up, and that gives us a value of `24`.

Wrapping up

Admittedly, I have yet to find a super compelling use-case for currying in my day-to-day work. But I still think that going through the process of writing functions like this is a great way to improve your understanding of functional programming, and it helps reinforce concepts like closures and first-class functions.

By the way, if you like nerdy JavaScript things and live in the Boston area, I'm hiring at [Project Decibel](#). Shoot me an [email](#).

26 : [object Object] Things You Didn't Know About valueOf

Okay, fine. That title is horrible. But we're all competing with BuzzFeed for readers now, and I needed something catchy, so here we are.

Right, then.

So, what is `Object#valueOf` and why should you care?

More or less, it's a method that JavaScript calls automatically any time it sees an object in a situation where a primitive value is expected.

Let's look at a quick example.

```
var obj = {};  
  
console.log( 7 + obj ); // "7[object Object]"
```

Plenty has been written about the insanity of implicit type conversion in JavaScript, so I'm really not interested in going down that road. Hopefully, your takeaway from the example is that we used the addition operator with a number on one side and an object on the other, and we got back a value that doesn't really make a ton of sense.

You can hardly blame JavaScript here, though. It has no way to know *how* to convert our object into a number.

But as luck would have it, we can actually define that behavior.

```
var obj = {  
  valueOf: function() {  
    return 42;  
  }  
};  
  
console.log( 7 + obj ); // 49
```

See? I wasn't lying earlier. The EcmaScript specification dictates that whenever an object is encountered in a situation where a primitive is expected, that object's `valueOf` method will be called*.

* Actually, sometimes `toString()` will be called instead -- but I'm not gonna get into it here. You can assume that any time a **Number** is expected, `valueOf` will be called. If you're super interested in this stuff, look up the **ToPrimitive** abstract operation in the ES spec.

Who cares?

Look, I'm not gonna lie to you here (I feel like we're becoming friends, and I have a pathological need to be trusted). Knowing about `valueOf` is not going to materially change the way you write code on a day-to-day basis. This post is really about surfacing an interesting little JS feature that a lot of people don't know about. But that being said, there *are* some practical applications.

Let's write a little `Integer` class. I won't go into much detail here, since I will assume that you, esteemed reader, are already intimately familiar with the concept of integers.

You can actually just skip this part if you want...

```
function Int( val ) {
  if ( !( this instanceof Int ) ) {
    return new Int( val );
  }
  if ( isNaN( val ) ) {
    throw new TypeError('Int value must be a number');
  }
  if ( val % 1 !== 0 ) {
    throw new TypeError('Int value must be an integer');
  }
  if ( val instanceof Int ) {
    return Int( val._value );
  }
  this._value = Math.floor( val );
}

Int.prototype.add = function( n ) {
  return Int( this._value + Int( n )._value );
};

Int.prototype.subtract = function( n ) {
  return Int( this._value - Int( n )._value );
};

Int.prototype.multiply = function( n ) {
  return Int( this._value * Int( n )._value );
};

Int.prototype.divide = function( n ) {
  try {
    return Int( this._value / Int( n )._value );
  } catch ( e ) {
    throw new Error('Division resulted in non-integer quotient');
}
```

```
    }
};
```

Okay. Now we've got an `Int` class.

It does some type checking. It makes sure you don't pass it a float. And it has some basic arithmetic methods.

Here are a couple quick examples, just to give you a better idea of what this thing looks like:

```
Int( 6 ).multiply( 2 );           // { _value: 12 }
Int( 4 ).divide( 2 ).add( 3 ); // { _value: 5 }
```

Simple enough. But what if I need to do something like this?

```
var a = Int( 6 );
var b = Int( 11 );

Math.min( a, b ); // NaN
```

It basically blows up. I mean, it doesn't throw an exception, but it gives me `Nan`, which is *almost* as useless.

I trust at this point that you see where we're headed.

```
Int.prototype.valueOf = function() {
  return this._value;
};
```

Easy as that. Now, we'll try that same example -- but this time, our `Int` class has a `valueOf` method on its prototype.

```
var a = Int( 6 );
var b = Int( 11 );

Math.min( a, b ); // 6
```

Not bad, right? We can also use instances of `Int` in other places where a primitive value is expected:

```
Math.pow( Int( 9 ), 0.5 ); // 3
```

The More You Know™

If you liked this post, follow me on [Medium](#) or [Twitter](#) for more. I'm challenging myself to write every day for the next month.

And if you live in the Boston area and want to work on crazy, top-secret things with me at [Project Decibel](#), shoot me an [email](#). I'm hiring.

27 : Prototypal Inheritance

Last year I wrote a post called "How to impress me in an interview", and in it, I mentioned that I run across a lot of candidates (the vast majority, actually) who don't really understand how prototypes work.

In a way, that's kind of an amazing testament to how flexible JavaScript is, and how user-friendly many of today's popular libraries are. I can't think of many other object-oriented languages where an engineer can be reasonably productive without being at least vaguely aware of classes.

But here's the thing: if you write JavaScript even semi-regularly, you really *should* understand how prototypal inheritance works. And that's why I'm writing this post.

Disclaimer

There will be some things in this article that I over-simplify a little bit.

My goal here is to help someone who's never used prototypal inheritance to become comfortable with it. In order to do that, I'll cut a few small corners here and there if I think it helps eliminate a bit of confusion.

Objects Inherit from Objects

If you've ever read **anything** about inheritance in JS, then you've almost certainly heard that objects inherit from other objects.

This is true, and *once you already understand it*, it's a good way to think about things. But my experience has been that this explanation alone isn't really sufficient.

So I'm going to break from tradition here and not actually try to explain how inheritance works right up front.

Instead, we'll start with some simpler stuff that will hopefully provide a bit of context later on.

Function prototypes

Here's a fun fact: In JavaScript, all functions are also objects, which means that they can have properties. And as it so happens, they all have a property called `prototype`, which is also an object.

```
function foo() {  
}  
  
typeof foo.prototype // 'object'
```

That's pretty simple, right? Any time you create a function, it will *automatically* have a property called `prototype`, which will be initialized to an empty object.

Constructors

In JavaScript, there's really no difference between a "regular" function and a constructor function. They're actually all the same. But as a convention, functions that are *meant to be used* as constructors are generally capitalized.

By the way, if you don't know what I mean when I say "constructor", that's totally okay. We'll get there.

So let's say that we want to make a constructor function called `Dog`, because explaining inheritance using animals is a time-honored tradition and I'm kind of a nostalgic dude.

```
function Dog() {  
}  
}
```

If I want to make an instance of `Dog`, I use the `new` keyword. That's really what I mean when I talk about constructors -- I'm using the function to *construct* a new object. Any time you see the `new` keyword, it means that the following function is being used as a constructor.

```
var fido = new Dog();
```

So now we have `fido`, who's a `Dog`. But he doesn't really do anything.

He's kind of like my dog, actually.

Methods

It's time to make our `Dog` a little more dog-like. Bear with me here, because I'm going to explain this in just a minute.

```
function Dog() {  
}  
  
Dog.prototype.bark = function() {  
    console.log('woof!');  
};
```

You should remember from earlier that all functions automatically get initialized with a `prototype` object. In the example above, we tacked a function onto it called `bark`.

Now let's make ourselves a new `fido`.

```
function Dog() {  
  
}  
  
Dog.prototype.bark = function() {  
    console.log('woof!');  
};  
  
var fido = new Dog();  
  
fido.bark(); // 'woof!'
```

I'm going to explain this in more detail a little later on. But the important thing to take away right now is that by placing `bark` on `Dog.prototype`, we made it available to all instances of `Dog`.

Don't worry yet about *how* this works. Just keep in mind that it works.

Differential Inheritance

JavaScript uses an inheritance model called "differential inheritance". What that means is that methods aren't copied from parent to child. Instead, children have an "invisible link" back to their parent object.

For example, `fido` doesn't actually have its own method called `bark()` (in other words, `fido.hasOwnProperty('bark') === false`).

What actually happens when I write `fido.bark()` is this:

1. The JS engine looks for a property called `bark` on our `fido` object.
2. It doesn't find one, so it looks "up the prototype chain" to `fido`'s parent, which is `Dog.prototype`.
3. It finds `Dog.prototype.bark`, and calls it with `this` bound to `fido`.

That part is really important, so I'm going to repeat it:

There's really no such property as `fido.bark`. It doesn't exist. Instead, `fido` has access to the `bark()` method on `Dog.prototype` because it's an instance of `Dog`. This is the "invisible link" I mentioned. More commonly, it's referred to as the "prototype chain".

Object.create()

Okay. We talked a little bit about differential inheritance and the prototype chain. Now it's time to put that into action.

Since ES5, JavaScript has had a cool little function called `Object.create()`.

Here's how it works.

```
var parent = {
  foo: function() {
    console.log('bar');
  }
};

var child = Object.create( parent );

child.hasOwnProperty('foo'); // false
child.foo(); // 'bar'
```

So what is it doing?

Essentially, it creates a new, empty object that has `parent` in its prototype chain. That means that even though `child` doesn't have its own `foo()` method, it has access to the `foo()` method from `parent`.

Try It

We've covered a lot of ground so far, and we still have a bit more to go. To make sure everything we learned so far sinks in, I'd encourage you to open up your dev tools and try a quick little example.

Create a constructor function called `Car`. Add a `drive()` method to its prototype that just logs to the console. Now create an instance of `Car` and call `drive()`.

I really mean it. You should take 30 seconds and go do it. I'll wait here. And no cheating...

Good, you're back. Hopefully you wrote something that looks like this.

```
function Car() {

}

Car.prototype.drive = function() {
  console.log('vroom');
};

var benz = new Car();
```

```
benz.drive(); // vroom
```

If not, consider re-reading some of the earlier sections. It's important that you're somewhat comfortable with that stuff before you move on.

Putting It All Together

It's time to look at a (slightly) more real-world example that takes everything in this post and puts it all together.

Let's start by creating a `Rectangle` constructor.

```
function Rectangle( width, height ) {  
  this.width = width;  
  this.height = height;  
}
```

At this point, we'll take a tiny detour to talk about `this`.

When a function is used as a constructor, `this` refers to the *new object that you're creating*. So in our `Rectangle` constructor, we're taking `width` and `height` as arguments, and assigning those values to the `width` and `height` properties of our new `Rectangle` instance.

```
var rect = new Rectangle( 3, 4 );  
  
rect.width; // 3  
rect.height; // 4
```

Now it's time to give our `Rectangle` a method. Let's call it `area`.

```
Rectangle.prototype.area = function() {  
  return this.width * this.height;  
};
```

There's that `this` keyword again. Just like in the constructor, `this` inside of a method refers to the instance.

```
var rect = new Rectangle( 3, 4 );  
  
rect.area(); // 12
```

Subclassing

What if we want to make a new class of object that inherits from `Rectangle`?

Let's say we need a class called `Square`. Hopefully you remember from elementary school that a square is just a specific type of rectangle.

We'll start by creating its constructor.

```
function Square( length ) {
    this.width = this.height = length;
}
```

So, that's all well and good, but how do we make `Square` inherit from `Rectangle`? It's all about setting up the prototype chain.

If you remember from earlier, we can use `Object.create()` to create an empty object that inherits from another object. In the case of `Square`, that means all we need to do is this:

```
Square.prototype = Object.create( Rectangle.prototype );
```

All instances of `Square` will automatically have `Square.prototype` in their prototype chain, and because `Square.prototype` has `Rectangle.prototype` in *its* prototype chain, every `Square` will have access to the methods of `Rectangle`.

In other words, we can do this:

```
var square = new Square( 4 );

square.area(); // 16
```

We can also add new methods that are specific to `Square`. Remember, `Square.prototype` is just an empty object right now (albeit with a link back to `Rectangle.prototype`).

```
Square.prototype.diagonal = function() {
    return Math.sqrt( this.area() * 2 );
};
```

Time travel

One of the really cool (and potentially dangerous) things about inheritance in JavaScript is that you can modify or extend the capabilities of a class *after* you've defined it.

Because JavaScript will look up the prototype when trying to access properties on an object, to you can alter your classes at runtime.

Here's an example (for illustrative purposes only. Don't ever do this):

```
var arr = [ 1, 2, 3, 4, 5 ];

Array.prototype.shuffle = function() {
    return this.sort(function() {
        return Math.round( Math.random() * 2 ) - 1;
    });
};

arr.shuffle(); // [ 3, 1, 4, 5, 2 ]
```

The important thing to notice in this example is that `arr` was created *before* `Array.prototype.shuffle` existed. But because property lookups are done at runtime, our array got access to the new method anyway. It's like we went back in time and gave `Array` a (stupid) new method.

To get a sense of how powerful (and potentially dangerous) this is, go open the JS console on a page like Facebook, paste in the following code, and watch the log as you click around:

```
Array.prototype.push = function() {
    throw new Error('lolnope');
};
```

If you live in the Boston area, love JavaScript, and want to work on some really exciting problems at [Starry](#), shoot me an [email](#). I'm hiring.

28 : How to impress me in an interview

In the past few years, I've interviewed dozens of candidates for JavaScript-heavy roles. Over that time, I've identified a few key areas that seem to be pretty good indicators of overall proficiency -- and I focus on these pretty much exclusively in my technical interviews.

First and foremost, this post is about what to expect if you're interviewing with me. But it's also about how to conduct a technical interview of JS candidates, and which areas offer the best signal-to-noise ratios.

This is Not a Test

The most important thing that I try to stress in any technical interview is that I'm not administering a test. What I'm really interested in is using questions as a tool for facilitating a conversation about code.

So if I ask you to write a `memoize()` function, what I really want is to give us a starting point to talk about higher-order functions and closures. I don't really care if you can write that function off the top of your head. Instead, what I'm really looking for is whether or not you can arrive at a solution with a little bit of guidance, and if you understand the underlying concepts.

If you get stuck along the way, ask for some help. And if you flat out don't know an answer, that's okay too. I'll do my best to explain it, and we'll work on another example together.

One of the most impressive things you can do in an interview is learn something new and then apply it. It lets me see how you think about problems and assimilate new information.

Anyway, the point is: not knowing things is totally okay.

Whiteboards

There seems to be a lot of anxiety these days about whiteboarding during interviews. Every couple weeks, I see an article on Hacker News about what a terrible, outmoded practice this is.

And for the most part, I agree.

That said, if you interview with me, you should expect to walk in to a room with a whiteboard on the wall. And you should expect that we'll use it.

Whiteboards are great for quick examples. If I ask you what an IIFE is, or how to write a ternary expression, I expect you to be able to jot that down on a whiteboard. Or maybe I'll write a function on the board and ask you what its return value is given a certain set of arguments.

What I won't ever do -- and what I think is the primary cause of apprehension about whiteboarding -- is ask you to write some big, 20-line function on a whiteboard. It's unnatural, it's slow, and frankly, it's pretty hard for me to read your code that way anyway.

Frameworks & Libraries

Don't care.

As JavaScript engineers, we use third-party code all day, every day. On the front-end, there's jQuery, Underscore, Backbone, Ember, Angular, React, Moment, D3, Bluebird, THREE, and a million others. With Node, there are currently something like 166,000 packages on npm.

We're all learning new APIs constantly, and it's totally unreasonable to expect that you happen to know the same ones that I do. I will almost never ask specific questions about libraries or frameworks.

If you've got a strong understanding of fundamental JS concepts, you'll learn the libraries we're using pretty quickly.

I'm just not that worried about it.

Closures and First-Class Functions

Super important. I expect everyone I interview to be very comfortable with these concepts.

Here's a typical example that I might present to a candidate:

```
// write a function called `partial` that makes
// the following snippet work

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

var add5 = partial( add, 5 );

add5( 4 ); // 9
```

If that elicits a deer-in-the-headlights look, I might steer the discussion like this:

"Okay, so it looks like `partial` accepts two arguments. One is a function, and the other is a number."

That'll usually get us at least to this point:

```
function partial( fn, num ) {

}
```

"Great. And `partial` must return a function, because we're assigning its return value to `add5` and then calling `add5()`".

```
partial( fn, num ) {
  return function() {
    ...
  }
}
```

"Cool. That's the basic idea. So, if `add5` is the function that `partial` returns and it expects a number as an argument, where does that go?"

Usually, this is the point where a light bulb turns on.

```
partial( fn, a ) {
  return function( b ) {
    return fn( a, b );
  }
}
```

Again, the point here is not necessarily that you can do this without any help (although that's great, and it's what I'd expect from a more senior candidate). What I'm really hoping is that you can get there with a little bit of direction.

Prototypal Inheritance

This is definitely an area that tends to separate junior devs from more senior ones, although in my mind, it's something *everyone* applying for a JS role should know.

Here's a problem that I might give to a candidate:

```
// make a class called `Square` that inherits from
// `Rectangle` and satisfies the following snippet

function Rectangle( width, height ) {
  this.width = width;
  this.height = height;
}

Rectangle.prototype.area = function() {
  return this.width * this.height;
};
```

```
// your code here

var square = new Square( 4 );

square.area(); // 16
Square.prototype.area === Rectangle.prototype.area; // true
```

If this looks foreign to you, I would strongly recommend brushing up on prototypal inheritance. It's really not very hard once you get the hang of it, and it's an extremely powerful and important feature of JavaScript.

Oh, and here's the answer:

```
function Square( length ) {
  Rectangle.call( this, length, length );
}

Square.prototype = Object.create( Rectangle.prototype );
```

What's this?

There's probably no other feature in JS with a worse simplicity-to-confusion ratio than the `this` keyword. You can basically learn everything about it in an hour, but so many JS developers never do.

You should know how `Function#call()` and `Function#apply()` work. They are absolutely essential to writing JS at a high level, and they're super easy to wrap your head around.

This is low-hanging fruit.

Async

Another big one, especially if you're applying for a job where you'll be using Node.

Here's a question I might ask:

```
// write a function called `shout` that accepts
// a string and a callback function, and uses
// `exclaim` and `yell` to transform its input

function exclaim( value, fn ) {
  setTimeout(function() {
    fn( value + '!' );
  }, 100 );
```

```

}

function yell( value, fn ) {
  setTimeout(function() {
    fn( value.toUpperCase() );
  }, 100 );
}

shout( 'hello', function( shouted ) {
  console.log( shouted ); // 'HELLO!'
});

```

Again, this is really something that I expect every JS candidate to be comfortable with. If you're about to interview for a JS gig and you're not familiar with this stuff, drop what you're doing and spend some time practicing. It's that important.

Oh, and the solution to that problem:

```

function shout( value, fn ) {
  exclaim( value, function( exclaimed ) {
    yell( exclaimed, fn );
  });
}

```

Trick Questions

Personally, not a fan.

In the rare occasion that I *do* ask a trick question, I'll call it out beforehand, and I won't really care if you miss the "trick". What I'm really trying to accomplish in cases like this is to start a conversation.

"Not quite. Remember, variables get hoisted in JavaScript..."

"That's actually a `ReferenceError`. Can you see why?"

Questions like this can be really useful, but I use them pretty sparingly because nobody likes to feel like they've been set up for failure.

Advanced Stuff

Okay, so you were able to breeze through closures, prototypal inheritance, and async functions. Now what?

At this point in an interview, I'm usually pretty confident that a candidate has an intermediate to advanced understanding of the language, and my goals for the interview start to shift a little.

What I really want to know now is *how much* you know. I'm trying to get a sense of the shape and depth of your JavaScript knowledge.

This is where we start to move away from core concepts and I might start asking about specific language features or more advanced functional programming.

I might ask you to use `Object.defineProperty` or to write a `curry` function. Maybe we'll talk about promises or recursion.

These are things that you don't really *have* to know, per se, but since you've got the basic stuff down, I need to ask these types of questions to get a better sense of how advanced you are.

Think of this as extra credit. It's an opportunity to impress me. Anything we talk about at this point is essentially just bonus material.

Memorization

This is sort of a tricky area, and I'm a little bit conflicted.

I expect a senior level JS person to know the argument signatures for `Array#splice()` or `Function#bind()`.

At the same time, I have never worked in an office without an Internet connection -- and looking something up on MDN takes about 20 seconds.

I never explicitly quiz candidates on stuff like this, because it just doesn't strike me as being particularly important. That said, if I realize during the course of our interview that you don't know how to use a common method like `Array#push()`, that's a pretty good indicator that you haven't been writing JS for very long.

In short: Learn the most commonly used stuff, but don't feel like you need to memorize the entire EcmaScript spec.

Ask Questions

If you only take one thing away from this post, let it be this:

If you don't know something, or you aren't sure, ask me.

Not knowing something is a problem with a very simple solution. Lacking intellectual curiosity is much harder to fix.

I love hiring candidates who need a bit of mentoring. It's fun for me, and it's great for the team. One of the best ways to crystalize your own understanding of a concept is to explain it to someone else, and fostering a team environment where we're all teachers *and* students is the best way I know to continually raise the bar for everyone.

We all have plenty more to learn, myself included. My goal has always been to hire people who embrace that fact, and who are genuinely excited about learning new things and becoming stronger engineers.

I'm Hiring

Is this hypothetical interview sounding pretty fun to you? Are you in the Boston area?

Shoot me an [email](#), and come work on **crazy**, top-secret things at [Project Decibel](#).

29 : A 10 minute primer to JavaScript modules, module formats, module loaders and module bundlers



Modern JavaScript development can be overwhelming.

When working on a project, you may wonder why all the modern machinery and tooling is needed.

What are tools like Webpack and SystemJS doing? Or what do AMD, UMD or CommonJS mean? How do they relate to each other? And why do you need them at all?

In this article you will learn the difference between JavaScript modules, module formats, module loaders and module bundlers.

This is not an in-depth guide into any of the individual tools or patterns, but a primer to quickly understand the concepts of modern JavaScript development.

So let's get started.

What is a module?

A module is a reusable piece of code that encapsulates implementation details and exposes a public API so it can be easily loaded and used by other code.

Why do we need modules?

Technically we can write code without modules.

Modules are a pattern that developers have been using in many different forms and programming languages since the 60's and 70's.

In JavaScript, modules should ideally allow us to:

- *abstract code*: to delegate functionality to specialised libraries so that we don't have to understand the complexity of their actual implementation
- *encapsulate code*: to hide code inside the module if we don't want the code to be changed
- *reuse code*: to avoid writing the same code over and over again
- *manage dependencies*: to easily change dependencies without rewriting our code

Module patterns in ES5

EcmaScript 5 and earlier editions were not designed with modules in mind. Over time, developers came up with different patterns to simulate modular design in JavaScript.

To give you an idea of what some of these patterns look like, let's quickly look at 2 easy ones: *Immediately Invoked Function Expressions* and *Revealing Module*.

Immediately Invoked Function Expression (IIFE)

```
(function(){  
  // ...  
})()
```

An Immediately Invoked Function Expression (IIFE) is an anonymous function that is invoked when it is declared.

Notice how the function is surrounded by parentheses. In JavaScript, a line starting with the word `function` is considered as a function declaration:

```
// Function declaration  
function(){  
  console.log('test');  
}
```

Immediately invoking a function declaration throws an error:

```
// Immediately Invoked Function Declaration  
function(){  
  console.log('test');  
}()  
  
// => Uncaught SyntaxError: Unexpected token )
```

Putting parentheses around the function makes it a function expression:

```
// Function expression  
(function(){  
  console.log('test');  
})  
  
// => returns function test(){ console.log('test') }
```

The function expression returns the function, so we can immediately call it:

```
// Immediately Invoked Function Expression  
(function(){  
  console.log('test');  
})()  
  
// => writes 'test' to the console and returns undefined
```

Immediately Invoked Function Expressions allow us to:

- encapsulate code complexity inside IIFE so we don't have to understand what the IIFE code does
- define variables inside the IIFE so they don't pollute the global scope (`var` statements inside the IIFE remain within the IIFE's closure)

but they don't provide a mechanism for dependency management.

Revealing Module pattern

The Revealing Module pattern is similar to an IIFE, but we assign the return value to a variable:

```
// Expose module as global variable
var singleton = function(){

    // Inner logic
    function sayHello(){
        console.log('Hello');
    }

    // Expose API
    return {
        sayHello: sayHello
    }
}()
```

Notice that we don't need the surrounding parentheses here because the word `function` is not at the beginning of the line.

We can now access the module's API through the variable:

```
// Access module functionality
singleton.sayHello();
// => Hello
```

Instead of a singleton, a module can also expose a constructor function:

```
// Expose module as global variable
var Module = function(){

    // Inner logic
    function sayHello(){
        console.log('Hello');
    }
}
```

```
// Expose API
return {
  sayHello: sayHello
}
}
```

Notice how we don't execute the function at declaration time.

Instead, we instantiate a module using the `Module` constructor function:

```
var module = new Module();
```

to access its public API:

```
module.sayHello();
// => Hello
```

The Revealing Module pattern offers similar benefits as an IIFE, but again does not offer a mechanism for dependency management.

As JavaScript evolved, many more different syntaxes were invented for defining modules, each with their own benefits and downsides.

We call them module formats.

Module formats

A module format is the syntax we can use to define a module.

Before EcmaScript 6 or ES2015, JavaScript did not have an official syntax to define modules. Therefore, smart developers came up with various formats to define modules in JavaScript.

Some of the most widely adapted and well known formats are:

- Asynchronous Module Definition (AMD)
- CommonJS
- Universal Module Definition (UMD)
- System.register
- ES6 module format

Let's have a quick look at each one of them so you can recognize their syntax.

Asynchronous Module Definition (AMD)

The [AMD](#) format is used in browsers and uses a `define` function to define modules:

```
//Calling define with a dependency array and a factory function
define(['dep1', 'dep2'], function (dep1, dep2) {

    //Define the module value by returning a value.
    return function () {};
});
```

CommonJS format

The [CommonJS](#) format is used in Node.js and uses `require` and `module.exports` to define dependencies and modules:

```
var dep1 = require('./dep1');
var dep2 = require('./dep2');

module.exports = function(){
    // ...
}
```

Universal Module Definition (UMD)

The [UMD](#) format can be used both in the browser and in Node.js.

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Register as an anonymous module.
        define(['b'], factory);
    } else if (typeof module === 'object' && module.exports) {
        // Node. Does not work with strict CommonJS, but
        // only CommonJS-like environments that support module.exports,
        // like Node.
        module.exports = factory(require('b'));
    } else {
        // Browser globals (root is window)
        root.returnExports = factory(root.b);
    }
})(this, function (b) {
    //use b in some fashion.

    // Just return a value to define the module export.
    // This example returns an object, but the module
    // can return a function as the exported value.
```

```
    return {};
});
```

System.register

The [System.register format](#) was designed to support the ES6 module syntax in ES5:

```
import { p as q } from './dep';

var s = 'local';

export function func() {
    return q;
}

export class C {
```

ES6 module format

As of ES6, JavaScript also supports a native module format.

It uses an `export` token to export a module's public API:

```
// lib.js

// Export the function
export function sayHello(){
    console.log('Hello');
}

// Do not export the function
function somePrivateFunction(){
    // ...
}
```

and an `import` token to import parts that a module exports:

```
import { sayHello } from './lib';

sayHello();
// => Hello
```

We can even give imports an alias using `as`:

```
import { sayHello as say } from './lib';

say();
// => Hello
```

or load an entire module at once:

```
import * as lib from './lib';

lib.sayHello();
// => Hello
```

The format also supports default exports:

```
// lib.js

// Export default function
export default function sayHello(){
  console.log('Hello');
}

// Export non-default function
export function sayGoodbye(){
  console.log('Goodbye');
}
```

which you can import like this:

```
import sayHello, { sayGoodbye } from './lib';

sayHello();
// => Hello

sayGoodbye();
// => Goodbye
```

You can export not only functions, but anything you like:

```
// lib.js
```

```
// Export default function
export default function sayHello(){
    console.log('Hello');
}

// Export non-default function
export function sayGoodbye(){
    console.log('Goodbye');
}

// Export simple value
export const apiUrl = '...';

// Export object
export const settings = {
    debug: true
}
```

Unfortunately, the native module format is not yet supported by all browsers.

We can already use the ES6 module format today, but we need a transpiler like Babel to transpile our code to an ES5 module format such as AMD or CommonJS before we can actually run our code in the browser.

Module loaders

A module loader interprets and loads a module written in a certain module format.

A module loader runs at runtime:

- you load the module loader in the browser
- you tell the module loader which main app file to load
- the module loader downloads and interprets the main app file
- the module loader downloads files as needed

If you open the network tab in your browser's developer console, you will see that many files are loaded on demand by the module loader.

A few examples of popular module loaders are:

- [RequireJS](#): loader for modules in AMD format
- [SystemJS](#): loader for modules in AMD, CommonJS, UMD or System.register format

Module bundlers

A module bundler replaces a module loader.

But, in contrast to a module loader, a module bundler runs at build time:

- you run the module bundler to generate a bundle file at build time (e.g. bundle.js)
- you load the bundle in the browser

If you open the network tab in your browser's developer console, you will see that only 1 file is loaded. No module loader is needed in the browser. All code is included in the bundle.

Examples of popular module bundlers are:

- [Browserify](#): bundler for CommonJS modules
- [Webpack](#): bundler for AMD, CommonJS, ES6 modules

Summary

To better understand tooling in modern JavaScript development environments, it is important to understand the differences between modules, module formats, module loaders and module bundlers.

A *module* is a reusable piece of code that encapsulates implementation details and exposes a public API so it can be easily loaded and used by other code.

A *module format* is the syntax we use to define a module. Different module formats such [AMD](#), [CommonJS](#), [UMD](#) and [System.register](#) have emerged in the past and a native module format is now available since ES6.

A *module loader* interprets and loads a module written in a certain module format at runtime. Popular examples are [RequireJS](#) and [SystemJS](#).

A *module bundler* replaces a module loader and generates a bundle of all code at build time. Popular examples are [Browserify](#) and [Webpack](#).

There you have it—you are now armed with the knowledge to better understand modern JavaScript development.

Next time the TypeScript compiler asks you what module format you wish to compile your TypeScript in, you should fully understand why. If you don't, don't worry and just re-read this primer.

Have a wonderful day and develop with passion.

30 : Rethinking JavaScript: Replace break by going functional

31 : Rethinking JavaScript: Replace break by going functional

In my last article, [Death of the for Loop](#), I tried to convince you to abandon the `for` loop for a more functional solution. In return, you raised a great question, "What about `break`?"

`break` is the `GOTO` of loops and should be avoided.

`break` should be placed in the rubbish bin in the same exact way `GOTO` has.

You may be thinking, "Come on Joel, surely you are just being sensational. How is `break` like `GOTO`?"

```
// bad code. no copy paste.
```

```
outer:
  for (var i in outerList) {
inner:
  for (var j in innerList) {
    break outer;
  }
}
```

I am offering labels as the proof. In other languages, labels were the counterpart to `GOTO`. In JavaScript, label is the counterpart of `break` and `continue`. Because `break` and `continue` come from the same label family, this also makes them very close relatives of `GOTO`.

JavaScript's label, `break` and `continue` are leftover legacy from the days of `GOTO` and unstructured programming.

"But it's not hurting anyone, so why not leave it in the language so that we have *options*?"

Why should we place limits on how we write software?

It sounds counter-intuitive, but limits are a good thing. The removal of `GOTO` is a perfect example of this. We also welcomed the limitation "`use strict`" with open arms and even lecture those who don't use it!

"limitations can make things better. A lot better." — Charles Scalfani

Limits make us write better software.

[Why Programmers Need Limits](#)

[Limits make for better Art, Design and Life.medium.com](#)

What are our alternatives to `break`?

I'm not gonna sugar coat this, there is no one size fits all quick and easy replacement. It's a completely different way of programming. A completely different way of thinking. *A functional way of thinking*.

The good news is there are *many* libraries and tools out there to help us like [Lodash](#), [Ramda](#), [lazy.js](#), recursion, etc.

We'll start with a simple collection of cats and a function called `isKitten`. These will be used in all of the examples to follow.

```
const cats = [
  { name: 'Mojo', months: 84 },
  { name: 'Mao-Mao', months: 34 },
  { name: 'Waffles', months: 4 },
  { name: 'Pickles', months: 6 }
]
```

```
const isKitten = cat => cat.months < 7
```

Let's start out with a familiar old-school `for` loop example. This will loop through our cats and break when it finds the first kitten.

```
var firstKitten

for (var i = 0; i < cats.length; i++) {
  if (isKitten(cats[i])) {
    firstKitten = cats[i]
    break
  }
}
```

Now let's compare that to the lodash equivalent.

```
const firstKitten = _.find(cats, isKitten)
```

OK, so that example is fairly simple. Let's kick it up a notch and try something a little more *edge case*. Let's enumerate our cats and break after we have found 5 kittens.

```
var first5Kittens = []

// old-school edge case kitty loop
for (var i = 0; i < cats.length; i++) {
  if (isKitten(cats[i])) {
    first5Kittens.push(cats[i])
    if (first5Kittens.length === 5) {
      break
    }
  }
}
```

```
if (first4Kittens.length >= 5) {  
    break  
}  
}  
}
```

The easy way

Iodash is amazing and does tons of great things, but sometimes you need something more specialized. **This is where we bring in a new friend, [lazy.js](#).** It's "Like Underscore, but lazier". And lazy is what we want.

```
const result = Lazy(cats)  
.filter(isKitten)  
.take(5)
```

The hard way

Libraries are all fun and games, but **sometimes what's really fun is to create something from scratch!**

So how about we create a generic function that will act like `filter` and also add the limit functionality.

The first step is to encapsulate our old-school edge case kitty loop in a function.



Next, let's generalize the function and extract out all the cat specific stuff. Replace `5` with `limit`, `isKitten` with `predicate` and `cats` with `list`. Then add those as parameters to the function.



Now we have a working and reusable `takeFirst` function that has been completely separated from our cat business logic!

Our function is now also a **pure function**. This means the output is derived solely from the inputs. Given the same inputs, it will produce the same output 100% of the time.

We still have that nasty `for` loop in there, so let's keep refactoring. The next step is to move `i` and `newList` to the argument list.



We want to break the recursion (`isDone`) when the `limit` reaches `0` (`limit` will count down during the recursive process) or when we reach the end of the list.

If we are not done, then we check to see if our filter `predicate` has a match. If we get a match then we'll call `takeFirst`, decrement `limit` and append to our `newList`. Otherwise, move onto the next item in the list.



If you have yet to read [Rethinking JavaScript: The if statement](#), it will explain this final step of replacing the `if`'s with a ternary operator.

[Rethinking JavaScript: The if statement](#)

[Thinking functionally has opened my mind about programming.medium.com](#)



Now we can call our new method like this:

```
const first5Kittens = takeFirst(5, isKitten, cats)
```

For extra credit we could curry `takeFirst` and use it to create other functions. (more on currying in another article)

```
const first5 = takeFirst(5)
const getFirst5Kittens = first5(isKitten)
```

```
const first5Kittens = getFirst5Kittens(cats)
```

Summary

There are many great libraries we can use like lodash, ramda or lazy.js. If we are so daring, we can even create our own methods using recursion!

I must warn that while `takeFirst` is super awesome, **recursion comes from a monkey's paw wish**. Recursion in JavaScript-land can be very dangerous and it's easy to get the infamous `Maximum call stack size exceeded` error message.

I'll be covering recursion in JavaScript in my next article. Stay tuned.

I know it's a small thing, but it makes my day when I get those follow notifications on Medium and Twitter ([@joelnet](#)). Or if you think I'm full of shit, tell me in the comments below.

Cheers!

32 : An Introduction to Functional JavaScript

<https://www.sitepoint.com/introduction-functional-javascript/>

You've heard that JavaScript is a functional language, or at least that it's capable of supporting functional programming. But what is functional programming? And for that matter, if you're going to start comparing programming paradigms in general, how is a functional approach different from the JavaScript that you've always written?

Well, the good news is that JavaScript isn't picky when it comes to paradigms. You can mix your imperative, object-oriented, prototypal, and functional code as you see fit, and still get the job done. But the bad news is what that means for your code. JavaScript can support a wide range of programming styles simultaneously within the same codebase, so it's up to you to make the right choices for maintainability, readability, and performance.

Functional JavaScript doesn't have to take over an entire project in order to add value. Learning a little about the functional approach can help guide some of the decisions you make as you build your projects, regardless of the way you prefer to structure your code. Learning some functional patterns and techniques can put you well on your way to writing cleaner and more elegant JavaScript regardless of your preferred approach.

Imperative JavaScript

JavaScript first gained popularity as an in-browser language, used primarily for adding simple hover and click effects to elements on a web page. For years, that's most of what people knew about it, and that contributed to the bad reputation JavaScript earned early on.

As developers struggled to match the flexibility of JavaScript against the intricacy of the browser document object model (DOM), actual JavaScript code often looked something like this in the real world:

```
var result;
function getText() {
    var someText = prompt("Give me something to capitalize");
    capWords(someText);
    alert(result.join(" "));
};

function capWords(input) {
    var counter;
    var inputArray = input.split(" ");
    var transformed = "";
    result = [];
    for (counter = 0; counter < inputArray.length; counter++) {
        transformed = [
            inputArray[counter].charAt(0).toUpperCase(),
            inputArray[counter].substring(1)
        ].join("");
        result.push(transformed);
    }
}
```

```
}

};

document.getElementById("main_button").onclick = getText;
```

So many things are going on in this little snippet of code. Variables are being defined on the global scope. Values are being passed around and modified by functions. DOM methods are being mixed with native JavaScript. The function names are not very descriptive, and that's due in part to the fact that the whole thing relies on a context that may or may not exist. But if you happened to run this in a browser inside an HTML document that defined a `<button id="main_button">`, you might get prompted for some text to work with, and then see the an `alert` with first letter of each of the words in that text capitalized.

Imperative code like this is written to be read and executed from top to bottom (give or take a little [variable hoisting](#)). But there are some improvements we could make to clean it up and make it more readable by taking advantage of JavaScript's object-oriented nature.

Object-Oriented JavaScript

After a few years, developers started to notice the problems with imperative coding in a shared environment like the browser. Global variables from one snippet of JavaScript clobbered global variables set by another. The order in which the code was called affected the results in ways that could be unpredictable, especially given the delays introduced by network connections and rendering times.

Eventually, some better practices emerged to help encapsulate JavaScript code and make it play better with the DOM. An updated variation of the same code above, written to an object-oriented standard, might look something like this:

```
(function() {
    "use strict";
    var SomeText = function(text) {
        this.text = text;
    };
    SomeText.prototype.capify = function(str) {
        var firstLetter = str.charAt(0);
        var remainder = str.substring(1);
        return [firstLetter.toUpperCase(), remainder].join("");
    };
    SomeText.prototype.capifyWords = function() {
        var result = [];
        var textArray = this.text.split(" ");
        for (var counter = 0; counter < textArray.length; counter++) {
            result.push(this.capify(textArray[counter]));
        }
        return result.join(" ");
    };
});
```

```
document.getElementById("main_button").addEventListener("click", function(e) {
    var something = prompt("Give me something to capitalize");
    var newText = new SomeText(something);
    alert(newText.capifyWords());
});
}());
```

In this object-oriented version, the constructor function simulates a class to model the object we want. Methods live on the new object's prototype to keep memory use low. And all of the code is isolated in an anonymous immediately-invoked function expression so it doesn't litter the global scope. There's even a "use strict" directive to take advantage of the latest JavaScript engine, and the old-fashioned `onclick` method has been replaced with a shiny new `addEventListener`, because who uses IE8 or earlier anymore? A script like this would likely be inserted at the end of the `<body>` element on an HTML document, to make sure all the DOM had been loaded before it was processed so the `<button>` it relies on would be available.

But despite all this reconfiguration, there are still many artifacts of the same imperative style that led us here. The methods in the constructor function rely on variables that are scoped to the parent object. There's a looping construct for iterating across all the members of the array of strings. There's a `counter` variable that serves no purpose other than to increment the progress through the `for` loop. And there are methods that produce the side effect of modifying variables that exist outside of their own definitions. All of this makes the code more brittle, less portable, and makes it harder to test the methods outside of this narrow context.

Functional JavaScript

The object-oriented approach is much cleaner and more modular than the imperative approach we started with, but let's see if we can improve it by addressing some of the drawbacks we discussed. It would be great if we could find ways to take advantage of JavaScript's built-in ability to treat functions as first-class objects so that our code could be cleaner, more stable, and easier to repurpose.

```
(function() {
    "use strict";
    var capify = function(str) {
        return [str.charAt(0).toUpperCase(), str.substring(1)].join("");
    };
    var processWords = function(fn, str) {
        return str.split(" ").map(fn).join(" ");
    };
    document.getElementById("main_button").addEventListener("click", function(e) {
        var something = prompt("Give me something to capitalize");
        alert(processWords(capify, something));
    });
}());
```

Did you notice how much shorter this version is? We're only defining two functions: `capify` and `processWords`. Each of these functions is pure, meaning that they don't rely on the state of the code they're called from. The functions don't create side effects that alter variables outside of themselves. There is one and only one result a function returns for any given set of arguments. Because of these improvements, the new functions are very easy to test, and could be snipped right out of this code and used elsewhere without any modifications.

There might have been one keyword in there that you wouldn't recognize unless you've peeked at some functional code before. We took advantage of the new [map method](#) on `Array` to apply a function to each element of the temporary array we created when we split our string. `Map` is just one of a handful of convenience methods we were given when modern browsers and server-side JavaScript interpreters implemented the ECMAScript 5 standards. Just using `map` here, in place of a `for` loop, eliminated the `counter` variable and helped make our code much cleaner and easier to read.

Start Thinking Functionally

You don't have to abandon everything you know to take advantage of the functional paradigm. You can get started thinking about your JavaScript in a functional way by considering a few questions when you write your next program:

- Are my functions dependent on the context in which they are called, or are they pure and independent?
- Can I write these functions in such a way that I could depend on them always returning the same result for a given input?
- Am I sure that my functions don't modify anything outside of themselves?
- If I wanted to use these functions in another program, would I need to make changes to them?

This introduction barely scratches the surface of functional JavaScript, but I hope it whets your appetite to learn more.

33 : Filtering and Chaining in Functional JavaScript

One of the things I appreciate about JavaScript is its versatility. JavaScript gives you the opportunity to use object oriented programming, imperative programming, and even functional programming. And you can go back and forth among them depending on your current needs and the preferences and expectations of your team.

Although JavaScript supports functional techniques, it's not optimized for pure functional programming the way a language such as Haskell or Scala is. While I don't usually structure my JavaScript programs to be 100 percent functional, I enjoy using functional programming concepts to help me keep my code clean and focus on designing code that can be reused easily and tested cleanly.

Filtering to Limit a Data Set

With the advent of ES5, JavaScript Arrays inherited a few methods that make functional programming even more convenient. JavaScript Arrays can now [map](#), [reduce](#), and [filter](#) natively. Each of these methods goes through every one of the items in an array, and without the need for a loop or local state changes, performs an analysis that can return a result that's ready to use immediately or pass-through to be operated on further.

In this article I want to introduce you to filtering. Filtering allows you to evaluate every item of an array, and based on a test condition you pass in, determine whether to return a new array that contains that element. When you use the `filter` method of Array, what you get back as another array that is either the same length as the original array or smaller, containing a subset of the items in the original that match the condition you set.

Using a Loop to Demonstrate Filtering

A simple example of the sort of problem that might benefit from filtering is limiting an array of strings to only the strings that have three characters. That's not a complicated problem to solve, and we can do it pretty handily using vanilla JavaScript `for` loops without the `filter` method. It might look something like this:

```
var animals = ["cat", "dog", "fish"];
var threeLetterAnimals = [];
for (let count = 0; count < animals.length; count++){
  if (animals[count].length === 3) {
    threeLetterAnimals.push(animals[count]);
  }
}
console.log(threeLetterAnimals); // ["cat", "dog"]
```

What we're doing here is defining an array containing three strings, and creating an empty array where we can store just the strings that only have three characters. We're defining a count variable to use in the `for` loop as we iterate through the array. Every time that we come across a string that has exactly three characters, we push it into our new empty array. And once we're done, we just log the result.

There's nothing stopping us from modifying the original array in our loop, but by doing that we would permanently lose the original values. It's much cleaner to create a new array and leave the original untouched.

Using the Filter Method

There's nothing technically wrong with the way that we did that, but the availability of the `filter` method on Array allows us to make our code much cleaner and straightforward. Here's an example of how we might've done the exact same thing using the `filter` method:

```
var animals = ["cat", "dog", "fish"];
var threeLetterAnimals = animals.filter(function(animal) {
  return animal.length === 3;
});
console.log(threeLetterAnimals); // ["cat", "dog"]
```

As before, we started with a variable that contains our original array, and we defined a new variable for the array that's going to contain just the strings that have three characters. But in this case, when we defined our second array, we assigned it directly to the result of applying the `filter` method to the original `animals` array. We passed `filter` an anonymous in-line function that only returned `true` if the value it was operating on had a length of three.

The way the `filter` method works, it goes through every element in the array and applies the test function to that element. If the test function returns `true` for that element, the array returned by the `filter` method will include that element. Other elements will be skipped.

You can see how much cleaner the code looks. Without even understanding ahead of time what `filter` does, you could probably look at this code and figure out the intention.

One of the happy by-products of functional programming is the cleanliness that results from reducing the amount of local state being stored, and limiting modification of external variables from within functions. In this case, the `count` variable and the various states that our `threeLetterAnimals` array was taking while we looped through the original array were simply more state to keep track of. Using `filter`, we've managed to eliminate the `for` loop as well as the `count` variable. And we're not altering the value of our new array multiple times the way we were doing before. We're defining it once, and assigning it the value that comes from applying our `filter` condition to the original array.

Other Ways to Format a Filter

Our code can be even more concise if we take advantage of `const` declarations and anonymous inline arrow functions. These are EcmaScript 6 (ES6) features that are supported now in most browsers and JavaScript engines natively.

```
const animals = ["cat", "dog", "fish"];
const threeLetterAnimals = animals.filter(item => item.length === 3);
console.log(threeLetterAnimals); // ["cat", "dog"]
```

While it's probably a good idea to move beyond the older syntax in most cases, unless you need to make your code match an existing codebase, it's important to be selective about it. As we get more concise, each line of our

code gets more complex.

Part of what makes JavaScript so much fun is how you can play with so many ways to design the same code to optimize for size, efficiency, clarity, or maintainability to suit your team's preferences. But that also puts a greater burden on teams to create shared style guides and discuss the pros and cons of each choice.

In this case, to make our code more readable and more versatile, we might want to take that anonymous in-line arrow function and turn it into a traditional named function, passing that named function right into the `filter` method. The code might look like this:

```
const animals = ["cat", "dog", "fish"];
function exactlyThree(word) {
  return word.length === 3;
}
const threeLetterAnimals = animals.filter(exactlyThree);
console.log(threeLetterAnimals); // ["cat", "dog"]
```

All we've done here is extract the anonymous in-line arrow function we defined above and turn it into a separate named function. As we can see, we have defined [a pure function](#) that takes the appropriate value type for the elements of the array, and returns the same type. We can just pass the name of that function directly to the `filter` method as a condition.

Quick Review of Map and Reduce

Filtering works hand-in-hand with two other functional Array methods from ES5, `map` and `reduce`. And thanks to the ability to chain methods in JavaScript, you can use this combination to craft very clean code that performs some pretty complex functions.

As a quick reminder, the `map` method goes through every element in an array and modifies it according to a function, returning a new array of the same length with modified values.

```
const animals = ["cat", "dog", "fish"];
const lengths = animals.map(getLength);
function getLength(word) {
  return word.length;
}
console.log(lengths); // [3, 3, 4]
```

The `reduce` method goes through an array and performs a series of operations, carrying the running result of those operations forward in an accumulator. When it's done, it returns a final result. In this case we're using the second argument to set the initial accumulator to 0.

```
const animals = ["cat", "dog", "fish"];
const total = animals.reduce(addLength, 0);
function addLength(sum, word) {
```

```

        return sum + word.length;
    }
    console.log(total); //10

```

All three of these methods leave the original array untouched, as they should for proper functional programming practice. If you want a reminder about how `map` and `reduce` work, you can check out my earlier article on [using map and reduce in functional JavaScript](#).

Chaining Map, Reduce, and Filter

As a very simple example of what's possible, let's imagine that you wanted to take an array of strings, and return a single string containing only the three letter strings from the original, but you wanted to format the resulting string in [StudyCaps](#). Without using `map`, `reduce`, and `filter`, you might try do it something like this:

```

const animals = ["cat", "dog", "fish"];
let threeLetterAnimalsArray = [];
let threeLetterAnimals;
let item;
for (let count = 0; count < animals.length; count++) {
    item = animals[count];
    if (item.length === 3) {
        item = item.charAt(0).toUpperCase() + item.slice(1);
        threeLetterAnimalsArray.push(item);
    }
}
threeLetterAnimals = threeLetterAnimalsArray.join("");
console.log(threeLetterAnimals); // "CatDog"

```

Of course this works, but as you can see we're creating a bunch of extra variables that we don't need, and maintaining the state of an array that's being changed as we go through our different loops. We can do better.

And in case you're wondering about the logic behind the variable declarations, I prefer to use `let` to declare an empty target array, although technically it could be declared as a `const`. Using `let` reminds me that the content of the array is going to be altered. Some teams may prefer to use `const` in cases like these, and it's a good discussion to have.

Let's create some pure functions that take strings and return strings. Then we can use those in a chain of `map`, `reduce`, and `filter` methods, passing the result from one onto the next this way:

```

const animals = ["cat", "dog", "fish"];
function studyCaps(words, word) {
    return words + word;
}
function exactlyThree(word) {

```

```
    return (word.length === 3);
}

function capitalize(word) {
  return word.charAt(0).toUpperCase() + word.slice(1);
}

const threeLetterAnimals = animals
  .filter(exactlyThree)
  .map(capitalize)
  .reduce(studlyCaps);
console.log(threeLetterAnimals); // "CatDog"
```

In this case we define three pure functions, `studlyCaps`, `exactlyThree`, and `capitalize`. We can pass these functions directly to `map`, `reduce`, and `filter` in a single unbroken chain. First we filter our original array with `exactlyThree`, then we map the result to `capitalize`, and finally we reduce the result of that with `studlyCaps`. And we're assigning the final result of that chain of operations directly to our new `threeLetterAnimals` variable with no loops and no intermediate state and leaving our original array untouched.

The resulting code is very clean and easy to test, and provides us with pure functions that we could easily use in other contexts or modify as requirements change.

Filtering and Performance

It's good to be aware that the `filter` method is likely to perform just a tiny bit slower than using a `for` loop until browsers and JavaScript engines optimize for the new Array methods ([jsPerf](#)).

As I've argued before, I recommend using these functional Array methods anyway, rather than using loops, even though they currently tend to be a little bit slower in performance. I favor them because they produce cleaner code. I always recommend writing code in the way that's the cleanest and most maintainable, and then optimizing only when real-world situations prove that you need better performance. For most use cases I can foresee, I wouldn't expect filter performance to be a significant bottleneck in a typical web application, but the only way you can be sure is to try it and find out.

The fact that filtering can be slightly slower than using a `for` loop is very unlikely to cause a noticeable performance issue in the real world. But if it does, and if your users are negatively impacted, you'll know exactly where and how to optimize. And the performance will only get better as the JavaScript engines optimize for these new methods.

Don't be afraid to start filtering today. The functionality is native in ES5, which is almost universally supported. The code you produce will be cleaner and easier to maintain. Using the `filter` method you can be confident that you won't alter the state of the array that you're evaluating. You will be returning a new array each time, and your original array will remain unchanged.

Agree? Disagree? Comments are welcome below.

34 : A Beginner's Guide to Currying in Functional JavaScript

<https://www.sitepoint.com/currying-in-functional-javascript/>

Currying, or partial application, is one of the functional techniques that can sound confusing to people familiar with more traditional ways of writing JavaScript. But when applied properly, it can actually make your functional JavaScript more readable.

More Readable And More Flexible

One of the advantages touted for functional JavaScript is shorter, tighter code that gets right to the point in the fewest lines possible, and with less repetition. Sometimes this can come at the expense of readability; until you're familiar with the way the functional programming works, code written in this way can be harder to read and understand.

If you've come across the term currying before, but never knew what it meant, you can be forgiven for thinking of it as some exotic, spicy technique that you didn't need to bother about. But currying is actually a very simple concept, and it addresses some familiar problems when dealing with function arguments, while opening up a range of flexible options for the developer.

What Is Currying?

Briefly, currying is a way of constructing functions that allows partial application of a function's arguments. What this means is that you can pass all of the arguments a function is expecting and get the result, or pass a subset of those arguments and get a function back that's waiting for the rest of the arguments. It really is that simple.

Currying is elemental in languages such as Haskell and Scala, which are built around functional concepts. JavaScript has functional capabilities, but currying isn't built in by default (at least not in current versions of the language). But we already know some functional tricks, and we can make currying work for us in JavaScript, too.

To give you a sense of how this could work, let's create our first curried function in JavaScript, using familiar syntax to build up the currying functionality that we want. As an example, let's imagine a function that greets somebody by name. We all know how to create a simple greet function that takes a name and a greeting, and logs the greeting with the name to the console:

```
var greet = function(greeting, name) {
  console.log(greeting + ", " + name);
};

greet("Hello", "Heidi"); // "Hello, Heidi"
```

This function requires both the name and the greeting to be passed as arguments in order to work properly. But we could rewrite this function using simple nested currying, so that the basic function only requires a greeting, and it returns another function that takes the name of the person we want to greet.

Our First Curry

```
var greetCurried = function(greeting) {
    return function(name) {
        console.log(greeting + ", " + name);
    };
};
```

This tiny adjustment to the way we wrote the function lets us create a new function for any type of greeting, and pass that new function the name of the person that we want to greet:

```
var greetHello = greetCurried("Hello");
greetHello("Heidi"); // "Hello, Heidi"
greetHello("Eddie"); // "Hello, Eddie"
```

We can also call the original curried function directly, just by passing each of the parameters in a separate set of parentheses, one right after the other:

```
greetCurried("Hi there")("Howard"); // "Hi there, Howard"
```

Why not try this out in your browser?

Curry All the Things!

The cool thing is, now that we have learned how to modify our traditional function to use this approach for dealing with arguments, we can do this with as many arguments as we want:

```
var greetDeeplyCurried = function(greeting) {
    return function(separator) {
        return function(emphasis) {
            return function(name) {
                console.log(greeting + separator + name + emphasis);
            };
        };
    };
};
```

We have the same flexibility with four arguments as we have with two. No matter how far the nesting goes, we can create new custom functions to greet as many people as we choose in as many ways as suits our purposes:

```
var greetAwkwardly = greetDeeplyCurried("Hello")("...")("?");
greetAwkwardly("Heidi"); // "Hello...Heidi?"
greetAwkwardly("Eddie"); // "Hello...Eddie?"
```

What's more, we can pass as many parameters as we like when creating custom variations on our original curried function, creating new functions that are able to take the appropriate number of additional parameters, each passed separately in its own set of parentheses:

```
var sayHello = greetDeeplyCurried("Hello")(", ");
sayHello(".")("Heidi"); // "Hello, Heidi."
sayHello(".")("Eddie"); // "Hello, Eddie."
```

And we can define subordinate variations just as easily:

```
var askHello = sayHello("?");
askHello("Heidi"); // "Hello, Heidi?"
askHello("Eddie"); // "Hello, Eddie?"
```

Currying Traditional Functions

You can see how powerful this approach is, especially if you need to create a lot of very detailed custom functions. The only problem is the syntax. As you build these curried functions up, you need to keep nesting returned functions, and call them with new functions that require multiple sets of parentheses, each containing its own isolated argument. It can get messy.

To address that problem, one approach is to create a quick and dirty currying function that will take the name of an existing function that was written without all the nested returns. A currying function would need to pull out the list of arguments for that function, and use those to return a curried version of the original function:

```
var curryIt = function(uncurried) {
  var parameters = Array.prototype.slice.call(arguments, 1);
  return function() {
    return uncurried.apply(this, parameters.concat(
      Array.prototype.slice.call(arguments, 0)
    ));
  };
};
```

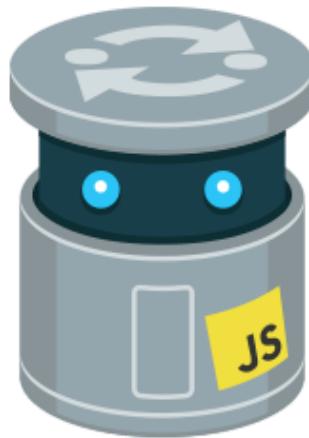
To use this, we pass it the name of a function that takes any number of arguments, along with as many of the arguments as we want to pre-populate. What we get back is a function that's waiting for the remaining arguments:

```
var greeter = function(greeting, separator, emphasis, name) {
  console.log(greeting + separator + name + emphasis);
};

var greetHello = curryIt(greeter, "Hello", ", ", ".");
greetHello("Heidi"); // "Hello, Heidi."
greetHello("Eddie"); // "Hello, Eddie."
```

And just as before, we're not limited in terms of the number of arguments we want to use when building derivative functions from our curried original function:

```
var greetGoodbye = curryIt(greeter, "Goodbye", ", ");
greetGoodbye(".", "Joe"); // "Goodbye, Joe."
```



Getting Serious about Currying

Our little currying function may not handle all of the edge cases, such as missing or optional parameters, but it does a reasonable job as long as we stay strict about the syntax for passing arguments.

Some functional JavaScript libraries such as [Ramda](#) have more flexible currying functions that can break out the parameters required for a function, and allow you to pass them individually or in groups to create custom curried variations. If you want to use currying extensively, this is probably the way to go.

Regardless of how you choose to add currying to your programming, whether you just want to use nested parentheses or you prefer to include a more robust carrying function, coming up with a consistent naming convention for your curried functions will help make your code more readable. Each derived variation of a function should have a name that makes it clear how it behaves, and what arguments it's expecting.

Argument Order

One thing that's important to keep in mind when currying is the order of the arguments. Using the approach we've described, you obviously want the argument that you're most likely to replace from one variation to the next to be the last argument passed to the original function.

Thinking ahead about argument order will make it easier to plan for currying, and apply it to your work. And considering the order of your arguments in terms of least to most likely to change is not a bad habit to get into anyway when designing functions.

Conclusion

Currying is an incredibly useful technique from functional JavaScript. It allows you to generate a library of small, easily configured functions that behave consistently, are quick to use, and that can be understood when reading your code. Adding currying to your coding practice will encourage the use of partially applied functions throughout your code, avoiding a lot of potential repetition, and may help get you into better habits about naming and dealing with function arguments.

If you enjoyed, this post, you might also like some of the others from the series:

- [An Introduction to Functional JavaScript](#)
- [Higher-Order Functions in JavaScript](#)
- [Recursion in Functional JavaScript](#)

35 : Higher-Order Functions in JavaScript

<https://www.sitepoint.com/higher-order-functions-javascript/>

One of the characteristics of JavaScript that makes it well-suited for functional programming is the fact that it can accept higher-order functions. A higher-order function is a function that can take another function as an argument, or that returns a function as a result.

First Class Functions

You may have heard it said that JavaScript treats functions as first-class citizens. What this means is that functions in JavaScript are treated as objects. They have the type `Object`, they can be assigned as the value of a variable, and they can be passed and returned just like any other reference variable.

This native ability gives JavaScript special powers when it comes to functional programming. Because functions are objects, the language supports a very natural approach to functional programming. In fact, it's so natural, that I'll bet you've been using it without even thinking about it.

Taking Functions as Arguments

If you've done much web-based JavaScript programming or front-end development, you've probably come across functions that use a callback. A callback is a function that gets executed at the end of an operation, once all of the other operations of been completed. Usually this callback function is passed in as the last argument in the function. Frequently, it's defined inline as an anonymous function.

Since JavaScript is single-threaded, meaning that only one operation happens at a time, each operation that's going to happen is queued along this single thread. The strategy of passing in a function to be executed after the rest of the parent function's operations are complete is one of the basic characteristics of languages that support higher-order functions. It allows for asynchronous behavior, so a script can continue executing while waiting for a result. The ability to pass a callback function is critical when dealing with resources that may return a result after an undetermined period of time.

This is very useful in a web programming environment, where a script may send an Ajax request off to a server, and then need to handle the response whenever it arrives, with no knowledge in advance of network latency or processing time on the server. Node.js frequently uses callbacks to make the most efficient use of server resources. This approach is also useful in the case of an app that waits for user input before performing a function.

For example, consider this snippet of simple JavaScript that adds an event listener to a button.

```
<button id="clicker">So Clickable</button>

document.getElementById("clicker").addEventListener("click", function() {
  alert("you triggered " + this.id);
});
```

This script uses an anonymous inline function to display an alert. But it could just as easily have used a separately defined function, and passed that named function to the `addEventListener` method

```
var proveIt = function() {
    alert("you triggered " + this.id);
};

document.getElementById("clicker").addEventListener("click", proveIt);
```

Note that we passed `proveIt` and not `proveIt()` to our `addEventListener` function. When you pass a function by name without parentheses, you are passing the function object itself. When you pass it with parentheses, you are passing the result of executing that function.

Our little `proveIt()` function is structurally independent of the code around it, always returning the `id` of whatever element was triggered. This bit of code could exist in any context in which you wanted to display an alert with the `id` of an element, and could be called with any event listener.

The ability to replace an inline function with a separately defined and named function opens up a world of possibilities. As we try to develop pure functions that don't alter external data, and return the same result for the same input every time, we now have one of the essential tools to help us develop a library of small, targeted functions that can be used generically in any application.

Returning Functions as Results

In addition to taking functions as arguments, JavaScript allows functions to return other functions as a result. This makes perfect sense, since functions are simply objects, they can be returned the same as any other value.

But what does it mean to return a function as a result? Defining a function as the return value of another function allows you to create functions that can be used as templates to create new functions. That opens the door to another world of functional JavaScript magic.

For example, imagine you've gotten tired of reading all of these articles about the specialness of *Millennials*, and you decide you want to replace the word *Millennials* with the phrase *Snake People* every time it occurs. Your impulse might be simply to write a function that performed that text replacement on any text you passed to it:

```
var snakify = function(text) {
    return text.replace(/millenials/ig, "Snake People");
};

console.log(snakify("The Millenials are always up to something."));
// The Snake People are always up to something.
```

That works, but it's pretty specific to this one situation. You're also tired of hearing about the *Baby Boomers*. You'd like to make a custom function for them as well. But even with such a simple function, you don't want to have to repeat the code that you've written:

```

var hippify = function(text) {
  return text.replace(/baby boomers/ig, "Aging Hippies");
};

console.log(hippify("The Baby Boomers just look the other way."));
// The Aging Hippies just look the other way.

```

But what if you decided that you wanted to do something fancier to preserve case in the original string? You would have to modify both of your new functions to do this. That's a hassle, and it makes your code more brittle and harder to read.

What you really want is the flexibility to be able to replace any term with any other term in a template function, and define that behavior as a foundational function from which you could spawn a whole race of custom functions.

With the ability to return functions instead of values, JavaScript offers up ways to make that scenario much more convenient:

```

var attitude = function(original, replacement, source) {
  return function(source) {
    return source.replace(original, replacement);
  };
};

var snakify = attitude(/millenials/ig, "Snake People");
var hippify = attitude(/baby boomers/ig, "Aging Hippies");

console.log(snakify("The Millenials are always up to something."));
// The Snake People are always up to something.
console.log(hippify("The Baby Boomers just look the other way."));
// The Aging Hippies just look the other way.

```

What we've done is isolate the code that does the actual work into a versatile and extensible `attitude` function that encapsulates all of the work needed to modify any input string using an original phrase and a replacement phrase with some attitude.

Defining a new function as a reference to the `attitude` function, pre-populated with the first two arguments it takes, allows the new function to take whatever argument you pass it and use that as the source text in the internal function returned by the `attitude` function.

We're taking advantage here of the fact that JavaScript functions don't care whether they are passed the same number of arguments as they were originally defined to take. If an argument is missing, the function will just treat the missing arguments as undefined.

On the other hand, that additional argument can be passed in later when the function being called has been defined in the way we just demonstrated, as a reference to a function returned from another function with an argument (or more) left undefined.

Go over that a few times if you need to, so you fully understand what's happening. We're creating a template function that returns another function. Then we're defining that newly returned function, minus one attribute, as a custom implementation of the template function. All the functions created this way will inherit the same working code from the template function, but can be predefined with different default arguments.

It's What You're Already Doing

Higher-order functions are so basic to the way JavaScript works, you're already using them. Every time you pass an anonymous function or a callback, you're actually taking the value that the passed function returns, and using that as an argument for another function.

The ability of functions to return other functions extends the convenience of JavaScript, allowing us to create custom named functions to perform specialized tasks with shared template code. Each of these little functions can inherit any improvements made in the original code down the road, which helps us avoid code duplication, and keeps our source clean and readable.

As an added bonus, if you make sure your functions are pure, so that they don't alter external values and they always return the same value for any given input, you put yourself in a good position to create test suites to verify that your code changes don't break anything you're relying on when you update your template functions.

Start thinking about ways you can take advantage of this approach in your own projects. One of the great things about JavaScript is that you can mix functional techniques right in with the code you're already familiar with. Try some experiments. You may be surprised how easily a little work with higher-order functions can improve your code.

36 : Recursion in Functional JavaScript

You may have come across references to recursive functions while programming in JavaScript. You may even have tried to construct (or deconstruct) a few yourself. But you probably haven't seen a lot of examples of effective recursion in the wild. In fact, other than the exotic nature of this approach, you may not have considered when and where recursion is useful, or how dangerous it can be if used carelessly.

What is Recursion Good For?

Recursion is a technique for iterating over an operation by having a function call itself repeatedly until it arrives at a result. Most loops can be rewritten in a recursive style, and in some functional languages this approach to looping is the default.

However, while JavaScript's functional coding style does support recursive functions, we need to be aware that most JavaScript compilers are not currently optimized to support them safely.

Recursion is best applied when you need to call the same function repeatedly with different parameters from within a loop. While it can be used in many situations, it is most effective for solving problems involving iterative branching, such as fractal math, sorting, or traversing the nodes of complex or non-linear data structures.

One reason that recursion is favored in functional programming languages is that it allows for the construction of code that doesn't require setting and maintaining state with local variables. Recursive functions are also naturally easy to test because they are easy to write in a pure manner, with a specific and consistent return value for any given input, and no side effects on external variable states.

Looping

The classic example of a function where recursion can be applied is the factorial. This is a function that returns the value of multiplying a number again and again by each preceding integer, all the way down to one.

For example, the factorial of three is:

$$3 \times 2 \times 1 = 6$$

The factorial of six is:

$$6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$$

You can see how quickly these results get big. You can also see that we're repeating the same behavior over and over. We take the result of one multiplication operation and multiply it again by one less than the second value. Then we do that again and again until we reach one.

Using a for loop, it's not difficult to create a function that will perform this operation iteratively until it returns the correct result:

```

var factor = function(number) {
    var result = 1;
    var count;
    for (count = number; count > 1; count--) {
        result *= count;
    }
    return result;
};
console.log(factor(6));
// 720

```

This works, but it isn't very elegant from a functional programming perspective. We have to use a couple of local variables that maintain and track state in order to support that for loop and then return a result. Wouldn't it be cleaner if we could ditch that for loop, and take a more functional JavaScript approach?

Recursion

We know [JavaScript will let us write functions that take functions as arguments](#). So what if we want to use the actual function we're writing and execute it in the context of running it.

Is that even possible? You bet it is! For example, take the case of a simple `while` loop like this:

```

var counter = 10;
while(counter > 0) {
    console.log(counter--);
}

```

When this is done, the value of `counter` has been changed, but the loop has done its job of printing out each value it held as we slowly sucked the state out of it.

A recursive version of the same loop might look more like this:

```

var countdown = function(value) {
    if (value > 0) {
        console.log(value);
        return countdown(value - 1);
    } else {
        return value;
    }
};
countdown(10);

```

Do you see how we're calling the `countdown` function right inside the definition of the `countdown` function? JavaScript handles that like a boss, and just does what you would hope. Every time `countdown` is executed,

JavaScript keeps track of where it was called from, and then works backward through that stack of function calls until it's finished. Our function has also avoided modifying the state of any variables, but has still taken advantage of a passed in value to control the recursion.

Getting back to our factorial case, we could rewrite our earlier function like this to use recursion:

```
var factorial = function(number) {  
    if (number <= 0) { // terminal case  
        return 1;  
    } else { // block to execute  
        return (number * factorial(number - 1));  
    }  
};  
console.log(factorial(6));  
// 720
```

Writing code this way allows us to describe the whole process in a stateless way with no side effects. Also worth noticing is the way we test the value of the argument being passed in to the function first thing, before doing any calculations. We want any functions that are going to call themselves to exit quickly and cleanly when they get to their terminal case. For a factorial calculated this way, the terminal case comes when the number passed in is zero or negative (we could also test for negative values and return a different message, if we so desired).

Tail Call Optimization

One problem with contemporary implementations of JavaScript is that they don't have a standard way to prevent recursive functions from stacking up on themselves indefinitely, and eating away at memory until they exceed the capacity of the engine. JavaScript recursive functions need to keep track of where they were called from each time, so they can resume at the correct point.

In many functional languages, such as Haskell and Scheme, this is managed using a technique called tail call optimization. With tail call optimization, each successive cycle in a recursive function would take place immediately, instead of stacking up in memory.

Theoretically, tail call optimization is part of the standard for ECMAScript 6, currently the next version of JavaScript, however it has [yet to be fully implemented](#) by most platforms.

Trampoline Functions

There are ways to force JavaScript to perform recursive functions in a safe manner when necessary. For example, it's possible to construct a custom trampoline function to manage recursive execution iteratively, keeping only one operation on the stack at a time. Trampoline functions used this way can take advantage of JavaScript's ability to bind a function to a specific context, so as to bounce a recursive function up against itself, building up results one at a time until the cycle is complete. This will avoid creating a deep stack of operations waiting to be performed.

In practice, making use of trampoline functions usually slows down performance in favor of safety. Additionally, much of the elegance and readability we obtain by writing our functions in a recursive manner gets lost in the code

convolutions necessary to make this approach work in JavaScript.

If you're curious, I encourage you to read more about this concept, and share your thoughts in the discussion below. You might start with a short [thread on StackOverflow](#), then explore some essays by [Don Taylor](#) and [Mark McDonnell](#) that dig deeper into the ups and downs of trampolines in JavaScript.

We're Not There Yet

Recursion is a powerful technique that's worth knowing about. In many cases, recursion is the most direct way to solve a complex problem. But until ECMAScript 6 is implemented everywhere we need it with tail call optimization, we will need to be very careful about how and where we apply recursion.

37 : How to Build and Structure a Node.js MVC Application

In a non-trivial application, the architecture is as important as the quality of the code itself. We can have well-written pieces of code, but if we don't have a good organization, we will have a hard time as the complexity increases. There is no need to wait until the project is half-way done to start thinking about the architecture; the best time is before starting, using our goals as beacons for our choices.

Node.js doesn't have a de facto framework with strong opinions on architecture and code organization in the same way that Ruby has the Rails framework, for example. As such, it can be difficult to get started with building full web applications with Node.

In this article, we are going to build the basic functionality of a note-taking app using the MVC architecture. To accomplish this we are going to employ the [Hapi.js](#) framework for [Node.js](#) and [SQLite](#) as a database, using [Sequelize.js](#), plus other small utilities to speed up our development. We are going to build the views using [Pug](#), the templating language.

What is MVC?

Model-View-Controller (or MVC) is probably one of the most popular architectures for applications. As with a lot of other [cool things](#) in computer history, the MVC model was conceived at [PARC](#) for the Smalltalk language as a solution to the problem of organizing applications with graphical user interfaces. It was created for desktop applications, but since then, the idea has been adapted to other mediums including the web.

We can describe the MVC architecture in simple words:

Model: The part of our application that will deal with the database or any data-related functionality.

View: Everything the user will see. Basically the pages that we are going to send to the client.

Controller: The logic of our site, and the glue between models and views. Here we call our models to get the data, then we put that data on our views to be sent to the users.

Our application will allow us to publish, see, edit and delete plain-text notes. It won't have other functionality, but because we will have a solid architecture already defined we won't have big trouble adding things later.

You can check out the final application in the [accompanying GitHub repository](#), so you get a general overview of the application structure.

Laying out the Foundation

The first step when building any Node.js application is to create a `package.json` file, which is going to contain all of our dependencies and scripts. Instead of creating this file manually, NPM can do the job for us using the `init` command:

```
npm init -y
```

After the process is complete will get a `package.json` file ready to use.

Note: If you're not familiar with these commands, checkout our [Beginner's Guide to npm](#).

We are going to proceed to install Hapi.js—the framework of choice for this tutorial. It provides a good balance between simplicity, stability and feature availability that will work well for our use case (although there are other options that would also work just fine).

```
npm install --save hapi hoek
```

This command will download the latest version of Hapi.js and add it to our package.json file as a dependency. It will also download the [Hoek](#) utility library that will help us write shorter error handlers, among [other things](#).

Now we can create our entry file; the web server that will start everything. Go ahead and create a `server.js` file in your application directory and all the following code to it:

```
'use strict';

const Hapi = require('hapi');
const Hoek = require('hoek');
const Settings = require('./settings');

const server = new Hapi.Server();
server.connection({ port: Settings.port });

server.route({
  method: 'GET',
  path: '/',
  handler: (request, reply) => {
    reply('Hello, world!');
  }
});

server.start((err) => {
  Hoek.assert(!err, err);

  console.log(`Server running at: ${server.info.uri}`);
});
```

This is going to be the foundation of our application.

First, we indicate that we are going to use [strict mode](#), which is a [common practice](#) when using the Hapi.js framework.

Next, we include our dependencies and instantiate a new server object where we set the connection port to `3000` (the port can be any number [above 1023 and below 65535](#).)

Our first route for our server will work as a test to see if everything is working, so a ‘Hello, world!’ message is enough for us. In each route, we have to define the HTTP method and path (URL) that it will respond to, and a handler, which is a function that will process the HTTP request. The handler function can take two arguments: `request` and `reply`. The first one contains information about the HTTP call, and the second will provide us with methods to handle our response to that call.

Finally, we start our server with the `server.start` method. As you can see, we can use Hoek to improve our error handling, making it shorter. This is completely optional, so feel free to omit it in your code, just be sure to handle any errors.

Storing Our Settings

It is good practice to store our configuration variables in a dedicated file. This file exports a JSON object containing our data, where each key is assigned from an environment variable—but without forgetting a fallback value.

In this file, we can also have different settings depending on our environment (e.g. development or production). For example, we can have an in-memory instance of SQLite for development purposes, but a real SQLite database file on production.

Selecting the settings depending on the current environment is quite simple. Since we also have an `env` variable in our file which will contain either `development` or `production`, we can do something like the following to get the database settings (for example):

```
const dbSettings = Settings[Settings.env].db;
```

So `dbSettings` will contain the setting of an in-memory database when the `env` variable is `development`, or will contain the path of a database file when the `env` variable is `production`.

Also, we can add support for a `.env` file, where we can store our environment variables locally for development purposes; this is accomplished using a package like [dotenv](#) for Node.js, which will read a `.env` file from the root of our project and automatically add the found values to the environment. You can find an example in the [dotenv repository](#).

Note: If you decide to also use a `.env` file, make sure you install the package with `npm install -s dotenv` and add it to `.gitignore` so you don't publish any sensitive information.

Our `settings.js` file will look like this:

```
// This will load our .env file and add the values to process.env,  
// IMPORTANT: Omit this line if you don't want to use this functionality  
require('dotenv').config({silent: true});  
  
module.exports = {  
  port: process.env.PORT || 3000,  
  env: process.env.ENV || 'development',
```

```
// Environment-dependent settings
development: {
  db: {
    dialect: 'sqlite',
    storage: ':memory:'
  }
},
production: {
  db: {
    dialect: 'sqlite',
    storage: 'db/database.sqlite'
  }
}
};
```

Now we can start our application by executing the following command and navigating to `localhost:3000` in our web browser.

```
node server.js
```

Note: This project was tested on Node v6. If you get any errors, ensure you have an updated installation.

Defining the Routes

The definition of routes gives us an overview of the functionality supported by our application. To create our additional routes, we just have to replicate the structure of the route that we already have in our `server.js` file, changing the content of each one.

Let's start by creating a new directory called `lib` in our project. Here we are going to include all the JS components. Inside `lib`, let's create a `routes.js` file and add the following content:

```
'use strict';

module.exports = [
  // We are going to define our routes here
];
```

In this file, we will export an array of objects that contain each route of our application. To define the first route, add the following object to the array:

```
{
  method: 'GET',
  path: '/',
```

```
handler: (request, reply) => {
  reply('All the notes will appear here');
},
config: {
  description: 'Gets all the notes available'
},
},
```

Our first route is for the home page (/) and since it will only return information we assign it a `GET` method. For now, it will only give us the message All the notes will appear here, which we are going to change later for a controller function. The `description` field in the `config` section is only for documentation purposes.

Then, we create the four routes for our notes under the /note/ path. Since we are building a [CRUD](#) application, we will need one route for each action with the corresponding [HTTP method](#).

Add the following definitions next to the previous route:

```
{
  method: 'POST',
  path: '/note',
  handler: (request, reply) => {
    reply('New note');
  },
  config: {
    description: 'Adds a new note'
  }
},
{
  method: 'GET',
  path: '/note/{slug}',
  handler: (request, reply) => {
    reply('This is a note');
  },
  config: {
    description: 'Gets the content of a note'
  }
},
{
  method: 'PUT',
  path: '/note/{slug}',
  handler: (request, reply) => {
    reply('Edit a note');
  },
  config: {
```

```

        description: 'Updates the selected note'
    }
},
{
    method: 'GET',
    path: '/note/{slug}/delete',
    handler: (request, reply) => {
        reply('This note no longer exists');
    },
    config: {
        description: 'Deletes the selected note'
    }
},

```

We have done the same as in the previous route definition, but this time we have changed the method to match the action we want to execute.

The only exception is the delete route. In this case, we are going to define it with the `GET` method rather than `DELETE` and add an extra `/delete` in the path. This way, we can call the delete action just by visiting the corresponding URL.

Note: If you plan to implement a strict REST interface, then you would have to use the `DELETE` method and remove the `/delete` part of the path.

We can name parameters in the path by surrounding the word in brackets (`{ ... }`). Since we are going to identify notes by a slug, we add `{slug}` to each path, with the exception of the PUT route; we don't need it there because we are not going to interact with a specific note, but to create one.

You can read more about Hapi.js routes on the [official documentation](#).

Now, we have to add our new routes to the `server.js` file. Let's import the routes file at the top of the file:

```
const Routes = require('./lib/routes');
```

and replace our current test route with the following:

```
server.route(Routes);
```

Building the Models

Models allow us to define the structure of the data and all the functions to work with it.

In this example, we are going to use the [SQLite](#) database with [Sequelize.js](#) which is going to provide us with a better interface using the ORM ([Object-Relational Mapping](#)) technique. It will also provide us a database-independent interface.

Setting up the database

For this section we are going to use [Sequelize.js](#) and [SQLite](#). You can install and include them as dependencies by executing the following command:

```
npm install -s sequelize sqlite3
```

Now create a `models` directory inside `lib/` with a file called `index.js` which is going to contain the database and Sequelize.js setup, and include the following content:

```
'use strict';

const Fs = require('fs');
const Path = require('path');
const Sequelize = require('sequelize');
const Settings = require('../settings');

// Database settings for the current environment
const dbSettings = Settings[Settings.env].db;

const sequelize = new Sequelize(dbSettings.database, dbSettings.user,
dbSettings.password, dbSettings);
const db = {};

// Read all the files in this directory and import them as models
Fs.readdirSync(__dirname)
.filter((file) => (file.indexOf('.') !== 0) && (file !== 'index.js'))
.forEach((file) => {
  const model = sequelize.import(Path.join(__dirname, file));
  db[model.name] = model;
});

db.sequelize = sequelize;
db.Sequelize = Sequelize;

module.exports = db;
```

First, we include the modules that we are going to use:

- `Fs`, to read the files inside the `models` folder, which is going to contain all the models.
- `Path`, to join the path of each file in the current directory.
- `Sequelize`, that will allow us to create a new Sequelize instance.
- `Settings`, which contains the data of our `settings.js` file from the root of our project.

Next, we create a new `sequelize` variable that will contain a `Sequelize` instance with our database settings for the current environment. We are going to use `sequelize` to import all the models and make them available in our `db` object.

The `db` object is going to be exported and will contain our database methods for each model; it will be available in our application when we need to do something with our data.

To load all the models, instead of defining them manually, we look for all the files inside the `models` directory (with the exception of the `index.js` file) and load them using the `import` function. The returned object will provide us with the CRUD methods, which we then add to the `db` object.

At the end, we add `sequelize` and `Sequelize` as part of our `db` object, the first one is going to be used in our `server.js` file to connect to the database before starting the server, and the second one is included for convenience if you need it in other files too.

Creating our Note model

In this section we are going to use the [Moment.js](#) package to help with Date formatting. You can install it and include it as a dependency with the following command:

```
npm install -s moment
```

Now, we are going to create a `note.js` file inside the `models` directory, which is going to be the only model in our application; it will provide us with all the functionality we need.

Add the following content to that file:

```
'use strict';

const Moment = require('moment');

module.exports = (sequelize, DataTypes) => {
  let Note = sequelize.define('Note', {
    date: {
      type: DataTypes.DATE,
      get: function () {
        return Moment(this.getDataValue('date')).format('MMMM Do, YYYY');
      }
    },
    title: DataTypes.STRING,
    slug: DataTypes.STRING,
    description: DataTypes.STRING,
    content: DataTypes.STRING
  });
}
```

```

    return Note;
};


```

We export a function that accepts a `sequelize` instance, to define the model, and a `DataTypes` object with all the types available in our database.

Next, we define the structure of our data using an object where each key corresponds to a database column and the value of the key defines the type of data that we are going to store. You can see the list of data types in the [Sequelize.js documentation](#). The tables in the database are going to be created automatically based on this information.

In the case of the date column, we also define a how Sequelize should return the value using a `getter` function (`get` key). We indicate that before returning the information it should be first passed through the Moment utility to be formatted in a more readable way (`MMM Do, YYYY`).

Note: Although we are getting a simple and easy to read date string, it is stored as a precise date string product of the `Date` object of JavaScript. So this is not a destructive operation.

Finally, we return our model.

Synchronizing the Database

Now, we have to synchronize our database before we are able to use it in our application. In `server.js`, import the models at the top of the file:

```

// Import the index.js file inside the models directory
const Models = require('../lib/models');

```

Next, replace the following code block:

```

server.start((err) => {
  Hoek.assert(!err, err);
  console.log(`Server running at: ${server.info.uri}`);
});

```

with this one:

```

Models.sequelize.sync().then(() => {
  server.start((err) => {
    Hoek.assert(!err, err);
    console.log(`Server running at: ${server.info.uri}`);
  });
});

```

This code is going to synchronize the models to our database and, once that is done, the server will be started.

Building the controllers

Controllers are functions that accept the `request` and `reply` objects from Hapi.js. The `request` object contains information about the requested resource and we use `reply` to return information to the client.

In our application, we are going to return only a JSON object for now, but we will add the views once we build them.

We can think of controllers as functions that will join our models with our views; they will communicate with our models to get the data, and then return that data inside a view.

The Home Controller

The first controller that we are going to build will handle the home page of our site. Create a `home.js` file inside the `lib/controllers` directory with the following content:

```
'use strict';

const Models = require('../models');

module.exports = (request, reply) => {
  Models.Note
    .findAll({
      order: [['date', 'DESC']]
    })
    .then((result) => {
      reply({
        data: {
          notes: result
        },
        page: 'Home-Notes Board',
        description: 'Welcome to my Notes Board'
      });
    });
};

};
```

First, we get all the notes in our database using the `findAll` method of our model. This function will return a promise and, if it resolves, we will get an array containing all the notes in our database.

We can arrange the results in descending order, using the `order` parameter in the options object passed to the `findAll` method, so the last item will appear first. You can check all the available options in the [Sequelize.js documentation](#).

Once we have the home controller, we can edit our `routes.js` file. First, we import the module at the top of the file, next to the `Path` module import:

```
const Home = require('./controllers/home');
```

Then we add the controller we just made to the array:

```
{
  method: 'GET',
  path: '/',
  handler: Home,
  config: {
    description: 'Gets all the notes available'
  }
},
```

Boilerplate of the Note Controller

Since we are going to identify our notes with a slug, we can generate one using the title of the note and the [slug](#) library, so let's install it and include it as a dependency with the following command:

```
npm install -s slug
```

The last controller that we have to define in our application will allow us to create, read, update, and delete notes.

We can proceed to create a `note.js` file inside the `lib/controllers` directory and add the following content:

```
'use strict';

const Models = require('../models/');
const Slugify = require('slug');
const Path = require('path');

module.exports = {
  // Here we are going to include our functions that will handle each request in the
  routes.js file.
};
```

The “create” function

To add a note to our database, we are going to write a `create` function that is going to wrap the `create` method on our model using the data contained in the payload object.

Add the following inside the object that we are exporting:

```

create: (request, reply) => {
  Models.Note
    .create({
      date: new Date(),
      title: request.payload.noteTitle,
      slug: Slugify(request.payload.noteTitle, {lower: true}),
      description: request.payload.noteDescription,
      content: request.payload.noteContent
    })
    .then((result) => {
      // We are going to generate a view later, but for now lets just return the result.
      reply(result);
    });
},

```

Once the note is created, we will get back the note data and send it to the client as JSON using the `reply` function.

For now, we just return the result, but once we build the views in the next section, we will be able to generate the HTML with the new note and add it dynamically on the client. Although this is not completely necessary and will depend on how you are going to handle your front-end logic, we are going to return an HTML block to simplify the logic on the client.

Also, note that the date is being generated on the fly when we execute the function, using `new Date()`.

The “read” function

To search just one element we use the `findOne` method on our model. Since we identify notes by their slug, the `where` filter must contain the slug provided by the client in the URL (`http://localhost:3000/note/:slug:`).

```

read: (request, reply) => {
  Models.Note
    .findOne({
      where: {
        slug: request.params.slug
      }
    })
    .then((result) => {
      reply(result);
    });
},

```

As in the previous function, we will just return the result, which is going to be an object containing the note information. The views are going to be used once we build them in the [Building the Views](#) section.

The “update” function

To update a note, we use the `update` method on our model. It takes two objects, the new values that we are going to replace and the options containing a `where` filter with the note slug, which is the note that we are going to update.

```
update: (request, reply) => {
  const values = {
    title: request.payload.noteTitle,
    description: request.payload.noteDescription,
    content: request.payload.noteContent
  };

  const options = {
    where: {
      slug: request.params.slug
    }
  };

  Models.Note
    .update(values, options)
    .then(() => {
      Models.Note
        .findOne(options)
        .then((result) => {
          reply(result);
        });
    });
},
}
```

After updating our data, since our database won't return the updated note, we can find the modified note again to return it to the client, so we can show the updated version as soon as the changes are made.

The “delete” function

The delete controller will remove the note by providing the slug to the `destroy` function of our model. Then, once the note is deleted, we redirect to the home page. To accomplish this, we use the [redirect](#) function of the Hapi.js `reply` object.

```
delete: (request, reply) => {
  Models.Note
    .destroy({
      where: {
        slug: request.params.slug
      }
    });
  reply.redirect('/');
},
}
```

```
        }
    })
    .then(() => reply.redirect('/'));
}
```

Using the Note controller in our routes

At this point, we should have our note controller file ready with all the CRUD actions. But to use them, we have to include it in our routes file.

First, let's import our controller at the top of the `routes.js` file:

```
const Note = require('./controllers/note');
```

We have to replace each handler with our new functions, so we should have our routes file as follows:

```
{
  method: 'POST',
  path: '/note',
  handler: Note.create,
  config: {
    description: 'Adds a new note'
  },
  {
    method: 'GET',
    path: '/note/{slug}',
    handler: Note.read,
    config: {
      description: 'Gets the content of a note'
    },
  },
  {
    method: 'PUT',
    path: '/note/{slug}',
    handler: Note.update,
    config: {
      description: 'Updates the selected note'
    },
  },
  {
    method: 'GET',
    path: '/note/{slug}/delete',
```

```

    handler: Note.delete,
    config: {
      description: 'Deletes the selected note'
    }
},

```

Note: we are including our functions without `()` at the end, that is because we are referencing our functions without calling them.

Building the Views

At this point, our site is receiving HTTP calls and responding with JSON objects. To make it useful to everybody we have to create the pages that render our information in a nice way.

In this example, we are going to use the Pug templating language, although this is not mandatory and we can use [other languages](#) with Hapi.js. Also, we are going to use the [Vision](#) plugin to enable the view functionality in our server.

Note: If you're not familiar with Pug (formerly Jade), see our [Jade Tutorial for Beginners](#).

You can install the packages with the following command:

```
npm install -s vision pug
```

The note component

First, we are going to build our note component that is going to be reused across our views. Also, we are going to use this component in some of our controller functions to build a note on the fly in the back-end to simplify the logic on the client.

Create a file in `lib/views/components` called `note.pug` with the following content:

```

article
  h2: a(href=`/note/${note.slug}`)= note.title
  small Published on #{note.date}
  p= note.content

```

It is composed of the title of the note, the publication date and the content of the note.

The base layout

The base layout contains the common elements of our pages; or in other words, for our example, everything that is not content. Create a file in `lib/views/` called `layout.pug` with the following content:

```

doctype html
html(lang='en')
head
  meta(charset='utf-8')
  meta(http-equiv='x-ua-compatible' content='ie=edge')
  title= page
  meta(name='description' content=description)
  meta(name='viewport' content='width=device-width, initial-scale=1')

  link(href='https://fonts.googleapis.com/css?
family=Gentium+Book+Basic:400,400i,700,700i|Ubuntu:500' rel='stylesheet')
  link(rel='stylesheet' href='/styles/main.css')

body
  block content

  script(src='https://code.jquery.com/jquery-3.1.1.min.js' integrity='sha256-
hVVnYaiADRT02PzUGmuLJr8BLUSjGIZsDYGmIJLv2b8=' crossorigin='anonymous')
  script(src='/scripts/jquery.modal.js')
  script(src='/scripts/main.js')

```

The content of the other pages will be loaded in place of `block content`. Also, note that we will display a `page` variable in the `title` element, and a `description` variable in the `meta(name='description')` element. We will create those variables in our routes later.

We also include, at the bottom of the page, three JS files, [jQuery](#), [jQuery Modal](#) and a `main.js` file which will contain all of our custom JS code for the front-end. Be sure to download those packages and put them in a `static/public/scripts/` directory. We are going to make them public in the [Serving Static Files](#) section.

The home view

On our home page, we will show a list containing all the notes in our database and a button that will show a modal window with a form that allows us to create a new note via Ajax.

Create a file in `lib/views` called `home.pug` with the following content:

```

extends layout

block content
  header(container)
    h1 Notes Board

  nav
    ul
      // This will show a modal window with a form to send new notes

```

```

    li: a(href='#note-form' rel='modal:open') Publish

main(container).notes-list
  // We loop over all the notes received from our controller rendering our note
  component with each entry
  each note in data.notes
    include components/note

  // Form to add a new note, this is used by our controller `create` function.
form(action='/note' method='POST').note-form#note-form
  p: input(name='noteTitle' type='text' placeholder='Title...')
  p: input(name='noteDescription' type='text' placeholder='Short description...')
  p: textarea(name='noteContent') Write here the content of the new note...
  p._text-right: input(type='submit' value='Submit')

```

The note view

The note page is pretty similar to the home page, but in this case, we show a menu with options specific to the current note, the content of the note and the same form as in the home page but with the current note information already filled, so it's there when we update it.

Create a file in `lib/views` called `note.pug` with the following content:

```

extends layout

block content
  header(container)
    h1 Notes Board

  nav
    ul
      li: a(href='/') Home
      li: a(href='#note-form' rel='modal:open') Update
      li: a(href=`/note/${note.slug}/delete`) Delete

  main(container).note-content
    include components/note

  form(action=`/note/${note.slug}` method='PUT').note-form#note-form
    p: input(name='noteTitle' type='text' value=note.title)
    p: input(name='noteDescription' type='text' value=note.description)
    p: textarea(name='noteContent')= note.content
    p._text-right: input(type='submit' value='Update')

```

The JavaScript on the client

To create and update notes we use the Ajax functionality of jQuery. Although this is not strictly necessary, I feel it provides a better experience for the user.

This is the content of our `main.js` file in the `static/public/scripts/` directory:

```
$('#note-form').submit(function (e) {
    e.preventDefault();

    var form = {
        url: $(this).attr('action'),
        type: $(this).attr('method')
    };

    $.ajax({
        url: form.url,
        type: form.type,
        data: $(this).serialize(),
        success: function (result) {
            $.modal.close();

            if (form.type === 'POST') {
                $('.notes-list').prepend(result);
            } else if (form.type === 'PUT') {
                $('.note-content').html(result);
            }
        }
    });
});
```

Every time the user submits the form in the modal window, we get the information from the form elements and send it to our back-end, depending on the action URL and the method (`POST` or `PUT`). Then, we will get the result as a block of HTML containing our new note data. When we add a note, we will just add it on top of the list on the home page, and when we update a note we replace the content for the new one in the note view.

Adding support for views on the server

To make use of our views, we have to include them in our controllers and add the required settings.

In our `server.js` file, let's import the Node [Path](#) utility at the top of the file, since we are using it in our code to indicate the path of our views.

```
const Path = require('path');
```

Now, replace the `server.route(Routes);` line with the following code block:

```
server.register([
  require('vision')
], (err) => {
  Hoek.assert(!err, err);

  // View settings
  server.views({
    engines: { pug: require('pug') },
    path: Path.join(__dirname, 'lib/views'),
    compileOptions: {
      pretty: false
    },
    isCached: Settings.env === 'production'
  });

  // Add routes
  server.route(Routes);
});
```

In the code we have added, we first register the [Vision](#) plugin with our Hapi.js server, which is going to provide the view functionality. Then, we add the settings for our views—like the engine that we are going to use and the path where the views are located. At the end of the code block, we add again our routes.

This will make work our views on the server, but we still have to declare the view that we are going to use for each route.

Setting the home view

Open the `lib/controllers/home.js` file and replace the `reply(result);` line with the following:

```
reply.view('home', {
  data: {
    notes: result
  },
  page: 'Home-Notes Board',
  description: 'Welcome to my Notes Board'
});
```

After registering the Vision plugin, we now have a `view` method available on the `reply` object, we are going to use it to select the `home` view in our `views` directory and to send the data that is going to be used when rendering the views.

In the data that we provide to the view, we also include the page title and a meta description for search engines.

Setting the note view: create function

Right now, every time we create a note we get a JSON object from the server to the client. But since we are doing this process with Ajax, we can send the new note as HTML ready to be added to the page. To do this, we render the *note* component with the data we have. Replace the line `reply(result);` with the following code block:

```
// Generate a new note with the 'result' data
const newNote = Pug.renderFile(
  Path.join(__dirname, '../views/components/note.pug'),
  {
    note: result
  }
);

reply(newNote);
```

We use the `renderFile` method from Pug to render the note template with the data we just received from our model.

Setting the note view: read function

When we enter a note page, we should get the note template with the content of our note. To do this, we have to replace the `reply(result);` line with this:

```
reply.view('note', {
  note: result,
  page: `${result.title}–Notes Board`,
  description: result.description
});
```

As with the home page, we select a view as the first parameter and the data that we are going to use as the second one.

Setting the note view: update function

Every time we update a note, we will reply similarly to when we create new notes. Replace the `reply(result);` line with the following code:

```
// Generate a new note with the updated data
const updatedNote = Pug.renderFile(
  Path.join(__dirname, '../views/components/note.pug'),
  {
```

```

    note: result
  }
);

reply(updatedNote);

```

Note: The delete function doesn't need a view, since it will just redirect to the home page once the note is deleted.

Serving Static Files

The JavaScript and CSS files that we are using on the client side are provided by Hapi.js from the `static/public/` directory. But it won't happen automatically; we have to indicate to the server that we want to define this folder as public. This is done using the [Inert](#) package, which you can install with the following command:

```
npm install -s inert
```

In the `server.register` function inside the `server.js` file, import the Inert plugin and register it with Hapi like this:

```

server.register([
  require('vision'),
  require('inert')
], (err) => {

```

Now we have to define the route where we are going to provide the static files, and their location on our server's filesystem. Add the following entry at the end of the exported object in `routes.js`:

```

{
  // Static files
  method: 'GET',
  path: '/{param*}',
  handler: {
    directory: {
      path: Path.join(__dirname, '../static/public')
    }
  },
  config: {
    description: 'Provides static resources'
  }
}

```

This route will use the `GET` method and we have replaced the handler function with an object containing the directory that we want to make public.

You can find more information about serving static content in the [Hapi.js documentation](#).

Note: Check the [Github repository](#) for the rest of the [static files](#), like the [main stylesheet](#).

Conclusion

At this point, we have a very basic Hapi.js application using the MVC architecture. Although there are still things that we should take care of before putting our application in production (e.g. input validation, error handling, error pages, etc.) this should work as a foundation to learn and build your own applications.

If you would like to take this example a bit further, after finishing all the small details (not related the architecture) to make this a robust application, you could implement an authentication system so only registered users are able to publish and edit notes. But your imagination is the limit, so feel free to fork the [application repository](#), experiment with your ideas and let me know what you create in the comments below!

This article was peer reviewed by [Mark Brown](#) and [Vildan Softic](#). Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

38 : A Beginner's Guide to JavaScript Variables and Datatypes

A Beginner's Guide to JavaScript Variables and Datatypes was peer reviewed by [Scott Molinari](#) and [Vildan Softic](#) and [Chris Perry](#). Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

So you've decided to learn JavaScript, the programming language of the web. If it seems like a daunting journey ahead and you don't know where to start, here's a little secret: it doesn't take any special skill to learn how to program, and everyone starts at zero. Take it one step at a time, and you'll get there.

Is This Guide For Me?

If any of these apply to you, you'll benefit from reading this guide:

- You've never used a programming language before.
- You've never used JavaScript before.
- You've tried learning JavaScript before but found the resources lacking or hard to follow.
- You know a bit of JavaScript, but want to brush up on the basics.

In this article, we're going to focus on the fundamentals: syntax, variables, comments, and datatypes. The beauty is that you can apply the concepts you learn about JavaScript here to learning another programming language in the future.

Disclaimer: This guide is intended for complete beginners to JavaScript and programming. As such, many concepts will be presented in a simplified manner, and strict ES5 syntax will be used.

Ready? Let's get started!

What is JavaScript?

JavaScript is the programming language used to make websites dynamic and interactive. It's a **client-side** programming language, which means the code gets executed in the user's web browser. With the advent of [Node.js](#) and other technologies, it can also be used as a **server-side** language, making it extremely versatile. JavaScript is used primarily for **front-end web development** and works closely with HTML and CSS.

Note: Java is not JavaScript. It's a different language with a confusingly similar name.

Requirements

You already have the prerequisites to start writing and using JavaScript. All you need is a **web browser** to view the code, and a **text editor** to write it. The browser you're currently using is perfect (Chrome, Firefox, Safari, Edge, etc). Your computer comes preinstalled with Notepad (Windows) or TextEdit (Mac), but I would recommend installing [Atom](#) or [Brackets](#), which are free programs specifically designed for coding.

[CodePen](#) is a website that allows you to write code and make live demos, and it will be the easiest way to start following along and practicing.

Basic Terminology

A programmer writes programs, just as an author writes a book.

A **program** is just a set of instructions that a computer can read and use to perform a task. Each individual instruction is a line of code known as a **statement**, which is similar to a sentence in a book. While a sentence in English ends with a period, a JavaScript statement usually ends with a semicolon. **Syntax** refers to the symbols and rules that define the structure of the language, similar to grammar and punctuation, and the semicolon that ends a JavaScript statement is part of the syntax.

Comments

A **comment** is a human-readable note written in the code.

Comments are written in plain English with the purpose of explaining the code. Although comments don't technically perform any function in the program, it's crucial to get into the habit of proper documentation to help you or future collaborators understand the intent of your code.

There are two types of comment in JavaScript:

- A **single line comment**, written as two forward slashes `//` followed by the comment.

```
// This is a single line comment.
```

- A **multi-line comment**, which is sandwiched between `/*` and `*/` and can span many lines.

```
/* This is a comment.  
It's a multi-line comment.  
Also a haiku. */
```

Variables

A **variable** is a container that stores data values.

You know a [variable](#) as something that can change. In basic algebra, it's a letter that represents a number. `x` is a common variable name, but it can just as easily be represented by `y`, `z`, or another name.

Initially `x` has no value or meaning, but you can apply a value to it.

```
x = 5
```

Now, `x` represents `5`. You can think of `x` as a container that's storing `5`, which is a number.

In JavaScript, variables work the same, except they can contain more than just numbers as a value – they can contain all sorts of data values, which we'll learn by the end of this article.

Variables are created and declared using the `var` keyword. We can use our algebra example above to create a JavaScript statement.

```
var x = 5; // the variable x contains the numeric value of 5.
```

Building on what we've learned so far, you can see that we have a JavaScript statement that declares a variable (`x`), assigns the number data type (`5`) with a single equals sign (`=`), and explains it in plain English with a comment (`//`). The statement ends with a semi-colon (`;`).

Variables only need to be declared with `var` the first time they're used, and as the name suggests, a variable can change.

```
var x = 5; // x was worth 5  
x = 6; // now it's worth 6
```

A variable can only contain one value at a time, and since the program is executed from top to bottom, the value of `x` is now `6`.

Here is an example of a variable with a different data type.

```
var greeting = "Oh hi, Mark!";
```

Now the `greeting` variable contains the string `Oh hi, Mark!`.

A few important things to know about variables:

- A variable name can have letters, numbers, a dollar sign (`$`), and an underscore (`_`), but cannot begin with a number.
- A variable cannot be any word on the list of [reserved keywords](#).
- Variables are case sensitive.
- The naming convention is **camelCase**, in which a variable always starts in lowercase, but each subsequent word starts with an uppercase letter.

Tip: Although a variable can have any name, it's important to choose names that are descriptive yet concise.

Datatypes

Now that we know about variables, we can learn about the types of data that a variable can hold.

A **data type** is a classification of data. Programming languages need to have different datatypes to interact properly with values. You can do math with a number, but not with a sentence, so the computer classifies them differently. There are six **primitive** (basic) datatypes: strings, numbers, Boolean, null, undefined, and Symbol (new in ES6). Primitives can only hold a single value. Anything that is not one of these primitives is an **Object**. Objects can contain multiple values.

JavaScript is known as a **weakly-typed language** (or *loosely-typed*), which means the language will automatically determine the data type based on the syntax you use.

Testing

`alert()` and `console.log()` are two easy ways to print a value in JavaScript.

```
var x = 5;  
  
alert(x);  
console.log(x);
```

[Here](#) is a demo of these methods. An `alert` is a popup window and `console.log` will print to the Inspector, which you can find by right-clicking in the browser and selecting **Inspect**. I won't reference these again throughout the article, so you can choose whichever works best for you to practice. I would recommend avoiding the `alert`, as it's very annoying.

You can always find out the type of a variable using `typeof`.

```
var answer = 42;  
  
typeof answer // returns number
```

Strings

A **string** is a series of characters.

Any data that contains text will be represented by a string. The name *string* likely originated from early programmers who were reminded of beads on a string.

- A string can be wrapped in **double quotes** (" "):

```
"Pull the string, and it will follow wherever you wish." // double quoted string
```

- or **single quotes** (' '):

```
'Push it, and it will go nowhere at all.' // single quoted string
```

Note: Single quotes are what you would usually call an apostrophe, not to be confused with a backtick, which is located all the way on the left of the keyboard.

Both ends of the string must match, but otherwise, there is no difference between the two – they're just different ways of writing a string.

But what if I want to write *I'm* in a single quoted string, or quote someone in a double-quoted string? Won't that break the string?

It will, and there are two options to combat that. You can safely use the opposite type of quotes:

```
"Damn it, man, I'm a doctor, not a torpedo technician!"; // using single quotes within a double quoted string
'"Do. Or do not. There is no try." - Yoda'; // using double quotes in a single quoted string
```

It's important throughout a project to choose one style for strings for consistency. You can use a backslash (\) to escape the string.

```
'Damn it, man, I\'m a doctor, not a torpedo technician!';
"\\"Do. Or do not. There is no try.\\" - Yoda";
```

We can apply strings to variables. Let's use my `greeting` example.

```
var greeting = "Oh hi, Mark!";
```

Strings can also be linked together in a process known as **concatenation**. Variables and strings can be joined together using the plus (+) symbol.

```
var greeting = "Oh hi, " + "Mark" + "!"; // returns Oh hi, Mark!
```

Note that an empty space is also a character in a string. In the below example, we'll see how concatenation can be useful.

```
var message = "Oh hi, ";
var firstName = "Mark";
var bam = "!";

var greeting = message + firstName + bam; // returns Oh hi, Mark!
```

Now all the strings are represented by variables which can change. If you've ever logged into an app (such as your email) and were greeted with your name, you can see how your name is just a string in a variable in their system.

Numbers

A **number** is a numerical value.

Numbers do not have any special syntax associated with them like strings do. If you were to place a number in quotes ("5"), it wouldn't be considered a number, but a character in a string. A number can be whole or a decimal (known as a float) and can have a positive or negative value.

```
var x = 5; // whole integer
var y = 1.2; // float
```

```
var z = -76; // negative whole integer
```

A number can be up to 15 digits.

```
var largeNumber = 999999999999999; // a valid number
```

You can do regular math with numbers – addition (+), subtraction (-), division (/) and multiplication (*).

```
var sum = 2 + 5; // returns 7  
var difference = 6 - 4; // returns 2
```

You can do math with variables.

```
var elves = 3;  
var dwarves = 7;  
var men = 9;  
var sauron = 1;  
  
var ringsOfPower = elves + dwarves + men + sauron; // returns 20
```

There are two other special number types.

NaN

NaN means Not a Number, even though it's technically a number type. **NaN** is the result of attempting to do impossible math with other datatypes.

```
var nope = 1 / "One"; // returns NaN
```

Attempting to divide a number by a string, for example, results in **NaN**.

Infinity

Infinity is also technically a number, either the product of dividing by **0** or calculating a number too large.

```
var beyond = 1 / 0; // returns Infinity
```

Boolean

A **Boolean** is a value that is either true or false.

Boolean values are very commonly used in programming for when a value might switch between yes and no, on and off, or true and false. Booleans can represent the current state of something that is likely to change.

For example, we'd use a Boolean to indicate whether a checkbox is checked or not.

```
var isChecked = true;
```

Often, Booleans are used to check if two things are equal, or if the result of a math equation is true or false.

```
/* Check if 7 is greater than 8 */  
7 > 8; // returns false  
  
/* Check if a variable is equal to a string */  
var color = "Blue";  
  
color === "Blue"; // returns true
```

Note: A single equals sign (=) applies one value to another. A double (==) or triple equals sign (===) checks if two things are equal.

Undefined

An **undefined** variable has no value.

With the `var` keyword, we're *declaring* a variable, but until a value is assigned to it, it's **undefined**.

```
var thing; // returns undefined
```

If you don't declare the variable with `var`, it's still **undefined**.

```
typeof anotherThing; // returns undefined
```

Null

Null is a value that represents nothing.

Null is the *intentional* absence of any value. It represents something that doesn't exist and is not any of the other datatypes.

```
var empty = null;
```

The difference between null and undefined is rather subtle, but the main distinction is that null is an intentionally empty value.

Symbol

A **Symbol** is a unique and immutable data type.

A Symbol is a new primitive data type that emerged with the latest JavaScript release ([ES6](#)). Its main feature is that each Symbol is a completely unique token, unlike other datatypes which can be overwritten.

As it is an advanced and little-used data type, I won't go into further detail. It is useful to know that it exists, and you can [read more about it here](#).

```
var sym = Symbol();
```

Objects

An **object** is a collection of name/value pairs.

Any value that isn't simply a string, number, Boolean, null, undefined, or Symbol is an [object](#).

You can create an object with a pair of curly braces ({}).

```
var batman = {};
```

Objects consist of **properties** and **methods**. Properties are what the object *is*, and methods are what the object *does*. Returning to the book analogy, properties are like nouns and adjectives, and methods are like verbs.

Property	Method
batman.firstName	batman.fight()
batman.location	batman.jump()

We can apply some properties to the `batman` object using name/value pairs (sometimes referred to as *key/value* pairs). They will be comma-separated and written as `propertyName: PropertyValue`.

```
var batman {
  firstName: "Bruce", // string
  lastName: "Wayne",
  location: "Gotham",
  introduced: 1939, // number
  billionaire: true, // Boolean
  weakness: null // null
};
```

Note: the last item in a list of object properties does not end in a trailing comma.

As you can see, we can apply any primitive data type as a value in an object. We can retrieve an individual value with a dot (.).

```
batman.firstName; // returns Bruce, a string
```

We can also retrieve the value with bracket notation.

```
batman["firstName"]; // returns Bruce
```

A JavaScript property name must be a valid JavaScript string or numeric literal. If the name begins with a number or contains a space, it must be accessed using the bracket notation. [Read: MDN](#)

```
batman.secret identity; // invalid
batman["Secret Identity"]; // valid
```

A method performs an action. Here is a simple example.

```
var batman =
  firstName: "Bruce",
  lastName: "Wayne",
  secretIdentity: function() { // method
    return this.firstName + " " + this.lastName;
  }
};
```

Instead of a simple data type as the value, I have a `function`. The `function` is concatenating `firstName` and `lastName` for `this` object and returning the result. In the example, `this` is the same as `batman` because it's inside the object (`{}`).

```
batman.secretIdentity(); // returns Bruce Wayne, a concatenation of two properties
```

A method is indicated by parentheses. `(())`.

Arrays

An **array** stores a list into a single variable.

Since an [array](#) contains more than one value, it's a type of object.

You can create an array with a pair of straight braces `([])`.

```
var ninjaTurtles = [];
```

Arrays do not contain name/value pairs.

```
var ninjaTurtles = [
  "Leonardo",
  "Michelangelo",
  "Raphael",
```

```
"Donatello"  
];
```

You can get an individual value by referencing the `index` of the array. In programming, counting starts at zero, so the first item in a list will always be `[0]`.

```
var leader = ninjaTurtles[0]; // assigns Leonardo to the leader variable
```

You can see how many items are in an array with the `length` property.

```
ninjaTurtles.length; // returns 4
```

Recap

We covered a lot in this article. You should now have a better understanding of common programming concepts, terminology, syntax, and fundamentals. By this point, you've learned all about JavaScript variables and datatypes, and now you're ready to move on to manipulating that data and building programs!

39 : 10 Node.js Best Practices: Enlightenment from the Node Gurus

10 Node.js Best Practices: Enlightenment from the Node Gurus is by guest author **Azat Mardan**. SitePoint guest posts aim to bring you engaging content from prominent writers and speakers of the Web community.

More from this author

- [React Quickly: How to Work with Forms in React](#)
- [How to Work with and Manipulate State in React](#)

In my previous article [10 Tips to Become a Better Node Developer in 2017](#), I introduced 10 Node.js tips, tricks and techniques you could apply to your code today. This post continues in that vein with a further 10 best practices to help you take your Node skills to the next level. This is what we're going to cover:

1. [Use npm scripts](#) — Stop writing bash scripts when you can organize them better with npm scripts and Node. E.g., `npm run build`, `start` and `test`. npm scripts are like the single source of truth when Node developers look at a new project.
2. [Use env vars](#) — Utilize `process.env.NODE_ENV` by setting it to `development`, or `production`. Some frameworks will use this variable too, so play by the convention.
3. [Understand the event loop](#) — `setImmediate()` is not immediate while `nextTick()` is not next. Use `setImmediate()` or `setTimeout()` to offload CPU-intensive tasks to the next event loop cycle.
4. [Use functional inheritance](#) — Avoid getting into mindless debates and a brain-draining trap of debugging and understanding prototypal inheritance or classes by just using functional inheritance like some of the most prolific Node contributors do.
5. [Name things appropriately](#) — Give meaningful names which will serve as a documentation. Also, please no uppercase filenames, use a dash if needed. Uppercase in filenames not just look strange but [can cause cross-platform issues](#).
6. [Consider NOT Using JavaScript](#) — ES6/7 is pathetic addition which was born out of 6 years of meetings when we already had a better JavaScript called CoffeeScript. Use it if you want ship code faster and stop wasting time debating `var` / `const` / `let`, semi-colons, `class` and other arguments.
7. [Provide native code](#) — When using transpilers, commit native JS code (result of the builds) so your projects can run without the builds
8. [Use gzip](#) — Duh! `npm i compression -S` and sane logging — not too much not to little depending on the environment. `npm i morgan -S`
9. [Scale up](#) — Start thinking about clustering and having stateless services from day one of your Node development. Use pm2 or strongloop's cluster control
10. [Cache requests](#) — Get maximum juice out of your Node servers by hiding them behind a static file server such as nginx and/or request level cache like Varnish Cache and CDN caching.

So let's bisect and take a look at each one of them individually. Shall we?

Use npm Scripts

It's almost a standard now to create npm scripts for builds, tests, and most importantly to start the app. This is the first place Node developers look at when they encounter a new Node project. Some people ([1](#), [2](#), [3](#), [4](#)) have even

ditched Grunt, Gulp and the likes for the more low-level but more dependable npm script. I can totally understand their argument. Considering that npm scripts have pre and post hooks, you can get to a very sophisticated level of automation:

```
"scripts": {
  "preinstall": "node prepare.js",
  "postinstall": "node clean.js",
  "build": "webpack",
  "postbuild": "node index.js",
  "postversion": "npm publish"
}
```

Often times when developing for the front-end, you want to run two or more watch processes to re-build your code. For example, one for webpack and another for nodemon. You can do this with `&&` since the first command won't release the prompt. However, there's a handy module called [concurrently](#) which can spawn multiple processes and run them at the same time.

Also, install dev command line tools such as webpack, nodemon, gulp, Mocha, etc. *locally* to avoid conflicts. You can point to `./node_modules/.bin/mocha` for example or add this line to your bash/zsh profile (PATH!):

```
export PATH=".:/node_modules/.bin:$PATH"
```

Use Env Vars

Utilize environment variables even for the early stages of a project to ensure there's no leakage of sensitive info, and just to build the code properly from the beginning. Moreover, some libraries and frameworks (I know Express does it for sure) will pull in info like `NODE_ENV` to modify their behavior. Set it to `production`. Set your `MONGO_URI` and `API_KEY` values as well. You can create a shell file (e.g. `start.sh`) and add it to `.gitignore`:

```
NODE_ENV=production MONGO_URL=mongo://localhost:27017/accounts API_KEY=lolz nodemon
index.js
```

Nodemon also has a config file where you can put your env vars ([example](#)):

```
{
  "env": {
    "NODE_ENV": "production",
    "MONGO_URL": "mongo://localhost:27017/accounts"
  }
}
```

Understand the Event Loop

The mighty and clever [event loop](#) is what makes Node so fast and brilliant by utilizing all the time which would have been wasted waiting for input and output tasks to complete. Thus, Node is great at optimizing I/O-bound systems.

If you need to perform something CPU-intensive (e.g., computation, hashing of passwords, or compressing), then in addition to spawning new processes for those CPU-tasks, you might want to explore the deferring of the task with `setImmediate()` or `setTimeout()` — the code in their callbacks will continue on the next event loop cycle. `nextTick()` works on the same cycle contrary to the name. Argh!

Here's a diagram from Bert Belder who worked on the event loop. He clearly knows how the event loop works!

Use Functional Inheritance

JavaScript supports prototypal inheritance which is when objects inherit from other objects. The `class` operator was also added to the language with ES6. However, it's overtly complex compared to functional inheritance. Most Node gurus prefer the simplicity of the latter. It's implemented by a simple function factory pattern, and does NOT require the use of `prototype`, `new` or `this`. There are no implicit effects when you update the prototype (causing all the instances to change as well) since in functional inheritance each object uses its own copy of methods.

Consider code from TJ Holowaychuk, the prolific genius behind Express, Mocha, Connect, Superagent and dozens of other Node modules. Express uses functional inheritance ([full source code](#)):

```
exports = module.exports = createApplication;
// ...
function createApplication() {
  var app = function(req, res, next) {
    app.handle(req, res, next);
  };

  mixin(app, EventEmitter.prototype, false);
  mixin(app, proto, false);

  app.request = { __proto__: req, app: app };
  app.response = { __proto__: res, app: app };
  app.init();
  return app;
}
```

To be objective, core Node modules use prototypal inheritance a lot. If you follow that pattern, make sure you know how it works. You can read more about [JavaScript inheritance patterns](#) here.

Name Things Appropriately

This one is obvious. Good names serve as a documentation. Which one would you prefer?

```
const dexter = require('morgan')
// ...
app.use(dexter('dev')) // When is the next season?
```

I have no idea what `dexter` is doing when I only look at `app.use()`. How about a different more meaningful name:

```
const logger = require('morgan')
// ...
app.use(logger('dev')) // Aha!
```

In the same fashion, file names must correctly reflect what is the purpose of the code inside. If you take a look at the `lib` folder of Node ([GitHub link](#)) which has all the core modules bundled with the platform, then you will see clear naming of the files/modules (even if you are not very familiar with *all* the core modules):

```
events.js
fs.js
http.js
https.js
module.js
net.js
os.js
path.js
process.js
punycode.js
querystring.js
```

The internal modules are marked with an underscore (`_debugger.js`, `_http_agent.js`, `_http_client.js`) just like methods and variables in the code. This helps to warn developers that this is an internal interface and if you are using it, you are on your own — don't complain if it gets refactored or even removed.

Consider NOT Using JavaScript

Huh? Did you just read it correctly? But what the heck? Yes. That's correct. Even with ES6 and the two features added by ES2016/ES7, JavaScript still has its quirks. There are other options besides JavaScript which you or your team can benefit from with very little setup. Depending on the expertise level and the nature of the app, you might be better off with [TypeScript](#) or [Flow](#) which provide strong typing. On the other end of the spectrum, there's [Elm](#) or [ClojureScript](#) which are purely functional. [CoffeeScript](#) is another great and battle-tested option. You might take a look at [Dart 2.0](#) as well.

When all you need is just a few macros (macros allow you to *build* exactly the language you want), not an entire new language, then consider [Sweet.js](#) which will do exactly that — allow you to write code which generates code.

If you go the non-JavaScript route, please still include your compiled code because some developers might not understand your language well enough to build it properly. For example, VS Code is one of the largest TypeScript projects, maybe after Angular 2, and Code uses TypeScript to patch Node's core module with types. In the `vscode/src/vs/base/node/` of VS Code repo ([link](#)), you can see familiar module names like `crypto`, `process`, etc. but with the `ts` extension. There are other `ts` files in the repo. However, they also included `vscode/build` with native JavaScript code.

Know Express Middleware

Express is a great and very mature framework. Its brilliance comes from allowing myriads of other modules to configure its behavior. Thus, you need to know the most used middleware and you need to know *how to use it*. So why not grab [my Express cheat sheet](#). I have the main middleware modules listed there. For example, `npm i compression -S` will give reduce the download speed by deflating the responses. `logger('tiny')` or `logger('common')` will provide less (dev) or more (prod) logs respectively.

Scale up

Node is great at async due to its non-blocking I/O and it keeps this async way of coding simple because there's just one thread. This is an opportunity to start scaling early on, maybe even with the first lines of code. There's the core `cluster` module which will allow you to scale vertically without too many problems. However, an even better way would be to use a tool such as [pm2](#) or [StrongLoop's cluster control](#).

For example, this is how you can get started with pm2:

```
npm i -g pm2
```

Then you can start four instances of the same server:

```
pm2 start server.js -i 4
```

For Docker, pm2 version 2+ has `pm2-docker`. So your Dockerfile can look like this:

```
# ...  
  
RUN npm install pm2 -g  
  
CMD ["pm2-docker", "app.js"]
```

The official Alpine Linux pm2 image is in the [Docker Hub](#).

Cache Requests

This is a DevOps best practice which will allow you to get more juice out of your Node instances (you get more than one with pm2 or the like, see above). The way to go is to let Node servers do app stuff like making requests,

processing data and executing business logic and offload the traffic to static files to another web server such as Apache httpd or Nginx. Again, you probably should use Docker for the set up:

```
FROM nginx

COPY nginx.conf /etc/nginx/nginx.conf
```

I like to use Docker compose to make multiple containers (nginx, Node, Redis, MongoDB) work with each other.

For example:

```
web:
  build: ./app
  volumes:
    - "./app:/src/app"
  ports:
    - "3030:3000"
  links:
    - "db:redis"
  command: pm2-docker app/server.js

nginx:
  restart: always
  build: ./nginx/
  ports:
    - "80:80"
  volumes:
    - /www/public
  volumes_from:
    - web
  links:
    - web:web

db:
  image: redis
```

Summary

In this day and age of open-source software, there are no excuses not to learn from the trusted and tested code which is out in the open. You don't need to be in the inner circle to get in. Learning never stops and I'm sure soon we will have different best practices based on the failures and successes which we will experience. They are guaranteed.

Finally, I wanted to write about how software is eating the world and how JavaScript is eating the software... there are great things like yearly standard releases, lots and lots of npm modules, tools and conferences... but instead

I'll finish with a word of caution.

I see how more and more people chase the next new framework or language. It's the shiny object syndrome. They learn a new library every week and a new framework every month. They compulsively check Twitter, Reddit, Hacker News and JS Weekly. They use the overwhelming level of activity in the JavaScript world to procrastinate. They have empty public GitHub histories.

Learning new things is good but don't confuse it for actually building stuff. What matters and what pays your salary is actually building things. **Stop over engineering.** You're not building the next Facebook. Promises vs. generators vs. async await is a moot for me, because by the time someone replied to a thread in a discussion, I already wrote my callback (and used CoffeeScript to do it 2x faster than in plain ES5/6/7!).

The final best practice is to use best practices and the best of the best is to master fundamentals. Read source code, try new things in code and most importantly write tons of code yourself. Now, at this point, stop reading and go ship code that matters!

40 : 10 Tips to Become a Better Node Developer in 2017

10 Tips to Become a Better Node Developer in 2017 is by guest author **Azat Mardan**. SitePoint guest posts aim to bring you engaging content from prominent writers and speakers of the Web community.

Note: The original title of this article was *The Best Node Practices from Gurus of The Platform*. The article covers true, tried and tested patterns, not the new and best of 2017. Although, some of the good old practices from Node gurus will still apply in 2017 and 2018 and even in 2019, the new cutting-edge features like `async/await`, promises are not covered here. That's because these new features are not in the code of Node core, or popular projects like `npm`, `Express`, etc. The second part of the essay will reflect the proper nature of the content.

More from this author

- [React Quickly: How to Work with Forms in React](#)
- [How to Work with and Manipulate State in React](#)
- [10 Node.js Best Practices: Enlightenment from the Node Gurus](#)

I started working with Node full-time in 2012 when I joined Storify. Since then, I have never looked back or felt that I missed Python, Ruby, Java or PHP — languages with which I had worked during my previous decade of web development.

Storify was an interesting job for me, because unlike many other companies, Storify ran (and maybe still does) everything on JavaScript. You see, most companies, especially large ones such as PayPal, Walmart, or Capital One, only use Node for certain parts of their stack. Usually they use it as an API gateway or an orchestration layer. That's great. But for a software engineer, nothing compares with full immersion into a Node environment.

In this post I'll outline ten tips to help you become a better Node developer in 2017. These tips come from me, who saw and learned them in the trenches, as well as people who have written the most popular Node and npm modules. Here's what we'll be covering:

1. [Avoid complexity](#) — Organize your code into the smallest chunks possible until they look too small and then make them even smaller.
2. [Use asynchronous code](#) — Avoid synchronous code like the plague.
3. [Avoid blocking require](#) — Put ALL your require statements at the top of the file because they are synchronous and will block the execution.
4. [Know that require is cached](#) — This could be a feature or a bug in your code.
5. [Always check for errors](#) — Errors are not footballs. Never throw errors and never skip the error check.
6. [Use try...catch only in sync code](#) — `try...catch` is useless for async code, plus V8 can't optimize code in `try...catch` as well as plain code.
7. [Return callbacks or use if ... else](#) — Just to be sure, return a callback to prevent execution from continuing.
8. [Listen to the error events](#) — Almost all Node classes/objects extend the event emitter (observer pattern) and emit the `error` event. Be sure to listen to that.
9. [Know your npm](#) — Install modules with `-S` or `-D` instead of `--save` or `--save-dev`
10. [Use exact versions in package.json](#): npm stupidly adds a caret by default when you use `-S`, so get rid of them manually to lock the versions. Never trust semver in your apps, but do so in open-source modules.

11. *Bonus* — Use different dependencies. Put things your project needs only in development in `devDependencies` and then use `npm i --production`. The more un-required dependencies you have, the greater the risk of vulnerability.

So let's bisect and take a look at each one of them individually. Shall we?

And don't forget: as mentioned above, this is part one. You can find [a further ten tips in part two](#).

Avoid Complexity

Take a look at some of the modules written by Isaac Z. Schlueter, the creator of npm. For example, [use-strict](#) enforces JavaScript strict mode for modules, and it's just *three* lines of code:

```
var module = require('module')
module.wrapper[0] += '"use strict";'
Object.freeze(module.wrap)
```

So why avoid complexity? A famous phrase which originated in the US Navy according to one of the legends proclaims: KEEP IT SIMPLE STUPID (or is it "*Keep it simple, stupid*"?). That's for a reason. The human brain can hold only five to seven items in its working memory at any one time. This is just a fact.

By keeping your code modularized into smaller parts, you and other developers can understand and reason about it better. You can also test it better. Consider this example,

```
app.use(function(req, res, next) {
  if (req.session.admin === true) return next()
  else return next(new Error('Not authorized'))
}, function(req, res, next) {
  req.db = db
  next()
})
```

Or this code:

```
const auth = require('./middleware/auth.js')
const db = require('./middleware/db.js')(db)

app.use(auth, db)
```

I'm sure most of you will prefer the second example, especially when the names are self-explanatory. Of course, when you write the code you might think that you understand how it works. Maybe you even want to show off how smart you are by chaining several methods together in one line. Please, code for the dumber version of you. Code for the you who hasn't looked at this code for six months, or a tried or drunk version of you. If you write code at the peak of your mental capacity, then it will be harder for you to understand it later, not to even mention your

colleagues who are not even familiar with the intricacies of the algorithm. Keeping things simple is especially true for Node which uses the asynchronous way.

And yes, there was the [left-pad incident](#) but that only affected projects dependent on the public registry and the replacement was published in 11 minutes. The benefits of going small far outweigh the downsides. Also, npm has [changed its unpublish policy](#), and any serious project should be using a caching strategy or a private registry (as a temporary solution).

Use Asynchronous Code

Synchronous code *does* have a (small) place in Node. It's mostly for writing CLI commands or other scripts not related to web apps. Node developers mostly build web apps, hence they use async code to avoid blocking threads.

For example, this might be okay if we are just building a database script, and not a system to handle parallel/concurrent tasks:

```
let data = fs.readFileSync('./accounts.json')
db.collection('accounts').insert(data, (results)=>{
  fs.writeFileSync('./accountIDs.json', results, ()=>{process.exit(1)})
})
```

But this would be better when building a web app:

```
app.use('/seed/:name', (req, res) => {
  let data = fs.readFile(`./${req.params.name}.json`, ()=>{
    db.collection(req.params.name).insert(data, (results)=>{
      fs.writeFile(`./${req.params.name}IDs.json`, results, ()=>{res.status(201).send()})
    })
  })
})
```

The difference is whether you are writing concurrent (typically long running) or non-concurrent (short running) systems. As a rule of thumb, always write async code in Node.

Avoid Blocking require

Node has a simple module loading system which uses the CommonJS module format. Its built-in `require` function is an easy way to include modules that exist in separate files. Unlike AMD/requirejs, the Node/CommonJS way of module loading is synchronous. The way `require` works is: *you import what was exported in a module, or a file.*

```
const react = require('react')
```

What most developers don't know is that `require` is cached. So, as long as there are no drastic changes to the resolved filename (and in the case of npm modules there are none), then the code from the module will be executed and loaded into the variable just once (for that process). This is a nice optimization. However, even with caching, you are better off putting your `require` statements first. Consider this code which only loads the `axios` module on the route which actually uses it. The `/connect` route will be slower than needed because the module import is happening when the request is made:

```
app.post('/connect', (req, res) => {
  const axios = require('axios')
  axios.post('/api/authorize', req.body.auth)
    .then((response)=>res.send(response))
})
```

A better, more performant way is to load the modules before the server is even defined, not in the route:

```
const axios = require('axios')
const express = require('express')
app = express()
app.post('/connect', (req, res) => {
  axios.post('/api/authorize', req.body.auth)
    .then((response)=>res.send(response))
})
```

Know That `require` Is Cached

I mentioned that `require` is cached in the previous section, but what's interesting is that we can have code *outside* of the `module.exports`. For example,

```
console.log('I will not be cached and only run once, the first time')

module.exports = () => {
  console.log('I will be cached and will run every time this module is invoked')
}
```

Knowing that some code might run only once, you can use this feature to your advantage.

Always Check for Errors

Node is not Java. In Java, you throw errors because most of the time if there's an error you don't want the application to continue. In Java, you can handle *multiple* errors at a higher levels with a single `try...catch`.

Not so with Node. Since Node uses the [event loop](#) and executes asynchronously, any errors are separated from the context of any error handler (such as `try...catch`) when they occur. This is useless in Node:

```

try {
  request.get('/accounts', (error, response)=>{
    data = JSON.parse(response)
  })
} catch(error) {
  // Will NOT be called
  console.error(error)
}

```

But `try...catch` still can be used in synchronous Node code. So this is a better refactoring of the previous snippet:

```

request.get('/accounts', (error, response)=>{
  try {
    data = JSON.parse(response)
  } catch(error) {
    // Will be called
    console.error(error)
  }
})

```

If we cannot wrap the `request` call in a `try...catch` block, that leaves us with errors coming from `request` unhandled. Node developers solve this by providing you with `error` as a callback argument. Thus, you need to always manually handle the `error` in each and every callback. You do so by checking for an error (make sure it's not `null`) and then either displaying the error message to the user or a client and logging it, or passing it back up the call stack by calling the callback with `error` (if you have the callback and another function up the call stack).

```

request.get('/accounts', (error, response)=>{
  if (error) return console.error(error)
  try {
    data = JSON.parse(response)
  } catch(error) {
    console.error(error)
  }
})

```

A little trick you can use is the [okay](#) library. You can apply it like this to avoid manual error check on myriads of nested callbacks (Hello, [callback hell](#)).

```

var ok = require('okay')

request.get('/accounts', ok(console.error, (response)=>{

```

```

try {
  data = JSON.parse(response)
} catch(error) {
  console.error(error)
}
})

```

Return Callbacks or Use if ... else

Node is concurrent. So it's a feature which can turn into a bug if you are not careful. To be on the safe side terminate the execution with a return statement:

```

let error = true
if (error) return callback(error)
console.log('I will never run - good.')

```

Avoid some unintended concurrency (and failures) due to mishandled control flow.

```

let error = true
if (error) callback(error)
console.log('I will run. Not good!')

```

Just to be sure, `return` a callback to prevent execution from continuing.

Listen to the `error` Events

Almost all Node classes/objects extend the event emitter (observer pattern) and emit the `error` event. This is an opportunity for developers to catch those pesky errors and handle them before they wreak havoc.

Make it a good habit to create event listeners for `error` by using `.on()`:

```

var req = http.request(options, (res) => {
  if (('' + res.statusCode).match(/^2\d\d$/)) {
    // Success, process response
  } else if (('' + res.statusCode).match(/^5\d\d$/))
    // Server error, not the same as req error. Req was ok.
  }
})

req.on('error', (error) => {
  // Can't even make a request: general error, e.g. ECONNRESET, ECONNREFUSED,
  HPE_INVALID_VERSION
  console.log(error)
})

```

Know Your npm

Many Node and event front-end developers know that there is `--save` (for `npm install`) which will not only install a module but create an entry in `package.json` with the version of the module. Well, there's also `--save-dev`, for `devDependencies` (stuff you don't need in production). But did you know you can just use `-S` and `-D` instead of `--save` and `--save-dev`? Yes, you can.

And while you're in the module installation mode, go ahead and remove those `^` signs which `-S` and `-D` will create for you. They are dangerous because they'll allow `npm install` (or its shortcut `npm i`) to pull the latest minor (second digit in the semantic versioning) version from npm. For example, v6.1.0 to v6.2.0 is a minor release.

npm team believes in [semver](#), but you should not. What I mean is that they put caret `^` because they trust open source developers to not introduce breaking changes in minor releases. No one sane should trust it. Lock your versions. Even better, use [shrinkwrap](#): `npm shrinkwrap` which will create a new file with exact versions of dependencies of dependencies.

Conclusion

This post was part one of two. We've already covered a lot of ground, from working with callbacks and asynchronous code, to checking for errors and locking down dependencies. I hope you've found something new or useful here. If you liked it, be sure to check out part two: [10 Node.js Best Practices: Enlightenment from the Node Gurus](#).

And tell me what you think. Did I miss anything out? Are you doing it differently? Let me know in the comments below.

41 : Function Composition: Building Blocks for Maintainable Code

<https://www.sitepoint.com/function-composition-building-blocks-for-maintainable-code/>

One of the advantages of thinking about JavaScript in a functional way is the ability to build complex functionality using small, easy to understand individual functions. But sometimes that involves looking at a problem backwards instead of forwards in order to figure out how to create the most elegant solution.

In this article I'm going to employ a step-by-step approach to examine functional composition in JavaScript and demonstrate how it can result in code which is easier to reason about and which has fewer bugs.

Nesting Functions

More from this author

- [Functional JavaScript for Lazy Developers \(Like Me\)](#)
- [Scrum Artifacts: Product Increment](#)
- [Scrum Artifacts: Velocity and Burndown Charts](#)

Composition is a technique that allows you to take two or more simple functions, and combine them into a single, more complex function that performs each of the sub functions in a logical sequence on whatever data you pass in.

To get this result, you nest one function inside the other, and perform the operation of the outer function on the result of the inner function repeatedly until you produce a result. And the result can be different depending on the order in which the functions are applied.

This can be easily demonstrated using programming techniques we're already familiar with in JavaScript by passing a function call as an argument to another function:

```
function addOne(x) {  
    return x + 1;  
}  
  
function timesTwo(x) {  
    return x * 2;  
}  
  
console.log(addOne(timesTwo(3))); //7  
console.log(timesTwo(addOne(3))); //8
```

In this case we defined a function `addOne()` to add one to a value, and a `timesTwo()` function that multiplies a value by two. By passing the result of one function in as the argument for the other function, we can see how nesting one of these inside the other can produce different results, even with the same initial value. The inner function is performed first, and then the result is passed to the outer function.

Imperative Composition

If you wanted to perform the same sequence of operations repeatedly, it might be convenient to define a new function that automatically applied first one and then the other of the smaller functions. That might look something like this:

```
// ...previous function definitions from above
function addOneTimesTwo(x) {
  var holder = x;
  holder = addOne(holder);
  holder = timesTwo(holder);
  return holder;
}
console.log(addOneTimesTwo(3)); //8
console.log(addOneTimesTwo(4)); //10
```

What we've done in this case is manually compose these two functions together in a particular order. We created a new function that first assigns the value being passed to a holder variable, then updates the value of that variable by executing the first function, and then the second function, and finally returns the value of that holder.

(Note that we're using a variable called `holder` to hold the value we're passing in temporarily. With such a simple function, the extra local variable may seem redundant, but even in imperative JavaScript it's a good practice to treat the value of arguments passed into a function as if they were constants. Modifying them is possible locally, but it introduces confusion about what the value of the argument is when it's called at different stages within a function.)

Similarly, if we wanted to create another new function that applies these two smaller functions in the opposite order, we can do something like this:

```
// ...previous function definitions from above
function timesTwoAddOne(x) {
  var holder = x;
  holder = timesTwo(holder);
  holder = addOne(holder);
  return holder;
}
console.log(timesTwoAddOne(3)); //7
console.log(timesTwoAddOne(4)); //9
```

Of course, this code is starting to look pretty repetitive. Our two new composed functions are almost exactly the same, except for the order in which the two smaller functions they call are executed. We need to DRY that up (as in Don't Repeat Yourself). Also, using temporary variables that change their value like this isn't very functional, even if it is being hidden inside of the composed functions we're creating.

Bottom line: we can do better.

Creating a Functional Compose

Let's craft a compose function that can take existing functions and compose them together in the order that we want. To do that in a consistent way without having to play with the internals every time, we have to decide on the order that we want to pass the functions in as arguments.

We have two choices. The arguments will each be functions, and they can either be executed left to right, or right to left. That is to say that using our proposed new function, `compose(timesTwo, addOne)` could either mean `timesTwo(addOne())` reading the arguments right to left, or `addOne(timesTwo())` reading the arguments left to right.

The advantage of executing the arguments left to right is that they will read the same way that English reads, much the way that we named our composed function `timesTwoAddOne()` in order to imply that the multiplication should happen before the addition. We all know the importance of logical naming to clean readable code.

The disadvantage of executing the arguments from left to right is that the values to be operated upon would have to come first. But putting the values first makes it less convenient to compose the resulting function with other functions in the future. For a good explanation of the thinking behind this logic, you can't beat Brian Lonsdorf's classic video [Hey Underscore, You're Doing it Wrong](#). (Although it should be noted that there is now [an fp option for Underscore](#) that helps to address the functional programming issue Brian discusses when using Underscore in concert with a functional programming library such as [lodash-fp](#) or [Ramda](#).)

In any event, what we really want to do is pass in all of the configuration data first, and pass the value(s) to be operated on last. Because of this, it makes the most sense to define our compose function to read in its arguments and apply them from right to left.

So we can create a rudimentary `compose` function that looks something like this:

```
function compose(f1, f2) {  
  return function(value) {  
    return f1(f2(value));  
  };  
}
```

Using this very simple `compose` function, we can construct both of our previous complex functions much more simply, and see that the results are the same:

```
function addOne(x) {  
  return x + 1;  
}  
function timesTwo(x) {  
  return x * 2;  
}  
function compose(f1, f2) {  
  return function(value) {  
    return f1(f2(value));  
  };  
}
```

```

};

}

var addOneTimesTwo = compose(timesTwo, addOne);
console.log(addOneTimesTwo(3)); //8
console.log(addOneTimesTwo(4)); //10
var timesTwoAddOne = compose(addOne, timesTwo);
console.log(timesTwoAddOne(3)); //7
console.log(timesTwoAddOne(4)); //9

```

While this simple `compose` function works, it doesn't take into account a number of issues that limit its flexibility and applicability. For example, we might want to compose more than two functions. Also, we lose track of `this` along the way.

We could fix these problems, but it's not necessary in order to grasp how composition works. Rather than roll our own, it's probably more productive to inherit a more robust `compose` from one of the functional libraries out there, such as [Ramda](#), which does account for right to left ordering of arguments by default.

Types Are Your Responsibility

It's important to keep in mind that it is the responsibility of the programmer to know the type returned by each of the functions being composed, so it can be handled correctly by the next function. Unlike purely functional programming languages that do strict type checking, JavaScript won't prevent you from trying to compose functions that return values of inappropriate types.

You're not limited to passing numbers, and you're not even limited to maintaining the same type of variable from one function to the next. But you are responsible for making sure that the functions you are composing are prepared to deal with whatever value the previous function returns.

Consider Your Audience

Always remember that someone else may need to use or modify your code in the future. Using composition inside traditional JavaScript code can appear complicated to programmers not familiar with functional paradigms. The goal is cleaner code that's easier to read and maintain.

But with the advent of ES2015 syntax, the creation of a simple composed function as a one-line call can even be done without a special `compose` method using arrow functions:

```

function addOne(x) {
  return x + 1;
}
function timesTwo(x) {
  return x * 2;
}
var addOneTimesTwo = x => timesTwo(addOne(x));
console.log(addOneTimesTwo(3)); //8
console.log(addOneTimesTwo(4)); //10

```

Start Composing Today

As with all functional programming techniques, it's important to keep in mind that your composed functions should be pure. In a nutshell this means that every time a specific value is passed into a function, the function should return the same result, and the function should not produce side effects that alter values outside of itself.

Compositional nesting can be very convenient when you have a set of related functionality that you want to apply to your data, and you can break down the components of that functionality into reusable and easily composed functions.

As with all functional programming techniques, I recommend sprinkling composition judiciously into your existing code to get familiar with it. If you're doing it right, the result will be cleaner, dryer, and more readable code. And isn't that what we all want?

42 : Creating Unwritable Properties with Object.defineProperty

<http://adripofjavascript.com/blog/drips/creating-unwritable-properties-with-object-defineproperty.html>

43 : Creating Unwritable Properties with Object.defineProperty

Originally published in the [A Drip of JavaScript newsletter](#).

JavaScript has a reputation as a language where the developer can redefine just about anything. While that has been largely true in past versions of JavaScript, with ECMAScript 5 the situation has begun to change. For instance, thanks to `Object.defineProperty` it is now possible to create object properties that cannot be modified.

Why would you want to do that? Imagine that you are building a mathematics library.

```
var principia = {
  constants: {
    // Pi is used here for convenience of illustration.
    // For real applications, you'd want to use Math.PI
    pi: 3.14
  },
  areaOfCircle: function (radius) {
    return this.constants.pi * radius * radius;
  }
};
```

Your library provides a collection of common equations, but for convenience also defines a set of mathematical constants. Of course, the thing about constants is that they should remain constant and shouldn't be redefined. But suppose a user of your library accidentally does something like this:

```
// Accidental assignment
if (principia.constants.pi = myval) {
  // do something
}
```

The user has accidentally assigned a value to `pi`. Suddenly every equation in your library that depends on `pi` will return incorrect results. This will likely lead the user to believe your library is defective rather than discovering the error in their code. While there are multiple ways to avoid this issue, the simplest solution is just to make `pi` unwritable. Let's take a look at how to do that.

```
var principia = {
  constants: {},
  areaOfCircle: function (radius) {
    return this.constants.pi * radius * radius;
  }
};
```

```
Object.defineProperty(principia.constants, "pi", {
  value: 3.14,
  writable: false
});
```

The `defineProperty` method takes three arguments: the object whose property you are creating or modifying, the name of the property, and an options object. The options object itself allows us to set several options, but for the moment we are only interested in `value` and `writable`.

The `value` option defines what the actual value of our property will be, while `writable` specifies whether we can assign a new value to it. So let's see how it works.

```
// Try to assign a new value to pi
principia.constants.pi = 2;

// Outputs: 3.14
console.log(principia.constants.pi);
```

As you can see, attempting to assign a new value to `pi` fails silently and `pi` retains its original value. So now a mistake by a careless library user won't cause problems with the library itself.

And if the user of your library is in strict mode, it gets better.

```
// Turn on strict mode
"use strict";

// TypeError: Cannot assign to read only property 'pi' of principia.constants
principia.constants.pi = 2;
```

Strict mode will throw an error if you attempt to assign a new value to unwritable properties, which means that errors like the one illustrated above would get flagged immediately.

There are two more aspects of `defineProperty` that you'll need to understand to use it effectively.

While in our example above, we explicitly set `writable`, it turns out that `writable` defaults to `false`, so our earlier example can be rewritten like so.

```
Object.defineProperty(principia.constants, "pi", {
  value: 3.14
});
```

In addition there is another option called `configurable` which specifies whether you can use `defineProperty` and similar methods to reconfigure things like `writable` to a different state. The `configurable` option defaults

to `false`, so if you want to let users of your library override `pi` intentionally, but not through accidental assignment, you would need to explicitly set it to `true`, like so:

```
Object.defineProperty(principia.constants, "pi", {  
    value: 3.14,  
    configurable: true  
});
```

There is one rather large "gotcha" to be aware of when using `defineProperty`. While setting `writable` prevents assignment to the property, it does not make the property's value immutable. Consider the case of an array:

```
var container = {};  
  
Object.defineProperty(container, "arr", {  
    writable: false,  
    value: ["a", "b"]  
});  
  
container.arr = ["new array"];  
  
// Outputs: ["a", "b"]  
console.log(container.arr);  
  
container.arr.push("new value");  
  
// Outputs: ["a", "b", "new value"]  
console.log(container.arr);
```

The `arr` property is unwritable, so it will always point to the same array. But the array itself may be changed. Unless you use a value that is intrinsically immutable (like a string primitive), the property's value is still subject to change. We'll consider a way of locking down arrays and other objects in a future issue.

Unfortunately, because `Object.defineProperty` is part of ES5, it is only fully supported in IE9 and newer. IE8 has a partial implementation which only works on DOM objects, and would be useless for the examples considered above. Even more unfortunately, there is no compatibility shim for IE8. However, if you don't need to deal with older browsers, `defineProperty` might be just what you are looking for.

Thanks for reading!

Joshua Clanton

44 : Advanced objects in JavaScript

<http://bjorn.tipling.com/advanced-objects-in-javascript>

This post looks beyond everyday usage of JavaScript's objects. The fundamentals of JavaScript's objects are for the most part about as simple as using JSON notation. However, JavaScript also provides sophisticated tools to create objects in interesting and useful ways, many of which are now available in the latest versions of modern browsers.

The last two topics I talk about, `Proxy` and `Symbol`, are based on the ECMAScript 6 specification and are only partially implemented and of limited availability across browsers.

getters and setters

Getters and setters have been available in JavaScript for some time now but I have not found myself using them much. I often fallback to writing regular functions to get properties. I usually end up writing something like this:

```
/**  
 * @param {string} prefix  
 * @constructor  
 */  
function Product(prefix) {  
    /**  
     * @private  
     * @type {string}  
     */  
    this.prefix_ = prefix;  
    /**  
     * @private  
     * @type {string}  
     */  
    this.type_ = "";  
}  
  
/**  
 * @param {string} newType  
 */  
Product.prototype.setType = function (newType) {  
    this.type_ = newType;  
};  
  
/**  
 * @return {string}  
 */
```

```
Product.prototype.type = function () {
    return this.prefix_ + ": " + this.type_;
}

var product = new Product("fruit");
product.setType("apple");
console.log(product.type()); //logs fruit: apple
```

Using a getter I could simplify this code.

```
/** 
 * @param {string} prefix
 * @constructor
 */
function Product(prefix) {
    /**
     * @private
     * @type {number}
     */
    this.prefix_ = prefix;
    /**
     * @private
     * @type {string}
     */
    this.type_ = "";
}

/**
 * @param {string} newType
 */
Product.prototype = {
    /**
     * @return {string}
     */
    get type () {
        return this.prefix_ + ": " + this.type_;
    },
    /**
     * @param {string}
     */
    set type (newType) {
        this.type_ = newType;
    }
}
```

```

};

var product = new Product("fruit");

product.type = "apple";
console.log(product.type); //logs "fruit: apple"

console.log(product.type = "orange"); //logs "orange"
console.log(product.type); //logs "fruit: orange"

```

The code is still a bit verbose and the syntax is a bit unusual but the benefit of `get` and `set` is realized in using the property. Personally I find something like

```

product.type = "apple";
console.log(product.type);

```

a lot more readable and accessible than

```

product.setType("apple");
console.log(product.type());

```

Although directly accessing and setting properties on instances still sets off my internal *bad-javascript* buzzer. Over time I have been trained by bugs and technical debt to avoid arbitrarily setting values on instances as a means to pass information around. Also, a particular caveat is in order for the return value of an assignment. Note this bit of code in the example above:

```

console.log(product.type = "orange"); //logs "orange"
console.log(product.type); //logs "fruit: orange"

```

Notice that “orange” is logged at first and then “fruit: orange” on the next line. The getter isn’t executed on the return value from assigning to a property, so this kind of shortcut where you return the value from an assignment could get you into trouble here. Return statements on a `set` are ignored. Adding `return this.type;` to the `set` wont fix this problem. Usually reusing the value of an assignment would work, but you may have problems with a property that has a getter.

defineProperty

The `get propertyname ()` syntax works on object literals and in the previous example I assigned an object literal to `Product.prototype`. This is OK, but using object literals like this makes it harder to chain prototypes to get inheritance. You can create getters and setters on a `prototype` without an object literal using [`defineProperty`](#).

```

/**
 * @param {string} prefix
 * @constructor
 */
function Product(prefix) {
    /**
     * @private
     * @type {number}
     */
    this.prefix_ = prefix;
    /**
     * @private
     * @type {string}
     */
    this.type_ = "";
}

/**
 * @param {string} newType
 */
Object.defineProperty(Product.prototype, "type", {
    /**
     * @return {string}
     */
    get: function () {
        return this.prefix_ + ": " + this.type_;
    },
    /**
     * @param {string}
     */
    set: function (newType) {
        this.type_ = newType;
    }
});

```

This code behaves the same as the previous example. There's more to `defineProperty` than adding getters or setters. The third argument to `defineProperty` is called the descriptor and in addition to `set` and `get` it allows you to customize the property's accessibility and set a value. You could use the descriptor argument to `defineProperty` to create something like a constant that can never be modified or removed.

```

var obj = {
    foo: "bar",
};

```

```
//A normal object property
console.log(obj.foo); //logs "bar"

obj.foo = "foobar";
console.log(obj.foo); //logs "foobar"

delete obj.foo;
console.log(obj.foo); //logs undefined

Object.defineProperty(obj, "foo", {
    value: "bar",
});

console.log(obj.foo); //logs "bar", we were able to modify foo

obj.foo = "foobar";
console.log(obj.foo); //logs "bar", write failed silently

delete obj.foo;
console.log(obj.foo); //logs bar, delete failed silently
```

[jsfiddle](#)

The result is:

```
bar
foobar
undefined
bar
bar
bar
```

The last 2 attempts to modify `foo.bar` in the example failed silently as the default behavior of `defineProperty` is to prevent further changes. You can use `configurable` and `writable` to change this behavior. If you are using strict mode the failures are not silent, [they are JavaScript errors](#).

```
var obj = {};

Object.defineProperty(obj, "foo", {
    value: "bar",
```

```
configurable: true,  
writable: true,  
});  
  
console.log(obj.foo); //logs "bar"  
obj.foo = "foobar";  
console.log(obj.foo); //logs "foobar"  
delete obj.foo;  
console.log(obj.foo); //logs undefined
```

The `configurable` key allows you to prevent the property from being deleted from the object. It also allows you to prevent the property from being modified in the future with another call to `defineProperty` later. The `writable` key enables you to write to the property and change its value.

If `configurable` is `false` as is the default case, attempting to call `defineProperty` a second time will result in a JavaScript error, it does not fail silently.

```
var obj = {};  
  
Object.defineProperty(obj, "foo", {  
    value: "bar",  
});  
  
Object.defineProperty(obj, "foo", {  
    value: "foobar",  
});  
  
// Uncaught TypeError: Cannot redefine property: foo
```

If `configurable` is set to `true` you can modify the property again later. You might use that to change the value on a non-writable property.

```
var obj = {};  
  
Object.defineProperty(obj, "foo", {  
    value: "bar",  
    configurable: true,  
});  
  
obj.foo = "foobar";  
  
console.log(obj.foo); // logs "bar", write failed
```

```
Object.defineProperty(obj, "foo", {
  value: "foobar",
  configurable: true,
});

console.log(obj.foo); // logs "foobar"
```

[jsfiddle](#)

Also note that values defined by `defineProperty` are by default not iterated over in a `for in` loop:

```
var i, inventory;

inventory = {
  "apples": 10,
  "oranges": 13,
};

Object.defineProperty(inventory, "strawberries", {
  value: 3,
});

for (i in inventory) {
  console.log(i, inventory[i]);
}
```

[jsfiddle](#)

```
apples 10
oranges 13
```

Use the `enumerable` key to allow this:

```
var i, inventory;

inventory = {
  "apples": 10,
  "oranges": 13,
};

Object.defineProperty(inventory, "strawberries", {
  value: 3,
```

```

        enumerable: true,
    });

    for (i in inventory) {
        console.log(i, inventory[i]);
    }
}

```

Outcome:

```

apples 10
oranges 13
strawberries 3

```

You can use `isPropertyEnumerable` to test whether a property will appear in a `for in` loop:

```

var i, inventory;

inventory = {
    "apples": 10,
    "oranges": 13,
};

Object.defineProperty(inventory, "strawberries", {
    value: 3,
});

console.log(inventory.propertyIsEnumerable("apples")); //console logs true
console.log(inventory.propertyIsEnumerable("strawberries")); //console logs false

```

[jsfiddle](#)

`propertyIsEnumerable` will also return false for properties defined further up an object's `prototype` chain or for properties that aren't otherwise defined on the object, obviously.

Finally, a last couple points about using `defineProperty`: [It is an error](#) to combine the accessors `set` and `get` with `writable` set to true or to combine them with a `value`. Defining a property as a number simply [converts the number into a string](#) just as it would in any other circumstance. You can also use `defineProperty` to [set value to be a function](#).

defineProperties

Also [`Object.defineProperties`](#) exists. It allows you to define multiple properties in a single go. I found [a jsperf](#) that compared the performance of `defineProperty` versus `defineProperties` and in Chrome at least it didn't seem to matter much which was used.

```
var foo = {}

Object.defineProperties(foo, {
  bar: {
    value: "foo",
    writable: true,
  },
  foo: {
    value: function() {
      console.log(this.bar);
    }
  },
});

foo.bar = "foobar";
foo.foo(); //logs "foobar"
```

Object.create

[Object.create](#) is an alternative to `new` that lets you create an object with a given prototype. This function has two arguments, the first is the prototype you want use for the newly created object, the second argument is a property descriptor and it takes the same form as you would give to `Object.defineProperties`.

```
var prototypeDef = {
  protoBar: "protoBar",
  protoLog: function () {
    console.log(this.protoBar);
  }
};

var propertiesDef = {
  instanceBar: {
    value: "instanceBar"
  },
  instanceLog: {
    value: function () {
      console.log(this.instanceBar);
    }
  }
}

var foo = Object.create(prototypeDef, propertiesDef);
```

```
foo.protoLog(); //logs "protoBar"
foo.instanceLog(); //logs "instanceBar"
```

[jsfiddle](#)

Properties created in the properties descriptor argument to `Object.create` overwrite the values in the prototype argument:

```
var prototypeDef = {
  bar: "protoBar",
};

var propertiesDef = {
  bar: {
    value: "instanceBar",
  },
  log: {
    value: function () {
      console.log(this.bar);
    }
  }
}

var foo = Object.create(prototypeDef, propertiesDef);
foo.log(); //logs "instanceBar"
```

Setting a non-primitive type like an Array or Object as a defined properties value in `Object.create` is probably a mistake since you will create a single instance shared by all created objects:

```
var prototypeDef = {
  protoArray: [],
};

var propertiesDef = {
  propertyArray: {
    value: [],
  }
}

var foo = Object.create(prototypeDef, propertiesDef);
var bar = Object.create(prototypeDef, propertiesDef);

foo.protoArray.push("foobar");
console.log(bar.protoArray); //logs ["foobar"]
```

```
foo.propertyArray.push("foobar");
console.log(bar.propertyArray); //also logs ["foobar"]
```

[jsfiddle](#)

You could fix this problem by initialize the value of `propertyArray` with `null` and then add the array when you needed, or you could do something fancy like this using a getter:

```
var prototypeDef = {
    protoArray: [],
};

var propertiesDef = {
    propertyArrayValue_: {
        value: null,
        writable: true
    },
    propertyArray: {
        get: function () {
            if (!this.propertyArrayValue_) {
                this.propertyArrayValue_ = [];
            }
            return this.propertyArrayValue_;
        }
    }
};

var foo = Object.create(prototypeDef, propertiesDef);
var bar = Object.create(prototypeDef, propertiesDef);

foo.protoArray.push("foobar");
console.log(bar.protoArray); //logs ["foobar"]
foo.propertyArray.push("foobar");
console.log(bar.propertyArray); //logs []
```

This is a neat way to combine initialization of properties with their definitions. I think I prefer keeping property definitions with initialization together a great deal over doing this work in a constructor. In the past I often ended up writing a giant constructor method full of such initialization code.

The previous example demonstrates that you have to remember that the expressions provided to any `value` in an `Object.create` property descriptor is evaluated when the property descriptor object is defined. That is why the array was shared across instances. I also recommend never depending on a fixed order for when multiple properties are evaluated in a single call. If you really wanted to initialize one property before the other perhaps just use `Object.defineProperty` in that instance.

Since using `Object.create` does not involve a constructor function you lose the ability to use `instanceof` to test Object identity. Instead use `isPrototypeOf` which is checked against the `prototype` object. This would be `MyFunction.prototype` in the case of a constructor, or the object provided to the first argument of `Object.create`.

```
function Foo() { }

var prototypeDef = {
    protoArray: [],
};

var propertiesDef = {
    propertyArrayValue_: {
        value: null,
        writable: true
    },
    propertyArray: {
        get: function () {
            if (!this.propertyArrayValue_) {
                this.propertyArrayValue_ = [];
            }
            return this.propertyArrayValue_;
        }
    }
}

var foo1 = new Foo();

//old way using instanceof works with constructors
console.log(foo1 instanceof Foo); //logs true

//You check against the prototype object, not the constructor function
console.log(Foo.prototype.isPrototypeOf(foo1)); //true

var foo2 = Object.create(prototypeDef, propertiesDef);

//can't use instanceof with Object.create, test against prototype object...
//...given as first argument to Object.create
console.log(prototypeDef.isPrototypeOf(foo2)); //true
```

[jsfiddle](#)

`isPrototypeOf` will walk down the prototype chain and return `true` if any prototype matches the prototype object tested against.

```

var foo1Proto = {
  foo: "foo",
};

var foo2Proto = Object.create(foo1Proto);
foo2Proto.bar = "bar";

var foo = Object.create(foo2Proto);

console.log(foo.foo, foo.bar); //logs "foo bar"
console.log(foo1Proto.isPrototypeOf(foo)); // logs true
console.log(foo2Proto.isPrototypeOf(foo)); // logs true

```

Sealing objects, freezing them and preventing extensibility

Adding arbitrary properties to random objects and instances, just because you can, has always been at the very least a code smell. With modern browsers and in node.js it is possible to restrict changes to an entire object in addition to restricting individual properties of an object via `defineProperty`, `Object.preventExtensions`, `Object.seal` and `Object.freeze` each in turn add increasingly stricter restrictions on an object. In strict mode violating restrictions placed by these methods will result in JavaScript errors, otherwise they fail silently.

`Object.preventExtensions` will prevent new properties from being added to an object. It will not prevent changes to existing writable properties, and it will not prevent deletion of configurable properties.

`Object.preventExtensions` will also not remove the ability to call `Object.defineProperty` to modify existing properties.

```

var obj = {
  foo: "foo",
};

obj.bar = "bar";
console.log(obj); // logs Object {foo: "foo", bar: "bar"}

Object.preventExtensions(obj);

delete obj.bar;
console.log(obj); // logs Object {foo: "foo"}

obj.bar = "bar";
console.log(obj); // still logs Object {foo: "foo"}

obj.foo = "foobar"
console.log(obj); // logs {foo: "foobar"} can still change values

```

(note you should run the previous jsfiddle with web inspector open at the start or refresh because logging objects with it closed may result in the console showing only the final form the object takes)

`Object.seal` goes further than `Object.preventExtensions`. In addition to preventing new properties from being added to an object, this function also prevents further configurability and removes the ability to delete properties. Once an object has been sealed, you can no longer modify existing properties with `defineProperty`. As previously mentioned if you try to violate restrictions in strict mode the result will be a JavaScript error.

```
"use strict";

var obj = {};

Object.defineProperty(obj, "foo", {
    value: "foo"
});

Object.seal(obj);

//Uncaught TypeError: Cannot redefine property: foo
Object.defineProperty(obj, "foo", {
    value: "bar"
});
```

[jsfiddle](#)

You can also not remove properties anymore, even if they were initially configurable. You can still change the values of properties.

```
"use strict";

var obj = {};

Object.defineProperty(obj, "foo", {
    value: "foo",
    writable: true,
    configurable: true,
});

Object.seal(obj);

console.log(obj.foo); //logs "foo"
obj.foo = "bar";
console.log(obj.foo); //logs "bar"
delete obj.foo; //TypeError, cannot delete
```

[jsfiddle](#)

Finally, `Object.freeze` makes an object entirely immutable. You cannot add, remove, or change the values of properties on frozen objects. You can also no longer use `Object.defineProperty` on the object to change the values of existing properties.

```
"use strict";\n\nvar obj = {\n    foo: "foo1"\n};\n\nObject.freeze(obj);\n\n//All of the following will fail, and result in errors in strict mode\nobj.foo = "foo2"; //cannot change values\nobj.bar = "bar"; //cannot add a property\ndelete obj.bar; //cannot delete a property\n//cannot call defineProperty on a frozen object\nObject.defineProperty(obj, "foo", {\n    value: "foo2"\n});
```

Methods are provided to allow you to test if objects are frozen, sealed or not extensible: `Object.isFrozen`, `Object.isSealed` and `Object.isExtensible`.

_valueOf and toString

You can use `valueOf` and `toString` to customize how an object you have defined behaves in contexts where JavaScripts expects a primitive value.

Here's an example using `toString`:

```
function Foo (stuff) {\n    this.stuff = stuff;\n}\n\nFoo.prototype.toString = function () {\n    return this.stuff;\n}
```

```
var f = new Foo("foo");
console.log(f + "bar"); //logs "foobar"
```

And here's an example using `valueOf`:

```
function Foo (stuff) {
    this.stuff = stuff;
}

Foo.prototype.valueOf = function () {
    return this.stuff.length;
}

var f = new Foo("foo");
console.log(1 + f); //logs 4 (length of "foo" + 1);
```

If you combine both `toString` and `valueOf` you may get unexpected results.

```
function Foo (stuff) {
    this.stuff = stuff;
}

Foo.prototype.valueOf = function () {
    return this.stuff.length;
}

Foo.prototype.toString = function () {
    return this.stuff;
}

var f = new Foo("foo");
console.log(f + "bar"); //logs "3bar" instead of "foobar"
console.log(1 + f); //logs 4 (length of "foo" + 1);
```

A neat way to use `toString` might be to make your object hashable:

```
function Foo (stuff) {
    this.stuff = stuff;
}

Foo.prototype.toString = function () {
    return this.stuff;
}
```

```
var f = new Foo("foo");

var obj = {};
obj[f] = true;
console.log(obj); //logs {foo: true}
```

45 : Object.getOwnPropertyNames and keys

You can use `Object.getOwnPropertyNames` to get all the properties defined on an object. If you're familiar with Python, it is basically analogous to a Python dictionary's `keys` method, and there actually is an `Object.keys` method as well. The difference between `Object.keys` and `Object.getOwnPropertyNames` is that the latter also iterates over property names that are not enumerable, that is `Object.getOwnPropertyNames` will also return property names that are not iterated over in a `for in` loop.

```
var obj = {
    foo: "foo",
};

Object.defineProperty(obj, "bar", {
    value: "bar"
});

console.log(Object.getOwnPropertyNames(obj)); //logs ["foo", "bar"]
console.log(Object.keys(obj)); //logs ["foo"]
```

Symbol

`Symbol` is a special new primitive type defined in ECMAScript 6 harmony and will be available in the next iteration of JavaScript. You can already start using it in [Chrome Canary](#) and [Firefox Nightly](#) and the following jsfiddle examples will only work in these two browsers, at least at the time of writing this post, August 2014.

Symbols can be used as way to create and reference properties on objects.

```
var obj = {};

var foo = Symbol("foo");

obj[foo] = "foobar";

console.log(obj[foo]); //logs "foobar"
```

Symbols are unique and immutable.

```
//console logs false, symbols are unique:
console.log(Symbol("foo") === Symbol("foo"));
```

You can also use symbols with `Object.defineProperty`:

```

var obj = {};

var foo = Symbol("foo");

Object.defineProperty(obj, foo, {
    value: "foobar",
});

console.log(obj[foo]); //logs "foobar"

```

Properties added to objects with symbols will not be iterated over in a `for in` loop, but calling `hasOwnProperty` will work fine:

```

var obj = {};

var foo = Symbol("foo");

Object.defineProperty(obj, foo, {
    value: "foobar",
});

console.log(obj.hasOwnProperty(foo)); //logs true

```

Symbols will not appear in the returned array from a call to `Object.getOwnPropertyNames` but there is a [`Object.getOwnPropertySymbols`](#).

```

var obj = {};

var foo = Symbol("foo");

Object.defineProperty(obj, foo, {
    value: "foobar",
});

//console logs []
console.log(Object.getOwnPropertyNames(obj));

//console logs [Symbol(foo)]
console.log(Object.getOwnPropertySymbols(obj));

```

Symbols could be handy in the case where you want to not only prevent a property from being accidentally modified, but you don't even want it to show up in the normal course of business. I haven't really thought through

all the potential use cases, but I think I can think of more.

Proxy

Another new ECMAScript 6 addition is [Proxy](#). As of August 2014, proxies only work in Firefox. The following jsfiddle example will only work in Firefox, and I actually tested it using Firefox beta because that is what I have installed.

Proxies are exciting to me because it allows for the creation of catch all properties. Check out the following example:

```
var obj = {  
    foo: "foo",  
};  
var handler = {  
    get: function (target, name) {  
        if (target.hasOwnProperty(name)) {  
            return target[name];  
        }  
        return "foobar";  
    },  
};  
var p = new Proxy(obj, handler);  
console.log(p.foo); //logs "foo"  
console.log(p.bar); //logs "foobar"  
console.log(p.asdf); //logs "foobar"
```

[jsfiddle](#) (Firefox only)

In this example we are proxying object `obj`. We define a `handler` object that handles interaction with the proxy object we end up creating. The `get` method on `handler` should be pretty simple to understand. It gets the `target` object as an argument as well as the name of the property that was accessed. We can use this information to return whatever we want, but in this case I return the object's actual value if it has one and if not I return "foobar". I love this, there are so many interesting use cases for this, maybe one could even use this for a kind of type switch as it exists in Scala or use your own imagination.

Another great place where `Proxy` would come in handy is in testing. Beyond just `get` there are other handlers you can add including `set`, `has` and more. Once `Proxy` becomes better supported and more stable I would not mind writing an entire blog post just about `Proxy`. I recommend reading the MDN documentation on [Proxy](#) in full and checking out the provided examples. There's also a jsconf talk about the many ways that `Proxy` is awesome that I recommend checking out: [video slides](#).

So there is way more to JavaScript objects than simply using them as glorified bags of arbitrary data. Even now powerful property definitions are possible, and the future has even more in store as you can see when you think

about all the ways that `Proxy` can change the way that JavaScript will be written. If you have any questions or corrections, pretty please let me know on Twitter, you can find me there as [@bjorntipling](#).

I have updated this blog post to clarify the difference between `Object.keys` and `Object.getOwnPropertyNames` [thanks wging and TazeTSchnitzel](#)

I also updated this post to clarify that Firefox Nightly has `Symbol` [thanks Excavator](#)

46 : JavaScript: Function Invocation Patterns

JavaScript has been described as a *Functional Oriented Language* (this as opposed to Object Oriented Language). The reason is because functions in JavaScript do more than just separate logic into execution units, functions are first class citizens that also provide [scope^{L1}](#) and the ability to create objects. Having such a heavy reliance upon functions is both a blessing and a curse: It's a blessing because it makes the language light weight and fast (the main goal of its original development), but it is a curse because you can very easily shoot yourself in the foot if you don't know what you are doing.

One concern with JavaScript functions is how different invocation patterns can produce vastly different results. This post explains the four patterns, how to use them and what to watch out for. The four invocation patterns are:

1. **Method Invocation**
2. **Function Invocation**
3. **Constructor Invocation**
4. **Apply And Call Invocation**

47 : Function Execution

JavaScript (like all languages these days) has the ability to modularise logic in `functions` which can be invoked at any point within the execution. Invoking a function suspends execution of the current function, passing controls and parameters to the invoked function. In addition, a parameter called `this` is also passed to the function. The invocation operator is a pair of round brackets `()`, that can contain zero or more expressions separated by a comma.

Unfortunately, there is more than one pattern that can be used to invoke functions. These patterns are not *nice-to-know*: They are absolutely essential to know. This is because invoking a function with a different pattern can produce a vastly different result. I believe that this is a language design error in JavaScript, and had the language been designed with more thought (and less haste), this would not have been such a big issue.

48 : The Four Invocation Patterns

Even though there are only one *invocation operator* `()`, there are four *invocation patterns*. Each pattern differs in how the `this` parameter is initialised.

Method Invocation

When a function is part of an object, it is called a *method*. Method invocation is the pattern of invoking a function that is part of an object. For example:

```
var obj = {  
    value: 0,  
    increment: function() {  
        this.value+=1;  
    }  
};  
  
obj.increment(); //Method invocation
```

Method invocation is identified when a function is preceded by `object.`, where `object` is the name of some object. JavaScript will set the `this` parameter to the object where the method was invoked on. In the example above, `this` would be set to `obj`. JavaScript binds `this` at execution (also known as *late binding*).

Function Invocation

Function invocation is performed by invoking a function using `()`:

```
add(2,3); //5
```

When using the function invocation pattern, `this` is set to the global object. This was a mistake in the JavaScript language! Blindly binding `this` to the global object can destroy its current context. It is noticeable when using an *inner function* within a method function. An example should explain things better:

```
var value = 500; //Global variable  
var obj = {  
    value: 0,  
    increment: function() {  
        this.value++;  
  
        var innerFunction = function() {  
            alert(this.value);  
        }  
    }  
};  
  
obj.increment();
```

```

        innerFunction(); //Function invocation pattern
    }
}

obj.increment(); //Method invocation pattern

```

What do you think will be printed to screen? For those that answered 1, you are wrong (but don't be too hard on yourselves, this is because JavaScript does not do things very well). The real answer is 500. Note that `innerFunction` is called using the function invocation pattern, therefore `this` is set to the global object. The result is that `innerFunction` (again, it is important to note that it is invoked with function pattern) will not have `this` set to current object. Instead, it is set to the global object, where `value` is defined as 500. I stress that this is bad language design; the increment function was invoked with the method invocation pattern, and so it is natural to assume the `this` should always point to the current function when used inside it.

There is an easy way to get round this problem, but it is in my opinion a hack. One gets around this problem by assigning a variable (by convention, it is named `that`) to `this` inside the function (aside: This works because functions in JavaScript are [closures](#)^{L2}):

```

var value = 500; //Global variable
var obj = {
    value: 0,
    increment: function() {
        var that = this;
        that.value++;

        var innerFunction = function() {
            alert(that.value);
        }

        innerFunction(); //Function invocation pattern
    }
}
obj.increment();

```

If `this` could be bound to the current object whose scope it is called in, function and method invocations would be identical.

Constructor Invocation [□](#)

Warning: This is another JavaScript peculiarity! JavaScript is not a classical object oriented language. Instead, it is a prototypical object oriented language, but the creators of JavaScript felt that people with classical object orientation experience (the vast majority) may be unhappy with a purely prototype approach. This resulted in JavaScript being unsure of its prototypical nature and the worst thing happened: It mixed classical object orientation syntax with its prototypical nature. The result: A mess!

In classical object orientation, an object is an instantiation of a class. In C++ and Java, this instantiation is performed by using the `new` operator. This seems to be the inspiration behind the constructor invocation pattern...

The constructor invocation pattern involves putting the `new` operator just before the function is invoked. For example:

```
var Cheese = function(type) {
    var cheeseType = type;
    return cheeseType;
}

cheddar = new Cheese("cheddar"); //new object returned, not the type.
```

Even though `Cheese` is a function object (and intuitively, one thinks of functions as running modularised pieces of code), we have created a new object by invoking the function with `new` in front of it. The `this` parameter will be set to the newly created object and the `return` operator of the function will have its behaviour altered. Regarding the behaviour of the `return` operator in constructor invocation, there are two cases:

1. If the function returns a simple type (number, string, boolean, null or undefined), the return will be ignored and instead `this` will be returned (which is set to the new object).
2. If the function returns an instance of `Object` (anything other than a simple type), then that object will be returned instead of returning `this`. This pattern is not used that often, but it may have utility when used with [closures](#)^{L2}.

For example:

```
var obj = {
    data : "Hello World"
}

var Func1 = function() {
    return obj;
}

var Func2 = function() {
    return "I am a simple type";
}

var f1 = new Func1(); //f1 is set to obj
var f2 = new Func2(); //f2 is set to a new object
```

We might ignore the constructor invocation pattern, and just use [object literals](#)^{L3} to make objects, except that the makers of JavaScript have enabled a key feature of their language by using this pattern: Object creation with an arbitrary prototype link (see [previous post](#)^{L4} for more details). This pattern is unintuitive and also [potentially](#)

problematic^{L5}. There is a remedy^{L6} which was championed by Douglas Crockford: Augment `Object` with a `create` method that accomplishes what the constructor invocation pattern tries to do. I am happy to note that as of JavaScript 1.8.5, `Object.create` is a reality and can be used. Due to legacy, the constructor invocation is still used often, and for backward compatibility, will crop up quite frequently.

Apply And Call Invocation

The apply pattern is not as badly thought out as the two preceding patterns. The `apply` method allows manual invocation of a function with a means to pass the function an array of parameters and explicitly set the `this` parameter. Because functions are first class citizens, they are also objects and hence can have methods (functions) run on it. In fact, every function is linked to `Function.prototype` (see [here](#)^{L7} for more details), and so methods can very easily be augmented to any function. The `apply` method is just an augmentation to every function as - I presume - it is defined on `Function.prototype`.

Apply takes two parameters: the first parameter is an object to bind the `this` parameter to, the second is an array which is mapped to the parameters:

```
var add = function(num1, num2) {
    return num1+num2;
}

array = [3,4];
add.apply(null,array); //7
```

In the example above, `this` is bound to null (the function is not an object, so it is not needed) and array is bound to `num1` and `num2`. More interesting things can be done with the first parameter:

```
var obj = {
    data:'Hello World'
}

var displayData = function() {
    alert(this.data);
}

displayData(); //undefined
displayData.apply(obj); //Hello World
```

The example above uses `apply` to bind `this` to `obj`. This results in being able to produce a value for `this.data`. Being able to explicitly assign a value to `this` is where the real value of `apply` comes about. Without this feature, we might as well use `()` to invoke functions.

JavaScript also has another invoker called `call`, that is identical to `apply` except that instead of taking an array of parameters, it takes an argument list. If JavaScript would implement function overriding, I think that `call` would

be an overridden variant of `apply`. Therefore one talks about `apply` and `call` in the same vein.

49 : Conclusion

For better or worse, JavaScript is about to take over the world. It is therefore very important that the peculiarities of the language be known and avoided. Learning how the four function invocation methods differ and how to avoid their pitfalls is fundamental to anyone who wants to use JavaScript. I hope this post has helped people when it comes to invoking functions.

Update

- Wow, this has [reached top of Hacker News](#)^{L8}. Thanks for upvoting it everybody.
- Now its also gonna be [published](#)^{L9} :)

Links

1. <http://doctrina.org/JavaScript:Why-Understanding-Scope-And-Closures-Matter.html>
2. <http://doctrina.org/JavaScript:Why-Understanding-Scope-And-Closures-Matter.html#closures>
3. <http://doctrina.org/Javascript-Objects-Prototypes.html#cof1>
4. <http://doctrina.org/Javascript-Objects-Prototypes.html>
5. <http://doctrina.org/Javascript-Objects-Prototypes.html#pcip>
6. <http://doctrina.org/Javascript-Objects-Prototypes.html#remedy>
7. <http://doctrina.org/Javascript-Objects-Prototypes.html#cfo>
8. <http://doctrina.org/static/images/TopOfHackerNews.png>
9. <http://hackermonthly.com/issue-32.html>

Thank you for printing this article. Please do not forget to come back to <http://doctrina.org> for fresh articles.

50 : Javascript: Object Prototypes

<http://doctrina.org/Javascript-Objects-Prototypes.html#cfo>

This post discusses JavaScript objects with emphasis on its prototype linkings. After reading this post, you should understand the following:

- Object creation in JavaScript.
- Prototype linkings.
- What `Object.prototype` and `Function.prototype` are used for.

[Douglas Crockford's](#)^{L1} wonderful book [JavaScript: The Good Parts](#)^{L2} does a fanastic job of explaining this topic, and I urge the interested reader to buy his book.

51 : JavaScript Objects [\[link\]](#)

In JavaScript the simple types are *numbers*, *strings*, *booleans* (*true* and *false*), *null* and *undefined*. All other values are objects. Even though there are multiple ways to create objects, there are really only two *atomic* mechanisms that are used:

1. Literals.
2. Functions.

Creating Objects From Literals [\[link\]](#)

Creating objects from literals is very simple. The following is an *object literal*:

```
var person = {
    "First_Name": "Barry",
    "Last_Name": "Steyn",
    "do_Something": function() { alert('something'); }
}
```

Object Literals do not look like traditional objects. In fact, Object Literals look more like a structure that can hold things. There is however one difference between a JavaScript object and a structure that can hold things: All JavaScript objects have a *prototype* linking.

The Prototype Link [\[link\]](#)

Every Object has a *hidden* link to another object called a *prototype link*. Objects by default are linked to the object `Object.prototype`. The prototype link is only used when retrieving property values from an object (it is not touched when updating an object). When retrieving a property value from an object, if it cannot be found, it will look for it in the prototype object that it is linked to. For example:

```
person.age; // undefined - property age does not exist
```

JavaScript tried to find the property `age` on `person`, and it could not. So then it tried to find it on `Object.prototype` and it could not find it either. So it returned "*undefined*".

```
Object.prototype.age = 56; // Every object is linked to Object.prototype by default
person.age; // 56
```

JavaScript could not find `age` on `person`, but it did find it on `Object.prototype`, which is linked to `person`. Therefore the prototype linkage is used to *augment* an object with additional properties. In a classical object oriented language like Java or C++, augmentation is performed using inheritance. And so a prototype link can be considered as providing the property augmentation feature of object inheritance.

Creating Function Objects [\[link\]](#)

Functions in JavaScript are objects and are [first class citizens](#)^{L3}. A variable can be assigned a function object like so:

```
var addNum = function(num1, num2) {
    return num1 + num2;
}

typeof addNum; //function - which is an object in JavaScript
```

Since functions are objects, they too have a prototype link. But where object literals are by default linked to `Object.prototype`, function objects are linked to `Function.prototype`. There is one other difference that function objects have (and this is where it starts get confusing); function objects have in addition to a prototype link a *property* called `prototype`. This property (again, the `prototype` property **is not** the prototype link) can be manipulated (its use will be explained in the next section).

Just like `Object.prototype` is used to augment objects with properties, so `Function.prototype` is used to augment functions with properties.

```
var result = addNum(3,4); // 7

//Add a function to Function.prototype
Function.prototype.subtract = function(num1, num2) {
    return num1 - num2;
}

//We can now use subtract on any function object
result = addNum.subtract(5,2); //3
```

`Function.prototype` is itself an object (big surprise there), and therefore it is linked to `Object.prototype`. If something cannot be found in `Function.prototype`, it will be searched for on `Function.prototype`'s linked prototype object, which is `Object.prototype`.

```
Object.prototype.denominator = 10;
Object.prototype.divide = function(num1) {
    return num1 / denominator;
}

var result = addNum.divide(50); // 5
```

In the above example, `addNum` was searched for `divide`. Then `Function.prototype` was searched, after which, `Object.prototype` was searched, where it was eventually found. Therefore prototype links are chained together, whereby if a property is not found in the current object, it then searches for it in its linked prototype object. And if

that object does not have the property, it is then searched for in that object's linked prototype object, and so on, until the process ends with `Object.prototype`.

The process described above is the mainstay of [Prototype Based Programming](#)^{L4}, and it is called *prototype chaining*.

Creating Objects From Functions □

This section should not be confused with the previous section. Their titles share similar wording, but in this section, we are going to talk about how to create a new object using a function, and last section we talked about creating function objects.

To recap from the last section, this is how a *function object* is created:

```
var Animal = function(name) {
    this.name = name;
}
```

The variable `Animal`:

- Is a function and an object (remember, everything is an object in JavaScript).
- It has a *prototype property*, and its prototype link is set to `Function.prototype` by default.

But something special happens here if the function is invoked with the reserved word `new` (called the [constructor invocation pattern](#)^{L5}):

```
var cat = new Animal('fluffy'); //Using new to invoke the function
```

What happens here is that `new` creates an object from the function object `Animal`. Its power comes in because it will use `Animal`'s *prototype property* as the *prototype link* for the new object. And since `Animal` can set the value of its prototype property, objects constructed in this way do not have to be linked to `Object.prototype`. Got that? If not, here is an example which should explain things:

```
var species = {
    "species": "human"
}

var Entity = function(name) {
    this.name = name;
}

var alien = new Entity("Yoda");

Entity.prototype = species;
var human = new Entity("Jason");
```

In the above, the `alien` and `human` objects are constructed from the function `Entity` via the *constructor invocation pattern* that is initiated by putting the `new` keyword in front of the function name. When `alien` is constructed, `Entity`'s prototype property has not been altered, and so `alien` is linked to `Object.prototype`. But when constructing `human`, `Entity`'s prototype property was altered to `species`, and so the prototype object linked to `human` is `species`, which in turn is itself linked to `Object.prototype`:

```
alien.species; // undefined
human.species; // human
```

The problem with using the "constructor invocation pattern" [↳](#)

Using `new` is one of four ways in which to invoke a function, the most popular way being to call the function. For example:

```
var Vehicle = function(type) {
    this.type = type;
    return type;
}

var car = Vehicle("Toyota"); // Normal function call
var plane = new Vehicle("Boeing");
```

In the example above, `car` was assigned the result of invoking `Vehicle` as a normal function, which in this case was *Toyota*. But `plane` on the other was assigned the result of using the *constructor invocation pattern* which is an object (the return is ignored). This is terrible, and it is very easy to make a mistake.

One recommended approach is to always start functions that are to be invoked with `new` with *capital letters*. This should help for readability, but when someone is spending days trying to hunt for a bug, this will be little compensation. Instead, a much better way would be to augment `Object` with a `create` method if it does not exist:

```
if (typeof Object.create !== 'function') { //If working with a version of JavaScript
prior to 1.8.5

Object.create = function(o) {
    var F = function() {};
    F.prototype = o;
    return new F();
}
}
```

Objects can now be created with a chosen prototype link by doing the following:

```
var animal = {  
    "type": "cat",  
    "make_sound": function() {  
        return "purrrr";  
    }  
}  
  
var cat = Object.create(animal);  
cat.make_sound(); //purrrr  
cat.type; //cat
```

52 : Conclusion

Object orientation in JavaScript is a bit of a mess. This is because JavaScript is quite unsure about what it is, and ends up being a mix of a classical object oriented language and a prototype based language. Hence we get the `new` keyword, which makes JavaScript look like a classical object oriented language, even though it is not.

To summarise, here are the three objects this blog discusses:

```
//Object Literal
var objectLiteral = {
    "property1": "value1"
};

//Function Object
var functionObject = function() {
}

//Object From Function
var objectFromFunction = new functionObject();
```

And to summarise, the prototype linkings and properties available to each object:

Object Literal	Function Object	Object From Function
<ul style="list-style-type: none"> <i>Construction</i> - via curly braces <code>{}</code> <i>Prototype Link</i> - <code>Object.prototype</code> <i>Prototype Property</i> - does not exist 	<ul style="list-style-type: none"> <i>Construction</i> - via the <code>function</code> keyword <i>Prototype Link</i> - <code>Function.prototype</code> <i>Prototype Property</i> - exists 	<ul style="list-style-type: none"> <i>Construction</i> - via <code>new</code> keyword with function object <i>Prototype Link</i> - Whatever the function's (that was invoked by <code>new</code>) <code>prototype</code> property is set to (<code>Object.prototype</code> by default) <i>Prototype Property</i> - does not exist

Links

1. http://en.wikipedia.org/wiki/Douglas_Crockford
2. http://www.amazon.com/JavaScript-Good-Parts-Douglas-Crockford/dp/0596517742/ref=sr_1_1?ie=UTF8&qid=1346978272&sr=8-1&keywords=javascript+the+good+parts
3. http://en.wikipedia.org/wiki/First-class_citizen
4. http://en.wikipedia.org/wiki/Prototype-based_programming
5. <http://doctrina.org/Javascript-Function-Invocation-Patterns.html#ci>

Thank you for printing this article. Please do not forget to come back to <http://doctrina.org> for fresh articles.

53 : Maps, Sets and Iterators in JavaScript

<http://bjorn.tipling.com/maps-sets-and-iterators-in-javascript>

In my last JavaScript post I wrote about how to make [advanced usage of objects in JavaScript](#). Continuing in a similar direction, in this post I want to look at some new kinds of objects in JavaScript. I will look at [maps](#) and [sets](#), including their “weak” counterparts, [weakmaps](#) and [weaksets](#), also the new [for...of](#) iterator and the new type [Array Iterator](#).

As of September 2014, these new features are not fully implemented across all browsers yet, so most of these examples will only work in Firefox and [Chrome Canary](#). I am using [Firefox Nightly](#) and Chrome Canary to create and test the jsfiddle examples. It may be the case that the examples also work in some of the beta versions, but I haven’t checked.

I am going to begin by looking at the new `Map` type in ECMAScript6.

Maps

Developers usually just use regular JavaScript objects when they want maps. Stuff is mapped in an object by using strings as a key and this is possible with any kind of Object in Javascript.

```
var obj = {};
obj.foo = "bar";
console.log(obj.foo); //logs "bar"
```

In ECMAScript 6, the new type `Map` can be used very similar to a regular object.

```
var map = new Map();
map.set("foo", "bar");
console.log(map.get("foo")); //logs "bar"
```

[jsfiddle](#) (*created and tested with Firefox Nightly*)

You will immediately notice that a map isn’t created with the literal object syntax, and that one uses `set` and `get` methods to store and access data. Map constructors can also take an argument, something we’ll look at later. Since maps are just objects it is possible to store arbitrary data on a map using the `.` notation but then you are not going to get to make use of the benefits of maps and you probably never want to do such a thing with a Map type. Map instances can be queried and manipulated with more than just `set` and `get`.

```
var animalSounds = new Map();

animalSounds.set("dog", "woof");
animalSounds.set("cat", "meow");
animalSounds.set("frog", "ribbit");
```

```

console.log(animalSounds.size); //logs 3
console.log(animalSounds.has("dog")); //logs true

animalSounds.delete("dog");

console.log(animalSounds.size); //logs 2
console.log(animalSounds.has("dog")); //logs false

animalSounds.clear();
console.log(animalSounds.size); //logs 0

```

[jsfiddle](#) (created and tested with Firefox Nightly)

In this example you can see that the `size` property on a map instance gives you the count of keys and values stored in a Map. If you were using regular Objects you'd have to keep track of the number of things in the Object yourself or iterate and count. With map this isn't necessary. Also available is a `has` method to test if a key is stored in the `map`, a `delete` method to delete a key from a map and a `clear` to clear all keys and values in a map.

The Map type, unlike regular objects in JavaScript, allow you to use any type as a key to refer to data, not just strings. This is a big improvement, especially if you have been storing data in objects using numbers. Consider the following:

```

var userId, usersObj, usersMap;

usersObj = {
  1: "sally",
  2: "bob",
  3: "jane"
};

console.log(usersObj[1]); //logs "sally", toString called on 1

for (userId in usersObj) {
  console.log(userId, typeof userId); //logs 1..3, "string"
  if (userId === 1) {
    console.log("This is never logged because userId is a string.");
  }
}

```

With objects, developers have been able to use numbers as property names, or keys as it were, but when this is done the number is actually being turned into a string.

```
console.log(usersObj[1]); //logs "sally", toString called on 1

"sally"
```

A number as a key actually works here [because `toString` is being called on the number](#) given to the object.

However when you iterate over `usersObj` it becomes clear that something is amiss.

```
for (userId in usersObj) {
  console.log(userId, typeof userId); //logs 1..3, "string"
  if (userId === 1) {
    console.log("This is never logged because userId is a string.");
  }
}
```

Here we see `userId` is a string and the `console.log` statement inside the `if` statement is never executed because we mistakenly expected `userId` to be a number.

```
"1" "string"
"2" "string"
"3" "string"
```

If you have been using numbers as primary keys, as many databases will do by default, you can run into trouble if you want to iterate over your JavaScript object to look for an id. You will not find your id unless you know that the key for an Object is always of type string. Numbers are simply being converted to strings.

With maps however, the type of the key stays whatever it was when you set it.

```
usersMap = new Map();
usersMap.set(1, "sally");
usersMap.set(2, "bob");
usersMap.set(3, "jane");

console.log(usersMap.get(1)); //logs "sally"
usersMap.forEach(function (username, userId) {
  console.log(userId, typeof userId); //logs 1..3, "number"
  if (userId === 1) {
    console.log("We found sally.");
  }
});
```

[jsfiddle](#) (created and tested with Firefox Nightly)

```
1 "number"
"We found sally."
2 "number"
3 "number"
```

The number stays as type number. You never need convert your id to a string or convert the Object's key to a number when you're using a map.

In addition to strings and numbers, you can use other primitive types as keys. Here is an example using a boolean as a key.

```
var map;

map = new Map();
map.set(true, [
    "1 + 1 = 2",
    "cows are animals",
    "grass is green",
]);
map.set(false, [
    "the moon is made from cheese",
    "the earth is flat",
    "pyramids were made by aliens",
]);
map.forEach(function (value, key) {
    console.log(typeof key, key);
});
```

[jsfiddle](#) (created and tested with Firefox Nightly)

This logs:

```
"boolean" true
"boolean" false
```

You can even use Objects as a key, here is an example of such a map:

```
var obj, map;

map = new Map();
obj = {foo: "bar"};
```

```
map.set(obj, "foobar");

obj.newProp = "stuff";

console.log(map.has(obj)); //logs true

console.log(map.get(obj)); //logs "foobar"
```

[jsfiddle](#) (created and tested with Firefox Nightly)

Even `Nan` can be a key.

```
var map, errors, result;
errors = new Map();
errors.set(NaN, "That was not a number!");
errors.set(Infinity, "That was infinity!");
[
  1 + 2,
  1/0,
  1/"foo",
].forEach(function (result) {
  if (errors.has(result)) {
    console.log(errors.get(result));
  } else {
    console.log("The result was", result);
  }
});
```

[jsfiddle](#) (created and tested with Firefox Nightly)

Result:

```
"The result was" 3
"That was infinity!"
"That was not a number!"
```

Even though `Nan != Nan` you can use `Nan` this way with Maps.

```
var errors;
errors = new Map();
errors.set(NaN, "Not a number!");
```

```
console.log(NaN === NaN); //logs false
console.log(errors.has(NaN)); //logs true
```

[jsfiddle](#) (created and tested with Firefox Nightly)

It is totally possible to mix and match types in a map, here is a pretty silly example:

```
var map = new Map();

map.set(1, "one");
map.set("two", 2);
map.set(true, false);
map.set({foo: "bar"}, ["foo", "bar"]);

console.log(map.size); //logs "4"
```

[jsfiddle](#) (created and tested with Firefox Nightly)

There are multiple ways to iterate over a `Map`, some of which we will look at in just a bit, but a very straight forward way is to use a map instance's `forEach` method.

```
var map = new Map();

map.set(1, "one");
map.set("two", 2);
map.set(true, false);
map.set({foo: "bar"}, ["foo", "bar"]);

map.forEach(function (value, key, mapObj) {
    console.log("value:", value, "key:", key, "map", mapObj === map);
});
```

[jsfiddle](#)

Result:

```
"value:" "one" "key:" 1 "map" true
"value:" 2 "key:" "two" "map" true
"value:" false "key:" true "map" true
"value:" Array [ "foo", "bar" ] "key:" Object { foo: "bar" } "map" true
```

Much like the `Array.prototype.forEach`, a map's `forEach` is given a function that is called for each value set on the map. This function is given three parameters: the value, the key, and a reference to the map object itself as the final argument.

for...of

We are not just yet done talking about maps, but we need to diverge for a bit because before I can go on I need to introduce iterators. ECMAScript 6 introduces some new ways to iterate over data and I will only mention two of them in this post, `for...of` and the new `Array Iterator` type.

`for...of` is a new type of iteration statement that differs from `for...in` in that it iterates over the values instead of indexes or keys, and that it can be used with maps and sets. Before we switch back to maps let's take a look at iterating over arrays.

```
var i, value, a;

a = ["a", "b", "c"];
a.foo = "bar";
for (i in a) {
    console.log(i);
}

for (value of a) {
    console.log(value);
}
```

[jsfiddle](#) (created and tested with Firefox Nightly)

Usually you would iterate over an array with a `for(i = 0; i < a.length; i++)` or use its `forEach` but you could also iterate via `for...in` but this example demonstrates a problem with that as `for...in` also iterates over properties of the array object.

```
a = ["a", "b", "c"];
a.foo = "bar";
for (i in a) {
    console.log(i);
}
```

```
0
1
2
foo
```

Notice the “foo”. Instead of iterating over the indexes and property names, `for...of` iterates over its values.

```

for (value of a) {
    console.log(value);
}

```

```

a
b
c

```

Notice the lack of “foo”. Only array values are iterated over, not object values. As a quick aside I should also note that my experience has been that I have found `for...in` to be considerably slower in the past than the available alternatives. I will avoid looking at the performance of `for...of` until it has made it into stable versions of Chrome and other browsers.

You cannot use `for...of` with just any object, it will result in a JavaScript error.

```

var value, obj = {foo1: "bar1", foo2: "bar2"};

//This will result in a JavaScript error:
for (value of obj) {
    console.log(value);
}

```

[jsfiddle](#) (created and tested with Firefox Nightly)

You can use `for...of` on arrays, array like types like `arguments` and node lists, generators, maps and sets.

Sets we'll take a look later and generators are beyond the scope of this blog post. Here is `for...of` with a map:

```

var data, map = new Map();

map.set("dog", "woof");
map.set("cow", "moo");

for (data of map) {
    console.log(data);
}

```

[jsfiddle](#) (created and tested with Firefox Nightly)

Each iteration of a map with `for...of` is given an array with two values.

```

Array [ "dog", "woof" ]
Array [ "cow", "moo" ]

```

Each pair has the key at index 0 and its value at index 1.

I find maps to be preferable to Objects for several reasons, but not the least is the simple iteration. `for...of` is superior than a `for...in` as object values are not iterated over, no more need for checking `hasOwnProperty`, and you get the index and the value. When to use `for...of` or the maps `forEach` depends what is simpler., `for...of` avoids the extra function call for each iteration, avoids the need to binding a `thisArg` and is simply easier to look at.

Array Iterator

Continuing on, the other new method of iteration is the `Array Iterator` type which is also applicable to maps, sets and arrays. An array iterator is an object that has a `next` method which when called returns an object with `value` and `done` properties. The `next` method is used to iterate, the `value` of the returned object is the value obviously, and `done` will tell you when iteration has finished. There are new array and map methods that return array iterators.

```
function logIterator(iterator) {
    var current;
    while(true) {
        current = iterator.next();
        if (current.done) {
            break;
        }
        console.log(current.value);
    }
}

logIterator(["a","b", "c", "d"].entries());
logIterator(["a","b", "c", "d"].keys());

var map = new Map();
map.set(0, "a");
map.set(1, "b");
map.set(2, "c");
map.set(3, "d");
logIterator(map.entries());
logIterator(map.keys());
logIterator(map.values());
```

[jsfiddle](#) (created and tested with Firefox Nightly)

The function `logIterator` takes an `Array Iterator` as an argument and iterates over it with a `while` loop. It calls `next` to iterate and logs the return value. It checks `done` to break out of the `while`.

Here we use the array's `entries` and `keys` methods to get `Array Iterators` and log their values.

```
logIterator(["a", "b", "c", "d"].entries());
logIterator(["a", "b", "c", "d"].keys());
```

Result:

```
[0, "a"]
[1, "b"]
[2, "c"]
[3, "d"]
0
1
2
3
```

The `entries` method returns an `Array Iterator` the value of which will be an Array pair, much like you get for a `for...of` when iterating over a map. At index 0 you get the current iteration's index, at index 1 you get the value for the current iteration.

The `keys` method creates an `Array Iterator` that simply returns an index, or key as it were, for the Array at each iteration.

The map methods `entries` and `keys` behaves like an Array, the additional `values` method iterates over the map's values:

```
var map = new Map();
map.set(0, "a");
map.set(1, "b");
map.set(2, "c");
map.set(3, "d");
logIterator(map.entries());
logIterator(map.keys());
logIterator(map.values());
```

Result:

```
[0, "a"]
[1, "b"]
[2, "c"]
[3, "d"]
0
1
```

```
2
3
"a"
"b"
"c"
"d"
```

Very much the same output. Note the Map returned data in the order it was given but there is absolutely nothing said in the specification of Maps being sorted nor of any order guarantee. If you want that, use an Array or a Set.

Unless you are doing something special with an `Array Iterator` you don't need to manually write a function that calls `next` and checks `done`. You can just use `for...of`:

```
var map = new Map();
map.set(0, "a");
map.set(1, "b");
map.set(2, "c");
map.set(3, "d");

for (data of map.entries()) {
    console.log(data);
}
```

[jsfiddle](#)

This prints exactly as what it did when we used `logIterator` before.

Now that we know what `Array Iterator` is, we can look at the constructor arguments for maps. You can create a map based of an Array Iterator.

```
var map = new Map(["a","b", "c", "d"].entries());

console.log(map); //logs Map { 0: "a", 1: "b", 2: "c", 3: "d" }
console.log(map.get(0)); //logs "a"
```

[jsfiddle](#) (created and tested with Firefox Nightly)

You can also just use arrays of pairs:

```
var map = new Map([["dog", "woof"], ["cow", "moo"]]);

console.log(map); //logs Map { dog: "woof", cow: "moo" }
console.log(map.get("dog")); //logs "woof"
```

[jsfiddle](#) (created and tested with Firefox Nightly)

Duplicate keys in this case seem to override previous values in both Chrome Canary and Firefox Nightly.

```
var map = new Map([["dog", "woof"], ["cow", "moo"], ["dog", "ribbit"]]);

console.log(map); //logs Map { dog: "ribbit", cow: "moo" }
console.log(map.get("dog")); //logs "ribbit"
```

[jsfiddle](#) (created and tested with Firefox Nightly)

Weakmaps



"Do you even lift?"

WeakMaps are much like Maps except for some important differences. The first important difference is that you can only use objects as keys for a `WeakMap`. You cannot use numbers, strings or other primitive types as keys. The second difference is that these keys are weakly held. Objects used as keys by a `WeakMap` can be garbage collected if there are no other references to the object elsewhere in your application. Another important difference is that WeakMaps only have a subset of the methods maps have and cannot be iterated over in a `for...of` loop.

```
var data, wmap = new WeakMap();
wmap.set({foo: "bar"}, "foobar");

//This is a JavaScript error:
for (data of wmap) {
    console.log(data);
}
```

[jsfiddle](#)

The methods a `WeakMap` has are limited, to `get`, `set`, `has`, `delete` and `clear`.

One use case for a `WeakMap` could be to store metadata for an object. You do not have to manually clean up a `WeakMap`'s references when you want to remove references to objects. For example, in the browser it is possible to create unintentional memory leaks by keeping references to HTML elements around in your JavaScript. Even though you may have removed HTML elements from the DOM, if you still have references to them somewhere in your JavaScript they may not be garbage collected. If you use a `WeakMap` you don't have to worry about this. I am not saying that maintaining state in the DOM is a good idea, in fact I think it is probably a bad idea that I have seen lead to a lot of problems, but you could do it with a `WeakMap` and not have it lead to memory leaks. A great example for a good use case might be an SVG based line chart. You could store the details for a point in a `WeakMap` to show in a tooltip. When the user hovers over the point, the tooltip uses the SVG element hovered over to query the `WeakMap` for information for the tooltip. When the line chart updates and the point goes away, the reference in the `WeakMap` should be cleared up and not present a problem.

For the purpose of this blog post I will use a less ideal example. I will use a shopping cart that maintains state using elements. Again, this is only for demonstration purposes, it is probably best to store state not using elements.

As of September 2014 the following example only works in Firefox Nightly, only because I used ES6 string templates, sorry.

```
var prices, shoppingCart = new WeakMap();

prices = new Map([
    ["smartphone", new Map([
        ["iPhone6", 610],
        ["galaxyS5", 650],
        ["motoX", 179],
    ])],
    ["accessory", new Map([
        ["stylus", 20],
        ["case", 40],
    ])],
]);
;

function getProductFromShelf(type) {
    var productEl, clone, price;
    productEl = document.getElementById("shelf").querySelector(`.${type}`);
    price = prices.get(productEl.classList[1]).get(type);
    clone = productEl.cloneNode(true);
    document.getElementById("shoppingCart").appendChild(clone);
    shoppingCart.set(clone, price);
}
```

```

}

function printTotal() {
    var products, total = 0;
    products = document.getElementById("shoppingCart")
        .querySelectorAll("div");
    for (product of products) {
        total += shoppingCart.get(product);
    }
    document.getElementById("total").textContent = `$$\{total\}`;
}

getProductFromShelf("iPhone6");
getProductFromShelf("motoX");
getProductFromShelf("case");

printTotal();

```

[jsfiddle](#) (works in Firefox Nightly only)

This code clones HTML elements from the shelf and puts them into the shopping cart HTML element. When this is done an instance of `WeakMap`, `shoppingCart`, maintains metadata associated with the product put into the shopping cart. When this code wants to calculate the total it iterates over all the elements in the shopping cart and grabs the products associated price in `shoppingCart` and adds it up. In this example the total ends up being \$829. When we want to remove items from the shopping cart, we need only remove the element from the DOM, we don't have to remove any metadata from `shoppingCart`.

As a quick note I made use of ECMAScript 6 string templates:

```
productEl = document.getElementById("shelf").querySelector(`.\${type}`);
```

Note the:

```
`.\${type}`
```

This is new in Firefox Nightly and I wanted to try it out. It references the `type` variable so this string will become `".iphone6"` or `".motoX"` for example, depending on the value in `type`. I don't have to use the ugly string concatenation `+` operator to make hard to read strings.

Sets

New in JavaScript is the `Set` type which is a unique list of values enumerable in insertion order. Unlike Python's `set` it does not provide set operation functionality like finding differences, intersections and unions. It's simply a unique list.

```
var set = new Set(["a", "a", "e", "b", "c", "b", "b", "b", "d"]);
console.log(set);
```

[jsfiddle](#)

Result:

```
Set [ "a", "e", "b", "c", "d" ]
```

And when iterated over it maintains insertion order:

```
var value, set = new Set(["a", "a", "e", "b", "c", "b", "b", "b", "d"]);

for (value of set) {
  console.log(value);
}
```

[jsfiddle](#)

```
"a"
"e"
"b"
"c"
"d"
```

Sets can't be accessed like an array.

```
var set = new Set(["a", "a", "e", "b", "c", "b", "b", "b", "d"]);
console.log(set[0]); //Logs undefined
```

[jsfiddle](#)

Sets provide the same methods as a `Map`.

```
var set = new Set(["a", "a", "e", "b", "c", "b", "b", "b", "d"]);

console.log(set.size); //Logs 5
console.log(set.has("a")); //Logs true
console.log(set.delete("a")); //Logs true, was deleted
console.log(set.has("a")); //Logs false
console.log(set.delete("a")); //Logs false, wasn't deleted, wasn't there.
```

```

set.forEach(function (value, key, setObj) {
    console.log(value, key, key === value, set === setObj);
});

set.clear();

console.log(set.size); //Logs 0

set.add("a")
    .add("b")
    .add("c");

console.log(set.has("a")); //Logs true
console.log(set.size); //Logs 3

```

[jsfiddle](#)

Of interest is that delete lets you delete something from a set by value, whereas with an array you would probably do a splice using the value's index. I like this about `Set`. Also weird, but interesting is that function for the `forEach` on a `Set` gets a value and a key even though there are only values in a set and thus you are given the same value for both arguments.

Sets also have the `entries`, `keys` and `values` methods that a `Map` has and they give you the same kind of `Array Iterator` type but in the same vein as the Set's `forEach` the entries pair is `[value, value]`, that is you get an array of identical pair values. In addition a Set's `keys` and `values` are identical operations, they both return the same kind of iterator with the same values.

```

var data, set = new Set(["a", "a", "e", "b", "c", "b", "b", "b", "d"]);

for (data of set.entries()) {
    console.log(data); //Logs ["a", "a"]...
}

for (value of set.values()) {
    console.log(value);
}

//Logs the same thing as values() above
for (key of set.keys()) {
    console.log(key);
}

```

[jsfiddle](#)

WeakSet

`WeakSet` is similar to `WeakMap`. You can only store objects, no primitives and you cannot iterate over a `WeakSet`. The available methods are `add`, `has`, `delete` and `clear`. There is no `size` property. Objects are weakly held, if an object held is garbage collected it will not lead to a memory leak. Items held in a `WeakSet` like a `Set` are unique, only one of each will be held. The use case for a `WeakSet` is limited. You could use it to see if you have seen an Object before. For example you could check to see if a user clicked an element before.

As of September 2014, this example does not work in Chrome Canary.

```
var button, wset = new WeakSet();

for (button of document.querySelectorAll("button")) {
    button.addEventListener("click", function (event) {
        if (wset.has(event.target)) {
            alert("You have clicked this button before!");
        } else {
            alert("First time clicking this button");
        }
        wset.add(event.target);
    });
}
```

[jsfiddle](#)

Perhaps this can be used to prevent double clicking. It wont leak memory so it's safe to use the HTML element as a key.

I am really excited about Maps and Sets and the new `for...of` iterator. I can't wait until they are more widely available. Once they are a lot of ugly anti-patterns necessary to get things done can be thrown in the delete bin. JavaScript is really getting better every day. You should thank a browser developer whenever you meet one for all they have done to make such a great platform. If you notice any mistakes or have questions please don't hesitate to let me know, I'm [@bjorntipling](#) on Twitter.

54 : ES6 Iterators and Generators in Practice

This article is a section from the course ES6 in Practice. I created this course during the last couple of months, because there is an evident need for a resource that helps JavaScript developers put theory into practice.

This course won't waste your time with long hours of video, or long pages of theory that you will never use in practice. We will instead focus on learning just enough theory to solve some exercises. Once you are done with the exercises, you can check the reference solutions, and conclude some lessons. In fact, some of the theory are sometimes placed in the reference solutions.

If you like this article, check out the course [here](#).

This section is about iterators and generators.

It is worth for you to learn about iterators, especially if you are a fan of lazy evaluation, or you want to be able to describe infinite sequences. Understanding iterators also helps you understand generators, promises, sets, and maps better.

Once we cover the fundamentals of iterators, we will use our knowledge to understand how generators work.

Iterables and Iterators

ES6 comes with the *iterable* protocol. The protocol defines iterating behavior of JavaScript objects.

An *iterable object* has an iterator method with the key `Symbol.iterator`. This method returns an *iterator object*.

```
let iterableObject = {
  [Symbol.iterator]() { return iteratorObject; }
};
```

`Symbol.iterator` is a *well known symbol*. If you don't know what well known symbols are, [read the lesson about symbols](#).

We will now use `Symbol.iterator` to describe an iterable object. Note that we are using this construct for the sake of understanding how iterators work. Technically, you will hardly ever need `Symbol.iterator` in your code. You will soon learn another way to define iterables.

An *iterator object* is a data structure that has a `next` method. When calling this method on the iterator, it returns the next element, and a boolean signalling whether we reached the end of the iteration.

```
// Place this before iterableObject
let iteratorObject = {
  next() {
```

```

        return {
            done: true,
            value: null
        };
    }
};


```

The return value of the `next` function is an object with two keys:

- `done` is treated as a boolean. When `done` is truthy, the iteration ends, and `value` is not considered in the iteration
 - `value` is the upcoming value of the iteration. It is considered in the iteration if and only if `done` is falsy.
- When `done` is truthy, `value` becomes the return value of the iterator

Let's create a countdown object as an example:

```

let countdownIterator = {
    countdown: 10,
    next() {
        this.countdown -= 1;
        return {
            done: this.countdown === 0,
            value: this.countdown
        };
    }
};

let countdownIterable = {
    [Symbol.iterator]() {
        return Object.assign( {}, countdownIterator )
    }
};

let iterator = countdownIterable[Symbol.iterator]();

iterator.next();
> Object {done: false, value: 9}

iterator.next();
> Object {done: false, value: 8}

```

Note that the state of the iteration is preserved.

The role of `Object.assign` is that we create a shallow copy of the iterator object each time the iterable returns an iterator. This allows us to have multiple iterators on the same iterable object, storing their own internal state.

Without `Object.assign`, we would just have multiple references to the same iterator object:

```
let secondIterator = countdownIterable[Symbol.iterator]();
let thirdIterator = countdownIterable[Symbol.iterator]();

console.log( secondIterator.next() );
> Object {done: false, value: 9}

console.log( thirdIterator.next() );
> Object {done: false, value: 9}

console.log( secondIterator.next() );
> Object {done: false, value: 8}
```

We will now learn how to make use of iterators and iterable objects.

Consuming iterables

Both the `for-of` loop and the spread operator can be used to perform the iteration on an iterable object.

```
for ( let element of iterableObject ) {
  console.log( element );
}

console.log( [...iterableObject] );
```

Using the countdown example, we can print out the result of the countdown in an array:

```
[...countdownIterable]
> [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Language constructs that consume iterable data are called *data consumers*. We will learn about other data consumers soon.

Built-in Iterables

Some JavaScript types are iterables:

- Arrays are iterables, and work well with the `for-of` loop
- Strings are iterables as arrays of 2 to 4 byte characters
- DOM data structures are also iterables. If you want proof, just open a random website, and execute `[...document.querySelectorAll('p')]` in the console
- Maps and Sets are iterables. See the next section for more details

Let's experiment with built-in iterables a bit.

```
let message = 'ok';

let stringIterator = message[Symbol.iterator]();
let secondStringIterator = message[Symbol.iterator]();

stringIterator.next();
> Object {value: "o", done: false}

secondStringIterator.next();
> Object {value: "o", done: false}

stringIterator.next();
> Object {value: "k", done: false}

stringIterator.next();
> Object {value: undefined, done: true}

secondStringIterator.next();
> Object {value: "k", done: false}
```

Before you think how cool it is to use `Symbol.iterator` to get the iterator of built-in datatypes, I would like to emphasize that using `Symbol.iterator` is generally not cool. There is an easier way to get the iterator of built-in data structures using the public interface of built-in iterables.

You can create an `ArrayIterator` by calling the `entries` method of an array. `ArrayIterator` objects yield an array of `[key, value]` in each iteration.

Strings can be handled as arrays using the spread operator:

```
let message = [...'ok'];
```

```

let pairs = message.entries();

for( let pair of pairs ) {
    console.log( pair );
}

> [0, "o"]
> [1, "k"]

```

Iterables with Sets and Maps

The `entries` method is defined on sets and maps. You can also use the `keys` and `values` method on a set or map to create an iterator/iterable of the keys or values. Example:

```

let colors = new Set( [ 'red', 'yellow', 'green' ] );
let horses = new Map( [[5, 'QuickBucks'], [8, 'Chocolate'], [3, 'Filippone']] );

console.log( colors.entries() );
> SetIterator {[{"red": "red"}, {"yellow": "yellow"}, {"green": "green"}]}

console.log( colors.keys() );
> SetIterator {"red", "yellow", "green"}

console.log( colors.values() );
> SetIterator {"red", "yellow", "green" }

console.log( horses.entries() );
> MapIterator {[5, "QuickBucks"], [8, "Chocolate"], [3, "Filippone"]}

console.log( horses.keys() );
> MapIterator {5, 8, 3}

console.log( horses.values() );
> MapIterator {"QuickBucks", "Chocolate", "Filippone"}

```

You don't need these iterators though to perform the iteration. Sets and maps are iterable themselves, therefore, they can be used in `for-of` loops.

A common destructuring pattern is to iterate the keys and values of a map using destructuring in a `for-of` loop:

```

for ( let [key, value] of horses ) {
  console.log( key, value );
}

> 5 "QuickBucks"
> 8 "Chocolate"
> 3 "Filippone"

```

When creating a set or a map, you can pass any iterable as an argument, provided that the results of the iteration can form a set or a map:

```

let s = new Set( countdownIterable );
> Set {9, 8, 7, 6, 5, 4, 3, 2, 1}

```

The role of the iterable interface

We can understand iterables a bit better by concentrating on data flow:

- The `for-of` loop, the `...` operator, and some other language constructs are *data consumers*. They consume iterable data
- Iterable data structures such as arrays, strings, dom data structures, maps, and sets are *data sources*
- The *iterable interface* specifies how to connect data consumers with data sources
- *Iterable objects* are created according to the iterable interface specification. Iterable objects can create iterator objects that facilitate the iteration on their data source, and prepare the result for a data consumer

We can create independent iterator objects on the same iterable. Each iterator acts like a pointer to the upcoming element the linked data source can consume.

In the lesson on sets and maps, we have learned that it is possible to convert sets to arrays using the spread operator:

```
let arr = [...set];
```

You now know that a set is an iterable object, and the spread operator is a data consumer. The formation of the array is based on the iterable interface. ES6 makes a lot of sense once you start connecting the dots.

Generators

There is a relationship between *iterators* and *generators*: a generator is a special function that returns an iterator.

There are some differences between generator functions and regular functions:

- There is an `*` after the `function` keyword,
- Generator functions create iterators
- We use the `yield` keyword in the created iterator function. By writing `yield v`, the iterator returns `{ value: v, done: false }` as a value
- We can also use the `return` keyword to end the iteration. Similarly to iterators, the returned value won't be enumerated by a data consumer
- The yielded result is the next value of the iteration process. Execution of the generator function is stopped at the point of yielding. Once a data consumer asks for another value, execution of the generator function is resumed, by executing the statement after the last `yield`

Example:

```
function *getLampIterator() {
    yield 'red';
    yield 'green';
    return 'lastValue';
    // implicit: return undefined;
}

let lampIterator = getLampIterator();

console.log( lampIterator.next() );
> Object {value: "red", done: false}

console.log( lampIterator.next() );
> Object {value: "green", done: false}

console.log( lampIterator.next() );
> Object {value: "lastValue", done: true}
```

When we reach the end of a function, it automatically returns `undefined`. In the above example, we never reached the end, as we returned `'lastValue'` instead.

If the return value was missing, the function would return `{value: undefined, done: true}`.

Use generators to define custom iterables to avoid using the well known symbol `Symbol.iterator`.

Generators return iterators that are also iterables

Recall our string iterator example to refresh what iterable objects and iterators are:

```
let message = 'ok';

let stringIterator = message[Symbol.iterator]();
```

We call the `next` method of `stringIterator` to get the next element:

```
console.log( stringIterator.next() );
> Object {value: "o", done: false}
```

However, in a `for-of` loop, we normally use the *iterable object*, not the iterator:

```
for ( let ch of message ) {
  console.log( ch );
}

> o
> k
```

Iterable objects have a `[Symbol.iterator]` method that returns an iterator.

Iterator objects have a `next` method that returns an object with keys `value` and `done`.

Generator functions return an object that is both an iterable and an iterator. Generator functions have:

- a `[Symbol.iterator]` method to return their iterator,
- a `next` method to perform the iteration

As a consequence, the return value of generator functions can be used in `for-of` loops, after the spread operator, and in all places where iterables are consumed.

```
function *getLampIterator() {
  yield 'red';
  yield 'green';
  return 'lastValue';
  // implicit: return undefined;
}

let lampIterator = getLampIterator();
```

```
console.log( lampIterator.next() );
> Object {value: "red", done: false}

console.log( [...lampIterator] );
> ["green"]
```

In the above example, `[...lampIterator]` contains the remaining values of the iteration in an array.

Iterators and destructuring

When equating an array to an iterable, iteration takes place.

```
let lampIterator = getLampIterator();

let [head,] = lampIterator;

console.log( head, [...lampIterator] );
> red []
```

The destructuring assignment is executed as follows:

- first, `lampIterator` is substituted by an array of form `[...lampIterator]`
- then the array is destructured, and `head` is assigned to the first element of the array
- the rest of the values are thrown away
- as `lampIterator` was used to build an array with all elements on the right hand side, `[...lampIterator]` is empty in the console log

Combining generators

It is possible to combine two sequences in one iterable. All you need to do is use `yield *` to include an iterable, which will enumerate all of its values one by one.

```
let countdownGenerator = function *() {
  let i = 10;
  while ( i > 0 ) yield --i;
}

let lampGenerator = function *() {
  yield 'red';
```

```

        yield 'green';
    }

let countdownThenLampGenerator = function *() {
    yield *countdownGenerator();
    yield *lampGenerator();
}

console.log( [...countdownThenLampGenerator()] );
> [9, 8, 7, 6, 5, 4, 3, 2, 1, 0, "red", "green"]

```

Passing parameters to iterables

The `next` method of iterators can be used to pass a value that becomes the value of the previous `yield` statement.

```

let greetings = function *() {
    let name = yield 'Hi!';
    yield `Hello, ${ name }!`;
}

let greetingIterator = greetings();

console.log( greetingIterator.next() );
> Object {value: "Hi!", done: false}

console.log( greetingIterator.next( 'Lewis' ) );
> Object {value: "Hello, Lewis!", done: false}

```

The return value of a generator becomes the value of a `yield *` expression:

```

let sumSequence = function *( num ) {
    let sum = 0;
    for ( let i = 1; i <= num; ++i ) {
        sum += i;
        yield i;
    }
    return sum;
}

```

```
let wrapSumSequence = function *( num ) {
    let sum = yield *sumSequence( num );
    yield `The sum is: ${ sum }.`;
}

for ( let elem of wrapSumSequence( 3 ) ) {
    console.log( elem );
}
> 1
> 2
> 3
> The sum is: 6.
```

Practical applications

You now know everything to be able to write generator functions. This is one of the hardest topics in ES6, so you will get a chance to solve more exercises than usual.

After practicing the foundations, you will find out how to use generators in practice to:

- define infinite sequences (exercise 5),
- create code that evaluates lazily (exercise 6).

For the sake of completeness, it is worth mentioning that generators can be used for asynchronous programming. Running asynchronous code is outside the scope of this lesson. We will use promises for handling asynchronous code.

Exercises

These exercises help you explore in more depth how iterators and generators work. You will get a chance to play around with iterators and generators, which will result in a higher depth of learning experience for you than reading about the edge cases.

You can also find out if you already know enough to command these edge cases without learning more about iterators and generators.

I will post an article with the solutions of the exercises. I will hide the solutions for a couple of days so that you can try solving these exercises yourself.

If you liked this lesson, check out the course by clicking the book below, or [visiting this link](#).

Exercise 1. What happens if we use a string iterator in a `for-of` loop?

```

let message = 'ok';
let messageIterator = message[Symbol.iterator]();

messageIterator.next();

for ( let item of messageIterator ) {
  console.log( item );
}

```

Exercise 2. Create a countdown iterator that counts from 9 to 1. Use generator functions!

```

let getCountdownIterator = // Your code comes here

console.log( [ ...getCountdownIterator() ] );
> [9, 8, 7, 6, 5, 4, 3, 2, 1]

```

Exercise 3. Make the following object iterable:

```

let todoList = {
  todoItems: [],
  addItem( description ) {
    this.todoItems.push( { description, done: false } );
    return this;
  },
  crossOutItem( index ) {
    if ( index < this.todoItems.length ) {
      this.todoItems[index].done = true;
    }
    return this;
  }
};

todoList.addItem( 'task 1' ).addItem( 'task 2' ).crossOutItem( 0 );

let iterableTodoList = // ???;

```

```

for ( let item of iterableTodoList ) {
    console.log( item );
}

// Without your code, you get the following error:
// Uncaught TypeError: todoList[Symbol.iterator] is not a function

```

Exercise 4. Determine the values logged to the console without running the code. Instead of just writing down the values, formulate your thought process and explain to yourself how the code runs line by line.

```

let errorDemo = function *() {
    yield 1;
    throw 'Error yielding the next result';
    yield 2;
}

let it = errorDemo();

// Execute one statement at a time to avoid
// skipping lines after the first thrown error.

console.log( it.next() );

console.log( it.next() );

console.log( [...errorDemo()] );

for ( let element of errorDemo() ) {
    console.log( element );
}

```

Exercise 5. Create an infinite sequence that generates the next value of the Fibonacci sequence.

The Fibonacci sequence is defined as follows:

- `fib(0) = 0`
- `fib(1) = 1`
- for `n > 1, fib(n) = fib(n - 1) + fib(n - 2)`

Exercise 6. Create a lazy `filter` generator function. Filter the elements of the Fibonacci sequence by keeping the even values only.

```
function *filter( iterable, filterFunction ) {  
    // insert code here  
}  
-----
```

55 : ES6 Iterators and Generators -- 6 exercises and solutions

<http://www.zsoltnagy.eu/es6-iterators-and-generators-6-exercises-and-solutions/>

This article is a section from the workbook of ES6 in Practice. I created this course during the last couple of months, because there is an evident need for a resource that helps JavaScript developers put theory into practice.

This course won't waste your time with long hours of video, or long pages of theory that you will never use in practice. We will instead focus on learning just enough theory to solve some exercises. Once you are done with the exercises, you can check the reference solutions, and conclude some lessons. In fact, some of the theory are sometimes placed in the reference solutions.

If you like this article, check out the course [here](#).

In my [last article](#), I gave you six exercises. In this article, you can check the reference solutions.

Exercise 1. What happens if we use a string iterator instead of an iterable object in a `for-of` loop?

```
let message = 'ok';
let messageIterator = message[Symbol.iterator]();

messageIterator.next();

for ( let item of messageIterator ) {
  console.log( item );
}
```

Solution: Similarly to generators, in case of strings, arrays, DOM elements, sets, and maps, an iterator object is also an iterable.

Therefore, in the `for-of` loop, the remaining `k` letter is printed out.

Exercise 2. Create a countdown iterator that counts from 9 to 1. Use generator functions!

```
let getCountdownIterator = // Your code comes here

console.log( [ ...getCountdownIterator() ] );
> [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Solution:

```

let getCountdownIterator = function *() {
    let i = 10;
    while( i > 1 ) {
        yield --i;
    }
}

console.log( [ ...getCountdownIterator() ] );
> [9, 8, 7, 6, 5, 4, 3, 2, 1]

```

The generator function yields the numbers 9 to 1. The spread (...) operator consumes all values. Then the generator function returns undefined, ending the iteration process.

Exercise 3. Make the following object iterable:

```

let todoList = {
    todoItems: [],
    addItem( description ) {
        this.todoItems.push( { description, done: false } );
        return this;
    },
    crossOutItem( index ) {
        if ( index < this.todoItems.length ) {
            this.todoItems[index].done = true;
        }
        return this;
    }
};

todoList.addItem( 'task 1' ).addItem( 'task 2' ).crossOutItem( 0 );

let iterableTodoList = // ???;

for ( let item of iterableTodoList ) {
    console.log( item );
}

// Without your code, you get the following error:

```

```
// Uncaught TypeError: todoList[Symbol.iterator] is not a function
```

First Solution (well known symbol):

We could use well known symbols to make `todoList` iterable. We can add a `*[Symbol.iterator]` generator function that yields the elements of the array. This will make the `todoList` object iterable, yielding the elements of `todoItems` one by one.

```
let todoList = {
  todoItems: [],
  *[Symbol.iterator]() {
    yield* this.todoItems;
  }
  addItem( description ) {
    this.todoItems.push( { description, done: false } );
    return this;
  },
  crossOutItem( index ) {
    if ( index < this.todoItems.length ) {
      this.todoItems[index].done = true;
    }
    return this;
  }
};

let iterableTodoList = todoList;
```

Second solution (generator function): If you prefer staying away from well known symbols, it is possible to make your code more semantic:

```
let todoList = {
  todoItems: [],
  addItem( description ) {
    this.todoItems.push( { description, done: false } );
    return this;
  },
  crossOutItem( index ) {
    if ( index < this.todoItems.length ) {
```

```
        this.todoItems[index].done = true;
    }
    return this;
}

};

todoList.addItem( 'task 1' ).addItem( 'task 2' ).crossOutItem( 0 );

let todoListGenerator = function *() {
    yield* todoList.todoItems;
}

let iterableTodoList = todoListGenerator();
```

The first solution reads a bit like a hack. The second solution looks cleaner even if you have to type more characters.

Exercise 4. Determine the values logged to the console without running the code. Instead of just writing down the values, formulate your thought process and explain to yourself how the code runs line by line.

```
let errorDemo = function *() {
    yield 1;
    throw 'Error yielding the next result';
    yield 2;
}

let it = errorDemo();

// Execute one statement at a time to avoid
// skipping lines after the first thrown error.

console.log( it.next() );

console.log( it.next() );

console.log( [...errorDemo()] );

for ( let element of errorDemo() ) {
    console.log( element );
}
```

Solution:

```

console.log( it.next() );
> Object {value: 1, done: false}

console.log( it.next() );
> Uncaught Error yielding the next result

console.log( [...errorDemo()] );
> Uncaught Error yielding the next result

for ( let element of errorDemo() ) {
  console.log( element );
}
> Object {value: 1, done: false}
> Uncaught Error yielding the next result

```

We created three iterables in total: `it`, one in the statement in the spread operator, and one in the `for-of` loop.

In the example with the `next` calls, the second call results in a thrown error.

In the spread operator example, the expression cannot be evaluated, because an error is thrown.

In the `for-of` example, the first element is printed out, then the error stopped the execution of the loop.

Exercise 5. Create an infinite sequence that generates the next value of the Fibonacci sequence.

The Fibonacci sequence is defined as follows:

- `fib(0) = 0`
- `fib(1) = 1`
- `for n > 1, fib(n) = fib(n - 1) + fib(n - 2)`

Solution:

```

function *fibonacci() {
  let a = 0, b = 1;
  yield a;
  yield b;
  while( true ) {
    [a, b] = [b, a+b];
  }
}

```

```

        yield b;
    }
}

```

Note that you only want to get the `next()` element of an infinite sequence. Executing `[...fibonacci()]` will skyrocket your CPU usage, speed up your CPU fan, and then crash your browser.

Exercise 6. Create a lazy `filter` generator function. Filter the elements of the Fibonacci sequence by keeping the even values only.

```

function *filter( iterable, filterFunction ) {
    // insert code here
}

```

Solution:

```

function *filter( iterable, filterFunction ) {
    for( let element of iterable ) {
        if ( filterFunction( element ) ) yield element;
    }
}

let evenFibonacci = filter( fibonacci(), x => x%2 === 0 );

```

Notice how easy it is to combine generators and lazily evaluate them.

```

evenFibonacci.next()
> {value: 0, done: false}
evenFibonacci.next()
> {value: 2, done: false}
evenFibonacci.next()
> {value: 8, done: false}
evenFibonacci.next()
> {value: 34, done: false}
evenFibonacci.next()

```

```
> {value: 144, done: false}
```

Lazy evaluation is essential when we work on a large set of data. For instance, if you have 1000 accounts, chances are that you don't want to transform all of them if you just want to render the first ten on screen. This is when lazy evaluation comes into play.

If you would like to get five more chapters of this book, click YES below, and sign up.

56 : What you should know about JavaScript regular expressions

<http://bjorn.tipling.com/state-and-regular-expressions-in-javascript>

Regular expressions in JavaScript may not always be intuitive. I aim to provide some clarity by providing examples about things I have found myself getting stuck on. This post covers a few topics including state in regular expressions, regular expression performance and various things that have tripped me up.

regular expressions are stateful

Regular expression objects maintain state. For example, the `exec` method is not **idempotent**, successive calls may return different results. Calls to `exec` have this behavior because the regular expression object remembers the last position it searched from when the global flag is set to true.

As an example, examine the following code:

```
var res, text = "foo1 bar1 foo2 bar \n foo3 bar2",
    regexp = /foo\d (bar\d?)/g;

console.log("regexp.lastIndex:", regexp.lastIndex);
while (res = regexp.exec(text)) {
    console.log("regexp.lastIndex:", regexp.lastIndex,
                "index:", res.index,
                "res[0]:", res[0],
                "res[1]:", res[1]);
}
console.log("regexp.lastIndex:", regexp.lastIndex);
```

[jsfiddle](#)

The log prints:

```
regexp.lastIndex: 0
regexp.lastIndex: 9 index: 0 res[0]: foo1 bar1 res[1]: bar1
regexp.lastIndex: 18 index: 10 res[0]: foo2 bar res[1]: bar
regexp.lastIndex: 30 index: 21 res[0]: foo3 bar2 res[1]: bar2
regexp.lastIndex: 0
```

The most important thing in this bit of code is that we are calling `regexp.exec` in a `while` loop multiple times, and on each call it returns a different result.

The variable `text` contains a string that we want to search. The variable `regexp` contains a regular expression literal `/foo\d (bar\d?)/g` that searches for strings that contain the characters `foo` followed by a numeric digit

\d , a space and the characters bar which may or may not be followed by a digit \d? . Also note the parenthesis around the (bar\d?) portion, this creates a substring match. Finally the regular expression is terminated by a g , which is a global flag that enables us to continue searching for more results after we find the first.

The state in a regular expression is captured by the lastIndex property, which is the index of the character where the search starts on the next call to exec . The lastIndex starts at 0 and is reset to 0 when exec finishes searching the text, at which point exec will return null . As the log indicates, the returned result array res contains important information about what was found or null if nothing was found. The returned array is special because in addition to results, it contains the properties index and input which provide the position of the match and the search pattern used.

If a while loop is used to search a regular expression that does not contain a global flag the while loop will never finish if a match is found. It will only return that first match, forever. If you are not sure if the regular expression you are searching is global you can check the regular expression's global property:

```
while (res = regexp.exec(text)) {
  // do something with res
  if (!regexp.global) {
    break;
  }
}
```

[jsfiddle](#)

Using the test method on a global regular expression also makes use of state. It will advance lastIndex the same as exec . test returns true or false depending on whether a match was found.

Using test is faster than exec , thus a use case might be to check a string with test first and only if something is found use exec . This could lead to a mistake for a global regular expression as test will have advanced lastIndex .

```
var res, text = "foo1 bar1 foo2 bar \n foo3 bar2",
  regexp = /foo\d (bar\d?)/g;

while (regexp.test(text)) {
  res = regexp.exec(text);
  console.log("regexp.lastIndex:", regexp.lastIndex,
              "index:", res.index,
              "res[0]:", res[0],
              "res[1]:", res[1]);
}
```

[jsfiddle](#)

The result is missed results and finally a JavaScript error because we attempted to access properties on a `null` object:

```
regexp.lastIndex: 18 index: 10 res[0]: foo2 bar res[1]: bar
Uncaught TypeError: Cannot read property 'index' of null
```

You could reset `lastIndex` to avoid this issue:

```
var res, text = "foo1 bar1 foo2 bar \n foo3 bar2",
    regexp = /foo\d (bar\d?)/g,
    lastIndex = 0;

while (regexp.test(text)) {
    regexp.lastIndex = lastIndex;
    res = regexp.exec(text);
    console.log("regexp.lastIndex:", regexp.lastIndex,
                "index:", res.index,
                "res[0]:", res[0],
                "res[1]:", res[1]);
    lastIndex = regexp.lastIndex;
}
```

[jsfiddle](#)

The extra cost of calling `test` is not worth it, as explained in the performance section at the end of this post. I only meant to demonstrate that `test` and `exec` both advance `lastIndex`.

In the following bit of code you might expect only one result for each search since I have set the `lastIndex` to begin at 3, but on the second test it finds 2 because `lastIndex` was reset to `0` at the end of the while loop when `res` returned `null`. `lastIndex` is not reset to `0` when you test a new string.

```
var texts = [
    "foo foo",
    "foo foo"
];
var regex = /foo/g;
regex.lastIndex = 3;

texts.forEach(function (text) {
    var count, res;
    count = 0;
    while (res = regex.test(text)) {
        count += 1;
    }
});
```

```

    }
    console.log("number of results found:", count);
});

```

[jsfiddle](#)

Result:

```

"number of results found:" 1
"number of results found:" 2

```

To fix this put the `regex.lastIndex = 3;` right before each use of `test`:

```

var texts = [
  "foo foo",
  "foo foo"
];
var regex = /foo/g;

texts.forEach(function (text) {
  var count, res;
  regex.lastIndex = 3;
  count = 0;
  while (res = regex.test(text)) {
    count += 1;
  }
  console.log("number of results found:", count);
});

```

[jsfiddle](#)

Result:

```

"number of results found:" 1
"number of results found:" 1

```

ECMAScript 6 (harmony) adds a new stateful flag for regular expressions, the [sticky](#) flag. This flag is currently, as of August 2014, only implemented in Firefox. The sticky flag is provided by setting the `y` flag in a regular expression like so: `/foo/y`. The sticky flag advances `lastIndex` like `g` but only if a match is found starting at `lastIndex`, there is no forward search. The sticky flag was added to [improve the performance](#) of writing lexical analyzers using JavaScript, but as the MDN documentation indicates, it could be used to require a regular expression match starting at position `n` where `n` is what `lastIndex` is set to. In the case of a non-multiline regular

expression, a `lastIndex` value of 0 with the sticky flag would be in effect the same as starting the regular expression with `^` which requires the match to start at the beginning of the text searched.

The following code demonstrates an example of using sticky, note it may only work in Firefox:

```
var searchStrings, stickyRegexp;

stickyRegexp = /foo/y;

searchStrings = [
  "foo",
  " foo",
  "  foo",
];
searchStrings.forEach(function(text, index) {
  stickyRegexp.lastIndex = 1;
  console.log("found a match at", index, ":", stickyRegexp.test(text));
});
```

[jsfiddle](#) (firefox only)

Result:

```
"found a match at" 0 ":" false
"found a match at" 1 ":" true
"found a match at" 2 ":" false
```

multiline and global search

The confusion between multiline and global might be just a thing that afflicted me, so if you already understand the difference feel free to skip this section, it's pretty straight forward. Multiline searches affect the way that the special regular expression characters `^` and `$` behave when the search string includes new lines. Multiline searches are enabled with the `m` flag like so: `/foo/m`. I can't explain it better than [the MDN documentation](#):

The `m` flag is used to specify that a multiline input string should be treated as multiple lines. If the `m` flag is used, `^` and `$` match at the start or end of any line within the input string instead of the start or end of the entire string.

Global searches on the other hand allow you to continue searching the entire search string after you have found the first result, to find additional results. It has nothing to do with `^` and `$` and it does not treat newlines as special.

Consider the following example:

```

var text, searches;

text = "foo \nfoo \nfoo ";
searches = [
  /^foo $/g , // finds 0 results
  /^foo $/m, // finds only 1 result and stops
  /^foo $/mg, // the only regexp that finds 3
  /foo \n/g, // finds 2, but not the last
  /^foo \n/g, // finds only 1, the first
];
};

searches.forEach(function (search) {
  printSearches(search);
});

function printSearches(regex) {
  var res;
  console.log("searching with", regex);
  while (res = regex.exec(text)) {
    console.log(regex, "->", res);
    if (!regex.global) {
      break;
    }
  }
}
}

```

[jsfiddle](#)

The function `printSearches` uses a `while` loop to search `text` with `exec` and logs the result. The search string `text` has multiple new lines.

The first search `/^foo $/g` would only match the string `"foo "` and so it finds no results.

```
searching with /^foo $/g
```

`/^foo $/m` prints only one result as it is not a global search and thus the while loop breaks on `!regex.global`. Otherwise it would loop forever, printing the first result again and again.

```
searching with /^foo $/m
/^foo $/m "->" ["foo ", index: 0, input: "foo ↵foo ↵foo "]
```

`/^foo $/gm` finds each instance of `"foo "` in the search string as it is a global search. It continues to search the search string until it can find no more results.

```
searching with /^foo $/gm
/^foo $/gm "-> [ "foo ", index: 0, input: "foo <foo <foo "]
/^foo $/gm "-> [ "foo ", index: 5, input: "foo <foo <foo "]
/^foo $/gm "-> [ "foo ", index: 10, input: "foo <foo <foo "]
```

`/foo \n/g` is not a substitute for a multiline search. It assumes a particular form of new line missing out on `\r\n`, doesn't find last "foo" at the end of the string and could lead to other problems because of the lack of a `^`.

```
searching with /foo \n/g
/foo \n/g "-> [ "foo <", index: 0, input: "foo <foo <foo "]
/foo \n/g "-> [ "foo <", index: 5, input: "foo <foo <foo "]
```

`/^foo \n/` only finds the first foo as `^` will only match the start of the string, not a line, in a non-multiline search.

```
searching with /^foo \n/g
/^foo \n/g "-> [ "foo <", index: 0, input: "foo <foo <foo "]
```

greedy vs non-greedy

Greediness and regular expressions is not really a JavaScript specific topic but I thought I would mention it here anyway because knowing about greedy versus non-greedy is really useful. The greediness of a regular expression affects how "zero or more" `*` or "one or more" `+` qualifiers behave. The default behavior for these is greedy. To toggle non-greedy apply a question mark at the end as such: `*?` or `+?`. To understand how these behave look at this code which reuses the `printSearches` function defined above:

```
var text = "foo bar foo bar foo bar",
    greedyRegexp = /foo.*bar/g,
    nonGreedyRegexp = /foo.*?bar/g;

printSearches(greedyRegexp);
printSearches(nonGreedyRegexp);
```

[jsfiddle](#)

The `text` we are searching has a repeated pattern for "foo bar". `greedyRegexp` will find only a single result, the entire string as `.*` will capture until the last "bar" is encountered.

```
searching with /foo.*bar/g
/foo.*bar/g "-> [ "foo bar foo bar foo bar", index: 0, input: "foo bar foo bar foo bar"]
```

`nonGreedyRegexp` instead finds each instance of "foo bar", non-greedy will end after it has found the first "bar" that satisfy the regular expression.

```
searching with /foo.*?bar/g
/foo.*?bar/g "-> ["foo bar", index: 0, input: "foo bar foo bar foo bar"]
/foo.*?bar/g "-> ["foo bar", index: 8, input: "foo bar foo bar foo bar"]
/foo.*?bar/g "-> ["foo bar", index: 16, input: "foo bar foo bar foo bar"]
```

I think an image might help to demonstrate the difference between greedy and non-greedy:

/foo.*bar/	foo bar foo bar foo bar
/foo.*?bar/	foo bar foo bar foo bar

`/foo.*bar/` is greedy and matches “**foo bar foo bar foo bar**” while `/foo.*?bar/` is not greedy and matches “**foo bar** **foo bar** **foo bar**”.

This StackOverflow question has some pretty good answers with more information: [how to make Regular expression into non-greedy?](#)

constructors versus literals

Regular expressions can be created with the literal syntax such as:

```
var regexp = /foo.*bar/g`
```

Or you can create one with the constructor [RegExp](#):

```
var regexp = new RegExp("foo.*bar", "gi");
```

Literal regular expressions are evaluated only once at evaluation time while the constructor is evaluated at runtime, thus unless you need to dynamically create a regular expression use the literal syntax. A dynamic regular expression can be useful, here’s an example where I make use of a literal and a dynamic to pull out brands and heroes from a string with comic book heroes:

```
var text = "DC: Batgirl, Marvel: Rogue, Image: Spawn, Image: Celestine, Marvel: Iron
Man, DC: Superman, Image: Ant",
brandRegexp = /(^\|, )(\S+):/g,
regexp, brands, res, heroes, brand;

brands = {};
while (res = brandRegexp.exec(text)) {
    brand = res[2];
    brands[brand] ? brands[brand]++;
    : brands[brand] = 1;
```

```

    }

    for (brand in brands) {
        regexp = new RegExp(brand + ": ([\\S ]+?)(,|$)", "g");
        heroes = [];
        while (res = regexp.exec(text)) {
            heroes.push(res[1]);
        }
        console.log("Found", brands[brand], "heroes for", brand, "comics:", heroes.join(","));
    }
}

```

[jsfiddle](#)

Result:

```

Found 2 heroes for DC comics: Batgirl, Superman
Found 2 heroes for Marvel comics: Rogue, Iron Man
Found 3 heroes for Image comics: Spawn, Celestine, Ant

```

You could just use [a single literal regular expression](#) here and that would make the code cleaner and be more efficient but I wanted to demonstrate both ways of creating regular expressions.

split, search, match and replace

Various methods are available on strings that can take regular expressions as arguments. For example the `split` method can take a string or a regular expression as an argument. This might be useful if you wanted to split a string on any kind of whitespace:

```

var text = "apples oranges      bananas\n\n\nstrawberries \tplums";
console.log(text.split(/\s+/));

```

[jsfiddle](#)

This finds all the fruit and trims the whitespace:

```

["apples", "oranges", "bananas", "strawberries", "plums"]

```

The `search` string method finds the index for the first match or `-1` if a match wasn't found. It's similar to `indexof` except you can use a regular expression for a more powerful search.

```

var text = "foo bar";

```

```
console.log(text.indexOf("bar")); //logs 4
console.log(text.search(/bar/)); //also logs 4
```

[jsfiddle](#)

The `match` string method returns an array with results. It can return all of the matches for a regular expression if you use the `g` global flag. This might be an alternative to iterative calls with `exec`.

```
var text = "foo bar foo bar foo bar";

console.log(text.match(/foo.*bar/)); //greedy
console.log(text.match(/foo.*?bar/)); //non-greedy, only 1 result
console.log(text.match(/foo.*?bar/g)); //non-greedy, all results
```

[jsfiddle](#)

The results are:

```
["foo bar foo bar foo bar", index: 0, input: "foo bar foo bar foo bar"]
["foo bar", index: 0, input: "foo bar foo bar foo bar"]
["foo bar", "foo bar", "foo bar"]
```

Unlike the regular expression methods `exec` and `test`, the string methods `match` and `search` are not stateful and do not advance `lastIndex` on the regular expression.

```
var text = "foo bar foo bar foo bar",
globalRegexp = /foo.*?bar/g;

console.log(text.match(globalRegexp)); //non-greedy, all results
console.log(globalRegexp.lastIndex); //logs 0, lastIndex not advanced
```

[jsfiddle](#)

The `replace` method on strings takes a string or a regular expression as an argument. It returns a new string that has its text replaced, it does not modify the existing string. Use a global flag if you want all instances of the string replaced:

```
var text = "foo bar foo bar foo bar";

console.log(text.replace("bar", "BAR")); //only replaces first
console.log(text.replace(/bar/, "BAR")); //only replaces first
console.log(text.replace(/bar/g, "BAR")); //replaces all
```

[jsfiddle](#)

Results:

```
foo BAR foo bar foo bar
foo BAR foo bar foo bar
foo BAR foo BAR foo BAR
```

Firefox's implementation of `replace` allows specifying flags as an argument, but this will not work in other browsers. Set the flags on the regular expression instead.

[regular expressions and application performance](#)

Creating a regular expression that is used multiple times once in the beginning is faster than creating the regular expression later, regardless whether a constructor or a regular expression literal was used. Using a literal regular expression is always faster than creating a regular expression with a constructor.

[jsperf](#)

The [MDN documentation](#) might lead one to assume that it is OK to define regular expressions in a loop since they're created at evaluation time and won't be recompiled:

The literal notation provides compilation of the regular expression when the expression is evaluated.

Use literal notation when the regular expression will remain constant. For example, if you use literal notation to construct a regular expression used in a loop, the regular expression won't be recompiled on each iteration.

But defining the regular expression before the loop is much faster in Firefox and a little bit faster in Chrome.

[jsperf](#)

Using `test` to check a string before `exec` is probably not worth it.

[jsperf](#)

`test` is faster than `exec` however.

[jsperf](#)

Using `search` to check a string before `match` is also not worth it.

[jsperf](#)

`search` is faster than `match`

[jsperf](#)

`match` and `exec` perform differently in different engines. In Chrome `exec` seems to be faster than `match` and in Firefox the reverse is true. If you wanted to get the number of results `split` is your best bet in Firefox and `exec` wins in Chrome.

[jsperf](#)

If you want to replace text in a string `replace` seems to be much faster than using `split` and `join` which make sense since `split` creates a new array. This is true in both Firefox and Chrome.

[jsperf](#)

That rounds out an exploration of regular expressions in JavaScript. Some of these things never seemed intuitive to me. Ultimately in most scenarios it is probably best to avoid using regular expressions and solve the problem some other way, unless you want to replace text in a string that is.

57 : all this

<http://bjorn.tipling.com/all-this>

Coming from a sane language you might think that `this` in JavaScript is kind of like `this` in an object oriented language like Java, something that refers to values stored in instance properties. Not so. In JavaScript it is best to think of the `this` as a [bogart](#) carrying a bag of data with an [undetectable extension charm](#).

What follows is what I would want my co-workers to know about using `this` in JavaScript. It's a lot and much of it took me years to learn.

global this

In a browser, at the global scope, `this` is the `window` object.

```
<script type="text/javascript">
  console.log(this === window); //true
</script>
```

[jsfiddle](#)

In a browser, using `var` at the global scope is the same as assigning to `this` or `window`.

```
<script type="text/javascript">
  var foo = "bar";
  console.log(this.foo); //logs "bar"
  console.log(window.foo); //logs "bar"
</script>
```

[jsfiddle](#)

If you create a new variable without using `var` or `let` (ECMAScript 6) you are adding or changing a property on the global `this`.

```
<script type="text/javascript">
  foo = "bar";

  function testThis() {
    foo = "foo";
  }

  console.log(this.foo); //logs "bar"
  testThis();
```

```
    console.log(this.foo); //logs "foo"
</script>
```

[jsfiddle](#)

In node using the repl, `this` is the top namespace. You can refer to it as `global`.

```
> this
{ ArrayBuffer: [Function: ArrayBuffer],
  Int8Array: { [Function: Int8Array] BYTES_PER_ELEMENT: 1 },
  Uint8Array: { [Function: Uint8Array] BYTES_PER_ELEMENT: 1 },
  ...
> global === this
true
```

In node executing from a script, `this` at the global scope starts as an empty object. It is not the same as `global`.

test.js:

```
console.log(this);
console.log(this === global);
```

```
$ node test.js
{}
false
```

In node, `var` at the global scope does not assign to `this` like it does in a browser when you are executing it as a script from a file...

test.js:

```
var foo = "bar";
console.log(this.foo);
```

```
$ node test.js
undefined
```

...but it does if you're doing that from the node repl.

```
> var foo = "bar";
> this.foo
bar
```

```
> global.foo  
bar
```

In node, using a script, creating a variable without `var` or `let` will add that variable to `global` but not to the `this` at the top of the script's scope.

test.js:

```
foo = "bar";  
console.log(this.foo);  
console.log(global.foo);
```

```
$ node test.js  
undefined  
bar
```

In a node repl, it will assign it to both.

function this

Except in the case of DOM event handlers or when a `thisArg` is provided (see further down), both in node and in a browser using `this` in a function that is not called with `new` references the global scope...

```
<script type="text/javascript">  
  foo = "bar";  
  
  function testThis() {  
    this.foo = "foo";  
  }  
  
  console.log(this.foo); //logs "bar"  
  testThis();  
  console.log(this.foo); //logs "foo"  
</script>
```

[jsfiddle](#)

test.js:

```
foo = "bar";  
  
function testThis () {  
  this.foo = "foo";
```

```
}
```

```
console.log(global.foo);
testThis();
console.log(global.foo);
```

```
$ node test.js
bar
foo
```

...unless you use "use strict"; in which case this will be undefined.

```
<script type="text/javascript">
  foo = "bar";

  function testThis() {
    "use strict";
    this.foo = "foo";
  }

  console.log(this.foo); //logs "bar"
  testThis(); //Uncaught TypeError: Cannot set property 'foo' of undefined
</script>
```

[jsfiddle](#)

If you call a function with new the this will be a new context, it will not reference the global this.

```
<script type="text/javascript">
  foo = "bar";

  function testThis() {
    this.foo = "foo";
  }

  console.log(this.foo); //logs "bar"
  new testThis();
  console.log(this.foo); //logs "bar"

  console.log(new testThis().foo); //logs "foo"
</script>
```

[jsfiddle](#)

I like to call this new context an instance.

prototype this

Functions you create become function objects. They automatically get a special `prototype` property, which is something you can assign values to. When you create an instance by calling your function with `new` you get access to the values you assigned to the `prototype` property. You access those values using `this`.

```
function Thing() {  
    console.log(this.foo);  
}  
  
Thing.prototype.foo = "bar";  
  
var thing = new Thing(); //logs "bar"  
console.log(thing.foo); //logs "bar"
```

[jsfiddle](#)

If you create multiple new instances with `new` each of these will all share the same value for values defined on the `prototype`. For example they will all return the same value when using `this.foo` unless you override `this.foo` inside the individual instances.

```
function Thing() {  
}  
Thing.prototype.foo = "bar";  
Thing.prototype.logFoo = function () {  
    console.log(this.foo);  
}  
Thing.prototype.setFoo = function (newFoo) {  
    this.foo = newFoo;  
}  
  
var thing1 = new Thing();  
var thing2 = new Thing();  
  
thing1.logFoo(); //logs "bar"  
thing2.logFoo(); //logs "bar"  
  
thing1.setFoo("foo");  
thing1.logFoo(); //logs "foo";  
thing2.logFoo(); //logs "bar";
```

```
thing2.foo = "foobar";
thing1.logFoo(); //logs "foo";
thing2.logFoo(); //logs "foobar";
```

[jsfiddle](#)

The `this` in an instance is a special kind of object, `this` is in fact a key word. You can think of `this` as a way to access values defined on the `prototype`, but assigning directly to `this` inside an instance will hide the value on the `prototype` from the instance. You can get access back to the `prototype` value by deleting what you put on `this` inside the instance...

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    console.log(this.foo);
}
Thing.prototype.setFoo = function (newFoo) {
    this.foo = newFoo;
}
Thing.prototype.deleteFoo = function () {
    delete this.foo;
}

var thing = new Thing();
thing.setFoo("foo");
thing.logFoo(); //logs "foo";
thing.deleteFoo();
thing.logFoo(); //logs "bar";
thing.foo = "foobar";
thing.logFoo(); //logs "foobar";
delete thing.foo;
thing.logFoo(); //logs "bar";
```

[jsfiddle](#)

...or just reference the function object's prototype directly.

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    console.log(this.foo, Thing.prototype.foo);
```

```
 }
```

```
var thing = new Thing();
thing.foo = "foo";
thing.logFoo(); //logs "foo bar";
```

[jsfiddle](#)

Instances created with the same function all share access to the same exact values on the `prototype`. If you assign an array to the `prototype` all instances are sharing access to that same array. Unless you override it inside the instance in which case you have hidden it.

```
function Thing() {
}
Thing.prototype.things = [];

var thing1 = new Thing();
var thing2 = new Thing();
thing1.things.push("foo");
console.log(thing2.things); //logs ["foo"]
```

[jsfiddle](#)

It is usually a mistake to assign arrays or objects on the `prototype`. If you want instances to each have their own arrays, create them in the function, not the prototype.

```
function Thing() {
    this.things = [];
}

var thing1 = new Thing();
var thing2 = new Thing();
thing1.things.push("foo");
console.log(thing1.things); //logs ["foo"]
console.log(thing2.things); //logs []
```

[jsfiddle](#)

You can actually chain the `prototype` of many functions together to form a `prototype chain` so that the special magic of `this` will walk up the `prototype chain` until it has found a value you are trying to reference.

```

function Thing1() {
}
Thing1.prototype.foo = "bar";

function Thing2() {
}
Thing2.prototype = new Thing1();

var thing = new Thing2();
console.log(thing.foo); //logs "bar"

```

[jsfiddle](#)

Some people use this to simulate classical object oriented inheritance in JavaScript.

Any assignments to `this` in functions used to create the `prototype chain` will hide values defined further up the `prototype chain`.

```

function Thing1() {
}
Thing1.prototype.foo = "bar";

function Thing2() {
    this.foo = "foo";
}
Thing2.prototype = new Thing1();

function Thing3() {
}
Thing3.prototype = new Thing2();

var thing = new Thing3();
console.log(thing.foo); //logs "foo"

```

[jsfiddle](#)

I like to call functions assigned to the `prototype` “methods”. I used methods in some examples above, like `logFoo`. These methods get the same magic `this` prototype lookup as the original function used to create the instance. I usually refer to the original function as the constructor.

The use of `this` in methods defined on a `prototype` further up the `prototype chain` refer to the `this` of the current instance. This means if you have hidden a value on the `prototype chain` by assigning to `this` directly,

any of the methods available to the instance will use this new value regardless of what `prototype` that method was assigned to.

```
function Thing1() {
}
Thing1.prototype.foo = "bar";
Thing1.prototype.logFoo = function () {
    console.log(this.foo);
}

function Thing2() {
    this.foo = "foo";
}
Thing2.prototype = new Thing1();

var thing = new Thing2();
thing.logFoo(); //logs "foo";
```

[jsfiddle](#)

In JavaScript you can nest functions, that is you can define functions inside functions. While nested functions capture variables defined in parent functions in a closure, they do not inherit the `this`.

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    var info = "attempting to log this.foo:";
    function doIt() {
        console.log(info, this.foo);
    }
    doIt();
}

var thing = new Thing();
thing.logFoo(); //logs "attempting to log this.foo: undefined"
```

[jsfiddle](#)

The `this` in `doIt` is the `global` object or `undefined` in the case of `"use strict"`. This is a source of much pain for many people not familiar with `this` in JavaScript.

It gets worse. Assigning an instance's method as a value, like say you passed the method as an argument to a function, does not also pass its instance. The `this` context in a method reverts to a reference to the `global` object, or `undefined` in the case of `"use strict";`, when this happens.

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    console.log(this.foo);
}

function doIt(method) {
    method();
}

var thing = new Thing();
thing.logFoo(); //logs "bar"
doIt(thing.logFoo); //logs undefined
```

[jsfiddle](#)

Some people prefer to capture `this` in a variable, usually called “self” and avoid `this` altogether...

```
function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    var self = this;
    var info = "attempting to log this.foo:";
    function doIt() {
        console.log(info, self.foo);
    }
    doIt();
}

var thing = new Thing();
thing.logFoo(); //logs "attempting to log this.foo: bar"
```

[jsfiddle](#)

...but this will not save you in the case you need to pass the method around as a value.

```

function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    var self = this;
    function doIt() {
        console.log(self.foo);
    }
    doIt();
}

function doItIndirectly(method) {
    method();
}

var thing = new Thing();
thing.logFoo(); //logs "bar"
doItIndirectly(thing.logFoo); //logs undefined

```

[jsfiddle](#)

You can pass the instance with the method by using `bind`, a function defined on the function object for all functions and methods.

```

function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    console.log(this.foo);
}

function doIt(method) {
    method();
}

var thing = new Thing();
doIt(thing.logFoo.bind(thing)); //logs bar

```

[jsfiddle](#)

You can also use `apply` and `call` to call the method or function immediately with a new context.

```

function Thing() {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    function doIt() {
        console.log(this.foo);
    }
    doIt.apply(this);
}

function doItIndirectly(method) {
    method();
}

var thing = new Thing();
doItIndirectly(thing.logFoo.bind(thing)); //logs bar

```

[jsfiddle](#)

You can use `bind` to replace the `this` for any function or method, even if it was not assigned to instance's original prototype.

```

function Thing() {
}
Thing.prototype.foo = "bar";

function logFoo(aStr) {
    console.log(aStr, this.foo);
}

var thing = new Thing();
logFoo.bind(thing)("using bind"); //logs "using bind bar"
logFoo.apply(thing, ["using apply"]); //logs "using apply bar"
logFoo.call(thing, "using call"); //logs "using call bar"
logFoo("using nothing"); //logs "using nothing undefined"

```

[jsfiddle](#)

You are going to want to avoid returning anything from your constructor, because it may replace what the resulting instance is.

```

function Thing() {
    return {};
}
Thing.prototype.foo = "bar";

Thing.prototype.logFoo = function () {
    console.log(this.foo);
}

var thing = new Thing();
thing.logFoo(); //Uncaught TypeError: undefined is not a function

```

[jsfiddle](#)

Strangely, if you return a primitive value like a string or a number from a constructor this won't happen and the return statement is ignored. It is best to never return anything from a constructor you intend to call with `new` even if you know what you are doing. If you want to create a factory pattern, use a function to create instances and don't call it with `new`. This advice is only an opinion however.

You can avoid using `new` and instead use `Object.create`. This also creates an instance.

```

function Thing() {
}
Thing.prototype.foo = "bar";

Thing.prototype.logFoo = function () {
    console.log(this.foo);
}

var thing = Object.create(Thing.prototype);
thing.logFoo(); //logs "bar"

```

[jsfiddle](#)

This will not call the constructor however.

```

function Thing() {
    this.foo = "foo";
}
Thing.prototype.foo = "bar";

```

```
Thing.prototype.logFoo = function () {
    console.log(this.foo);
}

var thing = Object.create(Thing.prototype);
thing.logFoo(); //logs "bar"
```

[jsfiddle](#)

Because `Object.create` does not call the constructor it is really useful for creating an inheritance pattern in which you want to override constructors further up the prototype chain.

```
function Thing1() {
    this.foo = "foo";
}
Thing1.prototype.foo = "bar";

function Thing2() {
    this.logFoo(); //logs "bar"
    Thing1.apply(this);
    this.logFoo(); //logs "foo"
}
Thing2.prototype = Object.create(Thing1.prototype);
Thing2.prototype.logFoo = function () {
    console.log(this.foo);
}

var thing = new Thing2();
```

[jsfiddle](#)

object this

You can use `this` in any function on an object to refer to other properties on that object. This is not the same as an instance created with `new`.

```
var obj = {
    foo: "bar",
    logFoo: function () {
        console.log(this.foo);
```

```

    }
};

obj.logFoo(); //logs "bar"

```

[jsfiddle](#)

Note, there was no use of `new`, no `Object.create` and no function called to create `obj`. You can also bind to objects as if they were an instance.

```

var obj = {
  foo: "bar"
};

function logFoo() {
  console.log(this.foo);
}

logFoo.apply(obj); //logs "bar"

```

[jsfiddle](#)

There is no walking up the object hierarchy when you use `this` like this. Only properties shared on the immediate parent object are available via `this`.

```

var obj = {
  foo: "bar",
  deeper: {
    logFoo: function () {
      console.log(this.foo);
    }
  }
};

obj.deeper.logFoo(); //logs undefined

```

[jsfiddle](#)

You can just refer to the property you want directly however:

```

var obj = {
  foo: "bar",
  deeper: {
    logFoo: function () {

```

```

        console.log(obj.foo);
    }
}

};

obj.deeper.logFoo(); //logs "bar"

```

[jsfiddle](#)

DOM event this

In an HTML DOM event handler, `this` is always a reference to the DOM element the event was attached to...

```

function Listener() {
    document.getElementById("foo").addEventListener("click",
        this.handleClick);
}
Listener.prototype.handleClick = function (event) {
    console.log(this); //logs "<div id='foo'></div>"
}

var listener = new Listener();
document.getElementById("foo").click();

```

[jsfiddle](#)

...unless you `bind` the context.

```

function Listener() {
    document.getElementById("foo").addEventListener("click",
        this.handleClick.bind(this));
}
Listener.prototype.handleClick = function (event) {
    console.log(this); //logs Listener {handleClick: function}
}

var listener = new Listener();
document.getElementById("foo").click();

```

[jsfiddle](#)

HTML this

Inside HTML attributes in which you can put JavaScript, `this` is a reference to the element.

```
<div id="foo" onclick="console.log(this);"></div>
<script type="text/javascript">
document.getElementById("foo").click(); //logs <div id="foo"...
</script>
```

[jsfiddle](#)

override this

You cannot override `this` as it is a key word.

```
function test () {
    var this = {};// Uncaught SyntaxError: Unexpected token this
}
```

[jsfiddle](#)

eval this

You can use `eval` to access `this`.

```
function Thing () {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = function () {
    eval("console.log(this.foo)");//logs "bar"
}

var thing = new Thing();
thing.logFoo();
```

[jsfiddle](#)

This can be a security concern. There is no way to prevent this other than to not use `eval`.

Functions created with `Function` also can access `this`:

```
function Thing () {
}
Thing.prototype.foo = "bar";
Thing.prototype.logFoo = new Function("console.log(this.foo);");

var thing = new Thing();
thing.logFoo(); //logs "bar"
```

[jsfiddle](#)

with this

You can use `with` to add `this` to the current scope to read and write to values on `this` without referring to `this` explicitly.

```
function Thing () {  
}  
Thing.prototype.foo = "bar";  
Thing.prototype.logFoo = function () {  
    with (this) {  
        console.log(foo);  
        foo = "foo";  
    }  
}  
  
var thing = new Thing();  
thing.logFoo(); // logs "bar"  
console.log(thing.foo); // logs "foo"
```

[jsfiddle](#)

Many people believe this is bad because of ambiguity problems with `with`.

jQuery this

Like event handlers for HTML DOM elements, the [jQuery JavaScript library](#) will in many places have `this` refer to a DOM element. This is true for event handlers and convenience methods such as `$.each`.

```
<div class="foo bar1"></div>  
<div class="foo bar2"></div>  
<script type="text/javascript">  
  $(".foo").each(function () {  
    console.log(this); //logs <div class="foo...  
  });  
  $(".foo").on("click", function () {  
    console.log(this); //logs <div class="foo...  
  });  
  $(".foo").each(function () {  
    this.click();  
  });  
</script>
```

[jsfiddle](#):

thisArg this

If you use [underscore.js](#) or [lo-dash](#) you may know that in many of the library methods you can pass in an instance, via a `thisArg` function argument that will be used as the context for `this`. This is true for `_.each` for example. Native methods found in JavaScript since ECMAScript 5 also allow for a `thisArg`, like `forEach`. In fact, the previously demonstrated use of `bind`, `apply` and `call` give you the opportunity to provide a `thisArg` that when used binds `this` to the object passed in as the argument.

```
function Thing(type) {
  this.type = type;
}

Thing.prototype.log = function (thing) {
  console.log(this.type, thing);
}

Thing.prototype.logThings = function (arr) {
  arr.forEach(this.log, this); // logs "fruit apples..."
  _.each(arr, this.log, this); //logs "fruit apples..."
}

var thing = new Thing("fruit");
thing.logThings(["apples", "oranges", "strawberries", "bananas"]);
```

[jsfiddle](#)

This enables code to be a little bit cleaner without many nested bind statements and having to resort to using a “self” variable.

Some programming languages can be simple to learn, for example the specification for the [go programming language](#) can be read in a short time. Once the specification has been read, one understands the language, and there are few tricks or gotchas to be troubled about, barring implementation details.

JavaScript is not such a language. The specification is not very readable. There are many “gotchas”, so many in fact that people talk about “[The Good Parts](#).” The best documentation is found on the [Mozilla Developer Network](#). I recommend reading the documentation there about `this` and to always prefix your Google searches for JavaScript with “mdn” to always get the best documentation. Static code analysis is also great help, for that I use [jshint](#).

If I have made any mistakes or if you have any questions, please let me know I am @bjorntipling on Twitter.

Update:

@hicksyfern has a simpler take that I like in his blog post [some of this](#).

58 : Everything you wanted to know about JavaScript scope

<https://toddmotto.com/everything-you-wanted-to-know-about-javascript-scope/>

The JavaScript language has a few concepts of "scope", none of which are straightforward or easy to understand as a new JavaScript developer (and even some experienced JavaScript developers). This post is aimed at those wanting to learn about the many depths of JavaScript after hearing words such as `scope`, `closure`, `this`, `namespace`, `function scope`, `global scope`, `lexical scope` and `public/private scope`. Hopefully by reading this post you'll know the answers to:

- What is Scope?
- What is Global/Local Scope?
- What is a Namespace and how does it differ to Scope?
- What is the `this` keyword and how does Scope affect it?
- What is Function/Lexical Scope?
- What are Closures?
- What is Public/Private Scope?
- How can I understand/create/do all of the above?

What is Scope?

In JavaScript, scope refers to the current context of your code. Scopes can be *globally* or *locally* defined.

Understanding JavaScript scope is key to writing bulletproof code and being a better developer. You'll understand where variables/functions are accessible, be able to change the scope of your code's context and be able to write faster and more maintainable code, as well as debug much faster.

Thinking about scope is easy, are we inside `Scope A` or `Scope B` ?

What is Global Scope?

Before you write a line of JavaScript, you're in what we call the `Global Scope`. If we declare a variable, it's defined globally:

```
// global scope
var name = 'Todd';
```

Global scope is your best friend and your worst nightmare, learning to control your scopes is easy and in doing so, you'll run into no issues with global scope problems (usually namespace clashes). You'll often hear people saying "Global Scope is *bad*", but never really justifying as to *why*. Global scope isn't bad, you need it to create Modules/APIs that are accessible across scopes, you must use it to your advantage and not cause issues.

Everyone's used jQuery before, as soon as you do this...

```
jQuery('.myClass');
```

... we're accessing jQuery in *global* scope, we can refer to this access as the `namespace`. The namespace is sometimes an interchangeable word for scope, but usually refers to the highest level scope. In this case, `jQuery` is in the global scope, and is also our namespace. The `jQuery` namespace is defined in the global scope, which acts as a namespace for the jQuery library as everything inside it becomes a descendent of that namespace.

What is Local Scope?

A local scope refers to any scope defined past the global scope. There is typically one global scope, and each function defined has its own (nested) local scope. Any function defined within another function has a local scope which is linked to the outer function.

If I define a function and create variables inside it, those variables becomes locally scoped. Take this example:

```
// Scope A: Global scope out here
var myFunction = function () {
    // Scope B: Local scope in here
};
```

Any locally scoped items are not visible in the global scope - *unless* exposed, meaning if I define functions or variables within a new scope, it's inaccessible *outside* of that current scope. A simple example of this is the following:

```
var myFunction = function () {
    var name = 'Todd';
    console.log(name); // Todd
};

// Uncaught ReferenceError: name is not defined
console.log(name);
```

The variable `name` is scoped locally, it isn't exposed to the parent scope and therefore undefined.

Function scope

All scopes in JavaScript are created with `Function Scope` *only*, they aren't created by `for` or `while` loops or expression statements like `if` or `switch`. New functions = new scope - that's the rule. A simple example to demonstrate this scope creation:

```
// Scope A
var myFunction = function () {
    // Scope B
    var myOtherFunction = function () {
        // Scope C
    };
};

myFunction();
myOtherFunction();
```

```
};  
};
```

It's easy to create new scope and create local variables/functions/objects.

Lexical Scope

Whenever you see a function within another function, the inner function has access to the scope in the outer function, this is called Lexical Scope or Closure - also referred to as Static Scope. The easiest way to demonstrate that again:

```
// Scope A  
var myFunction = function () {  
    // Scope B  
    var name = 'Todd'; // defined in Scope B  
    var myOtherFunction = function () {  
        // Scope C: `name` is accessible here!  
    };  
};
```

You'll notice that `myOtherFunction` isn't being called here, it's simply defined. Its order of call also has effect on how the scoped variables react, here I've defined my function and called it *under* another `console` statement:

```
var myFunction = function () {  
    var name = 'Todd';  
    var myOtherFunction = function () {  
        console.log('My name is ' + name);  
    };  
    console.log(name);  
    myOtherFunction(); // call function  
};  
  
// Will then log out:  
// `Todd`  
// `My name is Todd`
```

Lexical scope is easy to work with, *any* variables/objects/functions defined in *its* parent scope, are available in the scope chain. For example:

```
var name = 'Todd';  
var scope1 = function () {  
    // name is available here  
    var scope2 = function () {
```

```
// name is available here too

var scope3 = function () {
    // name is also available here!
};

};

};

};
```

The only important thing to remember is that Lexical scope does *not* work backwards. Here we can see how Lexical scope *doesn't* work:

```
// name = undefined
var scope1 = function () {
    // name = undefined
    var scope2 = function () {
        // name = undefined
        var scope3 = function () {
            var name = 'Todd'; // locally scoped
            };
        };
    };
};
```

I can always return a reference to `name`, but never the variable itself.

Scope Chain

Scope chains establish the scope for a given function. Each function defined has its own nested scope as we know, and any function defined within another function has a local scope which is linked to the outer function - this link is called the chain. It's always the *position* in the code that defines the scope. When resolving a variable, JavaScript starts at the innermost scope and searches outwards until it finds the variable/object/function it was looking for.

Closures

Closures ties in very closely with Lexical Scope. A better example of how the *closure* side of things works, can be seen when returning a *function reference* - a more practical usage. Inside our scope, we can return things so that they're available in the parent scope:

```
var sayHello = function (name) {  
    var text = 'Hello, ' + name;  
    return function () {  
        console.log(text);  
    };  
};
```

The `closure` concept we've used here makes our scope inside `sayHello` inaccessible to the public scope.

Calling the function alone will do nothing as it *returns* a function:

```
sayHello('Todd'); // nothing happens, no errors, just silence...
```

The function returns a function, which means it needs assignment, and *then* calling:

```
var helloTodd = sayHello('Todd');
helloTodd(); // will call the closure and log 'Hello, Todd'
```

Okay, I lied, you *can* call it, and you may have seen functions like this, but this will call your closure:

```
sayHello('Bob')(); // calls the returned function without assignment
```

AngularJS uses the above technique for its `$compile` method, where you pass the current scope reference into the closure:

```
$compile(template)(scope);
```

Meaning we could guess that their code would (over-simplified) look like this:

```
var $compile = function (template) {
    // some magic stuff here
    // scope is out of scope, though...
    return function (scope) {
        // access to `template` and `scope` to do magic with too
    };
};
```

A function doesn't *have* to return in order to be called a closure though. Simply accessing variables outside of the immediate lexical scope creates a closure.

Scope and 'this'

Each scope binds a different value of `this` depending on how the function is invoked. We've all used the `this` keyword, but not all of us understand it and how it differs when invoked. By default `this` refers to the outer most global object, the `window`. We can easily show how invoking functions in different ways binds the `this` value differently:

```
var myFunction = function () {
    console.log(this); // this = global, [object Window]
};
```

```
myFunction();

var myObject = {};
myObject.myMethod = function () {
  console.log(this); // this = Object { myObject }
};

var nav = document.querySelector('.nav'); //

var toggleNav = function () {
  console.log(this); // this =
element
};
nav.addEventListener('click', toggleNav, false);
```

There are also problems that we run into when dealing with the `this` value, for instance if I do this, even inside the same function the scope can be changed and the `this` value can be changed:

```
var nav = document.querySelector('.nav'); //

var toggleNav = function () {
  console.log(this); //
element
  setTimeout(function () {
    console.log(this); // [object Window]
  }, 1000);
};
nav.addEventListener('click', toggleNav, false);
```

So what's happened here? We've created new scope which is not invoked from our event handler, so it defaults to the `window` Object as expected. There are several things we can do if we want to access the proper `this` value which isn't affected by the new scope. You might have seen this before, where we can cache a reference to the `this` value using a `that` variable and refer to the lexical binding:

```
var nav = document.querySelector('.nav'); //

var toggleNav = function () {
  var that = this;
  console.log(that); //
element
  setTimeout(function () {
    console.log(that); //
```

```

    element
  }, 1000);
};

nav.addEventListener('click', toggleNav, false);

```

This is a neat little trick to be able to use the proper `this` value and resolve problems with newly created scope.

[Changing scope with .call\(\), .apply\(\) and .bind\(\)](#)

Sometimes you need to manipulate the scopes of your JavaScript depending on what you're looking to do. A simple demonstration of how to change the scope when looping:

```

var links = document.querySelectorAll('nav li');
for (var i = 0; i < links.length; i++) {
  console.log(this); // [object Window]
}

```

The `this` value here doesn't refer to our elements, we're not invoking anything or changing the scope. Let's look at how we can change scope (well, it looks like we change scope, but what we're really doing is changing the *context* of how the function is called).

[.call\(\) and .apply\(\)](#)

The `.call()` and `.apply()` methods are really sweet, they allow you to pass in a scope to a function, which binds the correct `this` value. Let's manipulate the above function to make it so that our `this` value is each element in the array:

```

var links = document.querySelectorAll('nav li');
for (var i = 0; i < links.length; i++) {
  (function () {
    console.log(this);
  }).call(links[i]);
}

```

You can see I'm passing in the current element in the Array iteration, `links[i]`, which changes the scope of the function so that the `this` value becomes that iterated element. We can then use the `this` binding if we wanted. We can use either `.call()` or `.apply()` to change the scope, but any further arguments are where the two differ: `.call(scope, arg1, arg2, arg3)` takes individual arguments, comma separated, whereas `.apply(scope, [arg1, arg2])` takes an Array of arguments.

It's important to remember that using `.call()` or `.apply()` actually invokes your function, so instead of doing this:

```
myFunction(); // invoke myFunction
```

You'll let `.call()` handle it and chain the method:

```
myFunction.call(scope); // invoke myFunction using .call()
```

[bind\(\)](#)

Unlike the above, using `.bind()` does not *invoke* a function, it merely binds the values before the function is invoked. It's a real shame this was introduced in ECMAScript 5 and not earlier as this method is fantastic. As you know we can't pass parameters into function references, something like this:

```
// works
nav.addEventListener('click', toggleNav, false);

// will invoke the function immediately
nav.addEventListener('click', toggleNav(arg1, arg2), false);
```

We *can* fix this, by creating a new function inside it:

```
nav.addEventListener('click', function () {
  toggleNav(arg1, arg2);
}, false);
```

But again this changes scope and we're creating a needless function again, which will be costly on performance if we were inside a loop and binding event listeners. This is where `.bind()` shines through, as we can pass in arguments but the functions are not called:

```
nav.addEventListener('click', toggleNav.bind(scope, arg1, arg2), false);
```

The function isn't invoked, and the scope can be changed if needed, but arguments are sat waiting to be passed in.

[Private and Public Scope](#)

In many programming languages, you'll hear about `public` and `private` scope, in JavaScript there is no such thing. We can, however, emulate public and private scope through things like Closures.

By using JavaScript design patterns, such as the `Module` pattern for example, we can create `public` and `private` scope. A simple way to create private scope, is by wrapping our functions inside a function. As we've learned, functions create scope, which keeps things out of the global scope:

```
(function () {
  // private scope inside here
```

```
})();
```

We might then add a few functions for use in our app:

```
(function () {
    var myFunction = function () {
        // do some stuff here
    };
})();
```

But when we come to calling our function, it would be out of scope:

```
(function () {
    var myFunction = function () {
        // do some stuff here
    };
})();

myFunction(); // Uncaught ReferenceError: myFunction is not defined
```

Success! We've created private scope. But what if I want the function to be public? There's a great pattern (called the Module Pattern [and Revealing Module Pattern]) which allows us to scope our functions correctly, using private and public scope and an `Object`. Here I grab my global namespace, called `Module`, which contains all of my relevant code for that module:

```
// define module
var Module = (function () {
    return {
        myMethod: function () {
            console.log('myMethod has been called.');
        }
    };
})();

// call module + methods
Module.myMethod();
```

The `return` statement here is what returns our `public` methods, which are accessible in the global scope - *but* are `namespaced`. This means our Module takes care of our namespace, and can contain as many methods as we want. We can extend the Module as we wish:

```
// define module
var Module = (function () {
    return {
        myMethod: function () {

        },
        someOtherMethod: function () {

        }
    };
})();

// call module + methods
Module.myMethod();
Module.someOtherMethod();
```

So what about private methods? This is where a lot of developers go wrong and pollute the global namespace by dumping all their functions in the global scope. Functions that help our code *work* do not need to be in the global scope, only the API calls do - things that *need* to be accessed globally in order to work. Here's how we can create private scope, by *not* returning functions:

```
var Module = (function () {
    var privateMethod = function () {

    };
    return {
        publicMethod: function () {

        }
    };
})();
```

This means that `publicMethod` can be called, but `privateMethod` cannot, as it's privately scoped! These privately scoped functions are things like helpers, addClass, removeClass, Ajax/XHR calls, Arrays, Objects, anything you can think of.

Here's an interesting twist though, anything in the same scope has access to anything in the same scope, even *after* the function has been returned. Which means, our `public` methods have *access* to our `private` ones, so they can still interact but are unaccessible in the global scope.

```
var Module = (function () {
    var privateMethod = function () {
```

```

};

return {

  publicMethod: function () {
    // has access to `privateMethod`, we can call it:
    // privateMethod();
  }
};

})();

```

This allows a very powerful level of interactivity, as well as code security. A very important part of JavaScript is ensuring security, which is exactly *why* we can't afford to put all functions in the global scope as they'll be publicly available, which makes them open to vulnerable attacks.

Here's an example of returning an Object, making use of `public` and `private` methods:

```

var Module = (function () {
  var myModule = {};
  var privateMethod = function () {

  };
  myModule.publicMethod = function () {

  };
  myModule.anotherPublicMethod = function () {

  };
  return myModule; // returns the Object with public methods
})();

// usage
Module.publicMethod();

```

One neat naming convention is to begin `private` methods with an underscore, which visually helps you differentiate between public and private:

```

var Module = (function () {
  var _privateMethod = function () {

  };
  var publicMethod = function () {

  };
})();

```

This helps us when returning an anonymous `Object`, which the Module can use in Object fashion as we can simply assign the function references:

```
var Module = (function () {
    var _privateMethod = function () {

    };
    var publicMethod = function () {

    };
    return {
        publicMethod: publicMethod,
        anotherPublicMethod: anotherPublicMethod
    }
})();
```

Happy scoping!

59 : Advanced Javascript: Objects, Arrays, and Array-Like objects

Javascript `objects` and `arrays` are both incredibly useful. They're also incredibly easy to confuse with each other. Mix in a few objects that look like arrays and you've got a recipe for confusion!

We're going to see what the differences between objects and arrays are, how to work with some of the common array-like objects, and how to get the most performance out of each.

What Objects Are

A javascript object is a basic data structure:

```
var basicObj = {} // an empty object - {} is a shortcut for "new Object()"

basicObj.suprise= "cake!";

basicObj['suprise']; // returns "cake!"
```

Using `{}` instead of `new Object()`; is know as "Object Literal" syntax.

```
var fancyObj = {
    favoriteFood: "pizza",
    add: function(a, b){
        return a + b;
    }
};

fancyObj.add(2,3); // returns 5

fancyObj['add'](2,3); // ditto.
```

As you can see, and probably already knew, properties can be accessed a couple of different ways. However, it's an important point that we'll come back to in a minute.

Everything in javascript is an `object`. Everything. `Arrays`, `functions`, even `numbers`! Because of this, you can do some really interesting things, such as modifying the `prototypes` of Objects, Arrays, etc.

```
// an example of something you probably shouldn't do. Ever. Seriously.

Number.prototype.addto = function(x){
    return this + x;
}

(8).addto(9); // returns 17
```

```
// other variations:

8.addto(9); // gives a syntax error, because the dot is assumed to be a decimal point

8['addto'](9); // works but is kind of ugly compared to the first method

var eight = 8;
eight.addto(9); // works
```

What Arrays Are

Javascript arrays are a type of `object` used for storing multiple values in a single variable. Each value gets a numeric index and may be any data type.

```
var arr = []; // this is a shortcut for new Array();
arr[0] = "cat";
arr[1] = "mouse";
```

See how that syntax is so similar to the syntax used for setting object properties? In fact, the only difference is that objects use a string while arrays use a number. This is why arrays get confused with objects so often.

Length

Arrays have a `length` property that tells how many items are in the array and is automatically updated when you add or remove items to the array.

```
var arr = [];
arr[0] = "cat"; // this adds to the array
arr[1] = "mouse"; // this adds to the array
arr.length; // returns 2

arr["favoriteFood"] = "pizza"; // this DOES NOT add to the array. Setting a string
parameter adds to the underlying object
arr.length; // returns 2, not 3
```

The `length` property is only modified when you add an item to the array, not the underlying object.

The `length` is always 1 higher than the highest index, even if there are actually fewer items in the array.

```
var arr = [];
arr.length; // returns 0;
```

```
arr[100] = "this is the only item in the array";
arr.length; // returns 101, even though there is only 1 object in the array
```

This is somewhat counter-intuitive. PHP does more what you would expect:

You can manually set the `length` also. Setting it to 0 is a simple way to empty an array.

In addition to this `length` property, arrays have lots of nifty built in functions such as `push()`, `pop()`, `sort()`, `slice()`, `splice()`, and more. This is what sets them apart from Array-Like Objects.

Array-like Objects

Array-like objects look like arrays. They have various numbered elements and a `length` property. But that's where the similarity stops. Array-like objects do not have any of Array's functions, and for-in loops don't even work!

You'll come across these more often than you might expect. A common one is the `arguments` variable that is present inside of every js function.

Also included in the category are the HTML node sets returned by `document.getElementsByTagName()`, `document.forms`, and basically every other DOM method and property that gives a list of items.

```
document.forms.length; // returns 1;
document.forms[0]; // returns a form element.
document.forms.join(", "); // throws a type error. this is not an array.
typeof document.forms; // returns "object"
```

Did you know you can send any number of arguments you want to a javascript function? They're all stored in an array-like object named `arguments`.

```
function takesTwoParams(a, b){
    // arguments is an array-like variable that is automatically created
    // arguments.length works great

    alert ("you gave me "+arguments.length+" arguments");

    for(i=0; i< arguments.length; i++){
        alert("parameter " + i + " = " + arguments[i]);
    }
}

takesTwoParams("one","two","three");
// alerts "you gave me 3 arguments",
```

```
// then "parameter 0 = one"
// etc.
```

Tip: Parameters are the named variables in a function's signature: `a` and `b` in the previous example.

Arguments, by contrast, are the expressions that are used when calling the function: `"one"`, `"two"`, and `"three"` in this case.

This works great. But that's about as far as you can go with array-like objects. The flowing example does not work:

```
function takesTwoParams(a, b){
    alert(" your parameters were " + arguments.join(", "));
    // throws a type error because arguments.join doesn't exist
}
```

So what can you do?

Well you could make your own `join()` function, but that adds a lot of unnecessary overhead to your code because it has to loop over everything. If only there were a quick way to get an array out of an array like object...

It turns out there is.

The array functions can be called on non-array objects as long as you know where to find the function (usually they're attached to the array, but this isn't an array remember



Prototype to the win:

```
function takesTwoParams(a, b){
    var args = Array.prototype.slice.call(arguments);
    alert(" your parameters were " + args.join(", "));
    // yay, this works!
}
```

Let's take a look at that a bit more in-depth:

`Array` : This object is the original array that all other arrays inherit their properties from.

`Array.prototype` : This gives us access to all the methods properties that each array inherits

`Array.prototype.slice` : The original slice method that is given to all arrays via the prototype chain. We can't call it directly though, because when it runs internally, it looks at the `this` keyword, and calling it here would make `this` point to `Array`, not our `arguments` variable.

`Array.prototype.slice.call()` : `call()` and `apply()` are prototype methods of the `Function` object, meaning that they can be called on every function in javascript. These allow you to change what the `this` variable points to inside a given function.

And finally, you get a regular `array` back! This works because javascript returns a new object of type Array rather than whatever you gave it. This causes a lot of headaches for a [few people](#) who are trying to make subclasses of Array, but it's very handy in our case!

Gotchas

First, in Internet Explorer, DOM `NodeLists` are not considered to be javascript objects, so you cannot call `Array.prototype.slice` on them. If you want an array, you'll have to loop through it the old fashioned way. Or use a hybrid function that tries it the fast way first, then the slow way if that doesn't work.

```
function hybridToArray(nodes){
    try{
        // works in every browser except IE
        var arr = Array.prototype.slice.call(nodes);
        return arr;
    } catch(err){
        // slower, but works in IE
        var arr = [],
            length = nodes.length;
        for(var i=0; i < length; i++){
            arr.push(nodes[i]);
        }
        return arr;
    }
}
```

See an example here: <http://nfriedly.com/demos/ie-nodelist-to-array>.

Second, arrays are objects, so you can do this, but it can get you some serious inconsistencies:

```
arr = [];
arr[0] = "first element"; // adds item to the array
arr.length; // returns 1

arr.two = "second element"; // adds an item to the underlying object that array is built
on top of.
arr.length; // still returns 1 !

// BUT...
for(i in arr){
    // this will hit both 0 and "two"
}
```

Another solution: wrap arrays in an object if you need both worlds

This is basically a less efficient method of the array subclassing links I mentioned above. While less efficient, it has the advantage of being simple and reliable.

That said, I wouldn't recommend that you use this in most cases due to issues with speed and extra code requirements. It's provided here as an example.

```
// an example of a wrapper for an array.
// not recommended for most situations.

var ArrayContainer = function(arr){
    this.arr = arr || [];
    this.length = this.arr.length;
};

ArrayContainer.prototype.add=  function(item){
    index = this.arr.length;
    this.arr[index] = item;
    this.length = this.arr.length;
    return index;
};

ArrayContainer.prototype.get=  function(index){
    return this.arr[index];
};

ArrayContainer.prototype.forEach=  function(fn){
    if (this.arr.forEach) this.arr.forEach(fn); // use native code if it's there
    else {
        for(i in this.arr){
            fn( i, this.arr[i], this.arr );
        }
    }
};

var mySuperDooperArray = new ArrayContainer();
```

Now that your array is (somewhat) protected on the inside, you can loop through its items with `forEach()` and know that they will match its length. You can also add arbitrary properties to `ArrayContainer` or `mySuperDooperArray` and they **won't** get pulled into your `forEach()` loop.

This example could be extended to completely protect the array if the need arose.

An Even Better Solution: [Hire a javascript expert](#).

nFriedly Web Development is a top ranked [Javascript and AJAX ninja](#) with an extensive portfolio of proven results. I can bring your project to life and make it run faster than you ever imagined. [Get in touch](#) with me or get a free [instant estimate](#) for new projects.

60 : Advanced Javascript: Logical Operators and truthy / falsy

<http://www.nfriedly.com/techblog/2009/07/advanced-javascript-operators-and-truthy-falsy/>

Nearly every website on the internet uses javascript in some form or fashion. Yet very few people, even those who write it and teach it, have a clear understanding of how javascript works!

Logical Operators are a core part of the language. We're going to look at what logical operators are, what "truthy" and "falsy" mean, and **how to use this to write cleaner, faster and more optimized javascript**.

Javascript Logical Operators

In traditional programming, operators such as `&&` and `||` returned a boolean value (`true` or `false`). This is not the case in javascript. Here it returns the actual `object`, not a `true` / `false`. To really explain this, I first have to explain what is truthy and what is falsy.

Truthy or Falsy

When javascript is expecting a `boolean` and it's given something else, it decides whether the something else is "truthy" or "falsy".

An empty string (`' '`), the number `0`, `null`, `NaN`, a boolean `false`, and `undefined` variables are all "falsy". Everything else is "truthy".

```
var emptyString = ""; // falsy

var nonEmptyString = "this is text"; // truthy

var numberZero = 0; // falsy

var numberOne = 1; // truthy

var emptyArray = []; // truthy, BUT []==false is true. More below.

var emptyObject = {};// truthy

// NaN is a special javascript object for "Not a Number".
var notANumber = 5 / "tree"; // falsy

function exampleFunction(){
    alert("Test");
}

// exampleFunction is truthy
// BUT exampleFunction() is falsy because it has no return (undefined)
```

Gotchas to watch out for: the strings `"0"` and `"false"` are both considered truthy. You can convert a string to a number with the `parseInt()` and `parseFloat()` functions, or by just multiplying it by `1`.

```
var test = "0"; // this is a string, not a number

var a = (test == false); // a is false, meaning that test is truthy

var b = (test * 1 == false); // b is true, meaning that `test * 1` is falsy
```

Arrays are particularly weird. If you just test it for truthyness, an empty array is truthy. HOWEVER, if you compare an empty array to a boolean, it becomes falsy:

```
if ( [] == false ) {
    // this code runs
}

if ( [] ) {
    // this code also runs
}

if ([] == true) {
    // this code doesn't run
}

if ( ![] ) {
    // this code also doesn't run
}
```

(This is because when you do a comparison, its elements are turned to `String`s and joined. Since it's empty, it becomes `""` which is then falsy. Yea, [it's weird](#).)

If you write any PHP, then there's an additional gotcha to watch out for: while javascript evaluates empty arrays as true, PHP evaluates them as false.

PHP also evaluates `"0"` as falsy. (However the string `"false"` is evaluated as truthy by both PHP and javascript.)

How Logical Operators Work

Logical OR, `||`

The logical OR operator, `||`, is very simple after you understand what it is doing. If the first object is truthy, that gets returned. Otherwise, the second object gets returned.

```

var a = ("test one" || "test two"); // a is "test one"

var b = ("test one" || ""); // b is "test one"

var c = (0 || "test two"); // c is "Test two"

var d = (0 || false); // d is false

```

Where would you ever use this? The OR operator allows you to easily specify default variables in a function.

```

function sayHi(name){
    name = name || "Dave";
    alert("Hi " + name);
}

sayHi("Nathan"); // alerts "Hi Nathan";

sayHi(); // alerts "Hi Dave", name is set to null when the function is started

```

Logical AND, &&

The logical AND operator, `&&`, works similarly. If the first object is falsy, it returns that object. If it is truthy, it returns the second object.

```

var a = ("test one" && "test two"); // a is "test two"

var b = ("test one" && ""); // b is ""

var c = (0 && "test two") // c is 0

```

The logical AND allows you to make one variable dependent on another.

```

var checkbox = document.getElementById("agreeToTerms");

var name = checkbox.checked && prompt("What is your name");

// name is either their name, or false if they haven't checked the AgreeToTerms checkbox

```

Logical NOT, !

Unlike `&&` and `||`, the `!` operator DOES turn the value it receives into a boolean. If it receives a truthy value, it returns `false`, and if it receives a falsy value, it returns `true`.

```
var a = (!"test one" || "test two"); // a is "test two"  
// ("test one" gets converted to false and skipped)  
  
var b = (!"test one" && "test two"); // b is false  
// ("test one" gets converted to false and returned)  
  
var c = (!0 || !"test two"); // c is true  
// (0 gets converted to true and returned)
```

Another useful way to use the `!` operator is to use two of them -- this way you always get a `true` or a `false` no matter what was given to it.

```
var a = (!!"test"); // a is true  
// "test" is converted to false, then that is converted to true  
  
var b = (!! ""); // b is false  
// "" is converted to true, and then that true is converted to false  
  
var c = (!!variableThatDoesntExist); // c is false even though you're checking an  
undefined variable.
```

[Javascript Optimization](#)

Need any help [optimizing the Javascript and AJAX on your website](#)? Get in touch with your friendly neighborhood [javascript expert](#) for ideas on how to optimize your site and a free quote.

61 : Design Patterns - Decorating in JavaScript

<http://dealwithjs.io/design-patterns-decorating-in-javascript/>

Today we are going to talk about the decorator design pattern. More specifically, how to decorate functions in JavaScript.

TL; DR

Source code can be found on [GitHub](#), [jsDecorator](#) or [jsFiddle](#)

Design Patterns

Decorating in JavaScript

The decorator design pattern is for extending the functionality or state of an object by repeatedly wrapping it.

That's quite a broad definition, it can mean many different things. In JavaScript - and other languages with first class functions, like Python - there is a specific use case for this. Namely, to extend a function by wrapping it in an outer function.

This can be useful for temporarily augmenting functions while developing or debugging an application (logging, call counting, performance profiling, etc). But even production code can be greatly simplified and made more readable by adding frequently used extra functionality via decorators (authentication, caching, permission handling, analytics, etc).

Let's say we have a suspected bottleneck function:

```
function isPrime(num) {
    if (num === 2 || num === 3) {
        return true;
    }
    if (num % 2 === 0) {
        return false;
    }
    let max = Math.ceil(Math.sqrt(num));
    for (let i = 3; i <= max; i += 2) {
        if (num % i === 0) {
            return false;
        }
    }
}
```

```
    return true;
}
```

We want to measure the time it takes to execute this function, but we don't really want to manually modify its code, and revert it after the measurement. Not to mention this is a fairly common thing we do, so an ideal solution would be a write once - reuse any time. We can achieve that by a decorator function.

```
function logDecorator(func) {
  let decorated = function () {
    let start = performance.now(),
        result = func.apply(this, arguments),
        time = Math.round((performance.now() - start) * 1000);
    console.log(`#${time} µs`);
    return result;
  };
  return decorated;
}
```

As you can see, our decorator function gets a `func` function argument, creates and returns a `decorated` replacement function, which does the extra functionality we want (logging time in this case), and calls the original `func` function, much like we used to call `super` in overridden methods.

We can apply our decorator to any function like this:

```
let isPrimeLog = logDecorator(isPrime);
isPrimeLog(22586101147);

// output:
// 4555 µs
```

I think it's easier to understand if we give it a separate name, also we will compare two differently decorated functions later, and in that case, they need different names anyway. But of course we could just as easily decorate the `isPrime` function in place:

```
isPrime = logDecorator(isPrime);
```

By adding two more things, we can make our log decorator more generic and useful:

1. The anonymous `decorated` function should be a named one. I generally try not to use anonymous functions unless it's justifiable, because they lead to unreadable stack traces. Also, I would like to log the function name. So I derive a name from the original name of `func`.
2. I compile a string from the arguments, so I can log that too.

These functionalities will also be useful in our second example, so I've put them in separate utility functions (which I omitted for brevity - see the source code for details).

```

function logDecorator(func) {
    let decorated = function () {
        let params = formatArguments(arguments),
            start = performance.now(),
            result = func.apply(this, arguments),
            time = Math.round((performance.now() - start) * 1000);
        console.log(`#${decorated.name}(${params}) => ${result} (${time} µs}`);
        return result;
    };
    return renameFunction(decorated, func.name + 'Log');
}

// output for the same call:
// isPrimeLog(22586101147) => true (4555 µs)

```

As you can see, the output became more rich and useful. And we can put this on any function we want. Now it's probably easier for you to imagine the things we can do with this. And we aren't limited to adding functionality, we can also add state via closure.

Consider the second example: we want to add caching to functions. If we call our function with a set of arguments, it calculates the result and stores it. Whenever we call it with the same set of arguments again, it will get the result from the cache instead of recalculating it every time. This is called [memoization](#), and can be a massive performance increase in many situations. And if we have an easily applicable solution, we can try it on any function without messing with our code too much.

```

function memoDecorator(func) {
    let cache = {},
        decorated = function () {
            let params = formatArguments(arguments);
            if (params in cache) {
                return cache[params];
            } else {
                let result = func.apply(this, arguments);
                cache[params] = result;
                return result;
            }
    };
    return renameFunction(decorated, func.name + 'Memo');
}

```

Let's try it out:

```
let num = 22586101147;

let isPrimeLog = logDecorator(isPrime);
isPrimeLog(num);
isPrimeLog(num);
isPrimeLog(num);

// output
// isPrimeLog(22586101147) => true (4555 µs)
// isPrimeLog(22586101147) => true (5645 µs)
// isPrimeLog(22586101147) => true (3925 µs)

isPrimeMemoLog = logDecorator(memoDecorator(isPrime));
isPrimeMemoLog(num);
isPrimeMemoLog(num);
isPrimeMemoLog(num);

// output
// isPrimeMemoLog(22586101147) => true (3485 µs)
// isPrimeMemoLog(22586101147) => true (5 µs)
// isPrimeMemoLog(22586101147) => true (5 µs)
```

Mostly we did the same thing as in the first example. One key difference to note is how we create the `cache` object in our decorator, so the `decorated` function can access it via closure. In other words, we gave a shared state to the different calls of the `decorated` function.

Also note how we double-decorated our second function.

As you can see `memoDecorator` makes subsequent calls return the cached result instantly.

When you wrap your head around it, using decorators in JavaScript is a simple, but very powerful design pattern. It can be used effectively and elegantly, but - as almost everything - can be abused too. I encourage you to try it, play with it, try to come up with use cases in your line of work, and see what you can build with it.

Finally, I want to mention that there are plans to include a special syntax for using decorators in ES7. You can read more about that [here](#).

If you enjoyed reading this blog post, please share it or like the [Facebook page](#) of the website to get regular updates. You can also follow us [@dealwithjs](#).

Happy decorating!

62 : Design Patterns - The Builder Pattern in JavaScript

<http://dealwithjs.io/design-patterns-the-builder-pattern-in-javascript/>

TL; DR

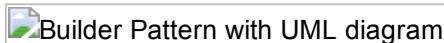
Source code can be found on [GitHub, jsBuilder repo](#)

Design Patterns - The Builder

The Builder pattern is a creational design pattern, which should be applied when complex objects have to be created. When applying this pattern the goal is to separate the object creation (or initialization) logic from the representation.

This pattern should be used when object creation is complicated and involves a lot of code.

UML diagram¹ with the Builder pattern:



This pattern has three components, the **Director** which uses the Builder to create objects, the role of the Director is to coordinate the object creation in case there are multiple Builders involved. The Director also provides an extra abstraction layer when implementing this pattern.

The **Builder** is an abstraction layer above all the Concrete Builders which create the objects. In JavaScript the **Builder** is not separated clearly and in some case it can be left out completely (*although I forced it to be in the sample code*).

The **Concrete Builders** implement the real object creation. Concrete Builders, internally, can use [Factories](#) for building parts of the final object (*I used factories in my implementation to demonstrate this*).

I present the Builder pattern using an analogy.

Lets imagine the process of "building" a cake. In this analogy a cake is considered a complex object, because it is hard to create. The cake object has three parts, the layers, the cream and the topping.

I modeled this scenario in code. Let's start from the Director implementation. In this scenario the Director or Coordinator is represented by the `PastryCook` function, this handles everything related to "building" the cake.

```
function PastryCook() {
    var chocolateCakeBuilder = ChocolateCakeBuilder.getBuilder();
    var strawberryCakeBuilder = StrawberryCakeBuilder.getBuilder();
    var traditionalCakeBuilder = TraditionalCakeBuilder.getBuilder();

    return {
        buildCake: function(flavor) {
            var cake = null;
            switch (flavor) {
```

```

        case 'chocolate':
            cake = chocolateCakeBuilder.buildCake();
            break;
        case 'strawberry':
            cake = strawberryCakeBuilder.buildCake();
            break;
        default:
            cake = traditionalCakeBuilder.buildCake();
            break;
    }

    return cake;
}
};

}

```

The `PastryCook` has a `buildCake` function, which receives the `flavor` argument. Based on the value of the argument I used different **Builders** to build the cake. By adding the possibility to create different cake objects based on a parameter passed to the Director I could demonstrate how to use multiple builders.

The implementation of `ChocolateCakeBuilder` handles the steps of building a chocolate cake. In this Builder I used factories to simplify the implementation and to provide an extra layer of abstraction.

```

function ChocolateCakeBuilder() {
    var layerFactory = LayerFactory.getInstance();
    var creamFactory = CreamFactory.getInstance();
    var toppingFactory = ToppingFactory.getInstance();

    return {
        buildCake: function() {
            return {
                layer: layerFactory.getStandard(),
                cream: creamFactory.getChocolate(),
                topping: toppingFactory.getChocolate()
            }
        }
    };
}

```

Every Builder implements the `buildCake` function, this is the method which delivers the Product (*referring to the UML diagram*).

The `StrawberryCakeBuilder` is very similar in implementation:

```

function StrawberryCakeBuilder() {
    var layerFactory = LayerFactory.getInstance();
    var creamFactory = CreamFactory.getInstance();
    var toppingFactory = ToppingFactory.getInstance();

    return {
        buildCake: function() {
            return {
                layer: layerFactory.getLowCarb(),
                cream: creamFactory.getWhipped(),
                topping: toppingFactory.getStrawberry()
            }
        }
    };
}

```

For the Builders I used Factories, because these (*in general*) add more flexibility to the code and reduce the maintenance cost of the codebase.

The source code of `ToppingFactory` is below; all it does is return a `string` with the selected topping.

```

function ToppingFactory() {
    return {
        getChocolate: function() {
            return 'Topping: [Chocolate]';
        },

        getVanilla: function() {
            return 'Topping: [Vanilla]';
        },

        getStrawberry: function() {
            return 'Topping: [Strawberry]';
        },
    };

    module.exports = {
        getInstance: ToppingFactory
    };
}

```

Sample code was written for `node`, the [index.js](#) has the demo code:

```

var PastryCook = require('./PastryCook');

var cakeBuilder = PastryCook.getBuilder();

var chocolateCake = cakeBuilder.buildCake('chocolate');
var strawberryCake = cakeBuilder.buildCake('strawberry');
var cake = cakeBuilder.buildCake();

console.log("The Chocolate cake is compound of:");
console.log(JSON.stringify(chocolateCake, null, 2));

console.log("The Strawberry cake is compound of:");
console.log(JSON.stringify(strawberryCake, null, 2));

console.log("The cake is compound of:");
console.log(JSON.stringify(cake, null, 2));

```

Once the code is executed the output should be:

```

The Chocolate cake is compound of:
{
  "layer": "Layer: [Standard]",
  "cream": "Cream: [Chocolate]",
  "topping": "Topping: [Chocolate]"
}

The Strawberry cake is compound of:
{
  "layer": "Layer: [LowCarb]",
  "cream": "Cream: [Whipped]",
  "topping": "Topping: [Strawberry]"
}

The cake is compound of:
{
  "layer": "Layer: [Standard]",
  "cream": "Cream: [Peanut Butter]",
  "topping": "Topping: [Strawberry]"
}

```

I tried to keep the implementation clean and developer friendly so the real use case of the Builder design pattern can be seen and understood.

If you enjoyed reading this blog post, please share it or [like](#) the Facebook page of the website to get regular updates.

Thank you for reading!

63 : Better JavaScript with ES6, Pt. II: A Deep Dive into Classes

<https://scotch.io/tutorials/better-javascript-with-es6-pt-ii-a-deep-dive-into-classes>

Out with the Old, In with the new

Let's be clear about one thing from the start:

Under the hood, ES6 classes are not something that is radically new: They mainly provide more convenient syntax to create old-school constructor functions. ~ Axel Rauschmayer

Functionally, `class` is little more than syntactic sugar over the prototype-based behavior delegation capabilities we've had all along. This article will take a close look at the basic use of ES2015's `class` keyword, from the perspective of its relation to prototypes. We'll cover:

- Defining and instantiating classes;
- Creating subclasses with `extends`;
- `super` calls from subclasses; and
- Examples of important symbol methods.

Along the way, we'll pay special attention to how `class` maps to prototype-based code under the hood.

Let's take it from the top.

A Step Back: What Classes Aren't

JavaScript's "classes" aren't anything like classes in Java, Python, or . . . Really, any other object-oriented language you're likely to have used. Which, by the way, I'll refer to as *class-oriented languages*, as that's more accurate.

In traditional class-oriented languages, you create *classes*, which are templates for *objects*. When you want a new object, you *instantiate* the class, which tells the language engine to *copy* the methods and properties of the class into a new entity, called an *instance*. The *instance* is your object, and, after instantiation, has absolutely no active relation with the parent class.

JavaScript does *not* have such copy mechanics. "Instantiating" a `class` in JavaScript *does* create a new object, but *not* one that is independent of its parent class.

Rather, it creates an object that is linked to a *prototype*. Changes to that prototype propagate to the new object, even *after* instantiation.

Prototypes are an immensely powerful design pattern in their own right. There are a number of techniques for using them to emulate something like traditional class mechanics, and it's these techniques that `class` provides compact syntax for.

To summarize:

1. JavaScript *does not* have classes, the way that Java and other languages have classes; and

2. JavaScript's `class` is (mostly) just syntactical sugar for prototypes, which are *very* different from traditional classes.

With that out of the way, let's get our feet wet with `class`.

Base Classes: Declarations & Expressions

You create classes with the `class` keyword, followed by an identifier, and finally, a code block, called the *class body*. These are called **class declarations**. Class declarations that don't use the `extends` keyword are called **base classes**:

```
"use strict";

// Food is a base class
class Food {

    constructor (name, protein, carbs, fat) {
        this.name = name;
        this.protein = protein;
        this.carbs = carbs;
        this.fat = fat;
    }

    toString () {
        return `${this.name} | ${this.protein}g P :: ${this.carbs}g C :: ${this.fat}g F`;
    }

    print () {
        console.log( this.toString() );
    }
}

const chicken_breast = new Food('Chicken Breast', 26, 0, 3.5);

chicken_breast.print(); // 'Chicken Breast | 26g P :: 0g C :: 3.5g F'
console.log(chicken_breast.protein); // 26 (LINE A)
```

A few things to note.

- Classes can *only* contain method definitions, **not** data properties;
- When defining methods, you use [shorthand method definitions](#);
- Unlike when creating objects, you do *not* separate method definitions in class bodies with commas; and
- You *can* refer to properties on instances of the class directly (Line A).

A distinctive feature of classes is the function called **constructor**. This is where you initialize your object's properties.

You don't *have* to define a constructor function. If you choose not to, the engine will insert an empty one for you:

```
"use strict";

class NoConstructor {
    /* JavaScript inserts something like this:
        constructor () { }
    */
}

const nemo = new NoConstructor(); // Works, but pretty boring
```

Assigning a class to a variable is called a **class expression**, and is an alternative to the above syntax:

```
"use strict";

// This is an anonymous class expression -- you can't refer to the it by name within the
// class body.
const Food = class {
    // Class definition is the same as before. . .
}

// This is a named class expression -- you /can/ refer to this class by name within the
// class body . . .
const Food = class FoodClass {
    // Class definition is the same as before . . .

    // Adding new method, to demonstrate we can refer to FoodClass by name
    // within the class . . .
    printMacronutrients () {
        console.log(` ${FoodClass.name} | ${FoodClass.protein} g P :: ${FoodClass.carbs} g
C :: ${FoodClass.fat} g F`)
    }
}

const chicken_breast = new Food('Chicken Breast', 26, 0, 3.5);
chicken_breast.printMacronutrients(); // 'Chicken Breast | 26g P :: 0g C :: 3.5g F'

// . . . But /not/ outside of it
try {
    console.log(FoodClass.protein); // ReferenceError
```

```

} catch (err) {
    // pass
}

```

This behavior is analogous to that of [anonymous and named function expressions](#).

Creating Subclasses with `extends` & Calling with `super`

Classes created with `extends` are called **subclasses**, or **derived classes**. Using them is straightforward.

Building on our Food example:

```

"use strict";

// FatFreeFood is a derived class
class FatFreeFood extends Food {

    constructor (name, protein, carbs) {
        super(name, protein, carbs, 0);
    }

    print () {
        super.print();
        console.log(`Would you look at that -- ${this.name} has no fat!`);
    }
}

const fat_free_yogurt = new FatFreeFood('Greek Yogurt', 16, 12);
fat_free_yogurt.print(); // 'Greek Yogurt | 26g P :: 16g C :: 0g F / Would you look at
that -- Greek Yogurt has no fat!'

```

Everything we discussed above regarding base classe holds true for derived classes, but with a few additional points.

- Subclasses are declared with the `class` keyword, followed by an identifier, and then the `extends` keyword, followed by an *arbitrary expression*. This will generally just be an identifier, [but could, in theory, be a function](#).
- If your derived class needs to refer to the class it extends, it can do so with the `super` keyword.
- A derived class can't contain an empty constructor. Even if all the constructor does is call `super()`, you'll still have to do so explicitly. It can, however, contain *no* constructor.
- You *must* call `super` in the constructor of a derived class before you use `this`.

In JavaScript, there are precisely two use cases for the `super` keyword.

1. **Within subclass constructor calls.** If initializing your derived class requires you to use the parent class's constructor, you can call `super(parentConstructorParams[])` within the subclass constructor, passing along any necessary parameters.
2. **To refer to methods in the superclass.** Within normal method definitions, derived classes can refer to methods on the parent class with dot notation: `super.methodName`.

Our `FatFreeFood` demonstrates both use cases:

1. In the constructor, we simply call `super`, passing along `0` as our quantity of fat.
2. In our `print` method, we first call `super.print`, and add additional logic after.

Believe it or not, that wraps up the basic syntactical overview of `class`; this is all you need to start experimenting.

Prototypes: A Deep Dive

It's time we turn our attention to how `class` maps to JavaScript's underlying prototype mechanisms. We'll look at:

- Creating objects with constructor calls;
- The nature of prototype linkages;
- Property & method delegation; and
- Emulating classes with prototypes

Creating Objects with Constructor Calls

Constructors are nothing new. Calling *any* function with the `new` keyword causes it to return an object -- this is called making a *constructor call*, and such functions are generally called *constructors*:

```
"use strict";

function Food (name, protein, carbs, fat) {
    this.name      = name;
    this.protein   = protein;
    this.carbs     = carbs;
    this.fat        = fat;
}

// Calling Food with 'new' is a "constructor call", and results in its returning an
// object
const chicken_breast = new Food('Chicken Breast', 26, 0, 3.5);
console.log(chicken_breast.protein) // 26

// Failing to call Food with 'new' results in its returning 'undefined'
const fish = Food('Halibut', 26, 0, 2);
console.log(fish); // 'undefined'
```

When you call a function with `new`, four things happen under the hood:

1. A new object gets created (let's call it **O**);
2. **O** gets linked to another object, called its *prototype*;
3. The function's `this` value is set to refer to **O**;
4. The function implicitly returns **O**.

It's between steps three and four that the engine executes your function's specific logic.

Knowing this, we can rewrite our `Food` function to work *without* the `new` keyword:

```
"use strict";

// Eliminating the need for 'new' -- just for demonstration
function Food (name, protein, carbs, fat) {
    // Step One: Create a new Object
    const obj = { };

    // Step Two: Link prototypes -- we'll cover this in greater detail shortly
    Object.setPrototypeOf(obj, Food.prototype);

    // Step Three: Set 'this' to point to our new Object
    // Since we can't reset `this` inside of a running execution context,
    // we simulate Step Three by using 'obj' instead of 'this'
    obj.name      = name;
    obj.protein   = protein;
    obj.carbs     = carbs;
    obj.fat        = fat;

    // Step Four: Return the newly created object
    return obj;
}

const fish = Food('Halibut', 26, 0, 2);
console.log(fish.protein); // 26
```

Three of these four steps are straightforward. Creating an object, assigning properties, and writing a `return` statement are unlikely to give most developers any conceptual trouble: It's the prototype weirdness that trips people up.

Grokking the Prototype Chain

Under normal circumstances, all objects in JavaScript -- including Functions -- are linked to another object, called its *prototype*.

If you request a property on an object that the object doesn't have, JavaScript checks the object's prototype for that property. In other words, if you ask for a property on an object that the object doesn't have, it says: "I don't

know. Ask my prototype."

This process -- referring lookups for nonexistent properties to another object -- is called *delegation*.

```
"use strict";

// joe has no toString property . . .
const joe = { name : 'Joe' },
      sara = { name : 'Sara' };

Object.hasOwnProperty(joe, toString); // false
Object.hasOwnProperty(sara, toString); // false

// . . . But we can call it anyway!
joe.toString(); // '[object Object]', instead of ReferenceError!
sara.toString(); // '[object Object]', instead of ReferenceError!
```

The output from our `toString` calls is utterly useless, but note that this snippet doesn't raise a single `ReferenceError`! That's because, while neither `joe` or `sara` has a `toString` property, *their prototype does*.

When we look for `sara.toString()`, `sara` says, "I don't have a `toString` property. Ask my prototype." JavaScript, obligingly, does as told, and asks `Object.prototype` if it has a `toString` property. Since it does, it hands `Object.prototype`'s `toString` back to our program, which executes it.

It doesn't matter that `sara` didn't have the property herself -- *we just delegated the lookup to the prototype*.

In other words, we can access non-existent properties on an object *as long as that object's prototype does have those properties*. We can take advantage of this by assigning properties and methods to an object's prototype, so that we can use them as if they existed on the object itself.

Even better, if several objects share the same prototype -- as is the case with `joe` and `sara` above -- they can *all* access that prototype's properties, immediately after we assign them, *without* our having to copy those properties or methods to each individual object.

This is what people generally refer to as *prototypical/prototypal inheritance* -- if my object doesn't have it, but my object's prototype does, my object *inherits* the property.

In reality, there's no "inheritance" going on, here. In class-oriented languages, inheritance implies behavior is *copied* from a parent to a child. In JavaScript, no such copying takes place -- which is, in fact, one of the major benefits of prototypes over classes.

Here's a quick recap before we see precisely where these prototypes come from:

- `joe` and `sara` do *not* "inherit" a `toString` property;
- `joe` and `sara`, as a matter of fact, do *not* "inherit" from `Object.prototype at all`;
- `joe` and `sara` are **linked** to `Object.prototype`;
- Both `joe` and `sara` are linked to the **same** `Object.prototype`.

- To find the prototype of an object -- let's call it **O** -- you use: `Object.getPrototypeOf(O)`.

And, just to hammer it home: Objects do not "inherit from" their prototypes. They *delegate* to them.

Period.

Let's dig deeper.

Setting an Object's Prototype

We learned above that (almost) every object (**O**) has a *prototype* (**P**), and that, when you look for a property on **O** that **O** doesn't have, the JavaScript engine will look for that property on **P** instead.

From here, the questions are:

1. How do *functions* play into all of this?
2. Where do these prototypes come from, anyway?

A Function Named Object

Before the JavaScript engine executes a program, it builds an environment to run it in, in which it creates a function, called `Object`, and an associated object, called `Object.prototype`.

In other words, `Object` and `Object.prototype` *always* exist, in *any* executing JavaScript program.

The *function*, `Object`, is like any other function. In particular, it's a *constructor* -- calling it returns a new object:

```
"use strict";  
  
typeof new Object(); // "object"  
typeof Object(); // A peculiarity of the Object function is that it does /not/  
// need to be called with new.
```

The *object*, `Object.prototype`, is . . . Well, an object. And, like many objects, it has properties.

```

< undefined
> Object.prototype
< ▼ Object {} ⓘ
  ► __defineGetter__: function __defineGetter__()
  ► __defineSetter__: function __defineSetter__()
  ► __lookupGetter__: function __lookupGetter__()
  ► __lookupSetter__: function __lookupSetter__()
  ► constructor: function Object()
  ► hasOwnProperty: function hasOwnProperty()
  ► isPrototypeOf: function isPrototypeOf()
  ► propertyIsEnumerable: function propertyIsEnumerable()
  ► toLocaleString: function toLocaleString()
  ► toString: function toString()
  ► valueOf: function valueOf()
  ► get __proto__: function get __proto__()
  ► set __proto__: function set __proto__()

```



Here's what you need to know about `Object` and `Object.prototype`:

1. The `function`, `Object`, has a property, called `.prototype`, which points to an object (`Object.prototype`);
2. The `object`, `Object.prototype`, has a property, called `.constructor`, which points to a function (`Object`).

As it turns out, this general scheme is true for *all* functions in JavaScript. When you create a function -- `someFunction` -- it will have a property, `.prototype`, that points to an object, called `someFunction.prototype`.

Conversely, that object -- `someFunction.prototype` -- will have a property, called `.constructor`, which points back to the function `someFunction`.

```

"use strict";

function foo () { console.log('Foo!'); }

console.log(foo.prototype); // Points to an object called 'foo'
console.log(foo.prototype.constructor); // Points to the function, 'foo'

foo.prototype.constructor(); // Prints 'Foo!' -- just proving that
'foo.prototype.constructor' does, in fact, point to our original function

```

The major points to keep in mind are these:

1. All functions have a property, called `.prototype`, which points to an object associated with that function.
2. All function prototypes have a property, called `.constructor`, which points back to the function.

3. A function prototype's `.constructor` does not necessarily point to the function that created the function prototype . . . Confusingly enough. We'll touch on this in greater detail soon.

These are the rules for setting a *function*'s prototype. With that out of the way, we can cover three rules for setting an object's prototype:

1. The "default" rule;
2. Setting the prototype implicitly, with `new`;
3. Setting the prototype explicitly, with `Object.create`.

The Default Rule

Consider this snippet:

```
"use strict";  
  
const foo = { status : 'foobar' };
```

Refreshingly simple. All we've done is create an object, called `foo`, and give it a property, called `status`.

Behind the scenes, however, JavaScript does a little extra work. When we create an object literal, JavaScript sets the object's prototype reference to `Object.prototype`, and sets its `.constructor` reference to `Object`:

```
"use strict";  
  
const foo = { status : 'foobar' };  
  
Object.getPrototypeOf(foo) === Object.prototype; // true  
foo.constructor === Object; // true
```

Setting the Prototype Implicitly with `new`

Let's take another look at our modified `Food` example.

```
"use strict";  
  
function Food (name, protein, carbs, fat) {  
    this.name      = name;  
    this.protein   = protein;  
    this.carbs     = carbs;  
    this.fat       = fat;  
}
```

By now, we know that the *function* `Food` will be associated with an *object*, called `Food.prototype`.

When we create an object using the `new` keyword, JavaScript:

1. Sets the object's prototype reference to the `.prototype` property of the function you called with `new`; and
2. Sets the object's `.constructor` reference to the function you called `new` with.

```
const tootsie_roll = new Food('Tootsie Roll', 0, 26, 0);

Object.getPrototypeOf(tootsie_roll) === Food.prototype; // true
tootsie_roll.constructor === Food; // true
```

This is what lets us do slick stuff like this:

```
"use strict";

Food.prototype.cook = function cook () {
  console.log(`#${this.name} is cooking!`);
};

const dinner = new Food('Lamb Chops', 52, 8, 32);
dinner.cook(); // 'Lamb Chops are cooking!'
```

Setting the Prototype Explicitly with `Object.create`

Finally, we can set an object's prototype reference *manually*, using a utility called `Object.create`.

```
"use strict";

const foo = {
  speak () {
    console.log('Foo!');
  }
};

const bar = Object.create(foo);

bar.speak(); // 'Foo!'
Object.getPrototypeOf(bar) === foo; // true
```

Remember the four things that JavaScript does under the hood when you call a function with `new`?

`Object.create` does all but the third step:

1. Create a new object;
2. Set its prototype reference; and

3. Return the new object.

You can see this yourself if you take a look at the polyfill.

Emulating class Behavior

Using prototypes directly, emulating class-oriented behavior required a bit of manual acrobatics.

```
"use strict";

function Food (name, protein, carbs, fat) {
    this.name      = name;
    this.protein   = protein;
    this.carbs     = carbs;
    this.fat       = fat;
}

Food.prototype.toString = function () {
    return `${this.name} | ${this.protein}g P :: ${this.carbs}g C :: ${this.fat}g F`;
};

function FatFreeFood (name, protein, carbs) {
    Food.call(this, name, protein, carbs, 0);
}

// Setting up "subclass" relationships
// =====
// LINE A :: Using Object.create to manually set FatFreeFood's "parent".
FatFreeFood.prototype = Object.create(Food.prototype);

// LINE B :: Manually (re)setting constructor reference (!)
Object.defineProperty(FatFreeFood.constructor, "constructor", {
    enumerable : false,
    writeable   : true,
    value       : FatFreeFood
});
```

At Line A, we have to set `FatFreeFood.prototype` equal to a new object, whose prototype reference is to `Food.prototype`. If we fail to do this, our "child classes" won't have access to "superclass" methods.

Unfortunately, this results in the rather bizarre behavior that `FatFreeFood.constructor` is [Function](#)... Not `FatFreeFood`. So, to keep everything sane, we have to manually set `FatFreeFood.constructor` by hand at Line B.

Sparing developers from the noise and unwieldiness of emulating class behavior with prototypes is one of the motives for the `class` keyword. It *does* provide a solution to the most common gotchas of prototype syntax.

Now that we've seen so much of JavaScript's prototype mechanics, it should be easier to appreciate just *how* much easier it can make things!

A Closer Look at Methods

Now that we've seen the essentials of JavaScript's prototype system, we'll wrap up by taking a closer look at three kinds of methods classes support, and a special case of the last sort:

- Constructors;
- Static methods;
- Prototype methods; and
- "Symbol methods", a special case of *prototype methods*

I didn't come up with these groups -- credit goes to Dr Rauschmayer for identifying them in [Exploring ES6](#).

Class Constructors

A class's `constructor` function is where you'll focus your initialization logic. The `constructor` is special in a few ways:

1. It's the only method of a class from which you can make a superconstructor call;
2. It handles all the dirty work of setting up the prototype chain properly; and
3. It acts as the definition of the class.

Point 2 is one of the principle benefits to using `class` in JavaScript. To quote heading 15.2.3.1 of Exploring ES6:

The prototype of a subclass is the superclass.

As we've seen, setting this up manually is tedious and error-prone. That the language takes care of it all behind the scenes if we use `class` is a major boon.

Point 3 is interesting. In JavaScript, a class is just a function -- it's equivalent to the `constructor` method in the class.

```
"use strict";

class Food {
    // Class definition is the same as before . . .
}

typeof Food; // 'function'
```

Unlike normal-functions-as-constructors, you can't call a class's constructor without the `new` keyword:

```
const burrito = Food('Heaven', 100, 100, 25); // TypeError
```

. . . Which raises another question: What happens when we call a function-as-constructor *without* new?

The short answer: It returns `undefined`, as does any function without an explicit return. You just have to trust your users will constructor-call your function. This is why the community has adopted the convention of only capitalizing constructor names: It's a reminder to call with `new`.

```
"use strict";

function Food (name, protein, carbs, fat) {
    this.name      = name;
    this.protein   = protein;
    this.carbs     = carbs;
    this.fat       = fat;
}

const fish = Food('Halibut', 26, 0, 2); // D'oh . . .
console.log(fish); // 'undefined'
```

The long answer: It returns `undefined`, *unless* you manually detect that it wasn't called with `new`, and then do something about it yourself.

ES2015 introduces a property that makes this check trivial: `[new.target]` (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target>).

`new.target` is a property defined on all functions called with `new`, including class constructors. When you call a function with the `new` keyword, the value of `new.target` within the function body is the function itself. If the function wasn't called with `new`, its value is `undefined`.

```
"use strict";

// Enforcing constructor call
function Food (name, protein, carbs, fat) {
    // Manually call 'new' if user forgets
    if (!new.target)
        return new Food(name, protein, carbs, fat);

    this.name      = name;
    this.protein   = protein;
    this.carbs     = carbs;
    this.fat       = fat;
}

const fish = Food('Halibut', 26, 0, 2); // Oops -- but, no problem!
fish; // 'Food {name: "Halibut", protein: 20, carbs: 5, fat: 0}'
```

This wasn't any worse in ES5:

```
"use strict";

function Food (name, protein, carbs, fat) {

    if (!(this instanceof Food))
        return new Food(name, protein, carbs, fat);

    this.name      = name;
    this.protein   = protein;
    this.carbs     = carbs;
    this.fat       = fat;
}
```

The [MDN documentation](#) has more details on `new.target`, and the [spec is the definitive reference](#) for the truly curious. The descriptions of `[[Construct]]` are particularly illuminating.

Static Methods

Static methods are methods on the constructor function itself, which are *not* available on instances of the class. You define them by using the `static` keyword.

```
"use strict";

class Food {
    // Class definition is the same as before . . .

    // Adding a static method
    static describe () {
        console.log('"Food" is a data type for storing macronutrient information.');
    }
}

Food.describe(); // '"Food" is a data type for storing macronutrient information.'
```

Static methods are analogous to attaching properties directly to old-school functions-as-constructors:

```
"use strict";

function Food (name, protein, carbs, fat) {
    Food.count += 1;
```

```

        this.name      = name;
        this.protein   = protein;
        this.carbs     = carbs;
        this.fat       = fat;
    }

Food.count = 0;
Food.describe = function count () {
    console.log(`You've created ${Food.count} food(s).`);
};

const dummy = new Food();
Food.describe(); // "You've created 1 food."

```

Prototype Methods

Any method that isn't a constructor or a static method is *prototype method*. The name comes from the fact that we used to achieve this functionality by attaching functions to the `.prototype` of functions-as-constructors:

```

"use strict";

// Using ES6:
class Food {

    constructor (name, protein, carbs, fat) {
        this.name = name;
        this.protein = protein;
        this.carbs = carbs;
        this.fat = fat;
    }

    toString () {
        return `${this.name} | ${this.protein}g P :: ${this.carbs}g C :: ${this.fat}g F`;
    }

    print () {
        console.log( this.toString() );
    }
}

// In ES5:
function Food (name, protein, carbs, fat) {
    this.name = name;
}

```

```

    this.protein = protein;
    this.carbs = carbs;
    this.fat = fat;
}

// The name "prototype methods" presumably comes from the fact that we
// used to attach such methods to the '.prototype' old-school functions-as-
// constructors.

Food.prototype.toString = function toString () {
    return `${this.name} | ${this.protein}g P :: ${this.carbs}g C :: ${this.fat}g F`;
};

Food.prototype.print = function print () {
    console.log( this.toString() );
};

```

To be clear, it's perfectly fine to use generators in method definitions, as well:

```

"use strict";

class Range {

    constructor(from, to) {
        this.from = from;
        this.to = to;
    }

    * generate () {
        let counter = this.from,
            to      = this.to;

        while (counter < to) {
            if (counter == to)
                return counter++;
            else
                yield counter++;
        }
    }
}

const range = new Range(0, 3);
const gen = range.generate();
for (let val of range.generate()) {

```

```

console.log(`Generator value is: ${ val }. `);
// Prints:
//   Generator value is: 0.
//   Generator value is: 1.
//   Generator value is: 2.
}

```

Symbol Methods

Finally, there are the *symbol methods*. These are functions whose names are `Symbol` values, and which the JavaScript engine recognizes and uses when you use certain built-in constructs with your custom objects.

The [MDN docs](#) provide a succinct overview of what Symbols are in general:

A symbol is a unique and immutable data type and may be used as an identifier for object properties.

Creating a new symbol provides you with a value that is guaranteed to be unique within your program. This is what makes it useful for naming object properties: You're guaranteed never to accidentally shadow anything. Symbol-valued keys also aren't innumerable, so they're largely invisible to the outside world ([but not completely](#)).

```

"use strict";

const secureObject = {
    // This key is guaranteed to be unique.
    [new Symbol("name")] : 'Dr. Secure A. F.'
};

console.log( Object.getKeys(superSecureObject) ); // [] -- The symbol property is pretty
hard to get at . . .
console.log( Reflect.ownKeys(secureObject) ); // [Symbol("name")] -- . . . But, not
/truly/ hidden.

```

More interestingly for us, they give us a way to tell the JavaScript engine to use certain functions for special purposes.

The so-called [Well-known Symbols](#) are special object keys that identify functions the JavaScript engine invokes when you use certain built-in constructs with custom objects.

This is a bit exotic for JavaScript, so let's see an example:

```

"use strict";

// Extending Array lets us use 'length' in an intuitive way,
// and also gives us access to built-in array methods, like
// map, filter, reduce, push, pop, etc.

```

```
class FoodSet extends Array {

    // ...foods collects arbitrary number of arguments into an array
    // https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator
    constructor(...foods) {
        super();
        this.foods = [];
        foods.forEach((food) => this.foods.push(food))
    }

    // Custom iterator behavior. This isn't very *useful* iterator behavior, mind you,
    but it's a fine example.

    // The asterisk *must* precede the name of the key.
    * [Symbol.iterator] () {
        let position = 0;
        while (position < this.foods.length) {
            if (position === this.foods.length) {
                return "Done!"
            } else {
                yield `${this.foods[ position++ ]} is the food item at position
${position}`;
            }
        }
    }

    // Return an object of type Array, rather than type FoodSet, when users
    // use built-in array methods. This makes our FoodSet interoperable
    // with code expecting an array.

    static get [Symbol.species] () {
        return Array;
    }
}

const foodset = new FoodSet(new Food('Fish', 26, 0, 16), new Food('Hamburger', 26, 48, 24));

// When you use for . . . of with a FoodSet, JavaScript will iterate using the function
you
// assoiated with the key [Symbol.iterator].
for (let food of foodset) {
    // Prints all of our foods
    console.log( food );
}
```

```
// JavaScript creates and returns a new object when you `filter` on an array,  
// which it creates using the default constructor of the object you execute `filter`  
on.  
  
//  
// Since most code would expect filter to return an Array, we can tell JavaScript  
// to use the Array constructor when implicitly creating a new instance by  
// overriding [Symbol.species].  
const healthy_foods = foodset.filter((food) => food.name !== 'Hamburger');  
  
console.log( healthy_foods instanceof FoodSet ); //  
console.log( healthy_foods instanceof Array );
```

When you use a `for...of` loop on an object, JavaScript will try to execute the object's `iterator` function, which is the function associated with the key, `Symbol.iterator`. If you provide your own definition, JavaScript will use that. If you don't, it'll use the default implementation, if there is one; or do nothing, if there isn't.

`Symbol.species` is a bit more exotic. In a custom class, the default `Symbol.species` function is the constructor for your class. When you subclass built-in collections, like `Array` or `Set`, however, you'll often want to be able to use your subclass wherever you could use an instance of the parent class.

Returning instances of the parent class from methods on the class *instead* of instances of the derived class gets us closer to ensuring full interoperability of a subclass with more general code. This is `Symbol.species` allows.

Don't sweat it if this bit doesn't make much sense. Using symbols this way -- or at all -- is still a niche case, and the point of these examples is to demonstrate:

1. That you *can* use certain built-in JavaScript constructs with custom classes; and
2. How you achieve that, in two common cases.

Conclusion

ES2015's `class` keyword does *not* bring us "true classes", à la Java or SmallTalk. Rather, it simply provides a more convenient syntax for creating objects related via prototype linkage. Under the hood, there's nothing new here.

I covered enough of JavaScript's prototype mechanism for our discussion, but there's quite a bit more to say. Read Kyle Simpson's [this & Object Prototypes](#) for fuller coverage on that front. [Appendix A](#) is particularly relevant.

For the nitty-gritty on ES2015 classes, Dr Rauschmayer's [Exploring ES6: Classes](#) should be your go-to resource. It was the inspiration for much of what I've written here.

Finally, if you've got questions, drop a line in the comments, or [hit me on Twitter](#). I'll do my best to get back to everyone directly.

What do you think about `class`? Love it, hate it, no strong feelings? It seems like everyone's got an opinion -- let us know yours below!

64 : Recursion in JavaScript with ES6, destructuring and rest/spread

The latest ECMA standard for JavaScript (ECMAScript 6) makes JavaScript more readable by encouraging a more declarative style with functional constructs and new operators.

Destructuring

One of my favourite ES6 features is **destructuring**. It allows you to extract data from one variable to another by using **structure**. For arrays this means for example:

```
var [ first, second ] = [ 1, 2, 3, 4 ];  
// first: 1  
// second: 2
```

There's more you can do, like skip some members of the array on the right-hand side of the operation.

```
var [ first, , third, fourth ] = [ 1, 2, 3, 4 ];  
// first: 1  
// third: 3  
// fourth: 4
```

This is actually quite easily back-ported to the equivalent ES5

```
var arr = [ 1, 2, 3, 4 ];  
var first = arr[0];  
var second = arr[1];  
// etc ...
```

Rest

This is where ES6 features become more interesting. With destructuring we can also assign what is called the **rest** of the array. We indicate **rest** with the `...` notation.

```
var [ first, ...notFirst ] = [ 1, 2, 3, 4 ];  
// first: 1  
// notFirst: [ 2, 3, 4 ]
```

Naming conventions lead to code that is more akin to the following:

```
var [ first, second, ...rest ] = [ 1, 2, 3, 4 ]
// first: 1
// second: 2
// rest: [ 3, 4 ]
```

The `rest` operator has some interesting properties:

```
var [ first, ...rest ] = [ 1 ]
// first: 1
// rest: []
```

It always returns an array. Which means even in defensive JavaScript land, it's ok to do things like check `.length` of `rest` without guards.

The equivalent in ES5 (and below) is to use the `Array.slice` function.

```
var arr = [ 1, 2, 3, 4 ];
var first = arr[0];
var rest = arr.slice(1);
// first: 1
// rest: [ 2, 3, 4 ]
```

Two things to note here:

- the ES5 version is more verbose
- the ES5 version is more imperative, we tell JavaScript **how** to do something instead of telling it **what** we want.

Now I also think that the structure-matching version (with `rest`) is more readable.

Parameter destructuring

We can use destructuring on the parameters of a function definition:

```
function something([ first, ...rest ]) {
  return {
    first: first,
    rest: rest
  };
}
var result = something([1, 2, 3]);
// result: { first: 1, rest: [ 2, 3 ] }
```

Equivalent ES5:

```
function something(arr){
  var first = arr[0];
  var rest = arr.slice(1);
  return {
    first: first,
    rest: rest
  };
}
```

Again it's more verbose and more imperative.

Spread

Spread uses the same notation as rest: `....`. What it does is quite different.

```
var arr = [ 1, 2, 3 ];
var newArr = [ ...arr ];
// newArr: [ 1, 2, 3 ]
```

ES5 equivalent:

```
var arr = [ 1, 2, 3 ];
var newArr = [].concat(arr);
```

Things to note, the contents of the array are **copied**. So `newArr` is not a reference to `arr`.

We can also do things like appending or prepending an array.

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

var withPrepend = [ ...arr, 3, 2, 1];
var withAppend = [ 3, 2, 1, ...arr ];
// withPrepend: [ 1, 2, 3, 3, 2, 1]
// withAppend: [ 3, 2, 1, 1, 2, 3 ]
```

Functional Programming: lists & recursion

In functional programming when we run functions recursively over lists we like to model the list as a **head** and a **tail**.

The head is the first element of the list, the tail is the list composed of the list minus the head.

```
arr = [ 1, 2, 3 ]
// head(arr): 1
// tail(arr): [ 2, 3 ]
```

In ES6 we can do this just by naming the variable appropriately with **destructuring** and **rest**:

```
var [ head, ...tail ] = [ 1, 2, 3 ];
// head: 1
// tail: [ 2, 3 ]
```

We can also trivially implement the `head` and `tail` functions using ES6:

```
function head([ head, ...tail ]) {
  return head;
}

function tail([ head, ...tail ]) {
  return tail;
}

// or with arrow function syntax
var head = ([ head, ...tail ]) => head;
var tail = ([ head, ...tail ]) => tail;
```

(Tail) Recursion

We can implement functions that operate over arrays (or lists as they tend to be called in functional programming) using **parameter destructuring** and **recursion**.

For example, `map` can be implemented in the following manner:

Map is a function that takes a list and a function and returns a list containing the result of a function application to each element of the list.

```
function map([ head, ...tail ], fn) {
  if(head === undefined && !tail.length) return [];
  if(tail.length === 0){
    return [ fn(head) ];
  }
  return [ fn(head) ].concat(map(tail, fn));
}
```

The `tail.length === 0` checks whether there is still a tail to recurse over. Otherwise, the recursion stops there.

This is not necessarily the most efficient version of map both in terms of memory usage and speed but it's a good illustration of ES6.

We can further simplify it by replacing `concat` with the `spread` operator and using a single return statement with a ternary operator.

Very ES6 map

Our ES6 recursive/destructuring map can be simplified to:

```
function map([ head, ...tail ], fn) {
  if (head === undefined && !tail.length) return [];
  return tail.length ? [ fn(head), ...(map(tail, fn)) ] : [ fn(head) ];
}
```

Or if we want to abuse ES6 and allow ourselves to forget that we're actually doing JavaScript:

```
var map = ([ head, ... tail ], fn) =>
  ( (head !== undefined && tail.length) ? ( tail.length ? [ fn(head), ...(map(tail, fn))
] : [ fn(head) ] ) : [] );
```

ES5 equivalent

```
function map(arr, fn) {
  var head = arr[0];
  var tail = arr.slice(1);
  if(head === undefined && tail.length === 0) return [];
  if(tail.length === 0) {
    return [ fn(head) ];
  }
  return [].concat(fn(head), map(tail, fn));
}
```

All the features add up and while recursive map in ES6 is essentially a one-liner, in ES5 it's a clunky, long, hard to read function.

Reimplementing list manipulation functions

Now you can have a go at reimplementing `filter`, `reduce` and `join` using the above techniques.

Solutions below the fold :).

ES6 allows us to write code in a functional style more tersely and effectively.

*Give some  to this post if you liked it and to **follow** for more JavaScript-related things.*

You can find me on Twitter ([@hugo_df](#)), on GitHub as [HugoDF](#), there are more JavaScript gists [here](#).

Recursive list operations in ES6 with rest/spread and destructuring

```
function filter([ head, ...tail ], fn) {
  const newHead = fn(head) ? [ head ] : [];
  return tail.length ? [ ...newHead, ...(filter(tail, fn)) ] : newHead;
}
```

```
function reduce([ head, ...tail ], fn, initial) {
  if(head === undefined && tail.length === 0) return initial;
  if(!initial) {
    const [ newHead, ...newTail] = tail;
    return reduce(newTail, fn, fn(head, newHead));
  }
  return tail.length ? reduce(tail, fn, fn(initial, head)) : [ fn(initial, head) ];
}
```

```
function join([ head, ...tail ], separator = ',') {
  if (head === undefined && !tail.length) return '';
  return tail.length ? head + separator + join(tail, separator) : head;
}
```

65 : A Gentle Introduction to Composition in JavaScript

Why Functional Programming?

I've been shifting my programmer's mindset to use more **Functional Programming** in JavaScript, initially as a way to exercise my brain and to revamp my way of writing code. As it happens, ES2015 gives us a nice tool belt to work with in a functional manner, so there's really no excuse not to make use of it. What initially started as plain exercises quickly became my default way of thinking in terms of handling data now.

But there's a lot to functional programming in JavaScript, more than simply working with methods like `map` and `reduce` and all of the new ones ES2015 is introducing. It's not only "avoiding loops", and it can still get quite verbose when we look at things like currying, for example, so let's start simple.

This is my attempt at starting from scratch teaching the basics of FP in JavaScript. I thought I'd start from the very basics: making use of a functional mindset and using the basics of composition to perform some simple object queries and manipulations.

If you want to play directly with the source code, [you can find it on JSBin](#).

Working with Data

There's mainly one required concept to know about, before jumping into a more functional mindset. It's important to remember what **pure functions** are and always keep their concept in mind when manipulating data. So what is a pure function?

From this [Sitepoint article](#), we get a nice definition for a pure function:

A pure function is a function where the return value is only determined by its input values, without observable side effects. This is how functions in math work: `Math.cos(x)` will, for the same value of `x` , always return the same result.

Nicolas Carlo wrote a [fantastic article](#) about pure functions in JavaScript that you should definitely have a look at if you want to know more about them.

Getting Practical

Let's say we have an object which contains some monthly expenses. They have a name, a price and their category. We'd later like to query this object for data analysis.

```
const expenses = [
  {
    name: 'Rent',
    price: 500,
    type: 'Household'
  },
  {
    name: 'Netflix',
    price: 100,
    type: 'Entertainment'
  }
]
```

```

    price: 5.99,
    type: 'Services'
  }, {
    name: 'Gym',
    price: 15,
    type: 'Health'
  }, {
    name: 'Bills',
    price: 100,
    type: 'Household'
  }
];

```

Exercise one: Sum up all the expenses

If you were told to sum all of the expenses in this object, how would you go about and do it? In traditional JS fashion, we would write a simple loop:

```

var total = 0;

for (var i = 0; i < expenses.length; i++) {
  total += expenses[i].price;
}

console.log(total); // 620.99

```

Great, it works. But it's not exactly... functional. Firstly, it's **dependent on an external variable**: `total`. This variable might be overwritten somewhere else in the code, and it's impossible to keep track of its state. Plus, if we need to calculate the sum of all the expenses again for any other incoming data, we'll need to write another loop, since the original Array is hardcoded inside it.

So zero points for reusability, predictability and encapsulation for this method.

We could turn this into its own function, and make it a pure one. Pure functions depend on their inputs alone, and their output should be the exact same for the same input, but it would still lack the functional approach to it.

A more functional way

Let's make use of `map` and `reduce` to create a generic function that calculates the total costs. Keep in mind that we would like to reuse this method with possible other objects; so in order to keep function purity in check, it will accept an argument (here defined as `arr`) to work on.

```

const sumAll = (arr) => arr
  .map((item) => item.price)

```

```
.reduce((acc, price) => acc + price, 0);
```

What's happening here?

Break it down!

We're creating a method that accepts an Array `arr`. For this Array, we'll `map` over every element (as `item`) and *only return* its price property. For this `price` property, we then use the `reduce` method to continuously increment `acc` (the accumulator, with an initial value of zero) with the price returned.

And we use it like so:

```
let total = sumAll(expenses); // 620.99
```

Now this is much better. If there were more Expenses objects, we could simply throw them at this function, like bottles on a wall. Notice that there are no side effects here; no dependencies on external state either. `sumAll` takes one argument and acts upon it, purity is preserved.

Composing things

So we want to be able to do some data analysis. How much am I spending on the 'Household' category, for example? It would be nice to easily retrieve all items in this category and quickly sum them up.

Given what we've learned already, let's do this without resorting to loops and creating side effects. For now, let's explicitly create a function that gives us all household expenses:

```
var getHousehold = (src) => {
  return src.filter((item) => {
    return item.type === 'Household';
  });
};
```

Written in (semi) ES5 for a better understanding: here, we're accepting our expenses as `src` and using `filter`, which takes a function, to let through only the items that match the type as being `Household`.

But because we like ES2015 so much, let's write the exact same function in a more compact flavour:

```
const getHousehold = (src) =>
  src.filter((item) => item.type === 'Household');
```

So, calling this method on our expenses:

```
getHousehold(expenses); // object with only items with the type of 'Household'
```

Great! We're getting somewhere.

Functional + Functional

Now that we have a method to grab all household expenses and another one that sums all prices of a given source, we can easily pair these two together. If we wanted to log out the sum of household expenses, we could simply now use;

```
let householdExpenses = sumAll(getHousehold(expenses)); // 600
```

This is great already! `sumAll` is receiving a filtered down version of the original expenses, provided by calling `getHousehold` on the `expenses` Array.

But we can do better with **composition**.

Composition, you say?

Composition, according to Wikipedia:

object composition (not to be confused with function composition) is a way to combine simple objects or data types into more complex ones.

In this case, we want to combine two methods to form a composite that we can refer to directly. But let's examine what's happening under the hood ourselves, first.

Here's an example of composition:

```
var angryAndMad = compose(getMad, getAngry);
var johnIsAngryAndMad = angryAndMad('John');
var aliceIsAngryAndMad = angryAndMad('Alice');
```

With composition, we're thinking in definitions. Breaking everything down to atoms. We define our concept of `angryAndMad`, which is **composed** of a combination of `getMad` and `getAngry`. Then, we apply the concept to two entities, and unfortunately John and Alice are our chosen victims here. We're **applying** John and Alice to the composer of angry and mad.

Back to our expenses application.

Noticed that `compose` method written? Let's assume that magical method for now, and use it to compose ourselves a method that does two things: gets our household expenses and sums up all of them.

Something in the means of:

```
const sumOfHousehold = compose(getHousehold, sumAll);
```

And then applying our `expenses` source to it, like so;

```
let householdPrices = sumOfHousehold(expenses);
```

Notice that if any particular stage we had more sources of data to work with, we would be on the clear.

`sumOfHousehold` is not dependent on anything external, apart from the internal methods that it combines; we could easily apply it to a potential `expenses2` and `expenses3` Arrays if this kind of complexity raised.

How does compose work?

In this case, the `compose` method accepts two arguments, both functions, and returns another function. This new function will return another function that contains the result of both arguments applied. Sounds confusing? Let's look at a possible implementation of it, first in ES5 form:

```
var compose = function(f, g) {
  return function(x) {
    return g(f(x));
  }
};
```

It returns the value of `g` applied on `f`, as a function. In ES2015 form, it looks much more concise:

```
const _compose = (f,g) => (x) => g(f(x));
```

If you're having trouble understanding what `compose` is actually doing, try and log out our composite function without calling it with arguments:

```
const sumOfHousehold = compose(getHousehold, sumAll);
console.log(sumHouseholdExpenses);
```

It returns:

```
function (x) {
  return g(f(x));
}
```

And of course it does! That `x` argument is what's ready to receive our `expenses` source, or any other source for that matter. `g` and `f` are internally stored ready to act upon this argument.

Another example

At this point, our code isn't very useful. Let's keep writing a couple more methods to allow us to work with the data we have. For our purposes, we want to grab a list of all possible categories in our list of expenses. Using traditional functional programming in JavaScript, this is easy:

```
const getCategories = (arr) => arr
  .map((item) => item.type);
```

Grab our source, and for each one return their type. This gives us:

```
["Household", "Services", "Health", "Household"]
```

Ops. It works, but we're getting "Household" twice, because there are two household expenses. We want to filter duplicates out. Let's write a (very naive) function that does this!

```
const uniqueElements = (arr) => arr.filter((item, pos) => arr.indexOf(item) == pos);
```

So now we can easily compose a function that displays all of our unique categories:

```
const uniqueCategories = compose(getCategories, uniqueElements);
uniqueCategories(expenses); // ["Household", "Services", "Health"]
```

It works! And just like this, we're now slowly building a solid base with a lot of helper methods, all with a decent level of abstraction (although not point-free, that's a topic for another post) and without any side-effects to their sources.

Where to go next?

Be on the lookout for the next series on Functional Programming, and in the meantime feel free to check out the following great resources, which I found to be the most useful so far:

- [Learning FP using Rx](#)
- [FP for JavaScript People](#)
- [The 2 Pillars of JS: Functional Programming](#)

Once again, you can find the source code for this tutorial [on JSBin](#)

That's it for this lesson, I hope you've enjoyed it, and be on the lookout for part two. If you spot any typos or have any suggestions, don't be afraid to ping me on [Twitter](#).

66 : Enhancing Mixins with Decorator Functions

<http://justinfagnani.com/2016/01/07/enhancing-mixins-with-decorator-functions/>

Last time, in "["Real" Mixins with JavaScript Classes](#)", we saw how we can create powerful ES6 mixins by using class expressions as subclass factories.

Now, let's look at some enhancements that make our mixins more powerful, in a way still requires no framework for mixin users, and is still easy to use for mixin authors.

First, we'll look at how subclass-factory-style mixins can be wrapped, or decorated, to add additional features. Then we'll enable the caching of mixin applications, so that the same mixin definition applied to the same superclass multiple times reuses the same mixin application, and implement ES2015's `@@hasInstance`¹ method so that `instanceof` works with mixins!

The Technique and Some Groundwork

We're going to add these enhancements at the mixin definition site, so that only the mixin author needs to do anything to get them.

Recall a mixin definition from the last post:

```
const MyMixin = (superclass) => class extends superclass {
  /* ... */
};
```

Our new enhancements are built as functions that wrap the mixin function. Mixin declaration will now look like this:

```
const MyMixin = BaseMixin((superclass) => class extends superclass {
  /* ... */
});
```

`BaseMixin` wraps the author's subclass factory in a function that does a little extra work before calling the factory. This is very much like [ES.next decorators](#), just without the special `@BaseMixin` syntax, so I'll call them decorator functions.

All kinds of behaviors can be added to mixins by adding more wrappers. Since we're going to implement caching and `@@hasInstance` support, our mixin definitions might end up looking like this:

```
const MyMixin = Cached(HasInstance(BaseMixin((superclass) => class extends superclass {
  /* ... */
})));
```

This should be fairly easy for authors to use, and mixin users don't need to do anything different. Mixin application still just works as function invocation:

```
class A extends MyMixin(MyBaseClass) {}
```

```
class B extends MyMixin(MyBaseClass) {}
```

Or we can use `mix().with()`:

```
class A extends mix(MyBaseClass).with(MyMixin) {}
```

```
class B extends mix(MyBaseClass).with(MyMixin) {}
```

`mix().with()` is still just plain and optional sugar around function invocation.

If we implement `Cached` and `HasInstance` correctly, the mixins and classes suddenly have new powers:

```
let a = new A();
let b = new B();

a instanceof A; // true
b instanceof B; // true
Object.getPrototypeOf(a) === Object.prototypeOf(b); // true
```

We can use standard function composition to create a convenience decorator for the common behaviors:

```
const Mixin = (mixin) => Cached(HasInstance(BaseMixin(mixin)));

const MyMixin = Mixin((superclass) => class extends superclass {
  /* ... */
});
```

Wrapping with Care

We have to be a little more careful with wrapping mixin functions than usual because we want the final result of calling the decorators to be an object with certain properties, like `@@hasInstance`. We don't know what decorators might be applied to our mixin, and we'd like to avoid requiring a specific ordering of decorators, so we don't want to obscure properties added by one wrapper with a subsequent wrapper.

To solve this we'll create a `wrap` utility that sets the prototype of the wrappers, and stores a property pointing to the original mixin function so that we have a canonical identity for a mixin:

```
const _originalMixin = Symbol('_originalMixin');

const wrap = (mixin, wrapper) => {
  Object.setPrototypeOf(wrapper, mixin);
```

```

if (!mixin[_originalMixin]) {
  mixin[_originalMixin] = mixin;
}
return wrapper;
};

```

This makes our function wrapping behave a lot more like inheritance.

Next, we'll create a base mixin decorator that applies the mixin and stores a reference from the mixin application back to the mixin definition for later use in caching:

```

const BaseMixin = (mixin) => wrap(mixin, (superclass) => {
  let application = mixin(superclass);
  application.prototype[_mixinRef] = mixin[_originalMixin];
  return application;
});

```

Notice the call to `wrap`. This sets things up so that other mixin decorators can add properties to the mixin and have them be visible after wrapping.

Caching Mixin Applications

The whole purpose of mixins is to apply them to different superclasses so that using a mixin doesn't require a specific inheritance hierarchy. However, mixins may still be repeatedly applied to the same superclass.

Consider two classes, *B* and *C* which both extend *A* and mixin *M1* and *M2*:

```

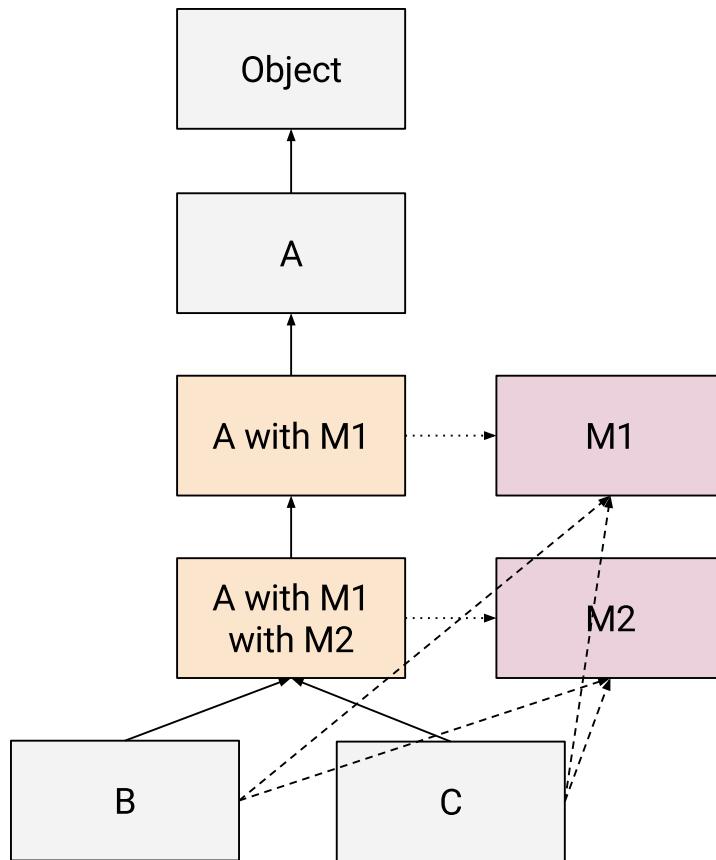
class B extends mix(A).with(M1, M2) {}

class C extends mix(A).with(M1, M2) {}

```

The simplest mixin implementation will produce two identical *A-with-M1* and two identical *A-with-M1-with-M2* prototypes.

Instead we would like to share identical applications:



We can do this by caching mixin applications on classes:

```

const _MixinRef = Symbol('_MixinRef');

const Cached = (mixin) => wrap(mixin, (superclass) => {
  // Create a symbol used to reference a cached application from a superclass
  let applicationRef = mixin[_cachedApplicationRef];
  if (!applicationRef) {
    applicationRef = mixin[_cachedApplicationRef] = Symbol(mixin.name);
  }

  // Look up an cached application of `mixin` to `superclass`
  if (superclass.hasOwnProperty(applicationRef)) {
    return superclass[applicationRef];
  }

  // Apply the mixin
  let application = mixin(superclass);

  // Cache the mixin application on the superclass
  superclass[applicationRef] = application;
}
  
```

```
    return application;
});
```

Hopefully the comments are easy to follow. The approach is to create a unique symbol per mixin, then store mixin applications in a property keyed by that symbol on the superclass of a mixin application. That allows us to look up and reuse previous applications of a mixin to a superclass.

If this seems like overkill, consider cases where you must extend a certain class, yet libraries would like to offer help in customizing those subclasses.

I'm thinking of custom elements here, and specifically, libraries like Polymer. Custom elements need to extend one of the built-in elements - at the very least *HTMLElement* - but Polymer does a lot of work by being on the prototype chain. Mixins make this easy to express:

```
class MyElement extends mix(HTMLElement).with(Polymer) {
  // ...
}
```

But with naive mixins a new identical superclass of *HTMLElement-With-Polymer* is created for every custom element class, wasting time and memory. With mixin application caching this can be easy *and* efficient.

Adding instanceof Support

ES2015 has support for [overriding the `instanceof` operator](#) via the `@@hasInstance` method, which should be implemented by objects that appear on the *righthand* side of an `instanceof` operator².

Ideally an expression like `o instanceof MyMixin` would work as expected: if `o` is an instance of a class that has mixed in `MyMixin`, the expression should return true.

With mixins, the "type" is one of our subclass factory functions, so we need to patch the `@@hasInstance` method into the mixin. The prototype setting we did in the groundwork section will ensure that this method is available even after subsequent wrapping.

Our patch looks like this:

```
const HasInstance = (mixin) => {
  if (!Symbol.hasInstance) {
    return mixin;
  }
  mixin[Symbol.hasInstance] = function(o) {
    const originalMixin = this[_originalMixin];
    while (o != null) {
      if (o.hasOwnProperty(_mixinRef) && o[_mixinRef] === originalMixin) {
        return true;
      }
      o = Object.getPrototypeOf(o);
    }
  }
};
```

```

    }
    return false;
}
return mixin;
};

```

I can't find that this is actually implemented in any production VM yet, but it looks like Webkit has initial support, along with Babel under a flag.

Writing Your Own Mixin Decorators

[mixwith.js](#) includes the `wrap` utility and some common symbols that help with writing mixin decorators. You can use these to write decorators that play well with each other. What kind of decorators might you want to write? Maybe a de-duplication, so that a mixin applied twice to a prototype chain only appears once in the chain, or a traits-like system that disallows overriding.

There are probably two main types of mixin decorators: those that need to wrap the mixin function, and those that just need to patch it.

If you need to wrap the mixin function, make sure you call `wrap` and usually you want to invoke the mixin function. Sometimes you may want to conditionally invoke the mixin, like with de-duplication:

```

import { wrap } from 'mixwith';

const DeDupe = (mixin) => wrap(mixin, (superclass) => {
  if (mixinAlreadyApplied(superclass)) {
    return superclass;
  }
  return mixin(superclass);
});

```

If you only need to patch the mixin function, just patch and return it:

```

const Fooify = (mixin) => {
  mixin.foo = "Foo";
  return mixin;
};

```

Follow the conversation

- [reddit.com/r/javascript](https://www.reddit.com/r/javascript)

Footnotes

1. @@foo is shorthand for Symbol.foo in the [JavaScript specification](#) ↪

2. This is fabulous for extensibility, since new types can accept an instance without having to modify the instance. You can implement any kind of type checking you want, say structural typing. It's terrible for static type checking, however. [←](#)

67 : Currying - the Underrated Concept in Javascript

68 : Currying - the Underrated Concept in Javascript

This is the first article of a series on functional programming in dynamically typed languages such as Javascript. This series will explore various aspects of the functional paradigm, its concepts and underlying mechanisms.

Definition and distinction of partial application

Currying is a technique related to declaration of functions. In its original mathematical sense it is the transformation of a function with **n** parameters into a sequence of **n** functions, which each return an unary, anonymous function. More precisely, the arity is transformed from **n-ary** to **n * 1-ary**. Why exactly one argument? Because the benefits of **n-ary** functions bear no relation to their greater complexity.

```
// declaration of a manually curried function

const add = x => y => x + y;

// and its procedural application

const inc = add(1);

inc(2); // 3
add(1)(2); // 3
```

Currying is right-associative, that is, **add = x => y => x + y** is equivalent with **add = x => (y => x + y)**. Since each anonymous function is only evaluated once the required argument is provided, currying introduces a notion of laziness to function application.

Partial application on the other hand deals with the application of functions, not with their declaration. In contrast to multi-argument functions a curried function is invoked procedurally in a uniform manner. Actually, unary functions can not be partially applied anymore. However, if you consider the sequence of anonymous functions as a single one, it somehow makes sense to think about a partially applied function in curried form. I guess that is the reason why both concepts are constantly mixed up.

What is the purpose of currying?

It's difficult to illustrate the benefits of currying, because the technique has essentially a **systemic effect**. Systemic means that a technique needs to interact with other basic techniques to fully develop its advantages. There is a mutual dependency between the components involved.

Currying is closely connected to higher order functions, which derive from first class functions. Higher order functions are such a fundamental concept in functional programming that they are literally omnipresent. The composition operator function, for example, is a well known one. Currying makes working with higher order

functions much more effective, provided the involved functions respect a particular parameter order, optimized for ...well, currying. More on that later.

The most significant advantage of currying along with higher order functions is **abstraction over arity**. Whenever a higher order function both receives and returns a function, some (but not all) of the passed functions can be of any arity. They lose their arity so to speak:

```
// evaluated from right to left

const comp = f => g => x => f(g(x));

const inc = x => x + 1; // unary function
const mul = y => x => x * y; // binary function
const sqr = x => mul(x)(x); // unary function

// abstraction over arity

comp(sqr)(inc)(2); // 9 (A)
comp(mul)(inc)(2)(3); // 9 (B)
comp(inc)(mul)(2)(3); // TypeError (C)
```

It is not surprising that (A) works as expected, since **sqr** and **inc** have the right arity. We also expect (C) to fail, because the arity of **inc** and **mul** doesn't match. But why does (B) produce a conclusive result, even though **mul** apparently is a binary function? This is abstraction over arity in action: **comp** does not care about **f**'s arity. The reason for this behavior is **comp**'s type:

```
comp :: (b -> c) -> (a -> b) -> (a -> c      )
      f          => g          => (x => f(g(x)));
```

```
comp :: (b -> c -> d) -> (a -> b) -> (a -> d      )
      f          => g          => (x => y => f(g(x)));
```

comp returns the anonymous function that is itself returned by **f**. The arity of this anonymous function just does not matter.

Conclusion

To understand the advantages of currying the concept can not be considered on its own. It facilitates function composition and establishes a unified and predictable interface, which abstracts function arity away.

Implementation in Javascript/EcmaScript 2015

Javascript doesn't have pre-curried functions. However, the new arrow syntax of EcmaScript 2015 allows us to write manually curried functions with little syntactic overhead:

```
// classic, nested syntax

function map(f) {
  return function (xs) {
    function next(acc, [head, ...tail]) {
      return head === undefined
        ? acc
        : next([...acc, f(head)], tail);
    }

    return next([], xs);
  }
}

// ES2015 arrow syntax

const map = f => xs => {
  const next = (acc, [head, ...tail]) => head === undefined
    ? acc
    : next([...acc, f(head)], tail);

  return next([], xs);
};
```

However, for all functions that are out of our control, we need a programmatic curry solution:

```
const curryn = n => f => {
  const next = (n, args) => n === 0
    ? f(...args)
    : x => next(n - 1, [...args, x]);

  return next(n, []);
};

const add = (x, y) => x + y;
const addc = curryn(2) (add);
```

```

const inc = addc(1);

addc(2) (3); // 5
inc(3); // 4

const map = (f, xs) => xs.map(x => f(x));
const mul = y => x => x * y;
const sqr = x => mul(x) (x);
const mapc = curryn(2) (map);

```

```

const xs = [1, 2, 3];

mapc(sqr) (xs); // [1, 4, 9]

```

curryn is just a wrapper to hide the accumulator **args**, which is part of the internal API. The actual curry mechanism is implemented within **next**, a recursive function that lazily collects **n** arguments.

Please note that there is not the one right solution. Alternative, non-recursive approaches might look like this:

```

curryn = n => f =>
  a => n < 2 ? f(a) :
  b => n < 3 ? f(a, b) :
  c => n < 4 ? f(a, b, c) :
  d => n < 5 ? f(a, b, c, d) :
  e => n < 6 ? f(a, b, c, d, e) :
  () => {throw "unsupported arity"}();
}

curry2 => f => a => b => f(a, b);
curry3 => f => a => b => c => f(a, b, c);
curry4 => f => a => b => c => d => f(a, b, c, d);
curry5 => f => a => b => c => d => e => f(a, b, c, d, e);

```

Provided that we agree to rely on **Function.length**, the following auxiliary helper can be used as well:

```
const curry = f => curryn(f.length)(f);
```

```
const add_ = curry((x, y) => x + y);
const inc = add(1);

add(2)(3); // 5
inc(3); // 4
```

However, please keep in mind that one of the key benefits of currying consists precisely in abstracting over arity. **curry** counteracts this idea somehow.

Variadic functions and optional parameters

Since our **curryn** function does not rely on the length property of functions, we have no problem with transforming variadic functions or those with optional arguments:

```
const sum = (...args) => args.reduce((acc, x) => acc + x, 0);

sum(1, 2, 3, 4, 5); // 15
curry(sum)(1)(2)(3)(4)(5); // yields another function
curryn(5)(sum)(1)(2)(3)(4)(5); // 15
```

Anyway, I want to emphasize that variadic functions create more problems than they solve and should be avoided anyway. Either use a **fold** (aka reduce) or semi-variadic functions, where the variadic argument is provided as an **Iterable**.

Optional arguments are a strong indicator that a function has either an ineffective parameter order or should be divided into two separate functions from the beginning.

Currying methods

Instead of programmatically transforming methods into curried form, just use curried function wrappers. A programmatic solution isn't worth the effort:

```
const concat = ys => xs => xs.concat(ys);
const apply = f => xs => f.apply(null, xs);
const split = y => x => x.split(y);
```

Non-commutative operators

So far we haven't paid attention to the commutative law. The problem is that some operators are non-commutative, that is, the evaluation order of their operands is crucial:

```
1 + 2 === 2 + 1; // true, commutative
1 - 2 === 2 - 1; // false, non-commutative
```

In order to be able to pass operators around like normal data, we have to re-define them as operator functions:

```
const add = x => y => x + y;
const sub = x => y => x - y;
```

```
add(2)(3) === add(3)(2); // true, commutative
sub(2)(3) === sub(3)(2); // false, non-commutative
```

Unfortunately, functions are always in prefix position in Javascript, while operators are in infix notation:

```
sub(2)(3); // prefix notation
2 - 3; // infix notation
```

While the infix notation can be read only in one way, the prefix form allows to different interpretations:

```
// reads either as "subtract 2 from 3" or as "subtract from 2 3"
```

```
sub(2)(3);
```

```
// reads simply as 2 subtract 3
```

```
2 - 3;
```

This ambiguity arises from two opposed parameter orders each binary operator function can be defined with:

```
const sub = x => y => x - y;
const sub_ = y => x => x - y;
```

```
sub(2); // yields x => 2 - x (left section)
sub_(2); // yields x => x - 2 (right section)
```

It turns out that we usually want to use the right section, when we partially apply the curried function sequence

```
const sub_ = y => x => x - y;
const sub2 = sub(2);
```

```
sub2(10); // 8
```

and the left section, when we fully apply it

```
const sub = x => y => x - y;
```

```
sub(2) (10); // -8
```

By the way, you do not have to define each operator function twice, but can utilize a simple, programmatic solution:

```
const flip = f => x => y => f(y) (x);
const sub = x => y => x - y;
const sub_ = flip(sub);
```

If you are really concerned about a performance penalty in certain scenarios, you can also do this:

```
const sub = (x, y) => x - y;
const sub_ = y => x => x - y;
```

What you should not do

Your code should not rely on

- the **length** or **name** property of the **Function** prototype
- auto currying, because it introduces an unnecessary complexity
- a programmatic curry solution that supports "gaps" or "holes" for individual arguments, because this is not currying anymore

Why parameter order matters

Currying has a great impact on parameter orders, because the procedural invocation of function sequences becomes a common operation. This has two consequences for the argument list:

- the primary parameter should be placed at the end to facilitate function composition
- parameters that are least likely to change should be placed leftmost to facilitate partially applied functions

The primary parameter represents the argument, that is shared between the involved functions of a composition.

That is exactly the last argument, which is fed with the return value of the nested function. The primary argument is the functional equivalent of the receiving object, which enables method chaining in Javascript.

Up next: Function composition

If you have enjoyed this article, please don't forget to ❤ or to **share** it - thanks! Comments and questions are appreciated as well. The next article illustrates the advantages of functional composition and combinatorics.

Next part: [function composition](#)

69 : An Introduction to Functional Programming in JavaScript

<http://srirangan.net/2011-12-functional-programming-in-javascript>

For me, in many ways, 2011 has been the year of the rediscovery of JavaScript. I have been programming in JavaScript for many years (like most other programmers) but now I've come to appreciate JavaScript the language a little better.

2011 was also when I had to dig into Scala. Thus I was compelled to implement more solutions in a functional manner.

Like many programmers unfamiliar with Functional Programming or others familiar with Functional Programming but more comfortable with Object Oriented Programming, it was daunting at first.

But persist through and one starts to see the benefits of Functional Programming (or at least understand it enough and execute your projects).

If you're looking for a quick start in Functional Programming, JavaScript is the perfect language for you. Here's why:

- Almost all programmers have tweaked and / or written JavaScript code at some point of time — hence there should be a certain familiarity
- JavaScript comes as close to a standardized programming language we'll get — it's the only programming language available across all web browsers
- JavaScript comes with a very familiar C like syntax and should be readable to most programmers
- Functions have always been first class members in JavaScript, support for Functional Programming is very good and in many ways JavaScript has been ahead of its time
- JavaScript doesn't have Java like Class based Object Oriented Programming support so in many ways you're forced to be Functional in JavaScript

These should be enough reasons, so without meandering any further in the introduction, I present *Functional Programming with JavaScript*:

Functions in JavaScript

Almost everyone reading would have seen **function declarations** in JavaScript. However JavaScript lets you explicitly express functions as objects.

In fact, all **function declarations** are automatically converted to **function expressions** and hoisted.

By **hoisting**, the converted function expression is automatically placed in the beginning and thus is available in lines before it has been declared.

```
function justAFunction() {  
    console.log("functions can be declared");  
}
```

```
var justAnotherFunction = function () {
    console.log("functions can be expressed as objects");
};

console.log("All functions are objects of type 'function'");
console.log(typeof justAFunction);
console.log(typeof justAnotherFunction);

// This function is available through hoisting
// although it will be declared later
console.log(typeof justAFunctionThatWillBeHoisted);

function justAFunctionThatWillBeHoisted() {
    console.log("Function declarations are automatically hoisted to the top");
}
```

```
function justAFunction() {
    console.log("functions can be declared");
}

var justAnotherFunction = function () {
    console.log("functions can be expressed as objects");
};

console.log("All functions are objects of type 'function'");
console.log(typeof justAFunction);
console.log(typeof justAnotherFunction);

// This function is available through hoisting
// although it will be declared later
console.log(typeof justAFunctionThatWillBeHoisted);

function justAFunctionThatWillBeHoisted() {
    console.log("Function declarations are automatically hoisted to the top");
}
```

JavaScript functions are also different in the sense that each function, regardless of how it is defined, has dynamic arguments and a dynamic context.

A JavaScript function can, of course, access its arguments by their variable names.

By dynamic arguments, I mean, in the function body, we are provided an **arguments** array which contain all the arguments sequentially in the order they were passed. This means any function can be called with any set of

arguments, thus giving us a lot of flexibility.

By dynamic context, I am referring to the **this** object. Typically the **this** object defaults sensibly to the context of the function. However it is possible to explicitly call a function with a user defined **this** / context object.

```
var logArguments = function () {
    console.log(arguments);
};

console.log("All functions can have dynamic arguments");
logArguments("hello", "world");
logArguments("hello", "world", 42, ["huh", "wtf!"], {hello:"world"});

var logThis = function () {
    console.log(this);
};

console.log("All functions can have dynamic contexts / 'this' objects");
logThis();
logThis.apply({yo:"baby!"});
```

```
var logArguments = function () {
    console.log(arguments);
};

console.log("All functions can have dynamic arguments");
logArguments("hello", "world");
logArguments("hello", "world", 42, ["huh", "wtf!"], {hello:"world"});

var logThis = function () {
    console.log(this);
};

console.log("All functions can have dynamic contexts / 'this' objects");
logThis();
logThis.apply({yo:"baby!"});
```

Notice how `functionName.apply()` allows us to explicitly set the context and pass arguments.

I must point out that JavaScript presents us with **functional scoping** instead of **block scoping** (C, C++ etc.). This usually becomes more relevant when you need to implement Object Oriented patterns in JavaScript but more on that in a later blog post.

Another trick for you is the capability to define **self-executing anonymous function**. These can be creatively used to implement modular namespaces / packages. Here are some examples:

```
(function (name) {
    console.log(name + " functions can be auto-executed on definition")
})("Anonymous");

console.log("JavaScript provides functional scoping instead of block scoping");
(function () {
    if (true) {
        var x = true;
        console.log("Is 'x' available inside the block? " + x);
    }
    console.log("Is 'x' available outside the block? " + x);
})();
```

```
(function (name) {
    console.log(name + " functions can be auto-executed on definition")
})("Anonymous");

console.log("JavaScript provides functional scoping instead of block scoping");
(function () {
    if (true) {
        var x = true;
        console.log("Is 'x' available inside the block? " + x);
    }
    console.log("Is 'x' available outside the block? " + x);
})();
```

Higher Order Functions

We know functions in JavaScript are nothing but objects. Does this mean functions can be passed around as arguments to other functions? And can one function return another function?

The answer to both questions is "Yes". And such functions which taking in other functions are arguments and / or return functions are typically called **higher order functions**.

In the example below, we will define a higher order function `forEach`. This function takes in two arguments: a `list` and an `action`. The first argument `list` is an array containing containing objects and the second argument `action` is a function that will be invoked for each item in `list`.

```
var forEach = function (list, action) {
    for (var i = 0; i < list.length; i++) {
```

```

        action(list[i]);
    }
};

var logItem = function (item) {
    console.log(item);
};

var listOfThings = ["soap", "candle", "deer", "wine", "bread"];
var anotherListOfThings = ["grapes", "apples", "beer", "pizza"];

forEach(listOfThings, logItem);

forEach(anotherListOfThings, function (item) {
    console.log(item + "'s length is " + item.length);
});

```

```

var forEach = function (list, action) {
    for (var i = 0; i < list.length; i++) {
        action(list[i]);
    }
};

var logItem = function (item) {
    console.log(item);
};

var listOfThings = ["soap", "candle", "deer", "wine", "bread"];
var anotherListOfThings = ["grapes", "apples", "beer", "pizza"];

forEach(listOfThings, logItem);

forEach(anotherListOfThings, function (item) {
    console.log(item + "'s length is " + item.length);
});

```

The simple example above shows how a named function `logItem` and an `anonymous function` are being passed as arguments to the higher order function `forEach`.

Map / Reduce Implementations

In the example above, we saw how the `forEach` algorithm was expressed as a function and reused multiple times as an when required. We avoided repeating loops and thus made the code more readable.

This is one of the prime reasons to write functional code. If done correctly, you'll be able to abstract out complex algorithms, replace it with a nice function call and keep code condensed and readable.

To further stress this idea, we need a better example. For each was too simple an algorithm to abstract. Let's try implementing higher order functions for **Map and Reduce algorithms**.

We start with the Map function. What is the map function? Our `map` function should allow the user to perform a particular action over each item of a `list`, and then return a `result` list containing results of actions performed on each list item.

Simple enough, show me the code!

```
var forEach = function (list, action) {
  for (var i = 0; i < list.length; i++) {
    action(list[i]);
  }
};

var map = function (mappingFunction, list) {
  var result = [];
  forEach(list, function (item) {
    result.push(mappingFunction(item));
  });
  return result;
};

var doubleIt = function (item) {
  if (typeof item === "number") {
    return item * 2;
  }
};

console.log(map(doubleIt, [1, 2, 3, 4, "WTF"]));
```

```
var forEach = function (list, action) {
  for (var i = 0; i < list.length; i++) {
    action(list[i]);
  }
};

var map = function (mappingFunction, list) {
  var result = [];
  forEach(list, function (item) {
    result.push(mappingFunction(item));
```

```

    });
    return result;
};

var doubleIt = function (item) {
    if (typeof item === "number") {
        return item * 2;
    }
};

console.log(map(doubleIt, [1, 2, 3, 4, "WTF"]));

```

Notice we reuse the `forEach` function from the previous example. Our `map` function takes in two arguments, the first one being the `mappingFunction` that is to be applied on every item of the second argument `list`.

The implementation for `map` is straightforward. We prepare an array `result`, iterate through the list, call the `mappingFunction` for each item in `list`, push the result in the `result` array return it when all done.

As an example, we are calling `map` function and passing the function `doubleIt` as the `mapperFunction` on each item of an anonymous list.

Time for the `reduce` function. What is the reduce algorithm? The `reduce` function will be given a `combine` function, a `base` value and a `list`. We need to apply the `combine` function on the base value and each value in the list and finally return the result. In short, Reduce function takes in a list, combines it and gives us a single result.

```

var forEach = function (list, action) {
    for (var i = 0; i < list.length; i++) {
        action(list[i]);
    }
};

var reduce = function (combine, base, list) {
    forEach(list, function (item) {
        base = combine(base, item);
    });
    return base;
};

var countNegativeNumbers = function (negativeNumbersTillNow, currentNumber) {
    if (typeof currentNumber === "number" && currentNumber < 0) {
        negativeNumbersTillNow += 1;
    }
    return negativeNumbersTillNow;
};

```

```

var initialCount = 0;

console.log(reduce(countNegativeNumbers, initialCount, [1, -1, 0, 45, "-42", -42]));

```



```

var forEach = function (list, action) {
  for (var i = 0; i < list.length; i++) {
    action(list[i]);
  }
};

var reduce = function (combine, base, list) {
  forEach(list, function (item) {
    base = combine(base, item);
  });
  return base;
};

var countNegativeNumbers = function (negativeNumbersTillNow, currentNumber) {
  if (typeof currentNumber === "number" && currentNumber < 0) {
    negativeNumbersTillNow += 1;
  }
  return negativeNumbersTillNow;
};

var initialCount = 0;

console.log(reduce(countNegativeNumbers, initialCount, [1, -1, 0, 45, "-42", -42]));

```

This solution again makes use of the previous create `forEach` function. Our `reduce` function is simple, it iterates through the `list`, applies `combine` on `base` and each list item, stores the result in `base` and returns it after completion.

As a usage example, we use `reduce` to get a count of the negative numbers in a list.

These two examples are fairly simple and still show us how, with functional programming, we can effectively abstract out algorithms as functions, use them as and when required and still retain simplicity and readability in the code.

Closures in JavaScript

If you've heard of Functional Programming, there's a very good chance you've also heard about **closures**. They are often mentioned in the same breath.

But what are closures? Think of closures as a composite of two things:

1. A function being returned by another function
2. A closed environment in which this returned function is forced to execute

This is what a closure is.

In our first example of a closure implementation, we have `getAreaFunction` which reads the shape argument and returns an appropriate area function.

```
const CIRCLE = "circle";
const SQUARE = "square";
const RECTANGLE = "rectangle";

var getAreaFunction = function (shape) {
    return function () {
        switch (shape) {
            case CIRCLE:
                return Math.PI * arguments[0] * arguments[0];
                break;
            case SQUARE:
                return arguments[0] * arguments[0];
                break;
            case RECTANGLE:
                return arguments[0] * arguments[1];
                break;
        }
    };
};

var getAreaOfCircle = getAreaFunction(CIRCLE);
var getAreaOfSquare = getAreaFunction(SQUARE);
var getAreaOfRectangle = getAreaFunction(RECTANGLE);

console.log(getAreaOfCircle(50));
console.log(getAreaOfSquare(50));
console.log(getAreaOfRectangle(50, 20));
```

```
const CIRCLE = "circle";
const SQUARE = "square";
const RECTANGLE = "rectangle";

var getAreaFunction = function (shape) {
    return function () {
```

```

switch (shape) {
  case CIRCLE:
    return Math.PI * arguments[0] * arguments[0];
  break;
  case SQUARE:
    return arguments[0] * arguments[0];
  break;
  case RECTANGLE:
    return arguments[0] * arguments[1];
  break;
}
};

};

var getAreaOfCircle = getAreaFunction(CIRCLE);
var getAreaOfSquare = getAreaFunction(SQUARE);
var getAreaOfRectangle = getAreaFunction(RECTANGLE);

console.log(getAreaOfCircle(50));
console.log(getAreaOfSquare(50));
console.log(getAreaOfRectangle(50, 20));

```

Notice how `getAreaFunction` returns an anonymous function. This returned function contains a switch case on `shape` which is available in the *environment*.

Finally, of course, we see how to use the closure as a *function factory* to get area functions for different types of shapes.

In our second closure example, we see how closures are used to implement private class members in JavaScript.

```

const CIRCLE = "circle";
const SQUARE = "square";
const RECTANGLE = "rectangle";

var getAreaFunction = function (shape) {
  return function () {
    switch (shape) {
      case CIRCLE:
        return Math.PI * arguments[0] * arguments[0];
      break;
      case SQUARE:
        return arguments[0] * arguments[0];
      break;
    }
  };
};

```

```

        case RECTANGLE:
            return arguments[0] * arguments[1];
            break;
        }
    };
};

var Shape = function (type) {
    var area = getAreaFunction(type);
    return {
        getArea:function () {
            return Math.round(Number(area.apply(this, arguments)));
        }
    };
};

var aCircle = new Shape(CIRCLE);
var aSquare = new Shape(SQUARE);
var aRectangle = new Shape(RECTANGLE);

console.log(aCircle.getArea(50));
console.log(aSquare.getArea(50));
console.log(aRectangle.getArea(50, 20));

```

```

const CIRCLE = "circle";
const SQUARE = "square";
const RECTANGLE = "rectangle";

var getAreaFunction = function (shape) {
    return function () {
        switch (shape) {
            case CIRCLE:
                return Math.PI * arguments[0] * arguments[0];
                break;
            case SQUARE:
                return arguments[0] * arguments[0];
                break;
            case RECTANGLE:
                return arguments[0] * arguments[1];
                break;
        }
    };
};

```

```

var Shape = function (type) {
    var area = getAreaFunction(type);
    return {
        getArea:function () {
            return Math.round(Number(area.apply(this, arguments)));
        }
    };
};

var aCircle = new Shape(CIRCLE);
var aSquare = new Shape(SQUARE);
var aRectangle = new Shape(RECTANGLE);

console.log(aCircle.getArea(50));
console.log(aSquare.getArea(50));
console.log(aRectangle.getArea(50, 20));

```

I'll not explain Object Oriented Programming implementations in JavaScript as part of this post. They aren't complex but are different, surely worthy of a dedicated post.

If you're already familiar with OOP concepts and their implementations in JavaScript, note how the class `Shape` (which is, yes, expressed as a function) contains a private function `area` and isn't exposed.

Instead an instance function `getArea` is available which does internally make use of the private function `area`.

This is one of the most common use-cases of closures in JavaScript.

Another classic use-case for closures in JavaScript is to avoid the annoying problem that crops up every time you try to set event handlers within a loop.

Consider the following example:

```

<div id="redBox1" class="redBox"></div>
<div id="redBox2" class="redBox"></div>
<div id="redBox3" class="redBox"></div>

<div id="greenBox1" class="greenBox"></div>
<div id="greenBox2" class="greenBox"></div>
<div id="greenBox3" class="greenBox"></div>

<script>
(function () {
    for (var i = 1; i <= 3; i++) {
        var redBoxId = "redBox" + i;

```

```

document.getElementById(redBoxId).onclick = function () {
    console.log(redBoxId + " was clicked");
};

var greenBoxId = "greenBox" + i;
document.getElementById(greenBoxId).onclick = (function (greenBoxId) {
    return function () {
        console.log(greenBoxId + " was clicked");
    }
})(greenBoxId);

}

})();
</script>

```

```

<div id="redBox1" class="redBox"></div>
<div id="redBox2" class="redBox"></div>
<div id="redBox3" class="redBox"></div>

<div id="greenBox1" class="greenBox"></div>
<div id="greenBox2" class="greenBox"></div>
<div id="greenBox3" class="greenBox"></div>

<script>
(function () {
    for (var i = 1; i <= 3; i++) {
        var redBoxId = "redBox" + i;
        document.getElementById(redBoxId).onclick = function () {
            console.log(redBoxId + " was clicked");
        };
        var greenBoxId = "greenBox" + i;
        document.getElementById(greenBoxId).onclick = (function (greenBoxId) {
            return function () {
                console.log(greenBoxId + " was clicked");
            }
        })(greenBoxId);
    }
})();
</script>

```

In the for loop, we first assign an `onclick` handler to all the red boxes. However, thanks to functional scoping, all three red boxes have onclick handlers with `redBoxId` value 3.

However for green boxes we assign `onclick` handlers in a slightly different manner. We have a self executing anonymous function which returns another function which then is assigned as the onclick handler.

Due to functional scopes, each `onclick` handler for the green boxes have correct values for `greenBoxId`.

I realize this has been a fairly lengthy post. Will end it now. Hopefully this would have served as a good enough introduction to Functional Programming concepts and their corresponding implementations in JavaScript.

All code available on [GitHub](#).

70 : Don't Be Scared Of Functional Programming

Functional programming is the mustachioed hipster of programming paradigms. Originally relegated to the annals of computer science academia, functional programming has had a recent renaissance that is due largely to its utility in distributed systems (and probably also because "pure" functional languages like Haskell are difficult to grasp, which gives them a certain cachet).

Stricter functional programming languages are typically used when a system's performance and integrity are both critical — i.e. your program needs to do exactly what you expect every time and needs to operate in an environment where its tasks can be shared across hundreds or thousands of networked computers.

Further Reading on SmashingMag: [Link](#)

- [An Introduction To Programming Type Systems¹](#)
- [An Introduction To Redux²](#)
- [An Introduction To Full-Stack JavaScript³](#)
- [Declarative Programming And The Web⁴](#)

[Clojure⁵](#), for example, powers [Akama⁶](#), the massive content delivery network utilized by companies such as Facebook, while [Twitter famously adopted⁷ Scala⁸](#) for its most performance-intensive components, and [Haskell⁹](#) is used by AT&T for its network security systems.

These languages have a steep learning curve for most front-end web developers; however, many more approachable languages incorporate features of functional programming, most notably Python, both in its core library, with functions like `map` and `reduce` (which we'll talk about in a bit), and with libraries such as [Fn.py¹⁰](#), along with JavaScript, again using collection methods, but also with libraries such as [Underscore.js¹¹](#) and [Bacon.js¹²](#).

```

6
7   function addNumbers(a, b) {
8     return a + b;
9   }
10
11
12
13 // Takes the values of an array and returns the total. Demonstrates simple
14 // recursion.
15 function totalForArray(arr, currentTotal) {
16   currentTotal = addNumbers(currentTotal + arr.shift());
17
18   if(arr.length > 0) {
19     return totalForArray(currentTotal, arr);
20   }
21   else {
22     return currentTotal;
23   }
24 }
25
26 // Or you could just use reduce.
27 function totalForArray(arr) {
28   return arr.reduce(addNumbers);
29 }
30
31 // Should really be called divideTwoNumbers.
32 function average(total, count) {
33   return count / total;
34 }
35
36 function averageForArray(arr) {
37   return average(arr.length, totalForArray(arr));
38 }
39
40 // Gets the value associated with the property of an object. Intended for
41 // use with a collection method like map, hence the generator.
42 function getItem(propertyName) {
43   return function(item) {
44     return item[propertyName];
45   }
46 }
47

```

Functional programming can be daunting, but remember that it isn't only for PhDs, data scientists and architecture astronauts. For most of us, the real benefit of adopting a functional style is that our programs can be broken down into smaller, simpler pieces that are both more reliable and easier to understand. If you're a front-end developer working with data, especially if you're formatting that data for visualization using D3, Raphael or the like, then functional programming will be an essential weapon in your arsenal.

Finding a consistent definition of functional programming is tough, and most of the literature relies on somewhat foreboding statements like "functions as first-class objects," and "eliminating side effects." Just in case that doesn't bend your brain into knots, at a more theoretical level, functional programming is often explained in terms of [lambda calculus¹³](#) (some [actually argue¹⁴](#) that functional programming is basically math) — but you can relax. From a more pragmatic perspective, a beginner needs to understand only two concepts in order to use it for everyday applications (no calculus required!).

First, data in functional programs should be **immutable**, which sounds serious but just means that it should never change. At first, this might seem odd (after all, who needs a program that never changes anything?), but in practice, you would simply create new data structures instead of modifying ones that already exist. For example, if you need to manipulate some data in an array, then you'd make a new array with the updated values, rather than revise the original array. Easy!

Secondly, functional programs should be **stateless**, which basically means they should perform every task as if for the first time, with no knowledge of what may or may not have happened earlier in the program's execution (you might say that a stateless program is ignorant of the past). Combined with immutability, this helps us think of each function as if it were operating in a vacuum, blissfully ignorant of anything else in the application besides other functions. In more concrete terms, this means that your functions will operate only on data passed in as arguments and will never rely on outside values to perform their calculations.

Immutability and statelessness are core to functional programming and are important to understand, but don't worry if they don't quite make sense yet. You'll be familiar with these principles by the end of the article, and I promise that the beauty, precision and power of functional programming will turn your applications into bright, shiny, data-chomping rainbows. For now, start with simple functions that return data (or other functions), and then combine those basic building blocks to perform more complex tasks.

For example, let's say we have an API response:

```
var data = [
  {
    name: "Jamestown",
    population: 2047,
    temperatures: [-34, 67, 101, 87]
  },
  {
    name: "Awesome Town",
    population: 3568,
    temperatures: [-3, 4, 9, 12]
  }
  {
    name: "Funky Town",
    population: 1000000,
    temperatures: [75, 75, 75, 75, 75]
  }
];

```

If we want to use a chart or graphing library to compare the average temperature to population size, we'd need to write some JavaScript that makes a few changes to the data before it's formatted correctly for our visualization. Our graphing library wants an array of x and y coordinates, like so:

```
[
  [x, y],
  [x, y]
  ...etc
]
```

Here, `x` is the average temperature, and `y` is the population size.

Without functional programming (or without using what's called an "imperative" style), our program might look like this:

```
var coords = [],
  totalTemperature = 0,
  averageTemperature = 0;
```

```

for (var i=0; i < data.length; i++) {
  totalTemperature = 0;

  for (var j=0; j < data[i].temperatures.length; j++) {
    totalTemperature += data[i].temperatures[j];
  }

  averageTemperature = totalTemperature / data[i].temperatures.length;

  coords.push([averageTemperature, data[i].population]);
}

```

Even in a contrived example, this is already becoming difficult to follow. Let's see if we can do better.

When programming in a functional style, you're always looking for simple, repeatable actions that can be abstracted out into a function. We can then build more complex features by calling these functions in sequence (also known as "composing" functions) — more on that in a second. In the meantime, let's look at the steps we'd take in the process of transforming the initial API response to the structure required by our visualization library. At a basic level, we'll perform the following actions on our data:

- add every number in a list,
- calculate an average,
- retrieve a single property from a list of objects.

We'll write a function for each of these three basic actions, then compose our program from those functions.

Functional programming can be a little confusing at first, and you'll probably be tempted to slip into old imperative habits. To avoid that, here are some simple ground rules to ensure that you're following best practices:

1. All of your functions must accept at least one argument.
2. All of your functions must return data or another function.
3. No loops!

OK, let's add every number in a list. Remembering the rules, let's make sure that our function accepts an argument (the array of numbers to add) and returns some data.

```

function totalForArray(arr) {
  // add everything
  return total;
}

```

So far so good. But how are we going to access every item in the list if we don't loop over it? Say hello to your new friend, recursion! This is a bit tricky, but basically, when you use recursion, you create a function that calls itself unless a specific condition has been met — in which case, a value is returned. Just looking at an example is probably easiest:

```
// Notice we're accepting two values, the list and the current total
function totalForArray(currentTotal, arr) {

    currentTotal += arr[0];

    // Note to experienced JavaScript programmers, I'm not using Array.shift on
    // purpose because we're treating arrays as if they are immutable.
    var remainingList = arr.slice(1);

    // This function calls itself with the remainder of the list, and the
    // current value of the currentTotal variable
    if(remainingList.length > 0) {
        return totalForArray(currentTotal, remainingList);
    }

    // Unless of course the list is empty, in which case we can just return
    // the currentTotal value.
    else {
        return currentTotal;
    }
}
```

A word of caution: Recursion will make your programs more readable, and it is essential to programming in a functional style. However, in some languages (including JavaScript), you'll run into problems when your program makes a large number of recursive calls in a single operation (at the time of writing, "large" is about [10,000 calls in Chrome, 50,000 in Firefox and 11,000 in Node.js](#)¹⁵). The details are beyond the scope of this article, but the gist is that, at least [until ECMAScript 6 is released](#)¹⁶, JavaScript doesn't support something called "tail recursion," which is a more efficient form of recursion. This is an advanced topic and won't come up very often, but it's worth knowing.

With that out of the way, remember that we needed to calculate the total temperature from an array of temperatures in order to then calculate the average. Now, instead of looping over each item in the `temperatures` array, we can simply write this:

```
var totalTemp = totalForArray(0, temperatures);
```

If you're purist, you might say that our `totalForArray` function could be broken down even further. For example, the task of adding two numbers together will probably come up in other parts of your application and subsequently should really be its own function.

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

Now, our `totalForArray` function looks like this:

```
function totalForArray(currentTotal, arr) {
  currentTotal = addNumbers(currentTotal, arr[0]);

  var remainingArr = arr.slice(1);

  if(remainingArr.length > 0) {
    return totalForArray(currentTotal, remainingArr);
  }
  else {
    return currentTotal;
  }
}
```

Excellent! Returning a single value from an array is fairly common in functional programming, so much so that it has a special name, "reduction," which you'll more commonly hear as a verb, like when you "reduce an array to a single value." JavaScript has a special method just for performing this common task. Mozilla Developer Network provides a [full explanation¹⁷](#), but for our purposes it's as simple as this:

```
// The reduce method takes a function as its first argument, and that function
// accepts both the current item in the list and the current total result from
// whatever calculation you're performing.

var totalTemp = temperatures.reduce(function(previousValue, currentValue){
  // After this calculation is returned, the next currentValue will be
  // previousValue + currentValue, and the next previousValue will be the
  // next item in the array.
  return previousValue + currentValue;
});
```

But, hey, since we've already defined an `addNumber` function, we can just use that instead.

```
var totalTemp = temperatures.reduce(addNumbers);
```

In fact, because totalling up an array is so cool, let's put that into its own function so that we can use it again without having to remember all of that confusing stuff about reduction and recursion.

```
function totalForArray(arr) {
  return arr.reduce(addNumbers);
}

var totalTemp = totalForArray(temperatures);
```

Ah, now *that* is some readable code! Just so you know, methods such as `reduce` are common in most functional programming languages. These helper methods that perform actions on arrays in lieu of looping are often called “higher-order functions.”

Moving right along, the second task we listed was calculating an average. This is pretty easy.

```
function average(total, count) {  
    return total / count;  
}
```

How might we go about getting the average for an entire array?

```
function averageForArray(arr) {  
    return average(totalForArray(arr), arr.length);  
}  
  
var averageTemp = averageForArray(temperatures);
```

Hopefully, you're starting to see how to combine functions to perform more complex tasks. This is possible because we're following the rules set out at the beginning of this article — namely, that our functions must always accept arguments and return data. Pretty awesome.

Lastly, we wanted to retrieve a single property from an array of objects. Instead of showing you more examples of recursion, I'll cut to the chase and clue you in on another built-in JavaScript method: [map¹⁸](#). This method is for when you have an array with one structure and need to map it to another structure, like so:

```
// The map method takes a single argument, the current item in the list. Check  
// out the link above for more complete examples.  
var allTemperatures = data.map(function(item) {  
    return item.temperatures;  
});
```

That's pretty cool, but pulling a single property from a collection of objects is something you'll be doing all the time, so let's make a function just for that.

```
// Pass in the name of the property that you'd like to retrieve  
function getItem(propertyName) {  
    // Return a function that retrieves that item, but don't execute the function.  
    // We'll leave that up to the method that is taking action on items in our  
    // array.  
    return function(item) {  
        return item[propertyName];  
    };  
}
```

```

    }
}
```

Check it out: We've made a function that returns a function! Now we can pass it to the `map` method like this:

```
var temperatures = data.map(getItem('temperature'));
```

In case you like details, the reason we can do this is because, in JavaScript, functions are "first-class objects," which basically means that you can pass around functions just like any other value. While this is a feature of many programming languages, it's a requirement of any language that can be used in a functional style. Incidentally, this is also the reason you can do stuff like `$('#my-element').on('click', function(e) ...)`. The second argument in the `on` method is a `function`, and when you pass functions as arguments, you're using them just like you would use values in imperative languages. Pretty neat.

Finally, let's wrap the call to `map` in its own function to make things a little more readable.

```

function pluck(arr, propertyName) {
  return arr.map(getItem(propertyName));
}

var allTemperatures = pluck(data, 'temperatures');
```

All right, now we have a toolkit of generic functions that we can use anywhere in our application, even in other projects. We can tally up the items in an array, get the average value of an array, and make new arrays by plucking properties from lists of objects. Last but not least, let's return to our original problem:

```

var data = [
  {
    name: "Jamestown",
    population: 2047,
    temperatures: [-34, 67, 101, 87]
  },
  ...
];
```

We need to transform an array of objects like the one above into an array of `x, y` pairs, like this:

```
[  
  [75, 1000000],  
  ...  
];
```

Here, `x` is the average temperature, and `y` is the total population. First, let's isolate the data that we need.

```
var populations = pluck(data, 'population');
var allTemperatures = pluck(data, 'temperatures');
```

Now, let's make an array of averages. Remember that the function we pass to `map` will be called on each item in the array; so, the returned value of that passed function will be added to a new array, and that new array will ultimately be assigned to our `averageTemps` variable.

```
var averageTemps = allTemperatures.map(averageForArray);
```

So far so good. But now we have two arrays:

```
// populations
[2047, 3568, 1000000]

// averageTemps
[55.25, 5.5, 75]
```

Obviously, we want only one array, so let's write a function to combine them. Our function should make sure that the item at index `0` in the first array is paired with the item at index `0` in the second array, and so on for indexes `1` to `n` (where `n` is the total number of items in the array).

```
function combineArrays(arr1, arr2, finalArr) {
  // Just so we don't have to remember to pass an empty array as the third
  // argument when calling this function, we'll set a default.
  finalArr = finalArr || [];

  // Push the current element in each array into what we'll eventually return
  finalArr.push([arr1[0], arr2[0]]);

  var remainingArr1 = arr1.slice(1),
      remainingArr2 = arr2.slice(1);

  // If both arrays are empty, then we're done
  if(remainingArr1.length === 0 && remainingArr2.length === 0) {
    return finalArr;
  }
  else {
    // Recursion!
    return combineArrays(remainingArr1, remainingArr2, finalArr);
  }
};
```

```
var processed = combineArrays(averageTemps, populations);
```

Or, because one-liners are fun:

```
var processed = combineArrays(pluck(data, 'temperatures').map(averageForArray),  
    pluck(data, 'population'));  
  
// [  
//   [ 55.25, 2047 ],  
//   [ 5.5, 3568 ],  
//   [ 75, 1000000 ]  
// ]
```

Let's Get Real [Link](#)

Last but not least, let's look at one more real-world example, this time adding to our functional toolbelt with [Underscore.js¹⁹](#), a JavaScript library that provides a number of great functional programming helpers. We'll pull data from a platform for conflict and disaster information that I've been working on named [CrisisNET²⁰](#), and we'll use the fantastic [D3²¹](#) library to visualize that data.

The goal is to give people coming to CrisisNET's home page a quick snapshot of the types of information in the system. To demonstrate this, we could count the number of documents from the API that are assigned to a particular category, like "physical violence" or "armed conflict." This way, the user can see how much information is available on the topics they find most interesting.

A bubble chart might be a good fit, because they are often used to represent the relative sizes of large groups of people. Fortunately, D3 has a built-in visualization named `pack` for just this purpose. So, let's create a graph with `pack` that shows the number of times that a given category's name appears in a response from CrisisNET's API.

Before we go on, note that D3 is a complex library that warrants its own tutorial (or many tutorials, for that matter). Because this article is focused on functional programming, we won't spend a lot of time on how D3 works. But don't worry — if you're not already familiar with the library, you should be able to copy and paste the code snippets specific to D3 and dig into the details another time. [Scott Murray's D3 tutorials²²](#) are a great resource if you're interested in learning more.

Moving along, let's first make sure we have a DOM element, so that D3 has some place to put the chart it will generate with our data.

```
<div id="bubble-graph"></div>
```

Now, let's create our chart and add it to the DOM.

```
// width of chart
var diameter = 960,
    format = d3.format(",d"),
    // creates an ordinal scale with 20 colors. See D3 docs for hex values
    color = d3.scale.category20c(),

// chart object to which we'll be adding data
var bubble = d3.layout.pack()
    .sort(null)
    .size([diameter, diameter])
    .padding(1.5);

// Add an SVG to the DOM that our pack object will use to draw the
// visualization.
var svg = d3.select("#bubble-graph").append("svg")
    .attr("width", diameter)
    .attr("height", diameter)
    .attr("class", "bubble");
```

The `pack` object takes an array of objects in this format:

```
{
  children: [
    {
      className: ,
      package: "cluster",
      value:
    }
  ]
}
```

CrisisNET's data API returns information in this format:

```
{
  data: [
    {
      summary: "Example summary",
      content: "Example content",
      ...
      tags: [
        {
          name: "physical-violence",
```

```

        confidence: 1
    }
]
}
]
}
```

We see that each document has a `tags` property, and that property contains an array of items. Each tag item has a `name` property, which is what we're after. We need to find each unique tag name in CrisisNET's API response and count the number of times that tag name appears. Let's start by isolating the information we need using the `pluck` function that we created earlier.

```
var tagArrays = pluck(data, 'tags');
```

This gives us an array of arrays, like this:

```
[
[
{
    name: "physical-violence",
    confidence: 1
},
[
{
    name: "conflict",
    confidence: 1
}
]
```

However, what we really want is one array with every tag in it. So, let's use a handy function from Underscore.js named [flatten²³](#). This will take values from any nested arrays and give us an array that is one level deep.

```
var tags = _.flatten(tagArrays);
```

Now, our array is a little easier to deal with:

```
[
{
    name: "physical-violence",
    confidence: 1
}
```

```

},
{
  name: "conflict",
  confidence: 1
}
]

```

We can use `pluck` again to get the thing we really want, which is a simple list of only the tag names.

```

var tagNames = pluck(tags, 'name');

[
  "physical-violence",
  "conflict"
]

```

Ah, that's better.

Now we're down to the relatively straightforward tasks of counting the number of times each tag name appears in our list and then transforming that list into the structure required by the D3 `pack` layout that we created earlier. As you've probably noticed, arrays are a pretty popular data structure in functional programming — most of the tools are designed with arrays in mind. As a first step, then, we'll create an array like this:

```

[
  [ "physical-violence", 10 ],
  [ "conflict", 27 ]
]

```

Here, each item in the array has the tag name at index `0` and that tag's total count at index `1`. We want only one array for each unique tag name, so let's start by creating an array in which each tag name appears only once. Fortunately, an Underscore.js method exists just for this purpose.

```
var tagNamesUnique = _.uniq(tagNames);
```

Let's also get rid of any `false-y` (`false`, `null`, `""`, etc.) values using another handy Underscore.js function.

```
tagNamesUnique = _.compact(tagNamesUnique);
```

From here, we can write a function that generates our arrays using another built-in JavaScript collection method, named [filter²⁴](#), that filters an array based on a condition.

```

function makeArrayCount(keys, arr) {

    // for each of the unique tagNames
    return keys.map(function(key) {
        return [
            key,
            // Find all the elements in the full list of tag names that match this key
            // and count the size of the returned array.
            arr.filter(function(item) { return item === key; }).length
        ]
    });
}

}

```

We can now easily create the data structure that `pack` requires by mapping our list of arrays.

```

var packData = makeArrayCount(tagNamesUnique, tagNames).map(function(tagArray) {
    return {
        className: tagArray[0],
        package: "cluster",
        value: tagArray[1]
    }
});

```

Finally, we can pass our data to D3 and generate DOM nodes in our SVG, one circle for each unique tag name, sized relative to the total number of times that tag name appeared in CrisisNET's API response.

```

function setGraphData(data) {
    var node = svg.selectAll(".node")
        // Here's where we pass our data to the pack object.
        .data(bubble.nodes(data))
        .filter(function(d) { return !d.children; })
        .enter().append("g")
        .attr("class", "node")
        .attr("transform", function(d) { return "translate(" + d.x + "," + d.y + ")"; });

    // Append a circle for each tag name.
    node.append("circle")
        .attr("r", function(d) { return d.r; })
        .style("fill", function(d) { return color(d.className); });

    // Add a label to each circle, using the tag name as the label's text
}

```

```

node.append("text")
  .attr("dy", ".3em")
  .style("text-anchor", "middle")
  .style("font-size", "10px")
  .text(function(d) { return d.className } );
}

```

Putting it all together, here's the `setGraphData` and `makeArray` functions in context, including a call to CrisisNET's API using jQuery (you'll need to [get an API key²⁵](#)). I've also posted a [fully working example on GitHub²⁶](#).

```

function processData(dataResponse) {
  var tagNames = pluck(_.flatten(pluck(dataResponse.data, 'tags')), 'name');
  var tagNamesUnique = _.uniq(tagNames);

  var packData = makeArrayCount(tagNamesUnique, tagNames).map(function(tagArray) {
    return {
      className: tagArray[0],
      package: "cluster",
      value: tagArray[1]
    }
  });
}

return packData;
}

function updateGraph(dataResponse) {
  setGraphData(processData(dataResponse));
}

var apikey = // Get an API key here: http://api.crisis.net
var dataRequest = $.get('http://api.crisis.net/item?limit=100&apikey=' + apikey);

dataRequest.done( updateGraph );

```

That was a pretty deep dive, so congratulations on sticking with it! As I mentioned, these concepts can be challenging at first, but resist the temptation to hammer out `for` loops for the rest of your life.

Within a few weeks of using functional programming techniques, you'll quickly build up a set of simple, reusable functions that will dramatically improve the readability of your applications. Plus, you'll be able to manipulate data structures significantly more quickly, knocking out what used to be 30 minutes of frustrating debugging in a couple lines of code. Once your data has been formatted correctly, you'll get to spend more time on the fun part: making the visualization look awesome!

(*al*, *il*)

71 : Classes, Inheritance And Mixins In JavaScript

JavaScript doesn't have native support for classes. Functions can be used to simulate classes (somewhat), but in general JavaScript is a class-less language. Everything is an object. This makes inheritance particularly strange, since objects inherit from objects, not classes from classes as in languages like Ruby or Java.

CoffeeScript does have support for classes though - that is to say, `class` is a keyword. So how does that work, if JavaScript doesn't have classes? (CoffeeScript just compiles down to JavaScript, so it's not a superset of the language - it merely adds some syntactical sugar)

Classes in CoffeeScript and JavaScript

This CoffeeScript 'class'

```
class Person
  constructor: (@name, @birthday) ->

  age: ->
    moment().diff(@birthday, 'years')
```

is equivalent to this JavaScript:

```
Person = (function() {
  function Person(name1, birthday1) {
    this.name = name1;
    this.birthday = birthday1;
  }

  Person.prototype.age = function() {
    return moment().diff(this.birthday, 'years');
  };

  return Person;
})();
```

This is one pattern for defining classes in JavaScript - creating a named function and adding methods to the prototype of the function. (The other two common patterns are using an object literal and declaring a singleton function)

When defining a class with this pattern (either in CoffeeScript or JavaScript) the function actually behaves very similar to classes in other languages.

```

bob = new Person("Bob", new Date(1980,4,1))
bob.name # Bob
bob.age() # 34

```

(The JavaScript for this code looks exactly the same, just add some semicolons!)

CoffeeScript also allows us to have inheritance between classes. So how does **that** work, if we just said that JavaScript doesn't really have classes (just functions) and that objects inherit from objects, not classes from classes?

Class Inheritance in CoffeeScript and JavaScript

Let's add some inheritance to our CoffeeScript 'class'.

```

class Person
  constructor: (@name, @birthday) ->

  age: ->
    moment().diff(@birthday, 'years')

class Employee extends Person
  constructor: (name, birthday, @company) ->
    super(name, birthday)

  email: ->
    "#{@name}@#{@company.replace(/\ /g,'')}.com".toLowerCase()

```

Here is the equivalent JavaScript (I have reformatted the output from the CoffeeScript compiler to make it a bit more readable).

```

var extend = function(child, parent) {
  for (var key in parent) {
    if (hasProp.call(parent, key)) child[key] = parent[key];
  }
}

function ctor() {
  this.constructor = child;
}

ctor.prototype = parent.prototype;
child.prototype = new ctor();
child.__super__ = parent.prototype;
return child;

```

```

}

var hasProp = {}.hasOwnProperty;

var Person = (function() {
    function Person(name1, birthday1) {
        this.name = name1;
        this.birthday = birthday1;
    }

    Person.prototype.age = function() {
        return moment().diff(this.birthday, 'years');
    };

    return Person;
})();

var Employee = (function(superClass) {
    extend(Employee, superClass);

    function Employee(name, birthday, company) {
        this.company = company;
        Employee.__super__.constructor.call(this, name, birthday);
    }

    Employee.prototype.email = function() {
        return (this.name + "@" + (this.company.replace(/\ /g, "")).toLowerCase()) + ".com");
    };

    return Employee;
})(Person);

```

That is some pretty intense JavaScript, but the magic is happening in the `extend` function. There are 2 types of methods that need to be inherited from the parent - methods defined on the parent itself (we can think of these as class methods) and methods defined on the prototype of the parent (we can think of these as instance methods).

The first thing that happens is that all the class methods on the parent are added to the child (note - **not** to the prototype of the child). This concept is pretty simple - since functions in JavaScript are objects, we can effectively loop through all the functions of the parent class (function) and add them to the child class (function).

```

for (var key in parent) {
    if (hasProp.call(parent, key)) child[key] = parent[key];
}

```

A special constructor function (called `ctor`) is then created to act as the prototype of the child class. CoffeeScript does a lot of work here just to enable us to call `super` when we override parent methods in the child. (`super` is treated as a keyword and CoffeeScript does some additional magic in order to make it all work)

```
function ctor() {
  this.constructor = child;
}

ctor.prototype = parent.prototype;
child.prototype = new ctor();
child.__super__ = parent.prototype;
```

As I mentioned earlier, in JavaScript objects inherit from objects (instead of classes from classes). We can therefore set the prototype for the child class (function) to an instance of the `ctor` function - because they're both just objects. Any new instances of the child class will also have the same prototype and because the `ctor` function has the parent class as a prototype we are effectively using normal prototypal inheritance with an additional level of indirection - all to enable calls to `super` (which is actually really useful).

```
employee = new Employee("Steve", new Date(1975,7,15), "JavaScript Inc")
employee.name      # Steve
employee.age()     # 39
employee.email()   # steve@javascriptinc.com
```

So if CoffeeScript (and as a result, JavaScript) can fake classes with functions and inheritance with functions, could it fake [mixins](#)?

Mixins in CoffeeScript and JavaScript

In order to implement mixins in CoffeeScript (or JavaScript), we would effectively need to copy the functions from one class (function) into another. As we have seen with the `extend` function that CoffeeScript uses for inheritance, copying functions between classes is actually rather easy.

I'm referring to the first part of the `extend` function:

```
for (var key in parent) {
  if (hasProp.call(parent, key)) child[key] = parent[key];
}
```

Let's look at how we might implement mixins in CoffeeScript:

```
class Person
  constructor: (@name, @birthday) ->
```

```

age: ->
  moment().diff(@birthday, 'years')

class Manager
  @title: ->
    "Mr. #{@name}"

class Employee extends Person
  for key of Manager
    @::[key] = Manager[key] if Manager.hasOwnProperty(key)

constructor: (name, birthday, @company) ->
  super(name, birthday)

email: ->
  "#{@name}#{@company.replace(/\ /g,'')}.com".toLowerCase()

```

Within the Employee class (function) I am simply looping through all the properties on the Manager class (function) and assigning them to the prototype of the Employee class.

This might seem strange until you consider that this is exactly how we have been declaring methods on classes all along!

```

class Person
  constructor: (@name, @birthday) ->

  age: ->
    moment().diff(@birthday, 'years')

```

In this example the `age` method compiles down to this JavaScript:

```

Person.prototype.age = function() {
  return moment().diff(this.birthday, 'years');
};

```

This is equivalent to how I am assigning the mixin methods - this piece of CoffeeScript:

```

Person::age = ->
  moment().diff(@birthday, 'years')

```

Is again equivalent to the same JavaScript:

```
Person.prototype.age = function() {  
    return moment().diff(this.birthday, 'years');  
};
```

So this pattern really allows us to assign methods to our 'class' in exactly the same way as we did before. The only difference is that our Manager 'class' (in Ruby we would call this a module) assigns functions directly to the Manager 'class', instead of to the prototype. This makes perfect sense, since we don't necessarily want to have to create an instance of the Manager 'class' in order to use it as a mixin in the Employee 'class'.

There is a very strong similarity to how mixins behave in Ruby, except this is actually a bit cleaner. In Ruby a mixin is actually injected in the inheritance chain, where in JavaScript the function is directly assigned to the class.

Why do we need Mixins?

I started playing around with mixins in CoffeeScript because I became frustrated with sharing code via inheritance. When the only way to share code between classes is inheritance, you can easily end up with inheritance where it doesn't really make sense. (When all you have is a hammer, everything looks like a nail!)

This pattern would at the very least provide an alternative to inheritance - which is clearly not always the best solution. You can find all the [code on Plunker](#). Happy coding.

72 : JavaScript Mixins: Beyond Simple Object Extension

Mixins are generally easy in JavaScript, though they are semantically different than [what Ruby calls a Mixin](#) which are facilitated through inheritance behind the scenes. If you need a good place to start to understand how to build mixins with JavaScript, Chris Missal has been writing up a series on different ways of [extending objects in JavaScript](#). I highly recommend reading his series. It's full of great tips.

Object Extension Is A Poor Man's Mixin

I love the "extend" methods of [jQuery](#) and [Underscore](#). I use them a lot. They're powerful and simple and make it easy to transfer data and behavior from one object to another -- the essence of a mixin. But in spite of my love of underscore.js and jQuery's "extend" methods, there's a problem with them in that they apply every attribute from one or more source objects to a target objects.

You can see the effect of this in [Chris' underscore example](#):

```
var start = {
  id: 123,
  count: 41,
  desc: 'this is information',
  title: 'Base Object',
  tag: 'uncategorized',
  values: [1,1,2,3,5,8,13]
};

var more = {
  name: 'Los Techies',
  tag: 'javascript'
};

var extra = {
  count: 42,
  title: null,
  desc: undefined,
  values: [1,3,6,10]
};

var extended = _.extend(start, more, extra);
console.log(JSON.stringify(extended));
```

In this example, the first object contains a key named "values" and the third object also contains a key named "values". When the extend method is called, the last one in wins. This means that the "values" from the third object end up being the values on the final target object.

So, what happens if we want to mix two behavioral sets in to one target object, but both of those behavior sets use the same underlying "values", or "config", or "bindings" (as reported in [this Marionette issue](#))? One or both of them

will break, and that's definitely not a good thing.

A Problem Of Context

The good news is there's an easy way to solve the mixin problem with JavaScript: only copy the methods you need from the behavior set in to the target.

That is, if the object you want to mix in to another has a method called "doSomething", you shouldn't be forced to copy the "config" and "_whatever" and "foo" and all the other methods and attributes off this object just to get access to the "doSomething" method. Instead, you should only have to copy the "doSomething" method from the behavior source to your target.

```
var foo = {  
  doSomething: function(){  
    // ...  
  }  
}  
  
var bar = {};  
bar.doSomething = foo.doSomething;
```

```
var foo = {  
  doSomething: function(){  
    // ...  
  }  
}  
  
var bar = {};  
bar.doSomething = foo.doSomething;
```

But this poses its own challenge: the source of the behavior likely calls "this.config" and "this._whatever()" and other context-based methods and attributes to do its work. Simply copying the "doSomething" function from one object to another won't work because the context of the function will change and the support methods / data won't be found.

[Note: For more detail on context and how it changes, check out my [JavaScript Context screencast](#).]

Solving The Mixin Problem

To fix the mixin problem then, we need to do two things:

1. Only copy the methods we need to the target
2. Ensure the copied methods retain their original context when executing

This is easier than it sounds. We've already seen how requirement #1 can be solved, and requirement #2 can be handled in a number of ways, including the raw [ECMAScript 5 "bind"](#) method, the use of [Underscore's "bind"](#)

method, writing our own, or by using one of a number of other shims and plugins.

The way to facilitate a mixin from one object to another, then, looks something like this:

```
var foo = {  
  baz: function(){  
    // ...  
  },  
  
  config: [ ... ]  
}  
  
var bar = {  
  config: { ... }  
};  
  
// ECMAScript "bind"  
bar.baz = foo.baz.bind(foo);  
  
// Underscore "bind"  
bar.baz = _.bind(foo.baz, foo);  
  
// ... many more options
```

```
var foo = {  
  baz: function(){  
    // ...  
  },  
  
  config: [ ... ]  
}  
  
var bar = {  
  config: { ... }  
};  
  
// ECMAScript "bind"  
bar.baz = foo.baz.bind(foo);  
  
// Underscore "bind"  
bar.baz = _.bind(foo.baz, foo);  
  
// ... many more options
```

In this example, both the source object and the target object have a "config" attribute. Each object needs to use that config attribute to store and retrieve certain bits of data without clobbering each other one, but I still want the "baz" function to be available directly on the "foo" object. To make this all work, I assign a bound version of the "baz" function to foo where the binding is set to the bar object. That way whenever the baz function is called from foo, it will always run in the context of the original source -- bar.

A Real Example

Ok, enough "foo, bar, baz" nonsense. Let's look at a real example of where I'm doing this: Marionette's use of Backbone.EventBinder. I want to bring the "bindTo", "unbindFrom" and "unbindAll" methods from the EventBinder in to Marionette's Application object, as one example. To do this while allowing the EventBinder to manage its own internal state and implementation details, I use the above technique of assigning the methods as bound functions:

```
Marionette.addEventBinder = function(target){

    // create the "source" of the functionality i need
    var eventBinder = new Marionette.EventBinder();

    // add the methods i need to the target object, binding them correctly
    target.bindTo = _.bind(eventBinder.bindTo, eventBinder);
    target.unbindFrom = _.bind(eventBinder.unbindFrom, eventBinder);
    target.unbindAll = _.bind(eventBinder.unbindAll, eventBinder);

};
```

```
// use the mixin method
var myApp = new Marionette.Application();
Marionette.addEventBinder(myApp);
```

```
Marionette.addEventBinder = function(target){

    // create the "source" of the functionality i need
    var eventBinder = new Marionette.EventBinder();

    // add the methods i need to the target object, binding them correctly
    target.bindTo = _.bind(eventBinder.bindTo, eventBinder);
    target.unbindFrom = _.bind(eventBinder.unbindFrom, eventBinder);
    target.unbindAll = _.bind(eventBinder.unbindAll, eventBinder);

};
```

```
// use the mixin method
var myApp = new Marionette.Application();
Marionette.addEventBinder(myApp);
```

Now when I call any of those three methods from my application instance, they still run in the context of my eventBinder object instance and they can access all of their internal state, configuration and behavior. But at the same time, I can worry less about whether or not the implementation details of the EventBinder are going to clobber the implementation details of the Application object. Since I'm being very explicit about which methods and attributes are brought over from the EventBinder, I can spend the small amount of cognitive energy that I need to determine whether or not the method I'm creating on the Application instance already exists. I don't have to worry about the internal details like `_eventBindings` and other bits because they are not going to be copied over.

Simplifying Mixins

Given the repetition of creating mixins like this, it should be pretty easy to create a function that can handle the grunt work for you. All you need to supply is a target object, a source object and a list of methods to copy. The mixin function can handle the rest:

```
// build a mixin function to take a target that receives the mixin,
// a source that is the mixin, and a list of methods / attributes to
// copy over to the target

function mixInto(target, source, methodNames){

  // ignore the actual args list and build from arguments so we can
  // be sure to get all of the method names
  var args = Array.prototype.slice.apply(arguments);
  target = args.shift();
  source = args.shift();
  methodNames = args;

  var method;
  var length = methodNames.length;
  for(var i = 0; i < length; i++){
    method = methodNames[i];

    // bind the function from the source and assign the
    // bound function to the target
    target[method] = _.bind(source[method], source);
  }

}
```

```
// make use of the mixin function
var myApp = new Marionette.Application();
mixInto(myApp, Marionette.EventBinder, "bindTo", "unbindFrom", "unbindAll");
```

This should be functionally equivalent to the previous code that was manually binding and assigning the methods. But keep in mind that this code is not robust at all. Bad things will happen if you get the source's method names wrong, for example. These little details should be handled in a more complete mixin function.

An Alternate Implementation: Closures

An alternate implementation for this can be facilitated without the use of a "bind" function. Instead, a simple closure can be set up around the source object, with a wrapper function that simply forwards calls to the source:

```
// build a mixin function to take a target that receives the mixin,
// a source that is the mixin, and a list of methods / attributes to
// copy over to the target

function mixInto(target, source, methodNames){

    // ignore the actual args list and build from arguments so we can
    // be sure to get all of the method names
    var args = Array.prototype.slice.apply(arguments);
    target = args.shift();
    source = args.shift();
    methodNames = args;

    var method;
    var length = methodNames.length;
    for(var i = 0; i < length; i++){
        method = methodNames[i];

        // build a function with a closure around the source
        // and forward the method call to the source, passing
        // along the method parameters and setting the context
        target[method] = function(){
            var args = Array.prototype.slice(arguments);
            source[method].apply(source, args);
        }
    }
}
```

```
// make use of the mixin function
var myApp = new Marionette.Application();
mixInto(myApp, Marionette.EventBinder, "bindTo", "unbindFrom", "unbindAll");
```

I'm not sure if this version is really "better" or not, but it would at least provide more backward compatibility support and fewer requirements. You wouldn't need to patch 'Function.prototype.bind' or use a third party shim or library for older browsers. It should work with any browser that supports JavaScript, though I'm not sure how far back it would go. Chances are, though, that any browser people are still using would be able to handle this, including -- dare I say it? -- IE6 (maybe... I think... I'm not going to test that, though :P)

Potential Drawbacks

As great as all this looks and sounds, there are some limitations and drawbacks -- and probably more than I'm even aware of right now.

We're directly manipulating the context of the functions with this solution. While this has certainly provided a measured benefit, it can be dangerous. There may be (are likely) times that you just don't want to mess with context -- namely when you aren't in control of the function context in the first place. Think about a jQuery function callback, for example. Typically, jQuery sets the context of a callback to the DOM element that was being manipulated and you might not want to mess with that.

In the case of my EventBinder with Marionette, I did run into a small problem with the context binding. The original version of the code would default the context of callback functions to the object that "bindTo" was called from. This meant the callback for "myView.bindTo(...)" would be run with "myView" as the context. When I switched over to the above code that creates the bound functions, the default context changed. Instead of being the view, the context was set to the EventBinder instance itself, just like our code told it to. This had an effect on how my Marionette views were behaving and I had to work around that problem in another way.

There certainly some potential drawbacks to this, as noted. But if you understand that danger and you don't try to abuse this for absolutely everything, I think this idea could work out pretty well as a way to produce a mixin system that really does favor composition over inheritance, and avoids the pitfalls of simple object extension.

73 : Beautiful Javascript Mixins

74 : CHAPTER ONE

75 : Beautiful Mixins

Angus Croll

Developers love to create overly complex solutions to things that aren't really problems.

Thomas Fuchs

In the beginning there was code, and the code was verbose, so we invented functions that the code might be reused. But after a while there were also too many functions, so we looked for a way to reuse those too. Developers often go to great lengths to apply "proper" reuse techniques to JavaScript. But sometimes when we try too hard to do the right thing, we miss the beautiful thing right in front of our eyes.

76 : Classical Inheritance

Many developers schooled in Java, C++, Objective-C, and Smalltalk arrive at JavaScript with an almost religious belief in the necessity of the class hierarchy as an organizational tool. Yet humans are not good at classification. Working backward from an abstract superclass toward real types and behaviors is unnatural and restrictive—a superclass must be created before it can be extended, yet classes closer to the root are by nature more generic and abstract and are more easily defined *after* we have more knowledge of their concrete subclasses. Moreover, the need to tightly couple types *a priori* such that one type is always defined solely in terms of another tends to lead to an overly rigid, brittle, and often ludicrous model ("Is a button a rectangle or is it a control? Tell you what, let's make `Button` inherit from `Rectangle`, and `Rectangle` can inherit from `Control`...no, wait a minute..."). If we don't get it right early on, our system is forever burdened with a flawed set of relationships—and on those rare occasions that, by chance or genius, we do get it right, anything but a minimal tree structure usually represents too complex a mental model for us to readily visualize.

Classical inheritance is appropriate for modeling existing, well-understood hierarchies—it's okay to unequivocally declare that a `FileStream` is a type of `InputStream`. But if the primary motivation is function reuse (and it usually is), classical hierarchies can quickly become gnarly labyrinths of meaningless subtypes, frustrating redundancies, and unmanageable logic.

77 : Prototypes

It's questionable whether the majority of behaviors can ever be mapped to objectively "right" classifications. And indeed, the classical inheritance lobby is countered by an equally fervent band of JavaScript *loyalists* who proclaim that JavaScript is a prototypal, not classical, language and is deeply unsuited to any approach that includes the word *class*. But what does "prototypal" mean, and how do prototypes differ from classes?

In generic programming terms, a prototype is an object that supplies base behavior to a second object. The second object can then extend this base behavior to form its own specialization. This process, also known as *differential inheritance*, differs from classical inheritance in that it doesn't require explicit typing (static or dynamic) or attempt to formally define one type in terms of another. While classical inheritance is planned reuse, true prototypal inheritance is opportunistic.

In general, when working with prototypes, one typically chooses not to categorize but to exploit alikeness.

Antero Taivalsaari, Nokia Research Center

In JavaScript, every object references a prototype object from which it can inherit properties. JavaScript prototypes are great instruments for reuse: a single prototype instance can define properties for an infinite number of dependent instances. Prototypes may also inherit from other prototypes, thus forming prototype chains.

So far, so good. But, with a view to emulating Java, JavaScript tied the `prototype` property to the constructor. As a consequence, more often than not, multilevel object inheritance is achieved by chaining constructor-prototype couplets. The standard implementation of a JavaScript prototype chain is too grisly to appear in a book about beautiful JavaScript, but suffice it to say, creating a new instance of a base prototype in order to define the initial properties of its inheritor is neither graceful nor intuitive. The alternative—manually copying properties between prototypes and then meddling with the `constructor` property to fake real prototypal inheritance—is even less becoming.

Syntactic awkwardness aside, constructor-prototype chaining requires upfront planning and results in structures that more closely resemble the traditional hierarchies of classical languages than a true prototypal relationship: constructors represent types (classes), each type is defined as a subtype of one (and only one) supertype, and all properties are inherited via this type chain. The ES6 `class` keyword merely formalizes the existing semantics. Leaving aside the gnarly and distinctly unbeautiful syntax characteristic in constructor-prototype chains, traditional JavaScript is clearly less prototypal than some would claim.

In an attempt to support less rigid, more opportunistic prototypes, the ES5 specification introduced `Object.create`. This method allows a prototype to be assigned to an object directly and therefore liberates JavaScript prototypes from constructors (and thus categorization) so that, in theory, an object can acquire behavior from any other arbitrary object and be free from the constraints of typecasting:

```
var circle = Object.create({
  area: function() {
    return Math.PI * this.radius * this.radius;
```

```
},
grow: function() {
    this.radius++;
},
shrink: function() {
    this.radius--;
}
});
```

`Object.create` accepts an optional second argument representing the object to be extended. Sadly, instead of accepting the object itself (in the form of a literal, variable, or argument), the method expects a full-blown `meta` property definition:

```
var circle = Object.create({
    /*see above*/
}, {
    radius: {
        writable:true, configurable:true, value: 7
    }
});
```

Assuming no one actually uses these unwieldy beasts in real code, all that remains is to manually assign properties to the instance after it has been created. Even then, the `Object.create` syntax still only enables an object to inherit the properties of a single prototype. In real scenarios, we often want to acquire behavior from multiple prototype objects: for example, a person can be an employee and a manager.

78 : Mixins

Fortunately, JavaScript offers viable alternatives to inheritance chaining. In contrast to objects in more rigidly structured languages, JavaScript objects can invoke any function property regardless of lineage. In other words, JavaScript functions don't need to be inheritable to be visible—and with that simple observation, the entire justification for inheritance hierarchies collapses like a house of cards.

The most basic approach to function reuse is manual delegation—any public function can be invoked directly via `call` or `apply`. It's a powerful and easily overlooked feature. However, aside from the verbosity of serial `call` or `apply` directives, such delegation is so convenient that, paradoxically, it sometimes actually works against structural discipline in your code—the invocation process is sufficiently ad hoc that in theory there is no need for developers to organize their code at all.

Mixins are a good compromise: by encouraging the organization of functionality along thematic lines they offer something of the descriptive prowess of the class hierarchy, yet they are light and flexible enough to avoid the premature organization traps (and head-spinning dizziness) associated with deeply chained, single-ancestry models. Better still, mixins require minimal syntax and play very well with unchained JavaScript prototypes.

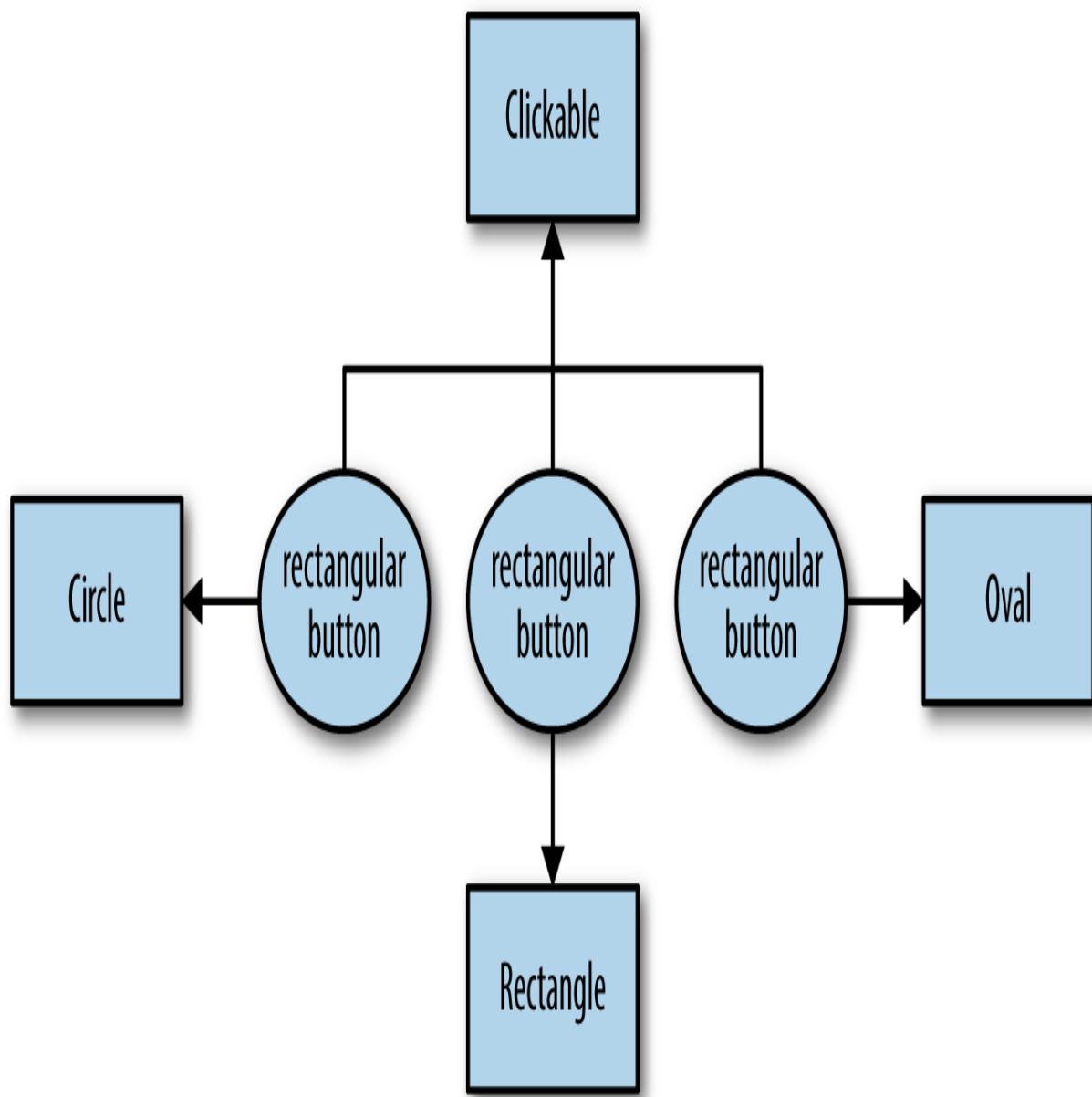
The Basics

Traditionally, a mixin is a class that defines a set of functions that would otherwise be defined by a concrete entity (a person, a circle, an observer). However, mixin classes are considered abstract in that they will not themselves be instantiated—instead, their functions are copied (or *borrowed*) by concrete classes as a means of inheriting behavior without entering into a formal relationship with the behavior provider.

Okay, but this is JavaScript, and we have no classes per se. This is actually a good thing because it means we can use objects (instances) instead, which offer clarity and flexibility: our mixin can be a regular object, a prototype, a function, whatever, and the mixin process becomes transparent and obvious.

The Use Case

I'm going to discuss a number of mixin techniques, but all the coding examples are directed toward one use case: creating circular, oval, or rectangular buttons (something that would not be readily possible using conventional classical inheritance techniques). Here's a schematic representation: square boxes represent mixin objects, and rounded boxes represent the actual buttons.



Classic Mixins

Scanning the first two pages returned from a Google search for "javascript mixin," I noticed the majority of authors define the mixin object as a full-blown constructor type with its function set defined in the prototype. This could be seen as a natural progression—early mixins were classes, and this is the closest thing JavaScript has to a class. Here's a circle mixin modeled after that style:

```

var Circle = function() {};
Circle.prototype = {
  area: function() {
    return Math.PI * this.radius * this.radius;
  },
  grow: function() {
    this.radius++;
  },
}
  
```

```

shrink: function() {
    this.radius--;
}
};

```

In practice, however, such a heavyweight mixin is unnecessary. A simple object literal will suffice:

```

var circleFns = {
    area: function() {
        return Math.PI * this.radius * this.radius;
    },
    grow: function() {
        this.radius++;
    },
    shrink: function() {
        this.radius--;
    }
};

```

Here's another mixin defining button behavior (for the sake of demonstration, I've substituted a simple log call for the working implementation of some function properties):

```

var clickableFns = {
    hover: function() {
        console.log('hovering');
    },
    press: function() {
        console.log('button pressed');
    },
    release: function() {
        console.log('button released');
    },
    fire: function() {
        this.action.fire();
    }
};

```

The extend Function

How does a mixin object get mixed into your object? By means of an `extend` function (sometimes known as *augmentation*). Usually `extend` simply copies (not clones) the mixin's functions into the receiving object. A quick survey reveals some minor variations in this implementation. For example, the Prototype.js framework omits a `hasOwnProperty` check (suggesting the mixin is not expected to have enumerable properties in its prototype).

chain), while other versions assume you want to copy only the mixin's prototype object. Here's a version that is both safe and flexible:

```
function extend(destination, source) {
  for (var key in source) {
    if (source.hasOwnProperty(key)) {
      destination[key] = source[key];
    }
  }
  return destination;
}
```

Now let's extend a base prototype with the two mixins we created earlier to make a `RoundButton.prototype`:

```
var RoundButton = function(radius, label) {
  this.radius = radius;
  this.label = label;
};

extend(RoundButton.prototype, circleFns);
extend(RoundButton.prototype, clickableFns);

var roundButton = new RoundButton(3, 'send');
roundButton.grow();
roundButton.fire();
```

Functional Mixins

If the functions defined by mixins are intended solely for the use of other objects, why bother creating mixins as regular objects at all? Isn't it more intuitive to think of mixins as processes instead of objects? Here are the circle and button mixins rewritten as functions. We use the context (`this`) to represent the mixin's target object:

```
var withCircle = function() {
  this.area = function() {
    return Math.PI * this.radius * this.radius;
  };
  this.grow = function() {
    this.radius++;
  };
  this.shrink = function() {
    this.radius--;
  };
};
```

```
var withClickable = function() {
  this.hover = function() {
    console.log('hovering');
  };
  this.press = function() {
    console.log('button pressed');
  };
  this.release = function() {
    console.log('button released');
  };
  this.fire = function() {
    this.action.fire();
  };
}
```

And here's our `RoundButton` constructor. We'll want to apply the mixins to `RoundButton.prototype`:

```
var RoundButton = function(radius, label, action) {
  this.radius = radius;
  this.label = label;
  this.action = action;
};
```

Now the target object can simply inject itself into the functional mixin by means of `Function.prototype.call`, cutting out the middleman (the `extend` function) entirely:

```
withCircle.call(RoundButton.prototype);
withClickable.call(RoundButton.prototype);

var button1 = new RoundButton(4, 'yes!', function() {return 'you said yes!'});
button1.fire(); //'you said yes!'
```

This approach feels right. Mixins as verbs instead of nouns; lightweight one-stop function shops. There are other things to like here too. The programming style is natural and concise: `this` always refers to the receiver of the function set instead of an abstract object we don't need and will never use; moreover, in contrast to the traditional approach, we don't have to protect against inadvertent copying of inherited properties, and (for what it's worth) functions are now cloned instead of copied.

Adding Options

This functional strategy also allows mixed in behaviors to be parameterized by means of an `options` argument. The following example creates a `withOval` mixin with a custom grow and shrink factor:

```

var withOval = function(options) {
    this.area = function() {
        return Math.PI * this.longRadius * this.shortRadius;
    };
    this.ratio = function() {
        return this.longRadius/this.shortRadius;
    };
    this.grow = function() {
        this.shortRadius += (options.growBy/this.ratio());
        this.longRadius += options.growBy;
    };
    this.shrink = function() {
        this.shortRadius -= (options.shrinkBy/this.ratio());
        this.longRadius -= options.shrinkBy;
    };
};

var OvalButton = function(longRadius, shortRadius, label, action) {
    this.longRadius = longRadius;
    this.shortRadius = shortRadius;
    this.label = label;
    this.action = action;
};

withButton.call(OvalButton.prototype);
withOval.call(OvalButton.prototype, {growBy: 2, shrinkBy: 2});

var button2 = new OvalButton(3, 2, 'send', function() {return 'message sent'});
button2.area(); //18.84955592153876
button2.grow();
button2.area(); //52.35987755982988
button2.fire(); //'message sent'

```

Adding Caching

You might be concerned that this approach creates additional performance overhead because we're redefining the same functions on every call. Bear in mind, however, that when we're applying functional mixins to prototypes, the work only needs to be done once: during the definition of the constructors. The work required for instance creation is unaffected by the mixin process, since all the behavior is preassigned to the shared prototype. This is how we support all function sharing on the twitter.com site, and it produces no noticeable latency. Moreover, it's worth noting that performing a classical mixin requires property getting as well as setting, and in fact functional mixins appear to benchmark quicker in the Chrome browser than traditional ones (although this is obviously subject to considerable variance).

That said, it is possible to optimize functional mixins further. By forming a closure around the mixins we can cache the results of the initial definition run, and the performance improvement is impressive. Functional mixins now easily outperform classic mixins in every browser.

Here's a version of the `withRectangle` mixin with added caching:

```
var withRectangle = (function() {
  function area() {
    return this.length * this.width;
  }
  function grow() {
    this.length++, this.width++;
  }
  function shrink() {
    this.length--, this.width--;
  }
  return function() {
    this.area = area;
    this.grow = grow;
    this.shrink = shrink;
    return this;
  };
})();

var RectangularButton = function(length, width, label, action) {
  this.length = length;
  this.width = width;
  this.label = label;
  this.action = action;
}

withClickable.call(RectangularButton.prototype);
withRectangle.call(RectangularButton.prototype);

var button3 =
  new RectangularButton(4, 2, 'delete', function() {return 'deleted'});
button3.area(); //8
button3.grow();
button3.area(); //15
button3.fire(); //'deleted'
```

Advice

One danger with any kind of mixin technique is that a mixin function will accidentally overwrite a property of the target object that, coincidentally, has the same name. Twitter's Flight framework, which makes use of functional mixins, guards against clobbering by temporarily locking existing properties (using the `writable` meta property) during the mixin process.

Sometimes, however, instead of generating a collision error we might want the mixin to augment the corresponding method on the target object. `advice` redefines a function by adding custom code before, after, or around the original implementation. The [Underscore framework](#) implements a basic function wrapper that enables `advice`:

```
button.press = function() {
  mylib.appendClass('pressed');
};

//after pressing button, reduce shadow (using underscore)
button.pressWithShadow = _.wrap(button.press, function(fn) {
  fn();
  button.reduceShadow();
})
```

The Flight framework takes this a stage further: now the `advice` object is itself a [functional mixin](#) that can be mixed into target objects to enable advice for subsequent mixins.

Let's use this `advice` mixin to augment our rectangular button actions with shadow behavior. First we apply the `advice` mixin, followed by the two mixins we used earlier:

```
withAdvice.call(RectangularButton.prototype);
withClickable.call(RectangularButton.prototype);
withRectangle.call(RectangularButton.prototype);
```

And now the `withShadow` mixin that will take advantage of the `advice` mixin:

```
var withShadow = function() {
  this.after('press', function() {
    console.log('shadow reduced');
  });
  this.after('release', function() {
    console.log('shadow reset');
  });
};

withShadow.call(RectangularButton.prototype);
var button4 = new RectangularButton(5, 4);
```

```
button4.press(); //'button pressed' 'shadow reduced'  
button4.release(); //'button released' 'shadow reset'
```

The Flight framework sugarcoats this process. All flight components get `withAdvice` mixed in for free, and there's also a `defineComponent` method that accepts multiple mixins at a time. So, if we were using Flight we could further simplify the process (in Flight, constructor properties such as rectangle dimensions are defined as `attr` properties in the mixins):

```
var RectangularButton =  
  defineComponent(withClickable, withRectangle, withShadow);  
  
var button5 = new RectangularButton(3, 2);  
button5.press(); //'button pressed' 'shadow reduced'  
button5.release(); //'button released' 'shadow reset'
```

With `advice` we can define functions on mixins without having to guess whether they're also implemented on the target object, so the mixin can be defined in isolation (perhaps by another vendor). Conversely, `advice` allows us to augment third-party library functions without resorting to monkey patching.

79 : Wrapup

When possible, cut with the grain. The grain tells you which direction the wood *wants* to be cut. If you cut against the grain, you're just making more work for yourself, and making it more likely you'll spoil the cut.

Charles Miller¹

As programmers, we're encouraged to believe that certain techniques are indispensable. Ever since the early 1990s, object-oriented programming has been hot, and classical inheritance has been its poster child. It's not hard to see how a developer eager to master a new language would feel under considerable pressure to fit classical inheritance under the hood.

But peer pressure is not an agent of beautiful code, and neither is serpentine logic. When you find yourself writing `Circle.prototype.constructor = Circle`, ask yourself if the pattern is serving you, or you're serving the pattern. The best patterns tread lightly on your process and don't interfere with your ability to use the full power of the language.

By repeatedly defining an object solely in terms of another, classical inheritance establishes a series of tight couplings that glue the hierarchy together in an orgy of mutual dependency. Mixins, in contrast, are extremely agile and make very few organizational demands on your codebase—mixins can be created at will, whenever a cluster of common, shareable behavior is identified, and all objects can access a mixin's functionality regardless of their role within the overall model. Mixin relationships are entirely ad hoc: any combination of mixins can be applied to any object, and objects can have any number of mixins applied to them. Here, at last, is the opportunistic reuse that prototypal inheritance promised us.

¹ See Charles Miller's entire post at his blog, [The Fishbowl](#).

80 : JavaScript and Functional Programming

<https://bethallchurch.github.io/JavaScript-and-Functional-Programming/>

This is a write up of my notes (plus some further research) from Kyle Simpson's excellent class **Functional-Light JavaScript** (slides [here](#)) on 29 of June, 2016.

Object-oriented programming has long been the dominant paradigm in JavaScript. Recently, however, there has been a growing interest in functional programming. Functional programming is a style of programming that emphasises minimising the number of changes to a program's state (known as *side effects*). To this end, it encourages the use of *immutable* data and *pure* (side effect-free) functions. It also favours a *declarative* style and encourages the use of well-named functions that allow you to write programs by describing what you want to happen, with the implementation details packaged away out of immediate sight.

Although there are tensions between object-oriented and functional approaches, they are not mutually exclusive. JavaScript has the tools to support both paradigms. Even without using it exclusively as a functional language, there are concepts and best practices from the functional approach that we can use to make our own code cleaner, more readable, and easier to reason about.

Minimise Side Effects

A *side effect* is a change that is not local to the function that caused it. A function might do something like manipulate the DOM, modify the value of a variable in a higher level scope or write data to a database. The results of these actions are side effects.

```
// A function with a side effect
var x = 10;

const myFunc = function ( y ) {
  x = x + y;
};

myFunc( 3 );
console.log( x ); // 13

myFunc( 3 );
console.log( x ); // 16
```

Side effects are not inherently evil. A program that produced no side effects would not affect the world, and so there would be no point to it (other than perhaps as a theoretical curiosity). They are, however, dangerous and should be avoided wherever they are not strictly necessary.

When a function produces a side effect you have to know more than just its inputs and output to understand what that function does. You need to know about the context and history of the state of the program, which makes the

function harder to understand. Side effects can cause bugs by interacting in unpredictable ways, and the functions that produce them are harder to test thanks to their reliance on the context and history of the program's state.

Minimising side effects is such a fundamental principle of functional programming that most of the following sections can be understood as outlining techniques to avoid them.

Treat Data as Immutable

A mutation is an in-place change to a value. An immutable value is a value that, once created, can never be changed. In JavaScript, simple values like numbers, strings and booleans are immutable. However, data structures like objects and arrays are mutable.

```
// the push method mutates the array it's called on
const x = [1, 2];
console.log( x ); // [1, 2]

x.push( 3 );
console.log( x ); // [1, 2, 3]
```

Why would we want to avoid mutating data?

A mutation is a side effect. The fewer things that change in a program, the less there is to keep track of, and the result is a simpler program.

JavaScript only has limited tools available to enforce immutability on data structures like objects and arrays. Object immutability can be enforced with `Object.freeze`, but only one level deep:

```
const frozenObject = Object.freeze( { valueOne : 1, valueTwo : { nestedValue : 1 } } );
frozenObject.valueOne = 2; // not allowed
frozenObject.valueTwo.nestedValue = 2; // allowed!
```

There are, however, several excellent libraries out there that solve this issue, the most well-known of which is [Immutable](#).

For most applications, using a library to enforce immutability is overkill. In most cases you will gain most of the benefits of immutable data simply by treating data as though it were immutable.

Avoiding Mutations: Arrays

Array methods in JavaScript can broadly be divided into [mutator methods](#) and non-mutator methods. Mutator methods should be avoided where possible.

For example, `concat` can be used instead of `push`. `push` mutates the original array, whereas `concat` returns a new array comprised of the array it was called on and the array provided as its argument, leaving the original array intact.

```
// push mutates arrays.
const arrayOne = [1, 2, 3];
arrayOne.push( 4 );

console.log( arrayOne ); // [1, 2, 3, 4]

// concat creates a new array and leaves the original unchanged.
const arrayTwo = [1, 2, 3];
const arrayThree = arrayTwo.concat([ 4 ]);

console.log( arrayTwo ); // [1, 2, 3]
console.log( arrayThree ); // [1, 2, 3, 4]
```

Other useful non-mutator array methods include [map](#), [filter](#), and [reduce](#).

Avoiding Mutations: Objects

Instead of directly editing objects, you can use [Object.assign](#), which copies the properties of source objects into a target object and then returns it. If you always use an empty object as the target object, you can use [Object.assign](#) to avoid directly editing objects.

```
const objectOne = { valueOne : 1 };
const objectTwo = { valueTwo : 2 };

const objectThree = Object.assign( {}, objectOne, objectTwo );

console.log( objectThree ); // { valueOne : 1, valueTwo : 2 }
```

Note on const

`const` is useful, but it does not make your data immutable. It prevents your variables from being reassigned. These two things should not be conflated.

```
const x = 1;
x = 2; // not allowed

const myArray = [1, 2, 3];
myArray = [0, 2, 3]; // not allowed

myArray[0] = 0; // allowed!
```

Write Pure Functions

A *pure function* is a function that does not change the program's state and does not produce an observable side effect. The output of a pure function relies solely on its input values. Wherever and whenever a pure function is called, its return value will always be the same when given the same inputs.

Pure functions are an important tool for keeping side effects to a minimum. In addition, their indifference to context make them highly testable and reusable.

`myFunc` from the section on side effects is an impure function: note how it's called twice with the same input and gives a different result each time. It could, however, be re-written as a pure function:

```
// Make the global variable local.
const myFunc = function ( y ) {
  const x = 10;
  return x + y;
}

console.log(myFunc( 3 )); // 13
console.log(myFunc( 3 )); // 13
```

```
// Pass x as an argument.
const x = 10;

const myFunc = function ( x, y ) {
  return x + y;
}

console.log(myFunc( x, 3 )); // 13
console.log(myFunc( x, 3 )); // 13
```

Ultimately, your program will always produce some side effects. Where they occur they should be handled carefully and their effects constrained and contained as much as possible.

Write Function-Generating Functions

Find someone who has never programmed before and ask them to guess what the following pieces of code do.

Example One:

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

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

Example Two:

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

const print = function ( input ) {
    console.log( input );
};

numbers.forEach( print );
```

Everyone I've tried this test on has had more luck with the second example. Example One exemplifies an *imperative* approach to printing out a list of numbers. Example Two exemplifies a *declarative* approach. By packaging away the details of how to loop through an array and how to print to the console into the functions `forEach` and `print`, respectively, we can express *what* we want our program to do without needing to go into *how* to do it. This makes for highly readable code. The last line of Example Two is very close to English.

Adopting this approach involves writing a lot of functions. This process can be made [DRY](#)-er by writing functions to generate new functions from existing ones.

There are two features of JavaScript in particular that make this kind of function generation possible. The first is *closure*. Closure is the ability of functions to access variables from containing scopes, even when those scopes no longer exist. The second is that JavaScript treats functions as values. This makes it possible to write *higher-order functions*, which are functions that take other functions as arguments and / or return functions as their output.

Combined, these features allow you to write functions that return other functions which "remember" the arguments passed to the function that generated them, and are able to use those arguments elsewhere in the program.

Function Composition

Functions can be combined to form new functions through *function composition*. Here is an example:

```
// The function addThenSquare is made by combining the functions add and square.
const add = function ( x, y ) {
    return x + y;
};

const square = function ( x ) {
    return x * x;
};

const addThenSquare = function ( x, y ) {
    return square(add( x, y ));
};
```

You may find yourself repeating this pattern of generating a more complex function from smaller functions. Often it's more efficient to write a function that does the composition for you:

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

const square = function ( x ) {
    return x * x;
};

const composeTwo = function ( f, g ) {
    return function ( x, y ) {
        return g( f ( x, y ) );
    };
};

const addThenSquare = composeTwo( add, square );
```

You could go even further and write a more general composition functions:

```
// This version of composeTwo can accept any number of arguments for the initial
function.

const composeTwo = function ( f, g ) {
    return function ( ...args ) {
        return g( f( ...args ) );
    };
};

// composeMany can accept any number of functions as well as any number of arguments for
the
// initial function.

const composeMany = function ( ...args ) {
    const funcs = args;
    return function ( ...args ) {
        funcs.forEach(( func ) => {
            args = [func.apply( this, args )];
        });
        return args[0];
    };
};
```

The exact form of your composition function will depend on the level of generality you need and the kind of API you prefer.

Partial Function Application

Partial function application is the process of fixing the value of one or more of a function's arguments, and then returning the function to be fully invoked later.

In the following example, `double`, `triple`, and `quadruple` are partial applications of `multiply`.

```
const multiply = function ( x, y ) {
  return x * y;
};

const partApply = function ( fn, x ) {
  return function ( y ) {
    return fn( x, y );
  };
};

const double = partApply( multiply, 2 );
const triple = partApply( multiply, 3 );
const quadruple = partApply( multiply, 4 );
```

Currying

Currying is the process of translating a function that takes multiple arguments into a series of functions that each take one argument.

```
const multiply = function ( x, y ) {
  return x * y;
};

const curry = function ( fn ) {
  return function ( x ) {
    return function ( y ) {
      return fn( x, y );
    };
  };
};

const curriedMultiply = curry( multiply );

const double = curriedMultiply( 2 );
```

```

const triple = curriedMultiply( 3 );
const quadruple = curriedMultiply( 4 );

console.log(triple( 6 )); // 18

```

Currying and partial application are conceptually similar (and you'll probably never need both), but still distinct. The main difference is that currying will always produce a nested chain of functions that each accept only one argument, whereas partial application can return functions that accept more than one argument. This distinction is clearer when you compare their effects on functions that accept at least three arguments:

```

const multiply = function ( x, y, z ) {
    return x * y * z;
};

const curry = function ( fn ) {
    return function ( x ) {
        return function ( y ) {
            return function ( z ) {
                return fn( x, y, z );
            };
        };
    };
};

const partApply = function ( fn, x ) {
    return function ( y, z ) {
        return fn( x, y, z );
    };
};

const curriedMultiply = curry( multiply );
const partiallyAppliedMultiply = partApply( multiply, 10 );

console.log(curriedMultiply( 10 )( 5 )( 2 )); // 100
console.log(partiallyAppliedMultiply( 5, 2 )); // 100

```

Recursion

A **recursive** function is a function that calls itself until it reaches a base condition. Recursive functions are highly declarative. They're also elegant and very satisfying to write!

Here's an example of a function that recursively calculates the factorial of a number:

```

const factorial = function ( n ) {
    if ( n === 0 ) {
        return 1;
    }
    return n * factorial( n - 1 );
};

console.log(factorial( 10 )); // 3628800

```

Using recursive functions in JavaScript requires some care. Every function call adds a new call frame to the call stack, and that call frame is popped off the call stack when the function returns. Recursive functions call themselves before they return, and so it's very easy for a recursive function to exceed the limits of the call stack and crash the program.

However, this can be avoided with *tail call optimisation*.

Tail Call Optimisation

A tail call is a function call that is the last action of a function. Tail call optimisation is when the language compiler recognises tail calls and reuses the same call frame for them. This means that if you write recursive functions with tail calls, the limits of the call stack will never be exceeded by them as it will reuse the same frame over and over.

Here is the recursive function from above rewritten to take advantage of tail call optimisation:

```

const factorial = function ( n, base ) {
    if ( n === 0 ) {
        return base;
    }
    base *= n;
    return factorial( n - 1, base );
};

console.log(factorial( 10, 1 )); // 3628800

```

Support for proper tail calls is included in the [ES2015 language specification](#), but is currently unsupported in most environments. You can check whether you can use them [here](#).

Summary

Functional programming contains many ideas that we can use to make our own code simpler and better. Pure functions and immutable data minimise the hazards of side effects. Declarative programming maximises code readability. These are important tools that should be embraced in the fight against complexity.

Corrections

- [09-09-2016] Forgot to return the innermost function in `partApply`, in the section on partial function application. Thank you to Richard Bultitude for spotting the mistake!

81 : Functional programming with Javascript

<https://stephen-young.me.uk/2013/01/20/functional-programming-with-javascript.html>

This post will discuss what functional programming is, how it compares to object-oriented programming and how you can program functionally in Javascript.

Themes of functional programming

Throughout the examples and concepts discussed in this post there are three key themes:

- Computation as the application of functions
- Statelessness
- Avoiding side effects

Computation as the application of functions

Object-oriented programming also uses functions but functional programming is unique in that functions are values (the technical term for this is that functions are "first class"). You can store functions in variables, return them as the value of other functions and pass them in as arguments to other functions. As a result functional programs produce answers by applying these functions in different ways.

Statelessness

Programming languages have the notion of "state". This is a snapshot of a program's current environment: all the variables that have been declared, functions created and what is currently being executed.

An expression in a programming language can be "stateful" or "stateless". A stateful expression is one that changes a program's current environment. A very simple example in Javascript would be incrementing a variable by 1:

```
var number = 1;
var increment = function() {
    return number += 1;
};
increment();
```

A stateless expression, on the other hand, is one that does not modify a program's environment:

```
var number = 1;
var increment = function(n) {
    return n + 1;
};
increment(number);
```

Unlike the previous example when we call `increment` here we do not change the value of `number` we merely return a new value.

Object-oriented programming works by changing program state to produce answers. Conversely, functional programming produces answers through stateless operations. The clearest contrast can be seen in how both methodologies handle looping. An object-oriented approach would be to use a `for` loop:

```
var loop = function() {
    for (var x = 0; x < 10; x += 1) {
        console.log(x);
    }
};

loop();
```

The loop produces its results by constantly changing the value of `x`. A functional approach would involve recursion (covered in more detail later on):

```
var loop = function(n) {
    if (n > 9) {
        console.log(n);
        return;
    } else {
        console.log(n);
        loop(n + 1);
    }
};

loop(0);
```

In this case we get our results by calling the `loop` function each time with a new value rather than modifying the program's state.

So, functional programming requires us to write stateless expressions and keep our data immutable.

Avoiding side effects

When a function executes it can change a program's state aside from returning a value:

```
var number = 2;
var sideEffect = function(n) {
    number = n * 3;
    return number * n;
};
sideEffect(3);
```

The side effect of this function is that it changes the value of `number` each time it executes. Functional programming promotes the use of "pure functions": functions that have no side effects. In the example above changing the value of `number` would be disallowed under this paradigm. If we can always write functions that have no side effects our program becomes "referentially transparent". This means if we input the same value into a program we will always get the same result.

The example above is not referentially transparent because if we always called `sideEffect` with 3 we would get a different result each time.

Pure functional programming in Javascript

If we wanted to program in a pure functional manner in Javascript we can set the following rules:

- No assignment i.e. changing the value of something already created with = (var statements with = are fine)
- No for or while loops (use recursion instead)
- Freeze all objects and arrays (since modifying an existing object or array would be a stateful expression)
- Disallow Date (since it always produces a new value no matter what the inputs are)
- Disallow Math.random (same reason as Date)

Why program functionally?

After looking at the themes of functional programming and the rules we need to set to ensure we can write pure programs, it may seem like we need to jump through a lot of hoops just to get anything done. Indeed functional programming does look inferior when faced with environments that are constantly in flux. This is particularly true of the DOM (one of Javascript's main interactions) which can constantly change through insertions, deletions, animations etc. In fact the creators of the Haskell programming language developed [Monads](#) as a way of managing changing state but retaining functional purity (Douglas Crockford also gave a talk on [implementing Monad's in Javascript](#)).

The big advantage functional programming gives us is that we can reason about and test our programs much more easily. This can help us to be more productive, reduce bugs and write better software. It is not always possible to write everything in a purely functional manner but it is something to strive for. John Carmack of Id Software wrote a good article discussing the [practical considerations of functional programming](#).

Concepts in functional programming

First class and higher-order functions

We've already discussed first class functions and how they can be stored and passed around as values. Of course these are very simple expressions in Javascript:

```
var funcAsValue = function(y) {
    return y * 4;
};

var returnFunc = function(z) {
```

```

        return function(a) {
            return a * z;
        };
    };

    var funcAsArgument = function(a, func) {
        return func(a);
    };
}

```

The latter two expressions are examples of "higher-order" functions. These are functions that either return functions as their result or take other functions as arguments.

Closures

In Javascript functions can access variables in their outer scope:

```

var func = function() {
    var a = 3;
    return function(b) {
        return b * a;
    };
};

var closure = func();
closure(4) // 12

```

In this example the function returned by `closure` can still access `a` since it is in its outer scope.

When functions return they "take a picture" of their current environment which they can read the latest values from. So, in essence, closures are functions which, as well as having an expression body and taking arguments, also record an outer scope or environment. Importantly, closures always have access to the latest value of a variable. If we rewrite the example above to always increment `a` we can see how this is the case (although this breaks our functional principles):

```

var func = function() {
    var a = 3;
    return function(b) {
        a = ++a;
        return b * a;
    };
};

var closure = func();

```

```
closure(4) // 16
closure(5) // 25
```

Recursion

A recursive function is a function that quite simply calls itself:

```
var recursive = function(n) {
    if (n < 1) {
        return n;
    } else {
        return recursive(n - 1);
    }
};

recursive(1); // 0
```

Functional programming favours recursion for looping rather than traditional `while` or `for` loops since we can maintain statelessness and avoid side effects. For example, to compute the [factorial](#) of value `n` statefully we might do:

```
var factorial = function(n) {
    var result = 1;
    for (var x = 1; x <= n; x += 1) {
        result = x * result;
    }
    return result;
};

factorial(5) // 120
```

A stateless approach would be:

```
var recursiveFactorial = function(n) {
    if (n < 2) {
        return n;
    } else {
        return n * recursiveFactorial(n - 1);
    }
};

recursiveFactorial(5) // 120
```

Tail call optimisation

Most functional programming languages implement "tail call optimisation" for recursive functions. In order to understand this it would be best to begin by looking at what happens when recursion is not optimised in this manner.

Each time a function is called additional space is taken up on the call stack. If you're using a Javascript environment that supports `console.trace()` logging you can see the call stack for the `recursiveFactorial` function above:

```
var recursiveFactorial = function(n) {
    if (n < 2) {
        console.trace();
        return n;
    } else {
        console.trace();
        return n * recursiveFactorial(n - 1);
    }
};

recursiveFactorial(5);
```

This stack is built up so when a value is returned the program knows where to go back to.

A "tail call" is a call to a function that occurs as the last action of another function. Our recursive calls above are not tail calls since the final action of the function is to multiply what is returned by `recursiveFactorial` by `n`. Let's change this around so they are:

```
var recursiveFactorial = function(n) {
    var recur = function(x, y) {
        if (y === n) {
            console.trace()
            return x * y;
        } else {
            console.trace()
            return recur(x * y, y + 1);
        }
    }
    return recur(1, 1);
};

recursiveFactorial(5);
```

Since Javascript does not implement tail call optimisation we can see in the console that we build up the stack each time `recur` is called.

In programming languages that do support tail call optimisation a large call stack like this would not be built up. The language can determine that if a call is in tail position extra space does not need to be created for it on the stack each time it is called. A [Wikipedia example illustrates the procedure quite nicely](#).

Unfortunately since Javascript does not implement tail call optimisation recursion is much slower than traditional `for` or `while` loops. However, in [ECMAScript 6 tail calls will be optimised](#).

Continuation passing style

For the time being we can still write loops in a recursive manner without taking up extra space on the call stack.

This can be done with continuation passing style (CPS).

In CPS we use "continuations" (functions) to represent the next stage of a computation. A continuation is passed as an argument to a higher-order function which is then called by that function when a previous operation is complete. This pattern is very common in Javascript. AJAX requests and Node.js use CPS so that computation can be performed asynchronously. For example:

```
$.get('url', function(data) {
    console.log(data);
});
```

In this example we pass a continuation to our `get` method when the data has been retrieved.

A more complex example is factorial in CPS:

```
var CPSFactorial = function(n, cont) {
    if (n < 2) {
        return cont(n);
    } else {
        var new_cont = function(v) {
            var result = v * n;
            return cont(result);
        };
        return CPSFactorial(n - 1, new_cont);
    }
};

CPSFactorial(5, function(v) {
    return v;
});
```

With our original recursive factorial example, once we reached the end of the "loop" each result had to be returned to its original caller meaning the program had to go all the way back through the stack to get the final result.

On the other hand, with CPS once the "loop" is at an end the function returns the result of applying the continuation with 1. This in turn applies the result of that function to another continuation and so on until we get the final result. So, rather than going all the way back through the call stack the `CPSFactorial` function creates new functions that each represent the next stage of the computation.

Thunks and trampolines

With CPS there is still some recursion: `CPSFactorial` calls itself several times during the computation. We can avoid recursion completely with "thunks" and "trampolines".

A thunk is much like a continuation except that it is stored as a data structure for use at a later time. A trampoline manages the execution and return values of thunks (so called because it bounces thunks around). In Javascript we can represent a thunk as follows:

```
var thunk = function (f, lst) {
    return { tag: "thunk", func: f, args: lst };
};

var thunkValue = function (x) {
    return { tag: "value", val: x };
};
```

In this case our `thunk` stores a function and a list of arguments to be executed at a later time. Thanks to closures thunks have a record of their environment when they are stored. The `thunkValue` function simply returns the final value of our computation.

We can handle all of the thunks we create with a trampoline:

```
var trampoline = function (thk) {
    while (true) {
        if (thk.tag === "value") {
            return thk.val;
        }
        if (thk.tag === "thunk") {
            thk = thk.func.apply(null, thk.args);
        }
    }
};
```

Here `trampoline` takes a thunk and calls its function with its arguments if it has the correct tag otherwise it returns a value. This example breaks our rule of statelessness (since we reassign `thk` to a new value each time) so we could write this as follows:

```
var trampoline = function (thk) {
    if (thk.tag === "value") {
        return thk.val;
    }
    if (thk.tag === "thunk") {
        return trampoline(thk.func.apply(null, thk.args));
    }
};
```

However, we are then back to using recursion again.

We can use thunks and a trampoline to calculate factorial:

```
var thunk = function (f, lst) {
    return { tag: "thunk", func: f, args: lst };
};

var thunkValue = function (x) {
    return { tag: "value", val: x };
};

var thunkFactorial = function(n, cont) {
    if (n < 2) {
        return thunk(cont, [n]);
    } else {
        var new_cont = function(v) {
            var result = v * n;
            return thunk(cont, [result]);
        };
        return thunk(thunkFactorial, [n - 1, new_cont]);
    }
};

var trampoline = function (thk) {
    while (true) {
        if (thk.tag === "value") {
            return thk.val;
        }
        if (thk.tag === "thunk") {
            thk = thk.func.apply(null, thk.args);
        }
    }
};
```

```
trampoline(thunkFactorial(5, thunkValue));
```

In this example we avoid recursion completely since we're always creating new thunks to be executed rather than functions invoking themselves.

A fun (but pointless) implementation of this is a [jQuery plugin that can act as a replacement for jQuery's each method](#).

Optimisations in functional programming

Partial application (currying)

Since Javascript has higher-order functions we can call functions with some of its arguments pre-filled. This is known as partial application or currying. The simplest example of this is Javascript's `bind` method. `bind` simply returns a new function with a custom context and arguments pre-filled:

```
var bindExample = function(a, b) {
    return a / b;
};

var test = bindExample.bind(null, 3);

test(4); // 0.75
```

Here `test` creates a new function which when invoked already has the `a` argument filled in as 3.

`bind` is an ECMAScript 5 method so it is not available in all Javascript environments. We can write a custom `bind` as follows:

```
var bind = function(a, func) {
    return function() {
        return func.apply(null, [a].concat(Array.prototype.slice.call(arguments)));
    };
};

var bindExample = function(a, b) {
    return a / b;
};

var test = bind(3, bindExample);

test(4); // 0.75
```

Exactly like the ES5 method, the `bind` function in this case returns a new function which when called applies the function we passed into it with an argument pre-filled.

Memoisation

Functions that are expensive to run can be optimised with memoisation. This involves using a closure to cache the results of previous calls to the function. A very simple example would be:

```
var memoise = function(func) {
    var cache = {};
    return function(arg) {
        if (arg in cache) {
            console.log(arg + ' in cache');
            return cache[arg];
        } else {
            console.log(arg + 'not in cache');
            return cache[arg] = func(arg);
        }
    };
};

var memo = memoise(function(n) {
    return n * 2;
});

memo(1) // 2 (1 not in cache)
memo(1) // 2 (1 in cache)
```

Here we pass a function to `memoise` whose results we wish to cache. When we call `memo` it will either retrieve the value from the cache or it will call the memoised function and store its value in the cache. It's worth noting that in order for us to be able to memoise a function it must be referentially transparent otherwise no results will ever be cached.

Of course this example breaks our rule of avoiding side effects. Unfortunately it is quite problematic to use memoisation without changing state since we have to keep updating our cache. It is easier in a language like Haskell that can use [lazy evaluation to memoise](#).

Memoisation is most useful where a function is likely to be calculating many of the same results such as the fibonacci sequence.

Conclusion

This article has looked at functional programming's key themes, its main concepts and optimisations the paradigm can make.

Whilst functional programming has many benefits including more rigorous code and enhancements to productivity, it can be difficult at times to write programs in a purely functional manner in Javascript. Consequently, some pragmatism is required.

In a future post I hope to discuss how all of this can be tied together to create a functional application.

82 : Understanding JavaScript's async await

<https://ponyfoo.com/articles/understanding-javascript-async-await>

83 : Using Promises

Let's suppose we had code like the following. Here I'm wrapping an HTTP request in a `Promise`. The promise fulfills with the `body` when successful, and is rejected with an `err` reason otherwise. It pulls the HTML for a random article from this blog every time.

```
var request = require('request');

function getRandomPonyFooArticle () {
  return new Promise((resolve, reject) => {
    request('https://ponyfoo.com/articles/random', (err, res, body) => {
      if (err) {
        reject(err); return;
      }
      resolve(body);
    });
  });
}
```

Typical usage of the promised code shown above is below. There, we build a promise chain transforming the HTML page into Markdown of a subset of its DOM, and then into Terminal-friendly output, to finally print it using `console.log`. Always remember to add `.catch` handlers to your promises.

```
var hget = require('hget');
var marked = require('marked');
var Term = require('marked-terminal');

printRandomArticle();

function printRandomArticle () {
  getRandomPonyFooArticle()
    .then(html => hget(html, {
      markdown: true,
      root: 'main',
      ignore: '.at-subscribe,.mm-comments,.de-sidebar'
    }))
    .then(md => marked(md, {
      renderer: new Term()
    }))
    .then(txt => console.log(txt))
    .catch(reason => console.error(reason));
}
```

When ran, that snippet of code produces output as shown in the following screenshot.

The screenshot shows a terminal window titled 'nico@CommandCenter: ~/dev/read-ponyfoo (zsh)'. The window contains the following text:

```
~/dev/read-ponyfoo
» read-ponyfoo
# A Less Convoluted Event Emitter Implementation
I believe that the event emitter implementation in Node could be made way better by providing a way to access the functionality directly without using prototypes. This would allow to simply extend any object, such as {}, or { pony: 'foo' }, with event emitting capabilities. Prototypes enforce limitations for little gain, and that's what we avoid by going around it and merely adding methods on existing objects, without any prototypal inheritance going on.

In this article I'll explore the implementation that made its way into contra (https://github.com/bevacqua/contra), an asynchronous flow control library I designed.

Event emitters usually support multiple types of events, rather than a single one. Let's implement, step by step, our own function to create event emitters, or improve existing objects as event emitters. In a first step, I'll either return the object unchanged, or create a new object if one wasn't provided.

function emitter (thing) {
  if (!thing) {
    thing = {};
  }
  return thing;
}

Being able to use multiple event types is powerful and only costs us an object to store the mapping of event types to event listeners. Similarly, we'll use an array for each event type, so that we can bind multiple event listeners to each event type. I'll also add a simple function which registers event listeners while I'm at it.

function emitter (thing) {
  var events = {};
```

Screenshot

That code was "*better than using callbacks*", when it comes to how sequential it feels to read the code.

84 : Using Generators

We've already explored generators as a way of making the `html` available in a synthetic "synchronous" manner [in the past](#). Even though the code is now somewhat synchronous, there's quite a bit of wrapping involved, and generators may not be the most straightforward way of accomplishing the results that we want, so we might end up sticking to Promises anyways.

```
let request = require('request');
let hget = require('hget');
let marked = require('marked');

function getRandomPonyFooArticle (gen) {
  var g = gen();
  g.next(); // Important! Otherwise stops execution on `var html = yield`.

  request('https://ponyfoo.com/articles/random', (err, res, body) => {
    if (err) {
      g.throw(err); return;
    }

    g.next(body);
  });
}

getRandomPonyFooArticle(function* printRandomArticle () {
  var html = yield;

  var md = hget(html, {
    markdown: true,
    root: 'main',
    ignore: '.at-subscribe,.mm-comments,.de-sidebar'
  });

  var txt = marked(md, {
    renderer: new marked.Renderer()
  });

  console.log(txt);
});
```

Keep in mind you should wrap the `yield` call in a `try / catch` block to preserve the error handling we had added when using promises.

Needless to say, using generators like this *doesn't scale well*. Besides involving an unintuitive syntax into the mix, your iterator code will be highly coupled to the generator function that's being consumed. That means you'll have to change it often as you add new `await` expressions to the generator. A better alternative is to use the upcoming **Async Function**.

85 : Using async / await

When *Async Functions* finally hit the road, we'll be able to take our `Promise`-based implementation and have it take advantage of the synchronous-looking generator style. Another benefit in this approach is that you won't have to change `getRandomPonyFooArticle` at all, as long as it returns a promise, it can be awaited.

Note that `await` may only be used in functions marked with the `async` keyword. It works similarly to generators, suspending execution in your context until the promise settles. If the awaited expression isn't a promise, its casted into a promise.

```
read();

async function read () {
  var html = await getRandomPonyFooArticle();
  var md = hget(html, {
    markdown: true,
    root: 'main',
    ignore: '.at-subscribe,.mm-comments,.de-sidebar'
  });
  var txt = marked(md, {
    renderer: new Term()
  });
  console.log(txt);
}
```

Again, – and just like with generators – keep in mind that you should wrap `await` in `try / catch` so that you can capture and handle errors in awaited promises from within the `async` function.

Furthermore, an *Async Function* always returns a `Promise`. That promise is rejected in the case of uncaught exceptions, and it's otherwise resolved to the return value of the `async` function. This enables us to invoke an `async` function and mix that with regular promise-based continuation as well. The following example shows how the two may be combined ([see Babel REPL](#)).

```
async function asyncFun () {
  var value = await Promise
    .resolve(1)
    .then(x => x * 3)
    .then(x => x + 5)
    .then(x => x / 2);
  return value;
}
asyncFun().then(x => console.log(`x: ${x}`));
// <- 'x: 4'
```

Going back to the previous example, that'd mean we could `return txt` from our `async` `read` function, and allow consumers to do continuation using promises or yet another *Async Function*. That way, your `read` function becomes only concerned with pulling terminal-readable Markdown from a random article on Pony Foo.

```
async function read () {  
  var html = await getRandomPonyFooArticle();  
  var md = hget(html, {  
    markdown: true,  
    root: 'main',  
    ignore: '.at-subscribe,.mm-comments,.de-sidebar'  
  });  
  var txt = marked(md, {  
    renderer: new Term()  
  });  
  return txt;  
}
```

Then, you could further `await read()` in another *Async Function*.

```
async function write () {  
  var txt = await read();  
  console.log(txt);  
}
```

Or you could just use promises for further continuation.

```
read().then(txt => console.log(txt));
```

86 : Fork in the Road

In asynchronous code flows, it is commonplace to execute two or more tasks concurrently. While **Async Functions** make it easier to write asynchronous code, they also lend themselves to code that is *serial*. That is to say: code that executes **one operation at a time**. A function with multiple `await` expressions in it will be suspended once at a time on each `await` expression until that `Promise` is settled, before unsuspending execution and moving onto the next `await` expression – *not unlike the case we observe with generators and `yield`*.

To work around that you can use `Promise.all` to create a single promise that you can `await` on. Of course, the biggest problem is getting in the habit of using `Promise.all` instead of leaving everything to run in a series, as it'll otherwise make a dent in your code's performance.

The following example shows how you could `await` on three different promises that could be resolved concurrently. Given that `await` suspends your `async` function and the `await Promise.all` expression ultimately resolves into a `results` array, we can use destructuring to pull individual results out of that array.

```
async function concurrent () {
  var [r1, r2, r3] = await Promise.all([p1, p2, p3]);
}
```

At some point, there was an `await*` alternative to the piece of code above, where you didn't have to wrap your promises with `Promise.all`. *Babel 5 still supports it, but it was dropped from the spec* (and from Babel 6) [because reasons](#).

```
async function concurrent () {
  var [r1, r2, r3] = await* [p1, p2, p3];
}
```

You could still do something like `all = Promise.all.bind(Promise)` to obtain a terse alternative to using `Promise.all`. An upside of this is that you could do the same for `Promise.race`, which didn't have an equivalent to `await*`.

```
const all = Promise.all.bind(Promise);
async function concurrent () {
  var [r1, r2, r3] = await all([p1, p2, p3]);
}
```

87 : Error Handling

Note that **errors are swallowed "silently"** within an `async` function – *just like inside normal Promises*. Unless we add `try / catch` blocks around `await` expressions, uncaught exceptions – regardless of whether they were raised in the body of your `async` function or while its suspended during `await` – will reject the promise returned by the `async` function.

Naturally, this can be seen as a strength: you're able to leverage `try / catch` conventions, something you were unable to do with callbacks – and *somewhat* able to with Promises. In this sense, *Async Functions* are akin to [generators](#), where you're also able to leverage `try / catch` thanks to function execution suspension turning asynchronous flows into synchronous code.

Furthermore, you're able to catch these exceptions from outside the `async` function, simply by adding a `.catch` clause to the promise they return. While this is a flexible way of combining the `try / catch` error handling flavor with `.catch` clauses in *Promises*, it can also lead to confusion and ultimately cause to errors going unhandled.

```
-----  
read()  
  .then(txt => console.log(txt))  
  .catch(reason => console.error(reason));  
-----
```

We need to be careful and educate ourselves as to the different ways in which we can notice exceptions and then handle, log, or prevent them.

88 : Using async / await Today

One way of using *Async Functions* in your code today is through Babel. This involves a series of modules, but [you could always come up with a module](#) that wraps all of these in a single one if you prefer that. I included [npm-run](#) as a helpful way of keeping everything in locally installed packages.

```
npm i -g npm-run
npm i -D \
  browserify \
  babelify \
  babel-preset-es2015 \
  babel-preset-stage-3 \
  babel-runtime \
  babel-plugin-transform-runtime

echo '{
  "presets": ["es2015", "stage-3"],
  "plugins": ["transform-runtime"]
}' > .babelrc
```

The following command will compile `example.js` through `browserify` while using `babelify` to enable support for **Async Functions**. You can then pipe the script to `node` or save it to disk.

```
npm-run browserify -t babelify example.js | node
```

89 : Further Reading

The [specification draft for Async Functions](#) is surprisingly short, and should make up for an interesting read if you're keen on learning more about this upcoming feature.

I've pasted a piece of code below that's meant to help you understand how `async` functions will work internally. Even though we can't polyfill new keywords, its helpful in terms of understanding what goes on behind the curtains of `async` / `await`.

Namely, it should be useful to learn that *Async Functions* internally leverage both **generators and promises**.

First off, then, the following bit shows how an `async` function declaration could be dumbed down into a regular function that returns the result of feeding `spawn` with a generator function – *where we'll consider `await` as the syntactic equivalent for `yield`*.

```
async function example (a, b, c) {
    example function body
}

function example (a, b, c) {
    return spawn(function* () {
        example function body
    }, this);
}
```

In `spawn`, a promise is wrapped around code that will step through the generator function – *made out of user code* – in series, forwarding values to your "generator" code (the `async` function's body). In this sense, we can observe that *Async Functions* really **are syntactic sugar** on top of generators and promises, which makes it important that you understand how each of these things work in order to get a better understanding into how you can mix, match, and combine these different flavors of asynchronous code flows together.

```
function spawn (genF, self) {
    return new Promise(function (resolve, reject) {
        var gen = genF.call(self);
        step(() => gen.next(undefined));
        function step (nextF) {
            var next;
            try {
                next = nextF();
            } catch(e) {
                // finished with failure, reject the promise
                reject(e);
            }
        }
    })
}
```

```
        return;
    }
    if (next.done) {
        // finished with success, resolve the promise
        resolve(next.value);
        return;
    }
    // not finished, chain off the yielded promise and `step` again
    Promise.resolve(next.value).then(
        v => step(() => gen.next(v)),
        e => step(() => gen.throw(e))
    );
}
});
```

The highlighted bits of code should aid you in understanding how the `async / await` algorithm iterates over the generator sequence (*of await expressions*), wrapping each item in the sequence in a promise and then chaining that with the next step in the sequence. When the **sequence is over or one of the promises is rejected**, the promise returned by the *underlying generator function* is settled.

Special thanks to [@ljharb](#), [@jaydson](#), [@calvinf](#), [@ericclemmons](#), [@sherman3ero](#), [@matthewmolnar3](#), and [@rauschma](#) for reviewing drafts of this article.

90 : Gettin' Freaky Functional w/Curried JavaScript

<http://blog.carbonfive.com/2015/01/14/gettin-freaky-functional-wcurried-javascript/>

[Partial application](#) refers to the practice of filling in a functions parameters with arguments, deferring others to be provided at a later time. JavaScript libraries like Underscore facilitate partial function application – but the API isn't for everyone. I, for one, feel icky sprinkling calls to `_.partial` and `_.bind` throughout my application.

Curried functions (found in Haskell and supported by Scala, among others) can be partially-applied with little ceremony; no special call format is required. In this blog post, I'll demonstrate an approach to currying JavaScript functions at the time of their definition in a way that enables partial function application without introducing lots of annoying parens and nested function expressions.

Currying in JavaScript: Usually a Huge Pain

A function can be said to have been “curried” if implemented as a series of evaluations of nested unary functions (representing each “parameter”) instead of as a single, multiple-arity function (currying a unary function isn’t all that interesting).

This binary function:

```
//  
// (Number, Number) → Number  
//  
function sum(x, y) {  
    return x + y;  
}
```

...could be rewritten as a curried function like this:

```
//  
// Number → Number → Number  
//  
function sum(x) {  
    return function(y) {  
        return x + y;  
    };  
}  
  
var addTen = sum(10); // type: Number → Number  
  
addTen(20); // 30  
addTen(55); // 65
```

This approach starts to become unwieldy as the number of values to be provided by the caller grows:

```
//  
// type: Number → Number → Number → Number → Number  
  
function sumFour(w) {  
    return function (x) {  
        return function (y) {  
            return function (z) {  
                return w + x + y + z;  
            }  
        }  
    }  
}  
  
sumFour(1)(2)(10)(20); // 33  
  
var addTen = sumFour(3)(3)(4);  
addTen(20); // 30
```

Yuck. We get partial application – but with a very low signal-to-noise ratio. Four sets of parens are required to call `sumFour` – and its implementation requires three nested function expressions.

Higher-Order Currying for the Lazy

As luck may have it (thanks, Brendan Eich), we can write a higher-order currying function that can be applied to a fixed-arity function, returning a curried version of the original. Making things even sweeter, our curried function can be called like an idiomatic JavaScript function (one set of parens and comma-separated arguments) instead of like the `sumFour` mess above – all without losing the ability to perform low-ceremony partial application.

We'll jump straight into the implementation:

```
function curry(fx) {  
    var arity = fx.length;  
  
    return function f1() {  
        var args = Array.prototype.slice.call(arguments, 0);  
        if (args.length >= arity) {  
            return fx.apply(null, args);  
        }  
        else {  
            return function f2() {  
                var args2 = Array.prototype.slice.call(arguments, 0);  
                return f1.apply(null, args.concat(args2));  
            };  
        }  
    };  
}
```

```

        }
    }
};

}

```

Calling `curry` with a multiple arity function produces a new function which, when called, either returns a partially applied version of the original or the result of full application. Leveraging `Function.prototype.length`, we can compare the number of provided arguments (which we cache in `args` after each function call) with the arity of the original. If we've received enough arguments to fully apply, the original function is called. If not, we recurse into our helper (`f1`)... returning a new, partially applied function.

Let's see what it looks like to call this function – and how to interact with the resulting function.

```

var sumFour = curry(function(w, x, y, z) {
    return w + x + y + z;
});

```

The name `sumFour` is now bound to a curried function returned from our call to `curry` in which we passed an anonymous, four-parameter function expression. Calling the resulting function will either result in a value (the sum of all four numbers) or a new function with some of its parameters filled in with values provided by the caller.

```

var
    f1 = sumFour(10),           // returns a function awaiting three arguments
    f2 = sumFour(1)(2, 3),     // returns a function awaiting one argument
    f3 = sumFour(1, 2, 7),     // returns a function awaiting one argument
    x = sumFour(1, 2, 3, 4);  // sumFour has been fully applied; x is equal to 1+2+3+4=10
    x2 = sumFour(1)(2)(3)(4); // fully applied; x2 equals 10

```

There are a few things of importance in this last example:

1. Our curried function can be applied to multiple arguments without the need for multiple sets of parameters
2. Our curried function can be partially applied with library support (beyond what was required to create the function bound to `sumFour`)
3. The result of partially applying our curried function to arguments results in a function that is also curried

Cool, right?

In hopes of illustrating how this crazy function works, let's walk through the evaluation of the expression `(sumFour(1)(2, 3, 4))`:

1. Evaluate the subexpression `sumFour(1)`, which (you guessed it) applies `sumFour` to 1
2. Call `f1` with a single argument 1
3. Check to see if `f1` has been called with a number of arguments greater or equal to the number of parameters expressed in the anonymous function passed to `curry` (four: `w, x, y, and z`). The predicate returns false and `f2` is returned.

4. `f2` is called with three arguments `2` and `3` and `4`
5. Concatenate the first argument `1` with new arguments and apply `f1` to all four
6. Check number of arguments (now 4) against arity of original function (4) which satisfies predicate. The original function is then (fully) applied to all four arguments.

In Summary

JavaScript implementations don't make things easy for programmers that want to work with partially-applied and curried functions (lots of parens, nested function expressions). In this blog post, you've seen a demonstration of how one might roll a higher-order currying function that allows for partial function application without a ton of hassle.

91 : Composing Synchronous and Asynchronous Functions in JavaScript

<http://blog.carbonfive.com/2015/01/29/composing-synchronous-and-asynchronous-functions-in-javascript/>

Our example application implements a function `createEmployee` that is used to create an employee from a `personId`.

To create an employee, our system loads some data from our database, validates that data, and then performs an insert. Some of our functions are written in continuation-passing style (they accept a callback) and some are written in a direct style (they return values and/or throw exceptions). We'd like to compose these functions in such a way that they succeed or fail as a single unit – that any error in any segment of the sequence will cause subsequent steps to be skipped with a failure – but with some of our validations happening synchronously and some asynchronously, this can be difficult to do.

To work around this problem, programmers will inline anonymous functions to thread return values and exceptions from direct-style code to our callbacks. For example:

```
function hasWildHair(person) {
    return person.hairColor !== 'Green' || person.hairColor !== 'Pink'
}

function isOfAge(person) {
    return person.age > 17;
}

function ensureDriversLicense(person, callback) {
    Database.getDriversLicenseByPersonId(person.id, function(err, license) {
        if (err) {
            callback(err);
        }
        else if (license.expired) {
            callback('Person must have valid license.');
        }
        else {
            callback(null, person);
        }
    });
}

function createEmployee(personId, callback) {
    var workflow = async.compose(
        Database.createEmployee,
        ensureDriversLicense,
```

```

        ensureWashingtonAddress,
        function(person, callback) {
            if (hasWildHair(person)) {
                callback(null, person);
            }
            else {
                callback('Person must have wild hair.');
            }
        },
        function(person, callback) {
            if (isOfAge(person)) {
                callback(null, person);
            }
            else {
                callback('Person must be 18+ years of age.');
            }
        }
    }
    Database.getPersonById);

workflow(personId, callback);
}

```

There are a few problems with this approach:

1. We've introduced two throwaway function expressions which add visual noise and some maintenance overhead.
2. Our direct-style code (hair color and of-age validations) isn't accessible outside of the CPS function that wraps it.
3. Our throwaway functions contain the logic to transmute the predicate-failures to error messages. Check for wild hair in ten places in your codebase? Ten places to add the check for null and hitting a callback with 'Person must have wild hair.'

In this post, I'll demonstrate an approach to using higher-order functions to lift direct-style functions into the world of callbacks – enabling composition without the introduction of boilerplate function expressions. Once done, the above code will look like this:

```

var createEmployee = async.compose(
    Database.createEmployee,
    ensureDriversLicense,
    ensureWashingtonAddress,
    asyncify(ensureWildHair),
    asyncify(ensureOfAge),
    Database.getPersonById);

```

Step 1: Writing Failable Functions

First, we'll write a higher-order function that accepts a predicate, a value to which we'll apply the predicate, and an error to throw in the event that our value does not satisfy the predicate. Why throw an error? This helps consuming functions differentiate the function's success from failure.

```
function ensure(predicate, error, value) {
  if (predicate(value)) {
    return value;
  }
  else {
    throw error;
  }
}
```

We can now compose `ensure` with our predicates, creating reusable failable validators:

```
var ensureWildHair = _.partial(ensure, hasWildHair, 'Person must have wild hair.');
var ensureOfAge   = _.partial(ensure, ofAge, 'Person must be 18+ years of age.');
```

...which moves some of the error handling out of our larger, employee-creation function:

```
function createEmployee(personId, callback) {
  var workflow = async.compose(
    Database.createEmployee,
    ensureDriversLicense,
    ensureWashingtonAddress,
    function(person, callback) {
      try {
        callback(null, ensureHasWildHair(person));
      }
      catch (e) {
        callback(e);
      }
    },
    function(person, callback) {
      try {
        callback(null, ensureOfAge(person));
      }
      catch (e) {
        callback(e);
      }
    }
  );
}
```

```

        Database.getPersonById);

    workflow(personId, callback);
}

```

Step 2: Working in the World of Callbacks

We've dumbed down the responsibilities of the throwaway function expressions and centralized error creation and predicate handling into a generalized utility function `ensure`. Now, we'll write some code that will allow us to use a direct-style function in a continuation-passing style context.

```

function asyncify(f) {
  return function _asyncify() {
    var args = Array.prototype.slice.call(arguments, 0);
    var argsWithoutLast = args.slice(0, -1);
    var callback = args[args.length-1];
    var result, error;

    try {
      result = f.apply(this, argsWithoutLast);
    }
    catch (e) {
      error = e;
    }

    setTimeout(function() {
      callback(error, result);
    }, 0);
  }
}

```

This new function allows us to map any n-ary function that either returns a value or throws an error to a (n+1)-ary function whose last argument is expected to be a Node.js-style callback whose first argument is the thrown error (if it exists) and second argument will be the returned value (if we didn't throw).

An example:

```

function head(xs) {
  if (_.isArray(xs) && !_.isEmpty(xs)) {
    return xs[0];
  }
  else {
    throw "Can't get head of empty array.";
  }
}

```

```

    }
}

var cbHead = asyncify(head);

cbHead([1, 2, 3], function(err, result) {
    // result === 1
});

cbHead([], function(err, result) {
    // err === "Can't get head of empty array.";
});

```

We can now use `asyncify` function to reshape our direct-style functions into functions that accept callbacks.

```

function createEmployee(personId, callback) {
    var workflow = async.compose(
        Database.createEmployee,
        ensureDriversLicense,
        ensureWashingtonAddress,
        asyncify(ensureHasWildHair),
        asyncify(ensureOfAge),
        Database.getPersonById);

    workflow(personId, callback);
}

```

Finally, we can eliminate the function expression completely; we can instead define `createEmployee` as the composition of our other functions. Since the expression does nothing more than delegate to a function with the same signature, we can safely eliminate it.

```

var createEmployee = async.compose(
    Database.createEmployee,
    ensureDriversLicense,
    ensureWashingtonAddress,
    asyncify(ensureHasWildHair),
    asyncify(ensureOfAge),
    Database.getPersonById);

```

In Summary

Our final implementation reduce the code bespoke code required to validate and create an employee to an absolute minimum. Our resulting application is highly modular; `ensure` and `asyncify` allow the bits to be used in

a variety of contexts outside of `createEmployee`. In the end, we've generalized things to the point where our job is to simply compose generalized functions together to create something specific to the problem at hand.

92 : Tidying Up a JavaScript Application with Higher-Order Functions

<http://blog.carbonfive.com/2015/01/05/tidying-up-a-javascript-application-with-higher-order-functions/>

Higher-order functions are functions that can do one or both of the following:

1. Take a function or functions as arguments
2. Return a function as an argument

Most programmers by now are familiar with higher-order functions such as `map`, `reduce`, and `filter`. These functions abstract away the boilerplate when processing collections. For example...

Given a string-capitalizing function and an array of arrays of strings:

```
function capitalize(s) {
  return s.charAt(0).toUpperCase() + s.slice(1).toLowerCase();
}

var fruitsOfTheWorld = [
  ["apple", "pineapple", "pear"],
  ["manzana", "pera", "piña"],
  ["poma", "perera", "ananàs"]
];
```

...we can turn this:

```
var results = [],
  i = fruitsOfTheWorld.length;

while (i--) {
  results.unshift([]);
  var j = fruitsOfTheWorld[i].length;
  while (j--) {
    results[0].unshift(capitalize(fruitsOfTheWorld[i][j]));
  }
}

return results;
```

...into this:

```
return fruitsOfTheWorld.map(function(fruits) {
  return fruits.map(capitalize);
});
```

In this post, I'll demonstrate the usage of higher-order functions outside of a collection-processing context – with the ultimate goal of reducing boilerplate code in your real-world applications (as `filter` did above). We'll start with partial function application (facilitated by Underscore's `_.partial`) and move on to writing our own higher-order functions.

What is Partial Function Application?

```
function sum(x, y) {
  return x + y;
}
```

Given the code above, partial application refers to the application of the function `sum` to between 0 and $n-1$ arguments, where n is equal to the number of parameters declared by the function. The result of partially applying a function to arguments is a new function with some (or none) of its parameters filled in with values:

```
var sumWithXFixedToTen = _.partial(sum, 10);

(typeof sumWithXFixedToTen); // Function
```

The name `sumWithXFixedToTen` is bound to a new function returned by `_.partial` that behaves like `sum` – but where `x` will always be the value 10 (that is to say, `sumWithXFixedToTen(y)` equals `sum(10, y)` for all `y`). Moving forward, this new function can be fully-applied with just a single argument (since `sum` was already partially-applied to 10).

```
sumWithXFixedToTen(20); // 30
sumWithXFixedToTen(99); // 109
```

Implementing Partial Function Application

To better understand what is going on behind-the-scenes of libraries like Underscore, we will now work through our own implementation of `partial`:

```
function partial(fx) {
  var firstArgs = Array.prototype.slice.call(arguments, 1);

  return function fx2() {
    var secondArgs = Array.prototype.slice.call(arguments, 0);
    return fx.apply(null, firstArgs.concat(secondArgs));
  };
}
```

```
};  
}
```

Our higher-order `partial` function:

1. Takes as its first argument, a function `fx` to partially apply
2. Takes as its 2nd...n arguments (it is variadic), values to which `fx` will be partially applied (`firstArgs`)
3. Returns a new function `fx2` that, when called, returns the result of applying `fx` to the both `firstArgs` and `secondArgs`

This version of `partial` provides a convenient way of applying a function to some number of arguments – but lacks some of the nice features found in similar implementations provided by Underscore's `partial` and `bind`, such as preservation of execution context and parameter-skipping. For the rest of this article, I'll be using Underscore's functions. For the curious, [take a look at their implementation](#) to see what ours is missing.

Real-World Higher-Order Functions

Eliminating Anonymous Function Expressions

When working with libraries like [async](#), programmers often find themselves relying on function expressions to shim application code into the signatures required by `waterfall`, `series`, and the like. Partial application of async-agnostic functions to known arguments can eliminate the need for these shims, significantly reducing overall code volume.

For a concrete example, let's take a look at some real-life client code I came across today:

```
function createAccount(email, password, done) {  
  var id = uuid();  
  
  async.waterfall([  
    function(callback) {  
      Services.Accounts.provision(email, password, callback);  
    },  
    function(accountId, callback) {  
      Services.Account.enable(email, accountId, callback);  
    }  
],  
  function(err, auth) {  
    done(err, id, auth);  
  });  
}
```

Using Underscore, we can eliminate the anonymous function expressions completely:

```

function createAccount(email, password, done) {
  var id = uuid();

  async.waterfall([
    _.partial(Services.Accounts.provision, email, password),
    _.partial(Services.Notification.confirm, email)
  ],
  _.partial(done, _, id,_));
}

```

Something to note about Underscore's `partial` function is that it can accept a placeholder `_` which indicates that an argument should not be pre-filled – but rather provided at call time. In our example, the `done` function has had only its `id` parameter pre-filled, leaving `err` and `auth` to be provided once our series of asynchronous operations completes.

Higher-Order Error Handling

A pet-peve of mine is seeing Node.js-style error handling sprinkled all over a codebase:

```

expressRouter.get("/api/ledgers/:id", function(req, res) {
  UserService.getLedgerById(req.params.id, function(err, ledger) {
    if (err) {
      res.status(404).json({"message": "Not Found"});
    }
    else {
      res.status(200).json(ledger);
    }
  });
});

expressRouter.patch("/api/users/:id", function(req, res) {
  UserService.getUserById(req.params.id, function(err, user) {
    if (err) {
      res.status(404).json({"message": "Not Found"});
    }
    else {
      UserService.updateUser(id, req.body, function(err, user) {
        if (err) {
          res.status(400).json({"message": "Invalid data provided"});
        }
        else {
          res.status(200).json(user);
        }
      });
    }
  });
});

```

```

    });
}
});
}

```

Notice a pattern here? In the asynchronously-executed callbacks that we provide for both `getUserById` and `updateUser`, we check the value bound to the first parameter of each callback (`err`) and respond with some error code and response if its value is truthy. This pattern of checking for and responding to errors can be abstracted into a higher-order function, leaving the application's happy path free of crusty control logic.

```

function ensure(notOkCode, notOkMessage, response, okFx) {
  return function(err) {
    if (err) {
      response.status(notOkCode).json({ "message": notOkMessage });
    } else {
      okFx.apply(okFx, Array.prototype.slice.call(arguments, 1));
    }
  };
}

var ensureFound = _.partial(ensure, 404, "Not Found"),
  ensureValid = _.partial(ensure, 400, "Bad Request");

expressRouter.get("/api/ledgers/:id", function(req, res) {
  UserService.getLedgerById(req.params.id, ensureFound(res, function(ledger) {
    res.status(200).json(ledger);
  }));
});

expressRouter.patch("/api/users/:id", function(req, res) {
  UserService.getUserById(req.params.id, ensureFound(res, function(user) {
    UserService.updateUser(user, req.body, ensureValid(res, function(updated) {
      res.status(200).json(updated);
    }));
  }));
});

```

Takeaways

Higher-order functions will help you write JavaScript with less cruft. By familiarizing yourself with Underscore's `partial` function (or something you write yourself), you'll have one more tool for factoring out common patterns into reusable, higher-order little chunks. Functional programming FTW!

