A Drip of JavaScript

THE COMPLETE COLLECTION

Joshua Clanton

A Drip of JavaScript

The Complete Collection

Joshua Clanton

This book is for sale at http://leanpub.com/a-drip-of-javascript-book

This version was published on 2014-09-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Joshua Clanton by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought ”A Drip of JavaScript: The Complete Collection.” #jsdripbook

The suggested hashtag for this book is #jsdripbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#jsdripbook

Also By Joshua Clanton

Jasmine Testing for JavaScript

Contents

Preface

Every Tuesday at 7:30 in the morning, hundreds of subscribers receive the latest issue of *A Drip of JavaScript*. Each issue is a bite-sized look at one aspect of programming in JavaScript.

This book is the living archive of *A Drip of JavaScript*. It is not intended to be a "how to program" manual, or a treatise on the language. Instead, it is meant to be a book you can dip into for five minutes at a time to learn something new.

Each month this book is updated to include the latest issues of the newsletter, as well as incorporate bug fixes and content feedback.

Resources

In addition to the newsletter itself[1], there is also an official Drip Discussion Group[2] for readers to talk about the concepts found in each drip.

Bug Reports & Feature Requests

Because this book is a "living archive," it is possible that from time mistakes may creep in. If you see any mistakes or have suggestions for improving the book, feel free to reach out to me on the discussion group[3], Twitter[4], or via email[5].

Thanks for reading!

Joshua Clanton

---

[1] http://adripofjavascript.com

[2] https://groups.google.com/forum/?hl=en&fromgroups#!forum/js-drip-discussions

[3] https://groups.google.com/forum/?hl=en&fromgroups#!forum/js-drip-discussions

[4] https://twitter.com/joshuacc

[5] mailto:jsdrip@designpepper.com

Credits

Many thanks to my coworkers at Hobsons for their feedback and encouragement.

Thanks also to the small army of proofreaders who look at each issue of *A Drip of JavaScript*.

Thanks to my parents for the Commodore 64 and the stacks of books they gave me as a child.

Most of all, thanks to my wife, without whose patience and support, this book would not be possible.

Default Parameters in JavaScript

Some languages – like Ruby, CoffeeScript, and upcoming versions of JavaScript[6] – have the ability to declare default parameters when defining a function. It works like this:

```
function myFunc(param1, param2 = "second string") {
    console.log(param1, param2);
}

// Outputs: "first string" and "second string"
myFunc("first string");

// Outputs: "first string" and "second string version 2"
myFunc("first string", "second string version 2");
```

Unfortunately, this construct isn't available in current versions of JavaScript. So what can we do to achieve this behavior with our current set of tools?

The simplest solution looks like this:

```
function myFunc(param1, param2) {
    if (param2 === undefined) {
        param2 = "second string";
    }

    console.log(param1, param2);
}

// Outputs: "first string" and "second string version 2"
myFunc("first string", "second string version 2");
```

This relies on the fact that a parameter omitted at call time is always `undefined`. And it's a good solution if you have only one parameter to deal with. But what if you have several?

Well if you have more than a few parameters, you should probably be passing in an object parameter, as that has the advantage of explicitly naming everything. And if you're passing in an object parameter, it makes sense to declare your defaults the same way.

[6] https://brendaneich.com/2012/10/harmony-of-dreams-come-true/

```
function myFunc(paramObject) {
    var defaultParams = {
        param1: "first string",
        param2: "second string",
        param3: "third string"
    };

    var finalParams = defaultParams;

    // We iterate over each property of the paramObject
    for (var key in paramObject) {
        // If the current property wasn't inherited, proceed
        if (paramObject.hasOwnProperty(key)) {
            // If the current property is defined,
            // add it to finalParams
            if (paramObject[key] !== undefined) {
                finalParams[key] = paramObject[key];
            }
        }
    }

    console.log(finalParams.param1,
                finalParams.param2,
                finalParams.param3);
}
```

That's a little unwieldy, so if you're using this pattern a lot, it makes sense to extract the iteration and filtering logic into its own function. Fortunately, the clever folks who write jQuery[7] and Underscore[8] have done just that with their respective `extend` methods.

Here's an updated version which uses Underscore's extend to achieve the same result.

[7] http://api.jquery.com/jQuery.extend/
[8] http://underscorejs.org/#extend

```
function myFunc(paramObject) {
    var defaultParams = {
        param1: "first string",
        param2: "second string",
        param3: "third string"
    };

    var finalParams = _.extend(defaultParams, paramObject);

    console.log(finalParams.param1,
                finalParams.param2,
                finalParams.param3);
}

// Outputs:
// "My own string" and "second string" and "third string"
myFunc({param1: "My own string"});
```

And that's how you can get default parameters in current versions of JavaScript.

Arbitrary Parameters with the `arguments` Object

In a previous drip we discussed default parameters. That gives us some flexibility when working with parameters, but what if we want something that would accept as many arguments as we can throw at it? Say for example that we want to create a function which adds together all the arguments we pass to it. How would we go about that?

```
function addAll () {
    // What do we do here?
}

// Should return 6
addAll(1, 2, 3);

// Should return 10
addAll(1, 2, 3, 4);
```

Fortunately, JavaScript does have an answer, though it is a little quirky. The answer is the `arguments` object.

Though it isn't exactly an array, the `arguments` object **acts** like an array that happens to contain every parameter passed to the function.

```
function myFunc () {
    console.log(arguments[0], arguments[1]);
}

// Outputs "param1" and "param2"
myFunc("param1", "param2");
```

Now that we know about `arguments`, it is easy to make an addition function that will operate on any number of parameters.

```
function addAll () {
    var sum = 0;

    for (var i = 0; i < arguments.length; i++) {
        sum = sum + arguments[i];
    }

    return sum;
}

// Returns 6
addAll(1, 2, 3);

// Returns 10
addAll(1, 2, 3, 4);
```

One problem to watch out for with `arguments` is that it isn't **really** an array. We can see this by running the following:

```
function myFunc () {
    console.log(Array.isArray(arguments));
}

// Will output 'false'
myFunc('param');
```

So it's not an array. Does that make a difference? Unfortunately, yes. It doesn't have any of the normal array methods like `push`, `pop`, `slice`,`indexOf`, or `sort`.

```
function sortArgs () {
    // This won't work!
    var sorted = arguments.sort();

    return sorted;
}
```

This can easily bite you, especially if you pass the contents of `arguments` to another function which expects a real array.

The solution is surprisingly easy to use, but a little more difficult to understand.

```
function sortArgs () {
    // Convert arguments object into a real array
    var args = [].slice.call(arguments);

    // Now this will work!
    var sorted = args.sort();

    return sorted;
}
```

What's going on here has several steps:

1. We create an empty array.
2. We use the array's `slice` method.
3. We use the `call` method to tell `slice` that it should operate on `arguments` rather than on the empty array.

Invoking `slice` without specifying which index the slice should begin at will return an unsliced **array**. And that's how we end up with exactly what we want: a real array that contains every parameter that was passed to the function.

Reordering Arrays with Array#sort

In the previous drip I briefly mentioned the standard `sort` method available on all JavaScript arrays. It does just what you'd expect.

```
// Produces [1, 2, 3]
[3, 2, 1].sort();

// Produces ["a", "b", "c"]
["c", "b", "a"].sort();

// Produces [1, 2, "a", "b"]
["b", 2, "a", 1].sort();
```

As you can see, it sorts by dictionary order (numbers first, followed by letters). But what if you want to sort your array differently? For example, listing "better" cars first? How would you go about that? Fortunately, `sort` accepts a custom comparison function.

```
var cars = [
    "Honda Civic",
    "Ford Taurus",
    "Chevy Malibu"
];

cars.sort(function(a, b) {
    // Default to 0, which indicates
    // no sorting is necessary
    var returnVal = 0;

    // If `a` is a Chevy, subtract 1
    // to move `a` "up" in the sort order
    // because Chevys are awesome.
    if (a.match(/Chevy/)) {
        returnVal = returnVal - 1;
    }

    // If `b` is a Chevy, add 1
    // to move `b` "up" in the sort order
```

```
    if (b.match(/Chevy/)) {
        returnVal = returnVal + 1;
    }

    return returnVal;
});

// Outputs:
// ["Chevy Malibu", "Honda Civic", "Ford Taurus"]
console.log(cars);
```

The comparison function compares two values and returns a number.

- If it returns a negative number, `a` will be sorted to a lower index in the array.
- If it returns a positive number, `a` will be sorted to a higher index.
- And if it returns `0` no sorting is necessary.

When you pass `sort` a comparison function, that function will be called repeatedly and given different values from the array until the array has been completely sorted.

It is important to note that returning `0` does not guarantee that `a` and `b` will remain in the same order. It merely means that sorting is not necessary. JavaScript engines may choose to keep them in the same order, but that is not part of the language spec.

What is an Array?

For an experienced developer, that question may seem simplistic, but in the case of JavaScript the answer is rather interesting.

Conceptually, an array is just a list of values which can be accessed by using an integer as the "key". The list starts at `0` and goes up from there. If we were to describe that with JavaScript's object notation, it would look like this:

```
var fakeArray = {
    0: "value 1",
    1: "value 2"
};
```

I feel like we're missing something. Oh, I know. It's that pesky "length" property.

```
var fakeArray = {
    0: "value 1",
    1: "value 2",
    length: 2
};
```

That's looking suspiciously array-like. And, as I've mentioned before, this is precisely how the `arguments` object works. What I haven't mentioned yet is that this is also how real arrays work "under the hood".

That's right, in JavaScript arrays are just a specialized type of object built into the language. It's easy to tell this by looking at the behavior of objects and arrays.

```
var fakeArray = {
    0: "value 1",
    1: "value 2",
    length: 2
};

// Outputs "value 1"
console.log(fakeArray[0]);

var realArray = ["value 1", "value 2"];
```

```
realArray.text = "some text";

// Outputs "some text"
console.log(realArray.text);
```

As you can see, you can access an object's properties in exactly the same way as an array's. And you can give an array additional properties just like any other object.

What about all those special array methods like `indexOf`, `slice`, and `sort`? Turns out, they're just functions attached to the array object. (More specifically, they belong to the Array prototype, but that's getting ahead of ourselves.)

```
var realArray = ["value 1", "value 2"];

// Outputs "0"
console.log(realArray.indexOf("value 1"));

realArray.indexOf = function() {
    return "I'll never tell.";
};

// Outputs "I'll never tell."
console.log(realArray.indexOf("value 1"));
```

In fact, given enough time, it is possible to reimplement almost all of the native array functionality purely in terms of manipulating objects.

Here's an example of reimplementing the `push` method:

```
var fakeArray = {
    length: 0,
    push: function (val) {
        // Place the new value into the
        // next available integer key
        this[this.length] = val;

        // Update the length property
        this.length = this.length + 1;

        // Return the updated length
        return this.length;
    }
};
```

```
15
16 fakeArray.push("first value");
17 fakeArray.push("second value");
18
19 // Outputs "first value"
20 console.log(fakeArray[0]);
21
22 // Outputs "second value"
23 console.log(fakeArray[1]);
```

The one important thing that can't be reimplemented is the convenient array literal syntax for creating arrays. (The square brackets.) But you can always use a constructor instead. In fact, the following two ways of creating an array are exactly equivalent.

```
1 var literalWay = ["value 1"];
2
3 var constructorWay = new Array("value 1");
```

If you're willing to give up the literal syntax, you can rebuild the concept of arrays in JavaScript entirely from scratch to achieve something like this:

```
1 var myCustomArray = new CustomArray("value 1");
```

And now you know (if you didn't already) how arrays work in JavaScript.

Transforming Arrays with Array#map

One of the most common tasks that developers perform in any language is taking an array of values and transforming those values. Up until recently, doing that in JavaScript took a fair bit of boilerplate code. For instance, here is some code for darkening RGB colors:

```
var colors = [
    {r: 255, g: 255, b: 255 }, // White
    {r: 128, g: 128, b: 128 }, // Gray
    {r: 0,   g: 0,   b: 0   }  // Black
];

var newColors = [];
var transformed;

for (var i = 0; i < colors.length; i++) {
    transformed = {
        r: Math.round( colors[i].r / 2 ),
        g: Math.round( colors[i].g / 2 ),
        b: Math.round( colors[i].b / 2 )
    };

    newColors.push(transformed);
}

// Outputs:
// [
//    {r: 128, g: 128, b: 128 },
//    {r: 64,  g: 64,  b: 64  },
//    {r: 0,   g: 0,   b: 0   }
// ];
console.log(newColors);
```

As you can see, there's quite a bit going on in that code that isn't really about what we want to accomplish, but is keeping track of trivial things like the current index and moving the values into the new array. What if we didn't have to do all of that?

Fortunately in ECMAScript 5 (the latest version of JavaScript), we don't. Here is the same example rewritten to take advantage of the `map` method:

```
var newColors = colors.map(function(val) {
    return {
        r: Math.round( val.r / 2 ),
        g: Math.round( val.g / 2 ),
        b: Math.round( val.b / 2 )
    };
});
```

Much nicer isn't it? Invoking `map` returns a new array created by running a transformation function over each element of the original array.

Now the only thing you need to keep track of is the logic of the transformation itself.

Of course, `map` isn't limited to simple transformations like this. Your function can also make use of two additional parameters, the current index and the array itself. Consider the following example:

```
var starter = [1, 5, 5];

function multiplyByNext (val, index, arr) {
    var next = index + 1;

    // If at the end of array
    // use the first element
    if (next === arr.length) {
        next = 0;
    }

    return val * arr[next];
}

var transformed = starter.map(multiplyByNext);

// Outputs: [5, 25, 5]
console.log(transformed);
```

As you can see, the additional parameters make it easy to create transformation functions which use the array element's neighbors. This can be useful in implementing something like Conway's Game of Life[9].

Browser support for `map` is pretty good, but not universal. It isn't supported in IE 8 and below. You have a few options for dealing with this.

1. Don't use `map`.

[9] https://en.wikipedia.org/wiki/Conway's_Game_of_Life

2. Use something like es5-shim[10] to make older IE's support `map`.
3. Use the `_.map` method in Underscore[11] or Lodash[12] for an equivalent utility function.

One of the most powerful techniques for avoiding programming bugs is to reduce the number of things that you are keeping track of manually. Array's `map` method is one more tool to help you do exactly that.

[10] https://github.com/kriskowal/es5-shim
[11] http://underscorejs.org/#map
[12] http://lodash.com/docs#map

Constructors in JavaScript

Despite the fact that they are very powerful, constructors are one of the most underused features of JavaScript. (Probably because they have a very influential detractor[13].) But if you want to really know JavaScript, you'll need to learn how they work.

What is a constructor? It's an ordinary function that is used with the `new` operator to produce a specialized type of object.

```
// `Color` is a constructor
var red = new Color(255, 0, 0);
```

In this example, red is a new "Color" object. But how does that work?

```
function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}

var red = new Color(255, 0, 0);
```

As you can see, the Color constructor is merely taking the arguments given to it and attaching them to the `this` object. That's because when the constructor is invoked by `new`, the constructor's `this` is set to the object that `new` will return.

Which means that the code above is roughly equivalent to:

```
var red = {
    r: 255,
    g: 0,
    b: 0
};
```

So why would we use a constructor? There are a couple of important reasons.

First, using a constructor means that all of these objects will be created with the same basic structure, and we are less likely to accidentally make a mistake than if we were creating a whole bunch of generic objects by hand. (Especially if we make the constructor throw errors on invalid input.)

Second, using a constructor means that the object is explicitly marked as an instance of `Color`.

[13] http://crockford.com/

```
var red = new Color(255, 0, 0);

var blue = { r: 255, g: 0, b: 0 };

// Outputs: true
console.log(red instanceof Color);

// Outputs: false
console.log(blue instanceof Color);
```

That gives us a means to ensure that we are receiving the correct type of data to process.

Third, using a constructor means that we can easily assign specialized methods to the constructor's prototype, and they will instantly be available to all objects created by the constructor.

```
function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}

Color.prototype.getAverage = function () {
    var total = this.r + this.g + this.b;
    var avg = total / 3;
    return parseInt(avg, 10);
};

var red = new Color(255, 0, 0);
var white = new Color(255, 255, 255);

// Outputs: 85
console.log(red.getAverage());

// Outputs: 255
console.log(white.getAverage());
```

It is important to **always** use the new keyword when invoking the constructor. Otherwise the constructor may clobber the `this` which was accidentally passed to it. In most cases that is the global object (`window` in the browser or `global` in Node.)

Because of this danger, it is customary to capitalize a constructor's name so that others know to invoke it with `new`.

An upcoming article will deal with how to further mitigate the danger inherent in working with constructors.

Dealing with the Dangers of `this` in Constructors

As I mentioned in the previous drip, invoking a constructor without `new` can be dangerous. Let's go over why.

```
function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}

// Safe invocation
var red = new Color(255, 0, 0);

// Dangerous invocation
var blue = Color(0, 0, 255);
```

When a constructor is invoked with the `new` keyword, the `this` value of the constructor is set to the new object that you are creating. But when a constructor is invoked like a normal function, its `this` defaults to the same `this` variable that any other function gets. And normally that is the global object (`window` in the browser.)

Here is an illustration of the problem.

```
// Global variable
r = "Rodent Of Unusual Size";

function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}

// Dangerous invocation
// Means `this` is the global object
var blue = Color(0, 0, 255);

```

```
// Outputs: 0
console.log(r);

// Outputs: undefined
console.log(blue);
```

In the example above, there is a global variable named `r`. Or to put it another way, the global object has a property named `r`. When the `Color` constructor is invoked without `new`, the constructor's `this` is set to the global object (in most cases). Which means that the constructor function has just overwritten the global `r` variable with something that was intended to be a property of the `blue` object.

Furthermore, because `Color` was invoked as an ordinary function, it didn't automatically return a new object, which means that `blue` is also undefined.

As you can imagine, debugging an issue like this can be time consuming and frustrating. So how do we prevent these sorts of problems? Fortunately the answer is pretty straightforward.

```
// Global variable
r = "Rodent Of Unusual Size";

function Color(r, g, b) {
    // Check whether `this` is a
    // `Color` object.
    if (this instanceof Color) {
        this.r = r;
        this.g = g;
        this.b = b;
    } else {
        // If not, then we should invoke
        // the constructor correctly.
        return new Color(r, g, b);
    }
}

// Dangerous invocation
// Means `this` is the global object
var blue = Color(0, 0, 255);

// Outputs: "Rodent Of Unusual Size"
console.log(r);

// Outputs: Color {r: 0, g: 0, b: 255}
console.log(blue);
```

In the updated `Color` constructor, the first thing we do is check whether `this` is an instance of `Color`. It works because the `new` keyword will have already created the new object as an instance of the constructor before the constructor function begins running.

If it isn't a `Color` object, then we know the constructor was invoked incorrectly, so we skip all the construction logic and have `Color` return the results of correctly invoking itself with `new`.

This means that the constructor is no longer in danger of clobbering the global object's properties.

Of course, using this approach also means that developers may get into the bad habit of invoking constructors without `new`. If you'd rather just force them to always use `new`, you could throw an error instead, like so:

```
function Color(r, g, b) {
    // Check whether `this` is a
    // `Color` object.
    if (this instanceof Color) {
        this.r = r;
        this.g = g;
        this.b = b;
    } else {
        // If not, throw error.
        throw new Error("`Color` invoked without `new`");
    }
}
```

And that's how you can make your custom constructors safely deal with a missing `new` keyword.

What are Prototype Properties and Methods?

A couple of drips ago, I mentioned prototype properties in passing. But what exactly are they? Let's take a look.

```
function Book(title, author) {
    this.title = title;
    this.author = author;
}

Book.prototype.getFormatted = function () {
    return this.title + " - " + this.author;
};

var hobbit = new Book("The Hobbit", "Tolkien");

// Outputs: "The Hobbit - Tolkien"
console.log(hobbit.getFormatted());
```

What is this magical `getFormatted`? And how did it get attached to my object? It's all through the power of prototypes.

Whenever you create an object using a constructor, that constructor has a property called `prototype` which points to an object. If you haven't done anything special, that prototype property is just an empty object. But it's an empty object with superpowers.

Let's see how that works.

```
function Book(title, author) {
    this.title = title;
    this.author = author;
}

Book.prototype.pubYear = "unknown";

var hobbit = new Book("The Hobbit", "Tolkien");

```

```
var caspian = new Book("Prince Caspian", "Lewis");
caspian.pubYear = 1951;

// Outputs: "unknown"
console.log(hobbit.pubYear);

// Outputs: 1951
console.log(caspian.pubYear);

// Outputs: "unknown"
console.log(Book.prototype.pubYear);
```

In the example above, we create a `Book` constructor, and then create a `pubYear` property "unknown" on it's prototype object.

When we try to access `hobbit.pubYear`, the JavaScript interpreter realizes that the `hobbit` object doesn't have a matching property, so it then checks to see if there is a matching property on the prototype object. Since there is, it will give us the value of `Book.prototype.pubYear`.

But because the `caspian` object has a `pubYear` property of it's own, the interpreter never has to go look at the prototype object.

While ordinary properties on a prototype can be useful, methods are more useful still. Let's go back to our original example.

```
function Book(title, author) {
    this.title = title;
    this.author = author;
}

Book.prototype.getFormatted = function () {
    return this.title + " - " + this.author;
};

var hobbit = new Book("The Hobbit", "Tolkien");

// Outputs: "The Hobbit - Tolkien"
console.log(hobbit.getFormatted());
```

Now you should know how the property lookup works. The interpreter realizes that there is no `getFormatted` property on `hobbit` itself, so looks for it on the prototype.

But you might notice something else special. The `getFormatted` method makes use of `this.title` and `this.author`. That works because when an object's method is invoked, the object itself becomes

the `this` for the method. And that's even if the method itself belongs to the prototype rather than directly to the object.

Why bother with all of this, though? Why not define `getFormatted` directly on the object, like so?

```
function Book(title, author) {
    this.title = title;
    this.author = author;

    this.getFormatted = function () {
        return this.title + " - " + this.author;
    };
}
```

Because if you do it like that, rather than defining a single method which can be used by all `Book` objects, each object has its own copy of the method. Now imagine a scenario where you are producing lots of `Book` objects.

```
var bookArray = [];

for (var i = 0; i < 100; i++) {
    bookArray[i] = new Book("Some Title", "Some Author");
}
```

Now you have have 100 copies of `getFormatted` taking up space in memory. In addition, if you need to change how `getFormatted` works, you'll need to manually update each instance of `Book`. Compare that to how simple it is to update the prototype method.

```
function Book(title, author) {
    this.title = title;
    this.author = author;
}

Book.prototype.getFormatted = function () {
    return this.title + " - " + this.author;
};

var hobbit = new Book("The Hobbit", "Tolkien");

// Outputs: "The Hobbit - Tolkien"
console.log(hobbit.getFormatted());

```

```
// Oops! We need to use commas, not hyphens
Book.prototype.getFormatted = function () {
    return this.title + ", " + this.author;
};

// Outputs: "The Hobbit, Tolkien"
console.log(hobbit.getFormatted());
```

Updating the method on the prototype means that all instances of `Book` get the updated method immediately, and you don't have the possibility of some instances having the outdated method.

So those are the basics of how prototype properties and methods work.

Using Dispatch Tables to Avoid Conditionals in JavaScript

When writing code, one of the surest ways to keep things simple and straightforward is to avoid conditionals when possible. Unfortunately, it is fairly common to see code with a lot of `if`, `switch`, and `case` statements like the following:

```
function processUserInput(command) {
    switch (command) {
        case "north":
            movePlayer("north");
            break;
        case "east":
            movePlayer("east");
            break;
        case "south":
            movePlayer("south");
            break;
        case "west":
            movePlayer("west");
            break;
        case "look":
            describeLocation();
            break;
        case "backpack":
            showBackpack();
            break;
    }
}
```

Above we have a function for processing user input from a text adventure game. While it isn't terribly difficult to understand, it is more complicated than necessary. And as the number of commands grow, the function can quickly become unwieldy. So what can we do to simplify it?

```
var commandTable = {
    north:    function() { movePlayer("north"); },
    east:     function() { movePlayer("east");  },
    south:    function() { movePlayer("south"); },
    west:     function() { movePlayer("west");  },
    look:     describeLocation,
    backpack: showBackpack
};

function processUserInput(command) {
    commandTable[command]();
}
```

In this refactored version we are using a dispatch table[14] to hold all the possible commands a user can give, and the functions the program should call. That changes the `processUserInput` function into a single line, and eliminates all conditionals.

If you are unfamiliar with bracket notation, it is just an alternate way of accessing a function's properties, with the advantage that you can use variables for the property's name. For example, if `command` is "north", then `commandTable[command]` is equivalent to `commandTable.north`.

The fundamental change we made is transforming the conditionals into a data structure. And data is much easier to manipulate than conditionals. In the future, if we want to add a new command, all we have to do is add it to `commandTable`. No messing around with `case` or `break` statements required.

[14] https://en.wikipedia.org/wiki/Dispatch_table

Testing Array Contents with Array#some

When working with arrays, it is quite common to perform different actions based on whether or not the array contains a particular item. A fairly straightforward implementation of this pattern can be found below:

```
var planets = [
    "mercury",
    "venus",
    "earth",
    "mars",
    "jupiter",
    "saturn",
    "uranus",
    "neptune"
];

// Default to false
var containsPluto = false;

for (var i = 0; i < planets.length && !containsPluto; i++) {
    if (planets[i] === "pluto") {
        containsPluto = true;
    }
}

// Outputs: false
console.log(containsPluto);
```

Here our for loop is doing the work of iterating over the array, and a simple if statement checks to see if our condition is met yet. This for loop is a little more complicated than most because it is set up to stop iterating as soon as a matching element is found in the array.

While it is not that difficult to understand, manually keeping track of the iteration and the "short circuiting" logic just isn't the problem that we're trying to solve. The fewer such things we have to keep in our head, the better. We could fix this by moving the loop into it's own function (probably a good idea), but fortunately the latest version of the JavaScript standard includes such a function for us.

The function is called `some` and it is available on all arrays. Here is an example of how it would apply to our problem.

```
function isPluto(element) {
    return (element === "pluto");
}

// Outputs: false
console.log(planets.some(isPluto));

var dwarfPlanets = [
    "ceres",
    "pluto",
    "haumea",
    "makemake",
    "eris"
];

// Outputs: true
console.log(dwarfPlanets.some(isPluto));
```

All we have to do to use `some` is to pass it a callback function. The callback will be executed once for each element until the callback returns true. At that point the `some` method itself will return true. If the callback never returns true, then `some` will return false.

The great thing about `some`, though is that you don't have to think about that unless you want to. You just have to pass in your callback.

In addition to the array element itself, the callback function has access to two other parameters: the current index, and the entire array. This can be useful in situations where you are comparing the current element against other members of the array. Consider this example:

```
function isLessThanPrev(el, index, arr) {
    // The first element doesn't have a predecessor,
    // so don't evaluate it.
    if (index === 0) {
        return;
    } else {
        return (el < arr[index - 1]);
    }
}

var evens = [2, 4, 6, 8];
```

```
12 var randoms = [0, 9, 2, 5];
13
14 // Outputs: false
15 console.log(evens.some(isLessThanPrev));
16
17 // Outputs: true
18 console.log(randoms.some(isLessThanPrev));
```

As with other features of ES5[15] that we've talked about, `some` is only supported in IE9 and higher. If you'd like to use it in older browsers, you will need to use a library like Underscore[16] which has an equivalent method or a compatibility shim[17] which ports `Array#some` back into older browsers.

[15] http://en.wikipedia.org/wiki/ECMAScript_5#ECMAScript.2C_5th_Edition
[16] http://underscorejs.org/#some
[17] https://github.com/kriskowal/es5-shim

Creating Chainable Interfaces in JavaScript

When first learning jQuery[18], one of the things that most strikes developers is the ease of using its chainable interface to just keep stringing commands together.

```
$("#myDiv")
    .addClass("myClass")
    .css("color", "red")
    .append("some text");
```

This approach is very powerful. But until you understand how it works, it can seem rather mysterious and complicated. Fortunately, it's very straightforward to implement, so that's what we'll be looking at.

That "jQuery style" chainability is also known as a fluent interface[19]. The fundamental thing that makes a fluent interface possible is for each method to return an object so that you can then call methods upon it.

That's a little abstract, so here is a concrete example:

```
function Book(name, author) {
    this.name = name;
    this.author = author;
}

Book.prototype.setName = function (name) {
    this.name = name;
    return this;
};

Book.prototype.setAuthor = function (author) {
    this.author = author;
    return this;
};

```

[18] http://jquery.com/

[19] https://en.wikipedia.org/wiki/Fluent_interface

```
var lotr = new Book("Lord of the Rings", "Tolkien");

// Outputs: {
//     name: "Lord of the Rings",
//     author: "Tolkien"
// }
console.log(lotr);

// Whoops! Details were slightly wrong.
// Let's fix that.
lotr.setAuthor("JRR Tolkien") // Returns `lotr`
    .setName("The Lord of the Rings"); // Returns `lotr`

// Outputs: {
//     name: "The Lord of the Rings",
//     author: "JRR Tolkien"
// }
console.log(lotr);
```

The trick here is in our prototype methods. We created `setName` and `setAuthor` which can be called on any `Book` object. But then we made sure that when the methods had done their work, they returned the object they were originally called on.

Since that object was a `Book`, we could immediately call any `Book` method upon it. And that's all that chaining requires.

Using Duck Typing to Avoid Conditionals in JavaScript

As I've mentioned before, one of the best ways to simplify your code is to eliminate unnecessary `if` and `switch` statements. In this drip, we're going to look at using duck typing[20] to do that.

Consider the example of a library filing system which has to deal with books, magazines, and miscellaneous documents:

```
1  function Book (title) {
2      this.title = title;
3  }
4
5  function Magazine (title, issue) {
6      this.title = title;
7      this.issue = issue;
8  }
9
10 function Document (title, description) {
11     this.title = title;
12     this.description = description;
13 }
14
15 var laic = new Book("Love Alone is Credible");
16 var restless = new Magazine("Communio", "Summer 2007");
17 var le = new Document("Laborem Exercens", "encyclical");
18
19 // A simple array where we keep track of things that are filed.
20 var filed = [];
21
22 function fileIt (thing) {
23     // We conditionally file in different places
24     // depending on what type of thing it is.
25
26     if (thing instanceof Book) {
27         console.log("File in main stacks.");
28     } else if (thing instanceof Magazine) {
```

[20] https://en.wikipedia.org/wiki/Duck_typing

```
        console.log("File on magazine racks.");
    } else if (thing instanceof Document) {
        console.log("File in document vault.");
    }

    // Mark as filed
    filed.push(thing);
}

// Outputs: "File in main stacks.""
fileIt(laic);

// Outputs: "File on magazine racks."
fileIt(restless);

// Outputs: "File in document vault."
fileIt(le);
```

That may not look like a lot of conditional logic, but once the library starts handling museum pieces, photographs, CDs, DVDs, etc. it can become quite unwieldy. That's where duck typing can come to our rescue.

The term "duck typing" comes from the saying, "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." The implication is that in our code we don't need to test that something is a duck if we know that it can quack.

Here's how a duck typed version of our code might look:

```
function Book (title) {
    this.title = title;
}

Book.prototype.file = function() {
    console.log("File in main stacks.");
};

function Magazine (title, issue) {
    this.title = title;
    this.issue = issue;
}

Magazine.prototype.file = function() {
    console.log("File on magazine racks.");
```

```
};

function Document (title, description) {
    this.title = title;
    this.description = description;
}

Document.prototype.file = function() {
    console.log("File in document vault.");
};

var laic = new Book("Love Alone is Credible");
var restless = new Magazine("Communio", "Summer 2007");
var le = new Document("Laborem Exercens", "encyclical");

// A simple array where we keep track of things that are filed.
var filed = [];

function fileIt (thing) {
    // Dynamically call the file method of whatever
    // `thing` was passed in.
    thing.file();

    // Mark as filed
    filed.push(thing);
}

// Outputs: "File in main stacks.""
fileIt(laic);

// Outputs: "File on magazine racks."
fileIt(restless);

// Outputs: "File in document vault."
fileIt(le);
```

Notice that for each different type of object we might file, we just define a `file` method on the prototype to encapsulate the logic specific to that type of object. The only thing that `fileIt` cares about is whether the `file` method exists.

If it quacks it must be a duck, and if it has a `file` method, then it must be fileable.

That means that not only is our conditional logic eliminated, but adding new types of objects to our filing system is a snap. For example:

```
function CompactDisc (title) {
    this.title = title;
}

CompactDisc.prototype.file = function() {
    console.log("File in music section.");
};

var coc = new CompactDisc("A Ceremony of Carols");

// Outputs: "File in music section."
fileIt(coc);
```

No modifications to `fileIt` were necessary. All we had to do was define an appropriate `file` method for `CompactDisc` objects.

Less conditional logic plus easy extensibility equals a win in my book.

Those of you with a background in more classical languages might be wondering about the advantages of using duck typing instead of more traditional subtype polymorphism[21]. Here is an illustration:

```
var weirdModernArtPiece = {
    title: "diffÃ©rance",
    file: function() {
        console.log("File this where no one can see it.");
    }
};

// Outputs: "File this where no one can see it."
fileIt(weirdModernArtPiece);
```

In this example, we created an *ad hoc* object and `fileIt` handled it just fine. Duck typing lets us use any object, regardless of its type, or even whether it has a specific type at all. The ability to use *ad hoc* objects means that our system can more easily adapt to changing requirements.

[21] https://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming

Function functions

One of the features of JavaScript that sometimes trips up developers used to other languages is the fact that functions are first-class objects. This means that functions can be assigned to a variable and passed around.

```
var one = function() { return 1; };

// Outputs: "function() { return 1; }"
console.log(one);

// Outputs: 1
console.log(one());
```

Why the difference between the results of these two logs? In the first instance, we are logging the value of `one`. And the value of `one` is the function itself. In the second case we are logging the value returned by invoking `one`.

So far, so good. But there is a further implication. Functions can return objects. And if functions are first-class objects, then that means it is possible for a function to return a function.

```
function outerFunction () {
    return function() { return "inner"; };
}

var whatIsIt = outerFunction();

// Outputs: function() { return "inner"; }
console.log(whatIsIt);

// Outputs: "inner"
console.log(whatIsIt());
```

This can be a little confusing at first, but is actually pretty simple. When we invoke `outerFunction`, it will immediately return our anonymous inner function. And we can easily assign that to a variable for later use.

Okay. But do functions that return functions have any real use? It turns out that they do. You see, functions created inside another function have a handy property. They hold onto or “close over” the variables that their parent defined.

```
function counterCreator () {
    var count = 0;
    return function() { return ++count; };
}

var counter = counterCreator();

// Outputs: function() { return ++count; }
console.log(counter);

// Outputs: 1
console.log(counter());

// Outputs: 2
console.log(counter());

// Outputs: ReferenceError: count is not defined
console.log(count);
```

Interesting! See how our `counter` function is holding onto the value of `count` even across multiple invocations? And that's despite the fact that the `count` variable isn't accessible in our top scope.

```
function counterCreator () {
    var count = 0;
    return function() { return ++count; };
}

var originalCounter = counterCreator();
var anotherCounter = counterCreator();

// Outputs: 1
console.log(originalCounter());

// Outputs: 2
console.log(originalCounter());

// Outputs: 1
console.log(anotherCounter());
```

Well, look at that. Each new counter function that we create is able to maintain its own private `count` variable independent of the other counter functions. In fact, this aspect of JavaScript's functions is often used to emulate the sort of private properties found in languages like Java.

That's a little beyond the scope of this issue, but how else can "function functions" be useful? Here's one example:

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

function timesCreator (a) {
    return function (b) {
        return multiply(a, b);
    };
}

var timesTwo = timesCreator(2);
var timesTwelve = timesCreator(12);

// Outputs: 4
console.log(timesTwo(2));

// Outputs: 24
console.log(timesTwelve(2));
```

This particular approach is called partial application[22], which is excellent at helping you ensure that there is one canonical location for your core algorithm. When you need more specific variants, you can just generate them automatically.

Or, going back to our example with the counters, perhaps you want to create a function that can be invoked many times, but will only execute it's main logic once.

```
function once (fn) {
    var used = false;
    return function (something) {
        // End function execution if previously invoked
        if (used) { return; }

        // Invoke the original function
        fn(something);

        // Mark the function as used
        used = true;
    };
```

[22] http://en.wikipedia.org/wiki/Partial_application

```
}

function logStuff (stuff) {
    console.log(stuff);
}

var logOnce = once(logStuff);

// Outputs: "One does not simply"
logOnce("One does not simply");

// No output!
logOnce("use a function twice");
```

Hopefully, this has given you some ideas about how you might use "function functions". We'll be looking more at their practical use in the future. If you're interested in learning more about them, I suggest looking through Underscore's documentation[23] and source code[24].

[23] http://underscorejs.org/#functions

[24] http://underscorejs.org/docs/underscore.html#section-57

Using JavaScript's `toString` Method

Have you ever wondered why if you try to alert a plain old JavaScript object, you get the text `[object Object]`? That's due to JavaScript's built-in `toString` method. Let's take a look at how it works.

Note: In some places, the examples will use `alert` rather than `console.log` in order to bypass the special handling that consoles give to objects.

```
// Alerts: '[object Object]'
alert({});

// Outputs: "function() {}"
console.log(function() {});

// Outputs:
// "Thu Oct 11 1962 00:00:00 GMT-0400 (Eastern Daylight Time)"
// May vary depending on your clock settings
console.log(new Date("October 11, 1962"));
```

What do each of these things have in common? They are each providing a string representation of their values, but their internal values are quite different. For instance, internally JavaScript dates are measured in milliseconds since midnight of January 1, 1970 UTC.

So how are these objects providing that string representation? All three of them have a prototype method called `toString` which converts their internal value into something that makes sense as a string of text.

Let's suppose that we are creating a new type of object and want to give a reasonable string representation to use in messages to a user. Without using `toString`, our code might look like this:

```
function Color (r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
    this.text = "rgb(" + r + ", " + g + ", " + b + ")";
}

var red = new Color(255, 0, 0);

// Outputs: "rgb(255, 0, 0)"
alert(red.text);
```

But there are a couple of problems with this. First of all, the `text` property will get out of sync with the actual values if they are changed. Second, we have to remember to explicitly use `red.text` instead of just using the value of `red`.

Here's how that code looks after it is rewritten to use `toString` instead:

```
function Color (r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}

Color.prototype.toString = function () {
    return "rgb(" + this.r + ", " +
                    this.g + ", " +
                    this.b + ")";
};

var red = new Color(255, 0, 0);

// Alerts: "rgb(255, 0, 0)"
alert(red);
```

By moving the string representation into `toString` on the prototype we get two benefits. The string representation will always stay up to date with the actual values. And any time that we want to deal with the string value, `toString` will be implicitly called by the JavaScript interpreter. For instance:

```
var message = "Red is: " + red;

// Outputs: "Red is: rgb(255, 0, 0)"
console.log(message);
```

While you may not need to use `toString` every day, it's a good tool to have in your utility belt.

Invoking Functions With `call` and `apply`

A couple of drips ago, we talked about some of the implications of functions being first-class citizens in JavaScript. Here is a further implication to consider: **If functions are objects and objects can have methods, then functions can have methods.**

In fact, JavaScript functions come with several methods built into `Function.prototype`. First let's take a look at `call`.

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

// Outputs: 3
console.log(add(1, 2));

// Outputs: 3
console.log(add.call(this, 1, 2));
```

Assuming that you're not using strict mode[25], these invocations of `add` are exactly equivalent. The first parameter given to `call` has a special purpose, but any subsequent parameters are treated the same as if `add` had been invoked normally.

The first parameter that `call` expects, though, will be set to `add`'s internal `this` value. When a function is invoked ordinarily, the `this` value is set implicitly.

If you're not in strict mode and the function isn't attached to an object, then it will inherit its `this` from the global object. If the function is attached to an object, its default `this` is the receiver of the method call.

Let's look at how that works:

[25] http://www.nczonline.net/blog/2012/03/13/its-time-to-start-using-javascript-strict-mode/

```
var palestrina = {
    work: "Missa Papae Marcelli",
    describe: function() {
        console.log(this.work);
    }
};

// Outputs: "Missa Papae Marcelli",
palestrina.describe();
```

But `call` gives us a way to "borrow" a method from one object to use for another.

```
var erasmus = {
    work: "Freedom of the Will"
};

// Outputs: "Freedom of the Will"
palestrina.describe.call(erasmus);
```

You may be wondering how this is useful. But we've seen this approach before. Last time, we used it to invoke `Array` methods on a non-array (though array-like) object.

```
function myFunc () {
    // Invoke `slice` with `arguments`
    // as it's `this` value
    var args = Array.prototype.slice.call(arguments);
}
```

It's use extends beyond the `arguments` object, though. For instance, you can invoke many array methods on strings:

```
var original = "There is 1 number.";

var updated = Array.prototype.filter.call(original, function(val) {
    return val.match(/1/);
});

// Outputs: ["1"]
console.log(updated);

// Outputs: "1"
console.log(updated.join(''));
```

Of course, the return values of those methods will be arrays, so you may need to convert them back to strings with `join`.

So far we've only talked about `call`. So what's the deal with `apply`? It turns out that `apply` works in almost exactly the same way as `call`. The difference is that instead of a series of arguments, `apply` takes an array of values to use in its invocation.

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

// Outputs: 3
console.log(add.call(this, 1, 2));

// Outputs: 3
console.log(add.apply(this, [1, 2]));
```

In the example above, `call` and `apply` are used in exactly equivalent ways. As you can see, the only real difference is that `apply` takes an array.

But that turns out to be a very important difference. Unlike a series of arguments, an array is very easy to manipulate in JavaScript. And that opens up much larger possibilities for working with functions.

In the next drip, we'll explore some of those possibilities.

Using `apply` to Emulate JavaScript's Upcoming Spread Operator

The next version of JavaScript, ECMAScript 6, is planned to introduce a handy spread operator[26] which makes using arrays to supply function arguments much simpler. Here's an example of how it will work:

```
var someArgs = ["a", "b", "c"];

// Using the spread operator with someArgs
console.log(...someArgs);

// Is equivalent to this
console.log("a", "b", "c");
```

Now some of you might be thinking that `apply` gives us a nice easy way to emulate that, and you'd be right. This is also equivalent to the above:

```
console.log.apply(console, someArgs);
```

The `apply` method is part of the prototype of all functions, so any time we have a function, we can invoke it using `apply`. It takes two arguments, a `this` parameter which sets the value of `this` within the function, and a single array which is converted into a list of arguments to the function that `apply` is called on.

So if `apply` gives us the same thing as the spread operator, why is the ES6 committee adding spread at all? Because the spread operator is much more flexible. See for example:

[26] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Spread_operator

```
var someArgs = ["a", "b", "c"];
var moreArgs = [1, 2, 3];

// Using the spread operator twice, with a non-spread argument
console.log(...someArgs, "between", ...moreArgs);

// Is equivalent to this
console.log("a", "b", "c", "between", 1, 2, 3);
```

In this case, there isn't a simple one to one mapping between using the spread operator and using `apply`. But we can still create a pretty decent emulation. Let's take a look at how.

```
function spreadify (fn, fnThis) {
    return function (/* accepts unlimited arguments */) {
        // Holds the processed arguments for use by `fn`
        var spreadArgs = [ ];

        // Caching length
        var length = arguments.length;

        var currentArg;

        for (var i = 0; i < length; i++) {
            currentArg = arguments[i];

            if (Array.isArray(currentArg)) {
                spreadArgs = spreadArgs.concat(currentArg);
            } else {
                spreadArgs.push(currentArg);
            }
        }

        fn.apply(fnThis, spreadArgs);
    };
}

var someArgs = ["a", "b", "c"];
var moreArgs = [1, 2, 3];

// Outputs: ["a", "b", "c"] [1, 2, 3]
console.log(someArgs, moreArgs);

```

```
// Outputs: a b c 1 2 3
spreadify(console.log, console)(someArgs, moreArgs);
```

What on earth is going on here? It's actually pretty simple. Our `spreadify` function takes two arguments, a function, and an optional parameter specifying what the function's `this` should be. In the case of `console.log`, that is `console`.

When the `spreadify` function is invoked, it immediately returns another function. That's why we have the doubled parentheses at the end of the example. But you could also do something like this:

```
// `spreadLog` is now the function that `spreadify` returns
var spreadLog = spreadify(console.log, console);

// Outputs: a b c 1 2 3
spreadLog(someArgs, moreArgs);
```

So we have our new function. That new function does two things.

1. It does some preprocessing on whatever arguments we pass it. (Either immediately with that second set of parentheses in the first example, or later on as in the second example.)
2. It then invokes our original function (`log` in this case) with the processed arguments.

In our preprocessing we use the `arguments` object to examine all the arguments which were passed in. If an argument is an array, then we add the array's contents to `spreadArgs`. If the argument is not an array, then we directly push the argument onto `spreadArgs`.

Finally, we invoke our original function by using `apply`, passing in the `this` value, as well the `spreadArgs` array.

In the event that you didn't pass a `this` value into `spreadify`, `fnThis` will be `undefined`, and `apply` will use the default value of `this` (usually the global object).

You may have noticed one problem with our emulation of the spread operator here. The `spreadify` function assumes that all arrays should be expanded into individual arguments. That may not always be the case. Fortunately there's a fairly simple workaround. Just wrap any array you want to preserve as an individual argument inside another array, like so:

```
var someArgs = ["a", "b", "c"];
var moreArgs = [1, 2, 3];
var myArray = ["my", "array"];

// `spreadLog` is now the function that `spreadify` returns
var spreadLog = spreadify(console.log, console);

// Outputs: a b c 1 2 3 ["my", "array"]
spreadLog(someArgs, moreArgs, [myArray]);
```

Of course, `spreadify` doesn't completely replace the spread operator, since in ES6 you can also spread into an array literal:

```
var alpha = ["a", "b", "c"];
var num = [1, 2, 3];

var all = ["all", ...alpha, ...num];
```

But `spreadify` can still be pretty useful. And hopefully walking through how it uses `apply` has helped give you a better grasp of how powerful JavaScript functions can be.

Boiling Down Arrays with Array#reduce

We've talked before about the advantages of using `Array`'s higher order functions (like `map` and `sort`) rather than manually iterating over an array's contents. But unlike `map` and `sort`, which produce new arrays, `reduce` can boil an array down to a new value of any type.

Consider the case of a library system which needs to show potential donors its effectiveness at spreading knowledge. In order to do that, it must determine how many book-checkouts there were last year. (Not total of all checkouts, since a single checkout may be for more than one book.)

Their data is stored in an object like this:

```
var books = [
    {
        title: "Showings",
        author: "Julian of Norwich",
        checkouts: 45
    },
    {
        title: "The Triads",
        author: "Gregory Palamas",
        checkouts: 32
    },
    {
        title: "The Praktikos",
        author: "Evagrius Ponticus",
        checkouts: 29
    }
];
```

The traditional C-style way to get at that number would be something like this:

```
var bookCheckouts = 0;

for (var i = 0; i < books.length; i++) {
    bookCheckouts = bookCheckouts + books[i].checkouts;
}
```

But using `reduce`, we can think more directly about the problem itself:

```
// Get an array of checkout values only
var bookCheckouts = books.map(function(item) {
    return item.checkouts;
});

// Sum the array's values from left to right
var total = bookCheckouts.reduce(function(prev, curr) {
    return prev + curr;
});
```

While it may look a little foreign at first, this example is more explicit about the problem than the `for` version is. Namely, dealing only with the `checkouts` property, and totaling them. And if you are familiar with the idioms, you can make it almost as concise.

```
// Get total of book checkouts
var total = books
            .map(function(b) { return b.checkouts; })
            .reduce(function(p, c) { return p + c; });
```

So how does that `reduce` method work? It takes an array of values and executes the function (almost!) once for each item in the array. The value of `prev` is the result of the calculations up to this point. Let's take a look:

```
[1, 2, 3].reduce(function(prev, curr, index) {
    console.log(prev, curr, index);
    return prev + curr;
});

// Outputs:
// 1, 2, 1
// 3, 3, 2
// 6
```

As you can see, the callback actually didn't iterate over the first element of the array. Instead, `prev` was set to the value of the first element. It has to work that way because otherwise `reduce` wouldn't have a starting value for `prev`.

The function ran three times, but on the last call, it wasn't given a current element since there were no elements left in the array. That behavior is a little confusing to read through, but is quite easy to use.

Here is another example:

```
[1, 2, 3].reduce(function(prev, curr, index) {
    console.log(prev, curr, index);
    return prev + curr;
}, 100);

// Outputs:
// 100 1 0
// 101 2 1
// 103 3 2
// 106
```

In addition to the callback function, we also passed in an initial value for `prev`. You'll note that in this case, the callback **did** iterate over the first element in the array, since it had an initial value to work with.

It should be noted that your callback function also has access to the array itself as a parameter.

```
[1, 2, 3].reduce(function(prev, curr, index, arr) {
    // Do something here
});
```

Is reduce only good for working with numbers? By no means. Consider this:

```
var relArray = [
    ["Viola", "Orsino"],
    ["Orsino", "Olivia"],
    ["Olivia", "Cesario"]
];

var relMap = relArray.reduce(function(memo, curr) {
    memo[curr[0]] = curr[1];
    return memo;
}, {});
```

```
11
12 // Outputs: {
13 //    "Viola": "Orsino",
14 //    "Orsino": "Olivia",
15 //    "Olivia": "Cesario"
16 // }
17 console.log(relMap);
```

This time we used `reduce` to transform the relationship data from an array of arrays into an object where each key value pair describes the relationship. You'll also see that instead of naming the parameter `prev` as I did before, I used the term `memo`. That's because in this case, `memo` is a more descriptive name for the actual value being passed in (an object collecting new properties.)

The `reduce` method is good for any sort of situation where we want to take an array and boil it down into a new value in steps that have access to one or two of the array elements at a time.

Want to use `reduce` in IE8 or IE7? Take a look at the Underscore[27] and Lo-Dash[28] libraries.

Date: May 14, 2013

[27] http://underscorejs.org/#reduce

[28] http://lodash.com/docs#reduce

Making Deep Property Access Safe in JavaScript

If you've been working with JavaScript for any length of time, you've probably run across the dreaded `TypeError: Cannot read property 'someprop' of undefined` and the similar error for `null`.

```
var rels = {
    Viola: {
        Orsino: {
            Olivia: {
                Cesario: null
            }
        }
    }
};

// Outputs: undefined
console.log(rels.Viola.Harry);

// TypeError: Cannot read property 'Sally' of undefined
console.log(rels.Viola.Harry.Sally);
```

The problem, of course, is that a `TypeError` immediately halts execution of your code. It's simple to deal with when you have predictable inputs, but when you need to access a deep object property that may or may not be there it can be quite problematic.

Sometimes you can solve this by merging with a default object, but at other times that doesn't make sense.

Often, what we really want is to be able to ask for a deep property and just find out whether it has a proper value. If the deep property's parent or grandparent is `undefined`, then for our purposes the property can be considered `undefined` as well.

Let's take a look at a solution:

```
function deepGet (obj, properties) {
    // If we have reached an undefined/null property
    // then stop executing and return undefined.
    if (obj === undefined || obj === null) {
        return;
    }

    // If the path array has no more elements, we've reached
    // the intended property and return its value.
    if (properties.length === 0) {
        return obj;
    }

    // Prepare our found property and path array for recursion
    var foundSoFar = obj[properties[0]];
    var remainingProperties = properties.slice(1);

    return deepGet(foundSoFar, remainingProperties);
}
```

The `deepGet` function will recursively search a given object until it reaches an `undefined` or `null` property, or until it reaches the final property specified in the `properties` array.

Let's try it out.

```
// Outputs: { Cesario: null }
console.log(deepGet(rels, ["Viola", "Orsino", "Olivia"]));

// Outputs: undefined
console.log(deepGet(rels, ["Viola", "Harry"]));

// Outputs: undefined
console.log(deepGet(rels, ["Viola", "Harry", "Sally"]));
```

Excellent!

Of course, we probably want to use this value in some way. And it's unlikely that `undefined` in itself will be all that useful.

```
var oliviaRel = deepGet(rels, ["Viola", "Orsino", "Olivia"]);
var sallyRel = deepGet(rels, ["Viola", "Harry", "Sally"]);

// Produces a pretty graph of Olivia's love interest
graph(oliviaRel);

// Tries to produce a graph of Sally's love interest
graph(sallyRel);
```

The problem here is that we have to explicitly handle `undefined` in our `graph` function. But what if we are using a third party library that doesn't check for `undefined`? We could use the "or" trick, like so:

```
graph(sallyRel || {});
```

But that's not very explicit about our intentions, and will also fail if `sallyRel` happens to be `false` or another falsy value like `0` or `""`.

Alternately, we could explicitly check for `null` and `undefined`.

```
if (sallyRel === undefined || sallyRel === null) {
    sallyRel = {};
}

graph(sallyRel);
```

But that seems unnecessarily verbose.

It would be much nicer if we could just specify a default value to return instead of `undefined`. So how would we do that?

```
function deepGet (obj, props, defaultValue) {
    // If we have reached an undefined/null property
    // then stop executing and return the default value.
    // If no default was provided it will be undefined.
    if (obj === undefined || obj === null) {
        return defaultValue;
    }

    // If the path array has no more elements, we've reached
    // the intended property and return its value
    if (props.length === 0) {
```

```
12          return obj;
13      }
14
15      // Prepare our found property and path array for recursion
16      var foundSoFar = obj[props[0]];
17      var remainingProps = props.slice(1);
18
19      return deepGet(foundSoFar, remainingProps, defaultValue);
20  }
21
22  var sallyRel = deepGet(rels, ["Viola", "Harry", "Sally"], {});
23
24  // Will output a graph based on the empty object
25  graph(sallyRel);
```

Now we have a nice safe way to do deep property access and even get back a useful value when the property doesn't have one.

If you find this utility useful or interesting, I have open-sourced it on GitHub[29]. I've even added some syntactic sugar so you can use a string-based property list, like `Viola.Harry.Sally`.

[29] https://github.com/joshuacc/drabs

Testing Array Contents with Array#every

We've talked before about how common it is to test the contents of an array to figure out what to do next, and how Array's some method will let us test whether some elements meet our criteria. But sometimes we need to make sure that **every** element in an array meets certain criteria.

Here is an example of the pattern at work:

```
var heroes = [
    { name: "Superman",     universe: "DC"     },
    { name: "Batman",       universe: "DC"     },
    { name: "Spider-Man",   universe: "Marvel" },
    { name: "Wonder Woman", universe: "DC"     }
];

// Default to true
var areAllDC = true;

for (var i = 0; i < heroes.length && areAllDC; i++) {
    if (heroes[i].universe !== "DC") {
        areAllDC = false;
    }
}

// Outputs: false
console.log(areAllDC);
```

We iterate over the array with a for loop, but with an extra check on `areAllDC` to halt iteration as soon as we've determined that there is a non-matching element.

So, that's the "classical" looping approach. Fortunately, JavaScript also gives us a very nice built-in function to do this work for us. And we can do so without having to constantly write loops and keep track of extraneous variables and code.

The `every` function is available on all arrays, and we can use it like this:

```
function isDC(element) {
    return (element.universe === "DC");
}

// Outputs: false
console.log(heroes.every(isDC));

var villains = [
    { name: "Brainiac", universe: "DC" },
    { name: "Sinestro", universe: "DC" },
    { name: "Darkseid", universe: "DC" },
    { name: "Joker",    universe: "DC" }
];

// Outputs: true
console.log(villains.every(isDC));
```

The callback will continue to be executed on each element until it returns a falsy value (`false`, `undefined`, etc.) or it reaches the end of the array. If it reaches the end without returning a falsy value, then `every` will return `true`.

The callback function also has access to two other parameters: the current index, and the array as a whole. You can use them to evaluate the current element in the context of the entire array.

For example:

```
function isSameUniverse(el, index, arr) {
    // The first element doesn't have a predecessor,
    // so don't evaluate it.
    if (index === 0) {
        return true;
    } else {
        // Do the universe properties of
        // current and previous elements match?
        return (el.universe === arr[index - 1].universe);
    }
}

// Outputs: false
console.log(heroes.every(isSameUniverse));

// Outputs: true
console.log(villains.every(isSameUniverse));
```

Because `every` is part of the ES5 specification[30], it isn't available in IE8 and below. But if you want to use it in older browsers, you can use Underscore[31], Lo-Dash[32], or an ES5 shim[33] to make it available.

[30] http://en.wikipedia.org/wiki/ECMAScript_5#ECMAScript.2C_5th_Edition
[31] http://underscorejs.org/#every
[32] http://lodash.com/docs#every
[33] https://github.com/kriskowal/es5-shim

The Difference Between Boolean Objects and Boolean Primitives

One of the unintuitive things about JavaScript is the fact that there are constructors for each of the primitive value types (boolean, string, etc), but what they construct isn't actually the same thing as the primitive.

Take booleans, for example. In most code, the primitive values are used, like so:

```
var primitiveTrue = true;
var primitiveFalse = false;
```

There is also the `Boolean` function, which can be used as an ordinary function which returns a boolean primitive:

```
var functionTrue = Boolean(true);
var functionFalse = Boolean(false);
```

But the `Boolean` function can also be used as a constructor with the `new` keyword:

```
var constructorTrue = new Boolean(true);
var constructorFalse = new Boolean(false);
```

The tricky thing here is that when `Boolean` is used as a constructor, it doesn't return a primitive. Instead it returns an object.

```
// Outputs: true
console.log(primitiveTrue);

// Outputs: true
console.log(functionTrue);

// Outputs: Boolean {}
console.log(constructorTrue);
```

It turns out that using the `Boolean` constructor can be quite dangerous. Why? Well, JavaScript is pretty aggressive about type coercion. If you try adding a string and a number, the number will be coerced into a string.

```
// Outputs: "22"
console.log("2" + 2);
```

Likewise, if you try to use an object in a context that expects a boolean value, the object will be coerced to `true`.

```
// Outputs: "Objects coerce to true."
if ({}) { console.log("Objects coerce to true."); }
```

And since the `Boolean` object is an object, it will also coerce to `true`, even if its internal value is `false`.

```
// Outputs: "My false Boolean object is truthy!"
if (constructorFalse) {
    console.log("My false Boolean object is truthy!");
} else {
    console.log("My false Boolean object is falsy!");
}
```

If you actually need to get at the internal value of a `Boolean` object, then you'll need to use the `valueOf` method.

```
// Outputs: "The value of my false Boolean object is falsy!"
if (constructorFalse.valueOf()) {
    console.log("The value of my false Boolean object is truthy!");
} else {
    console.log("The value of my false Boolean object is falsy!");
}
```

But because of the quirks surrounding `Boolean` objects, you're probably best off avoiding them altogether. In fact, linting tools like JSHint and JSLint will flag the `Boolean` constructor as a potential error in your code.

In the event that you need to explicitly coerce another type of value into `true` or `false`, you're better off using `Boolean` as an ordinary function, or using the not operator twice.

```
// Two approaches to coercing 0 into false
var byFunction = Boolean(0);
var byNotNot = !!0;
```

The double not above is pretty simple, though it can be confusing if you haven't seen it before. Using a single not operator coerces the value into a boolean primitive and then reverses it. (To `true` in this case). The second use of the not operator reverses the value again, so that it is flipped back to the correct boolean representation of the original value.

You're reasonably likely to see both of these approaches in JavaScript code, though in my experience the double not is more common.

An Introduction to Writing Your Own JavaScript Compatibility Shims

We've talked in several past issues about the need to use compatibility shims to get newer features of JavaScript ported back into older browsers. Today we're going to talk about how those shims work and how you can write your own.

We are going to create a shim for the previously discussed Array#map, which is not available in IE8 and below. To recap, the `map` method returns a new array which is created by running a transformation function over each element of the original array, like so:

```
var evens = [2, 4, 6, 8, 10];

var odds = evens.map(function(val) { return val - 1; });

// Outputs: [1, 3, 5, 7, 9]
console.log(odds);
```

Perhaps the most important part of writing a compatibility shim is using feature detection to ensure that we don't accidentally overwrite the `map` method in a modern browser.

```
// Only use the shim if map isn't defined
if (!Array.prototype.map) {
    // Create and attach shim here
}
```

Once we know that we are in a `map`less browser, we can actually create the shim itself.

```
// Only use the shim if map isn't defined
if (!Array.prototype.map) {
    // Map will accept a single function as an argument
    Array.prototype.map = function(fn) {
        var i,
            newVal,
            mappedArr = [];

        for (i = 0; i < this.length; i++) {
```

```
            // Run the callback function over each element
            // to get the updated value
            newVal = fn(this[i]);

            // Push the new value onto the new array
            mappedArr.push(newVal);
        }

        // Return our new array
        return mappedArr;
    };
}
```

Now if you load that code up in IE8 and below, you should have a functional (if incomplete) `map` method where it was completely missing before.

Of course, it's worth pointing out that this particular implementation of `map` isn't production-worthy. For one thing, it doesn't take into account the fact that `map` will pass in the index and array as arguments to the callback function. It also fails to account for the fact that `map` itself will accept a second argument to set its `this` value. For details, see our previous discussion of map, as well as the detailed documentation on MDN[34].

And that leads me to the second most important part of writing a compatibility shim. You need to make sure that you get the functionality right. A lot of the time it is really simple to code for the most common use cases, but much more difficult to ensure that you've covered all of the uncommon ones. If you'd like to see how much of a difference that makes, take a look at ES5-shim's implementation of map[35]. It is significantly more complex.

Because of this complexity, it is generally best to rely on an established library with lots of unit tests to do the heavy lifting for you. However, if your goal is to learn about a new feature of JavaScript, building your own implementation will teach you more of the ins and outs than anything else.

[34] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/map

[35] https://github.com/kriskowal/es5-shim/blob/master/es5-shim.js#L349

Redefining `undefined`

When developers start listing the quirks of JavaScript one of the things you are likely to hear is that it is possible to assign a new value to `undefined`. While under certain circumstances this is correct, the full truth is a bit more complex.

Let's start with the simplest case. Can you assign a new value to `undefined`?

```
undefined = "my string";

console.log(undefined);
```

It turns out that the outcome of this code depends on which version of the ECMAScript standard your JavaScript engine supports. In ES3 engines (like IE8 & below) "my string" is logged, while in ES5 engines `undefined` is logged. The reason is that ES5 silently prevents assigning a new value to `undefined`.

Then of course, there is ES5's strict mode:

```
"use strict";

undefined = "my string";
```

If you turn on strict mode as in this example, you'll find that attempting to assign a value to `undefined` will throw an error instead of failing silently. (This is one of the benefits of using strict mode.) But of course, this only works in engines that conform to ES5.

So far we have three different types of behaviors. But that's not the end of the story. What exactly does it mean to assign a new value to `undefined` in older JS engines? Isn't `undefined` one of JavaScript's built-in types?

Let's explore this idea in an ES3 browser. (I'm using IE8.)

```
undefined = "my string";

// Because no value is assigned, normally it would be undefined
var unassigned;

// Outputs: "my string"
console.log(undefined);

// Outputs: undefined
console.log(unassigned);
```

What? We just redefined what `undefined` means! Shouldn't `unassigned` be equivalent to "my string" instead of just `undefined`?

Though it is counterintuitive, this behavior actually does make sense when you add one more crucial bit of information. There is more than one meaning for the term `undefined` in JavaScript.

1. The `undefined` type.
2. The immutable value `undefined` which is the only instance of the type.
3. The global variable `undefined` which is a reference to the value `undefined`. In ES3 this variable is mutable, while in ES5 it is immutable.

Now can you see why it was possible to assign a new value to `undefined`? It's because we were assigning a new value to the global variable. And the `unassigned` variable above gave us a reference to the immutable value rather than the (potentially mutable) variable.

This situation seems a bit odd. Is there really a global variable `undefined`? And if so, is there a way to detect this global variable in ES5 engines? Certainly.

Since global variables are really properties of the global object (`window` in the browser or `global` in Node.js), we can use JavaScript's `in` operator to look for it. (I'm using Chrome.)

```
var isUndefinedAGlobalVariable = "undefined" in window;

// Outputs: true
console.log(isUndefinedAGlobalVariable);
```

How does this compare to something like `null`? After all, they are fairly similar sorts of values.

```
var isNullAGlobalVariable = "null" in window;

// Outputs: false
console.log(isNullAGlobalVariable);
```

That's confusing. Why the difference? Because unlike `undefined`, `null` is a JavaScript keyword. Due to that, `null` is inherently unassignable, while ES5 had to add a special rule to cover the `undefined` global variable.

While it is becoming less of a problem over time, popular libraries that support older browsers (like jQuery) have to defend themselves against the possibility of other code on the page redefining `undefined`. To do so, they often use this pattern:

```
// Misbehaving code
undefined = "some other value";

// Outputs: "some other value"
console.log(undefined);

// Library code
var libraryWrapper = function (undefined) {
    console.log(undefined);
};

// Outputs: undefined
libraryWrapper();
```

By declaring a parameter named `undefined` for the function, but not passing in a value when it is invoked, they guarantee that within their function the local variable `undefined` will reference the value `undefined`, even if the global variable `undefined` does not.

And that is the story behind redefining `undefined`.

The Uses of `in` vs `hasOwnProperty`

Last issue I briefly mentioned JavaScript's `in` operator, but didn't go into much detail about its use. That's the subject we'll be tackling first today.

The `in` operator will tell you whether an object (or array) has a property name which matches a given string.

```
var fantasyLit = {
    tolkien: "The Lord of the Rings",
    lewis: "The Chronicles of Narnia"
};

// Outputs: true
console.log("tolkien" in fantasyLit);

// Outputs: false
console.log("asimov" in fantasyLit);
```

Looks simple enough, right? But consider this:

```
// Outputs: true
console.log("constructor" in fantasyLit);
```

What's going on here? It turns out that the `in` operator doesn't distinguish between properties created specifically on an object and properties that the object inherited from the prototype chain. In this case `in` is seeing the `constructor` property of `Object.prototype` which all objects inherit from.

It will also return `true` for user-defined prototype properties. For instance:

```
function litList () {}

litList.prototype.addToList = function (author, work) {
    this[author] = work;
};

var fantasyLit = new litList();

fantasyLit.addToList("tolkien", "The Lord of the Rings");

// Outputs: true
console.log("tolkien" in fantasyLit);

// Outputs: false
console.log("asimov" in fantasyLit);

// Outputs: true
console.log("addToList" in fantasyLit);
```

Because of this, using `in` to detect whether an object possesses a given property can be a bit deceptive. Usually we only want to check for properties that belong to the object itself, not its prototype chain. Fortunately, JavaScript has a solution for that. It is called `hasOwnProperty`.

It is a method on `Object.prototype`, which means it is available to all JavaScript objects.

Here is how you use it:

```
function litList () {}

litList.prototype.addToList = function (author, work) {
    this[author] = work;
};

var fantasyLit = new litList();

fantasyLit.addToList("tolkien", "The Lord of the Rings");

// Outputs: true
console.log(fantasyLit.hasOwnProperty("tolkien"));

// Outputs: false
console.log(fantasyLit.hasOwnProperty("asimov"));

```

```
// Outputs: false
console.log(fantasyLit.hasOwnProperty("addToList"));
```

Because in JavaScript arrays also inherit from `Object`, they can use `hasOwnProperty` as well, though it is often less useful.

```
var summerMovies = [
    "Iron Man 3",
    "Star Trek: Into Darkness",
    "Man of Steel"
];

// Outputs: true
summerMovies.hasOwnProperty("2");
```

It's important to keep in mind the limits of both `in` and `hasOwnProperty`. While they can tell you that a given property has been declared, they can't tell you whether the property has a "real value".

Consider these examples:

```
// Puts a "declared" property on the global object
// (window in browsers)
var declared;

// Outputs: true
console.log("declared" in window);

// Outputs: true
console.log(window.hasOwnProperty("declared"));

// Outputs: undefined
console.log(declared);

var obj = { myUndefined: undefined };

// Outputs: true
console.log("myUndefined" in obj);

// Outputs: true
console.log(obj.hasOwnProperty("myUndefined"));

// Outputs: undefined
console.log(obj.myUndefined);
```

Another limit that you may encounter is that the `hasOwnProperty` method can be rendered useless if an object happens to define a property named `hasOwnProperty`. For instance:

```
var voldemort = {
    hasOwnProperty: function () { return true; }
};

// Outputs: true
console.log(voldemort.hasOwnProperty("ridikulus"));
```

Because `voldemort` defines its own `hasOwnProperty`, the call never makes it to `Object.prototype.hasOwnProperty`. It's unlikely that you'll run into an object as maliciously constructed as `voldemort`, but it's good to be aware of the possibility. Here is the workaround:

```
// Returns false
Object.prototype.hasOwnProperty.call(voldemort, "ridikulus");
```

Finally, depending on the JavaScript engine, you may have trouble detecting the special `__proto__` property with either of these methods.

That's a brief introduction to the use of `in` and `hasOwnProperty`.

An Introduction to IIFEs - Immediately Invoked Function Expressions

It doesn't take very long working with JavaScript before you come across this pattern:

```
(function () {
    // logic here
})();
```

Your first encounter is likely to be quite confusing. But fortunately the concept itself is simple. The pattern is called an immediately invoked function expression, or IIFE (pronounced "iffy").

In JavaScript functions can be created either through a function declaration or a function expression. A function declaration is the "normal" way of creating a named function.

```
// Named function declaration
function myFunction () { /* logic here */ }
```

On the other hand, if you are assigning a function to a variable or property, you are dealing with a function expression.

```
// Assignment of a function expression to a variable
var myFunction = function () { /* logic here */ };

// Assignment of a function expression to a property
var myObj = {
    myFunction: function () { /* logic here */ }
};
```

A function created in the context of an expression is also a function expression. For example:

```
// Anything within the parentheses is part of an expression
(function () { /* logic here */ });

// Anything after the not operator is part of an expression
!function () { /* logic here */ };
```

The key thing about JavaScript expressions is that they return values. In both cases above the return value of the expression is the function.

That means that if we want to invoke the function expression right away we just need to tackle a couple of parentheses on the end. Which brings us back to the first bit of code we looked at.

```
(function () {
    // logic here
})();
```

Now we know what the code is doing, but the question "Why?" still remains.

The primary reason to use an IIFE is to obtain data privacy. Because JavaScript's `var` scopes variables to their containing function, any variables declared within the IIFE cannot be accessed by the outside world.

```
(function () {
    var foo = "bar";

    // Outputs: "bar"
    console.log(foo);
})();

// ReferenceError: foo is not defined
console.log(foo);
```

Of course, you could explicitly name and then invoke a function to achieve the same ends.

```
function myImmediateFunction () {
    var foo = "bar";

    // Outputs: "bar"
    console.log(foo);
}

myImmediateFunction();

// ReferenceError: foo is not defined
console.log(foo);
```

However, this approach has a few downsides. First, it unnecessarily takes up a name in the global namespace, increasing the possibility of name collisions. Second, the intentions of this code aren't as self-documenting as an IIFE. And third, because it is named and isn't self-documenting it might accidentally be invoked more than once.

It is worth pointing out that you can easily pass arguments into the IIFE as well.

```
var foo = "foo";

(function (innerFoo) {
    // Outputs: "foo"
    console.log(innerFoo);
})(foo);
```

And that's the story behind IIFEs. Soon we'll be building on this by taking a look at the module pattern in JavaScript.

Variable and Function Hoisting in JavaScript

One of the trickier aspects of JavaScript for new JavaScript developers is the fact that variables and functions are "hoisted." Rather than being available after their declaration, they might actually be available beforehand. How does that work? Let's take a look at variable hoisting first.

```
// ReferenceError: noSuchVariable is not defined
console.log(noSuchVariable);
```

This is more or less what one would expect. An error is thrown when you try to access the value of a variable that doesn't exist. But what about this case?

```
// Outputs: undefined
console.log(declaredLater);

var declaredLater = "Now it's defined!";

// Outputs: "Now it's defined!"
console.log(declaredLater);
```

What is going on here? It turns out that JavaScript treats variables which will be declared later on in a function differently than variables that are not declared at all. Basically, the JavaScript interpreter "looks ahead" to find all the variable declarations and "hoists" them to the top of the function. Which means that the example above is equivalent to this:

```
var declaredLater;

// Outputs: undefined
console.log(declaredLater);

declaredLater = "Now it's defined!";

// Outputs: "Now it's defined!"
console.log(declaredLater);
```

One case where this is particularly likely to bite new JavaScript developers is when reusing variable names between an inner and outer scope. For example:

```
var name = "Baggins";

(function () {
    // Outputs: "Original name was undefined"
    console.log("Original name was " + name);

    var name = "Underhill";

    // Outputs: "New name is Underhill"
    console.log("New name is " + name);
})();
```

In cases like this, the developer probably expected `name` to retain its value from the outer scope until the point that `name` was declared in the inner scope. But due to hoisting, `name` is `undefined` instead.

Because of this behavior JavaScript linters and style guides often recommend putting all variable declarations at the top of the function so that you won't be caught by surprise.

So that covers variable hoisting, but what about function hoisting? Despite both being called "hoisting," the behavior is actually quite different. Unlike variables, a function declaration doesn't just hoist the function's name. It also hoists the actual function definition.

```
// Outputs: "Yes!"
isItHoisted();

function isItHoisted() {
    console.log("Yes!");
}
```

As you can see, the JavaScript interpreter allows you to use the function before the point at which it was declared in the source code. This is useful because it allows you to express your high-level logic at the beginning of your source code rather than the end, communicating your intentions more clearly.

```
travelToMountDoom();
destroyTheRing();

function travelToMountDoom() { /* Traveling */ }
function destroyTheRing() { /* Destruction */ }
```

However, **function definition hoisting only occurs for function declarations**, not function expressions. For example:

```
// Outputs: "Definition hoisted!"
definitionHoisted();

// TypeError: undefined is not a function
definitionNotHoisted();

function definitionHoisted() {
    console.log("Definition hoisted!");
}

var definitionNotHoisted = function () {
    console.log("Definition not hoisted!");
};
```

Here we see the interaction of two different types of hoisting. Our variable `definitionNotHoisted` has its declaration hoisted (thus it is `undefined`), but not its function definition (thus the `TypeError`.)

You might be wondering what happens if you use a named function expression.

```
// ReferenceError: funcName is not defined
funcName();

// TypeError: undefined is not a function
varName();

var varName = function funcName() {
    console.log("Definition not hoisted!");
};
```

As you can see, the function's name doesn't get hoisted if it is part of a function expression.

And that is how variable and function hoisting works in JavaScript.

Creating Bound Functions with Function#bind

Continuing from some of our previous discussions of functions as first-class values, it's time to tackle `Function`'s `bind` method. As you might guess, the purpose of `bind` is to "bind" a function. But what does that mean, exactly?

```
:::javascript
var hero = {
    name: "Batman",
    signal: function () {
        console.log(this.name + " has been signaled.");
    }
};

// Outputs: "Batman has been signaled."
hero.signal();
```

Consider if we wanted to let another object signal Batman. We could do something like this:

```
:::javascript
var commissioner = {
    name: "Jim Gordon",
    signalBatman: function() {
        hero.signal();
    }
};

// Outputs: "Batman has been signaled."
commissioner.signalBatman();
```

But what if the `hero` variable gets redefined?

```
:::javascript
hero = {
    name: "Superman",
    signal: function () {
        console.log(this.name + " has been signaled.");
    }
};

// Outputs: "Superman has been signaled."
commissioner.signalBatman();
```

Commissioner Gordon is signaling Superman? That can't be right. Let's try using `bind` instead.

```
:::javascript
var hero = {
    name: "Batman",
    signal: function () {
        console.log(this.name + " has been signaled.");
    }
};

var commissioner = {
    name: "Jim Gordon",
    signalBatman: hero.signal.bind(hero)
};

hero = {
    name: "Superman",
    signal: function () {
        console.log(this.name + " has been signaled.");
    }
};

// Outputs: "Batman has been signaled."
commissioner.signalBatman();
```

As you can see, the `bind` method allows us to create a new function which is permanently bound to a given value of `this`. You can't even override its `this` value using `call` or `apply`. This can be quite handy when you need to pass around a function that needs a certain `this` value in order to function correctly.

For instance, consider good old `console.log`:

```
:::javascript
// Outputs: "logging"
console.log("console.logging");

var justLog = console.log;

// TypeError: Illegal invocation
justLog("just logging");
```

It turns out that `log` just won't work without `console`. But is there a way we can just pass around the function instead of the entire console object? With `bind` there is.

```
:::javascript
// Outputs: "logging"
console.log("console.logging");

var justLog = console.log.bind(console);

// Outputs: "just logging"
justLog("just logging");
```

Unfortunately, `bind` is only supported in Internet Explore 9 or higher. If you need this functionality in older browsers, you can use Underscore[36], Lo-Dash[37], or the ES5 shim[38] library.

That's a brief introduction to creating bound functions with `bind`. Next time we'll look at using `bind` for partial application.

[36] http://underscorejs.org/#bind
[37] http://lodash.com/docs#bind
[38] https://github.com/kriskowal/es5-shim

Partial Application with Function#bind

In the last drip, we covered using `bind` to create bound functions. But `bind` is also a very convenient way of implementing partial application. First, let's take a look at exactly what partial application is.

According to Wikipedia[39], partial application "refers to the process of fixing a number of arguments to a function, producing another function of smaller arity." Arity refers to the number of arguments that a function takes.

Here's an example:

```
:::javascript
function multiply (x, y) {
    return x * y;
}

function double (num) {
    return multiply(2, num);
}
```

That's a very rudimentary form of partial application, where inside `double` we permanently fix `multiply`'s first argument as `2`, producing a new function `double`.

But explicitly declaring the body of the new function isn't actually necessary if you use `bind`. Here's how we could rewrite that:

```
:::javascript
function multiply (x, y) {
    return x * y;
}

var double = multiply.bind(null, 2);
```

As we covered last time, the first argument to `bind` sets it's internal `this` value. Since our function doesn't depend on `this`, we just pass in `null`. But any subsequent arguments that we supply will be used to permanently fix the arguments of the function we are binding. In this case, we are only fixing one argument, but we can potentially fix as many as we want.

[39] https://en.wikipedia.org/wiki/Partial_application

```
:::javascript
function greet (salutation, person, delivery) {
    var message = '"'+ salutation + ', ' + person + ',"' ' +
                  delivery + ' the greeter.';

    console.log(message);
}

var hail = greet.bind(null, "Hail");

// Outputs: '"Hail, Lord Elrond," said the greeter.'
hail("Lord Elrond", "said");

var begone = greet.bind(null, "Begone", "Wormtongue", "commanded");

// Outputs: '"Begone, Wormtongue," commanded the greeter.'
begone();
```

One thing you may have noticed is that `bind` always fixes the arguments from left to right. This means that it isn't suitable for all forms of partial application, but it does cover the most common ones.

You may be wondering if partial application is actually all that useful in real code. To answer, let's revisit an example from our discussion of dispatch tables:

```
:::javascript
var commandTable = {
    north:    function() { movePlayer("north"); },
    east:     function() { movePlayer("east");  },
    south:    function() { movePlayer("south"); },
    west:     function() { movePlayer("west");  },
    look:     describeLocation,
    backpack: showBackpack
};

function processUserInput(command) {
    commandTable[command]();
}
```

In this example we are taking user input from a text adventure game, and then calling the appropriate command from `commandTable`. As you can see, we're using the same rudimentary form of partial application that we saw in our first example.

Using `bind` we can clean this up a bit.

```
:::javascript
var commandTable = {
    north:    movePlayer.bind(null, "north"),
    east:     movePlayer.bind(null, "east"),
    south:    movePlayer.bind(null, "south"),
    west:     movePlayer.bind(null, "west"),
    look:     describeLocation,
    backpack: showBackpack
};

function processUserInput(command) {
    commandTable[command]();
}
```

As I mentioned last time, `bind` isn't available in IE8 and older, so you may want to use a polyfill or library to achieve the same functionality. If you only want to implement partial application and don't need to create bound functions, you might want to consider the `partial` methods in Underscore[40] and Lo-Dash[41].

[40] http://underscorejs.org/#partial

[41] http://lodash.com/docs#partial

Determining if a String Contains a Substring

One of the most basic tasks in any programming language is determining whether a string contains a given substring. Unfortunately, JavaScript's built-in tools for doing so leave quite a bit to be desired. First of all, let's take a look at using `String.prototype`'s `indexOf` method.

```
var philosophers = "Aquinas, Maimonedes, and Avicenna";
var me = "Joshua";

function printPhilosopherStatus (person) {
    if (philosophers.indexOf(person) >= 0) {
        console.log(person + " is a philosopher.");
    } else {
        console.log(person + " is NOT a philosopher.");
    }
}

// Outputs: "Joshua is NOT a philosopher."
printPhilosopherStatus(me);
```

While `indexOf` is often recommended as a simple way to test for the presence of a substring, that's not really its purpose. Its job is to return the index at which a given substring is found. In the event that no match is found, it will return `-1`. That means that we can use it, but the clarity of the code suffers. Ideally, what we're looking for is a method with a name that matches our intention (determining if x contains y), and returns a simple `true` or `false`.

Looking through the documentation for `String.prototype`, the `search` method looks promising due to its name. Unfortunately, with the exception of matching on a regular expression rather than a string, the behavior is identical to `indexOf`.

However, that does point us toward something else useful. `RegExp.prototype` has a `test` method which returns a boolean. Let's try it out.

```
var philosophers = "Aquinas, Maimonedes, and Avicenna";
var me = "Joshua";

function printPhilosopherStatus (person) {
    var personRegExp = new RegExp(person);
    if (personRegExp.test(philosophers)) {
        console.log(person + " is a philosopher.");
    } else {
        console.log(person + " is NOT a philosopher.");
    }
}

// Outputs: "Joshua is NOT a philosopher."
printPhilosopherStatus(me);
```

This is a bit better because the method itself returns `true` or `false`. The method name also communicates intent more clearly than `indexOf`.

Unfortunately, if we are trying to match a string which uses characters like `?` or `.`, we have a problem. Because they have special meanings in regular expressions, we have to deal with escaping them. That means this isn't a very good general purpose solution. In addition, the code could still use some improvement in clearly communicating its intent.

Finally we come to `String.prototype`'s `contains` method.

```
var philosophers = "Aquinas, Maimonedes, and Avicenna";
var me = "Joshua";

function printPhilosopherStatus (person) {
    if (philosophers.contains(person)) {
        console.log(person + " is a philosopher.");
    } else {
        console.log(person + " is NOT a philosopher.");
    }
}

// Outputs: "Joshua is NOT a philosopher."
printPhilosopherStatus(me);
```

This has all the features that we've been looking for. It returns a boolean value, and the method name clearly conveys the intent of our code.

Unfortunately, there is a problem. The `contains` method is a proposal for the next version of JavaScript (ECMAScript 6) and has only been implemented in FireFox 19+ so far.

If you'd like to use something similar to `contains`, for now your best bet is to use a third-party library like String.js[42], a "prolly-fill" like ES6 Shim[43], or wrap `indexOf` in your own custom utility function, like so:

```
function aContainsB (a, b) {
    return a.indexOf(b) >= 0;
}

var philosophers = "Aquinas, Maimonedes, and Avicenna";
var me = "Joshua";

function printPhilosopherStatus (person) {
    if (aContainsB(philosophers, person)) {
        console.log(person + " is a philosopher.");
    } else {
        console.log(person + " is NOT a philosopher.");
    }
}

// Outputs: "Joshua is NOT a philosopher."
printPhilosopherStatus(me);
```

And that is an overview of some the ways you can determine if a string contains substrings in JavaScript.

[42] http://stringjs.com/#methods/contains-ss

[43] https://github.com/paulmillr/es6-shim/blob/master/es6-shim.js#L162

Equals Equals Null in JavaScript

One of the strongest injunctions that new JavaScript developers receive is to always use strict equality (===) in comparisons. Douglas Crockford recommends this approach in JavaScript: The Good Parts[44], and it is considered by some parts of the JavaScript community to be a best practice.

But even in code that follows this practice, you're likely to run across one exception: `== null`. Or possibly `!= null`. Here is an example from jQuery's source[45].

```
:::javascript
if ( val == null ) {
    val = "";
}
```

What does this code do? And why is it a common exception to the triple equals rule? Let's take a look at an example where it might be useful.

```
var ethos = {
    achilles: "glory",
    aeneas: "duty",
    hades: null // Beyond human understanding
};

function printEthos (name) {
    console.log(ethos[name]);
}

// Outputs: "glory"
printEthos("achilles");

// Outputs: "null"
printEthos("hades");

// Outputs: "undefined"
printEthos("thor");
```

[44] http://www.amazon.com/gp/product/0596517742/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0596517742&linkCode=as2&tag=joshuacc-20

[45] https://github.com/jquery/jquery/blob/410cf4ee6bd454fa7e2bb5dc37a0b051f15d3e96/src/attributes.js#L203

The `printEthos` function will be called in response to a user inputting a name. And while this works fine for `"achilles"` and `"aeneas"`, it doesn't really work for `"hades"` or `"thor"`. The problem here is that they give us two different types of nothing, and neither of them means anything to the end user. It would be much better to capture any "nothings" and display a user-friendly message. So let's do that.

```
function printEthos (name) {
    if (ethos[name]) {
        console.log(ethos[name]);
    } else {
        console.log(name + " has no recorded ethos.");
    }
}

// Outputs: "hades has no recorded ethos."
printEthos("hades");

// Outputs: "thor has no recorded ethos."
printEthos("thor");
```

We've solved our problem. Or have we? By testing the truthiness of `ethos[name]`, we've opened ourselves up to a different sort of issue. Consider what happens in the case of Pythagoras.

```
ethos.pythagoras = 0; // The sublimity of Number

// Outputs: "pythagoras has no recorded ethos"
printEthos("pythagoras");
```

Because `0` is a falsy value, `"pythagoras"` isn't recognized as having an ethos. So we need something that won't accidentally catch falsy values when we really just want to catch nothingness as opposed to somethingness. And that's what `== null` does.

```
function printEthos (name) {
    if (ethos[name] != null) {
        console.log(ethos[name]);
    } else {
        console.log(name + " has no recorded ethos.");
    }
}

// Outputs: "hades has no recorded ethos."
```

```
10 printEthos("hades");
11
12 // Outputs: "thor has no recorded ethos."
13 printEthos("thor");
14
15 // Outputs: 0
16 printEthos("pythagoras");
```

Despite the fact that `null` is a falsy value (i.e. it evaluates to `false` if coerced to a boolean), it isn't considered loosely equal to any of the other falsy values in JavaScript. In fact, the only values that `null` is loosely equal to are `undefined` and itself.

Because of this interesting property, it has become somewhat conventional to use `== null` as a more concise way of checking whether a given value is "nothing" (`null`/`undefined`) or "something" (anything else). In addition to jQuery[46], you can find examples of this convention in Underscore[47].and Less[48]. Indeed, JSHint, one of the more popular JavaScript linting tools, provides an option[49] to allow the use of loose equality only when comparing to null.

Of course, you can choose to be more explicit about the fact that you are checking for both `null` and `undefined`, but next time you run across `== null`, don't automatically think, "Bad practice!" Instead, take a look at the context and see if it is being used strategically.

[46] https://github.com/jquery/jquery/blob/410cf4ee6bd454fa7e2bb5dc37a0b051f15d3e96/src/attributes.js#L203

[47] https://github.com/jashkenas/underscore/blob/71099af43d735444ff033fefe5264d0b0988eeef/underscore.js#L77

[48] https://github.com/less/less.js/blob/49888fe6063d01e98011802ca926b54cdb7576b9/dist/less-1.4.2.js#L2422

[49] http://www.jshint.com/docs/options/#eqnull

Creating Unwritable Properties with Object.defineProperty

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 explicity 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.

JavaScript's void Operator

If you've been doing web development for any length of time, you've probably seen links that look something like this:

```
<a href="javascript:void 0;" onclick="doSomething()">Do something</a>
```

While this is a rather terrible line of HTML, there is something interesting going on in it: namely, the `void 0` that is executed when a user clicks the link. What exactly does `void 0` do? Lets ask our trusty console.

```
// Outputs: undefined
console.log(void 0);
```

That's interesting. If `void 0` is equivalent to `undefined`, what about using `void` with other values?

```
// Outputs: undefined
console.log(void "test");

// Outputs: undefined
console.log(void {});
```

It turns out that `void`'s one and only purpose is to make an expression evaluate to `undefined`. While this isn't useful in most situations, it can come in handy occasionally.

As we've talked about before, `undefined` isn't a JavaScript keyword, but a property of the global object. That means that `undefined` can be "shadowed" within a function's scope. Here is an example:

```
function shadowLog () {
    var undefined = "shadowed";
    console.log(undefined);
}

// Outputs: "shadowed"
shadowLog();
```

Under ordinary circumstances one can expect the `undefined` variable to refer to the `undefined` primitive. However, if you are dealing with variable shadowing or redefining undefined, then it can be helpful to use `void` as a guarantee that you are working with the `undefined` primitive, like so:

```
(function shadowFunc () {
    var undefined = "shadowed";
    var undefinedVar;

    // Outputs: false
    console.log(undefined === void 0);

    // Outputs: false
    console.log(undefinedVar === undefined);

    // Outputs: true
    console.log(undefinedVar === void 0);
})();
```

When using `void`, it is conventional to use `0` as the operand. Using `void 0` here allows us to easily check whether a variable is `undefined`, even though the name `undefined` was shadowed in the function's scope.

Filtering Arrays with Array#filter

Working with arrays is a daily task for most developers. And one of the most common tasks is taking an array and filtering it down to a subset of the elements it contains.

A straightforward way of doing the filtering manually might look like this:

```
var sidekicks = [
    { name: "Robin",     hero: "Batman"   },
    { name: "Supergirl", hero: "Superman" },
    { name: "Oracle",    hero: "Batman"   },
    { name: "Krypto",    hero: "Superman" }
];

var batKicks = [];

for (var i = 0; i < sidekicks.length ; i++) {
    if (sidekicks[i].hero === "Batman") {
        batKicks.push(sidekicks[i]);
    }
}

// Outputs: [
//    { name: "Robin",  hero: "Batman"   },
//    { name: "Oracle", hero: "Batman"   }
// ]
console.log(batKicks);
```

Fortunately, in JavaScript, arrays have the handy `filter` method which we can use to do the filtering for us instead of manually looping through the array ourselves.

```
var sidekicks = [
    { name: "Robin",     hero: "Batman"   },
    { name: "Supergirl", hero: "Superman" },
    { name: "Oracle",    hero: "Batman"   },
    { name: "Krypto",    hero: "Superman" }
];

var batKicks = sidekicks.filter(function (el) {
    return (el.hero === "Batman");
});

// Outputs: [
//    { name: "Robin",  hero: "Batman"   },
//    { name: "Oracle", hero: "Batman"   }
// ]
console.log(batKicks);

var superKicks = sidekicks.filter(function (el) {
    return (el.hero === "Superman");
});

// Outputs: [
//    { name: "Supergirl", hero: "Superman" },
//    { name: "Krypto",    hero: "Superman" }
// ]
console.log(superKicks);
```

The filter method accepts a callback function. In that callback function we examine each element of the array indidually, and return `true` if we want the element to pass the filter.

In addition to the individual array element, the callback also has access to the index of the current element and the full array, like so:

```
var filtered = sidekicks.filter(function (el, index, arr) {
    // Filtering here
});
```

This allows you to create more complex filters that depend on the element's relationship with other elements, or the array as a whole.

Because the filter method returns an array, it can be chained together with other array methods like `sort` and `map` (see issues #3 and #6.) For example:

```
var sortedBatKickNames = sidekicks.filter(function (el) {
    return (el.hero === "Batman");
}).map(function(el) {
    return el.name;
}).sort();

// Outputs: ["Oracle", "Robin"];
console.log(sortedBatKickNames);
```

Because the `filter` method is part of the ECMAScript 5 specification, it isn't available in older browsers like IE8. If you need to use it in older browsers, then you will need to use a compatibility shim[50] or the equivalent method in a library like Underscore[51] or Lo-Dash[52].

Hopefully this has given you some idea of how you can use `filter` in your everyday work.

[50] https://github.com/kriskowal/es5-shim

[51] http://underscorejs.org/#filter

[52] http://lodash.com/docs#filter

Using JavaScript's Array Methods on Strings

We've talked before about using `apply` and `call` to let methods from one object operate on a different object. But this time we'll take a detailed look at how that works with strings and array methods specifically.

Because in many ways strings behave as if they were arrays of characters, many of JavaScript's standard array methods can operate on strings as well. Many, however, is not all. For example:

```
var ontologist = "Anselm";
[].push.call(ontologist, " of Canterbury");

// Outputs: "Anselm"
console.log(ontologist);
```

As you can see, the `ontologist` string was not modified, even though the whole point of `push` is to modify the array. This is because in JavaScript, strings are immutable.

Because strings are immutable (cannot be modified) any method which attempts to change the string will fail. That immediately rules out methods like `push`, `pop`, `shift`, and `splice`.

However, other methods which treat the string as read-only are fair game. Consider these:

```
var ontologist = "Anselm";

var hasSomeA = [].some.call(ontologist, function(val) {
    return val === "A";
});

// Outputs: true
console.log(hasSomeA);

var everyCharIsE = [].every.call(ontologist, function(val) {
    return val === "E";
});

// Outputs: false
console.log(everyCharIsE);
```

```
16
17 var beforeM = [].filter.call(ontologist, function(val) {
18     return val < "m";
19 });
20
21 // Outputs: ["A", "e", "l"]
22 console.log(beforeM);
```

It's great that all of these methods can operate on strings, but you may have noticed that `filter` returns an array rather than a string. If you think about it, this makes sense. Using `call` or `apply` doesn't change the function's logic, just what value it operates on.

Any array method which returns an array will continue to do so even if called on a string value. If you need your final result to be a string again, then you will need to convert it back to a string with `join`, like so:

```
1 var beforeM = [].filter.call(ontologist, function(val) {
2     return val < "m";
3 }).join("");
4
5 // Outputs: "Ael"
6 console.log(beforeM);
```

Above, I mentioned that many array methods will fail to operate on strings because they are immutable. But there is a fairly simple way around this. We can first convert the string into an array and then convert it back again.

For example:

```
1 var reversed;
2
3 reversed = [].reverse.call(ontologist);
4
5 // Outputs: "Anselm"
6 console.log(reversed);
7
8 reversed = [].slice.call(ontologist).reverse().join("");
9
10 // Outputs: "mlesnA"
11 console.log(reversed);
```

Our first attempt at reversing the string fails because the `reverse` method attempts to mutate the value it is called upon. But in our second attempt we first use the `slice` method to convert the

string into an array, and only then attempt to reverse it, using join to convert it into a new string after the mutatation. This basic approach means that you can use the entire range of array methods to manipulate strings.

As you can see, using array methods on strings can be a little quirky at times, but can also be very powerful. Hopefully this brief introduction has helped you see some places where it can help you in your day to day work.

Numbers and JavaScript's Dot Notation

In JavaScript almost everything behaves like an object. And that is true even when the thing in question isn't an object. For instance:

```
var numOfIs = "Odin".match(/i/).length;

// Outputs: 1
console.log(numOfIs);

var strOfTrue = true.toString();

// Outputs: "true"
console.log(strOfTrue);
```

Here we see two examples. First, the primitive string `"Odin"` lets us treat it as if it were an object by accessing the `match` method. Second, the primitive boolean value `true` lets us treat it as if it were an object by accessing the `toString` method.

However, if we try to do the same thing with a number, we are likely to run into an error.

```
// SyntaxError: Unexpected token ILLEGAL
var meaningOfLife = 42.toString();
```

Because of this error, many beginning JavaScript developers get the impression that you can't access properties on a primitive number with dot notation. You might try bracket notation instead:

```
var meaningOfLife = 42["toString"]();

// Outputs: "42"
console.log(meaningOfLife);
```

That works, but it's not especially nice. And it turns out that dot notation does work with **some** numbers.

```
var platform = 9.75.toString();

// Outputs: "9.75"
console.log(platform);
```

Why does dot notation work with `9.75` but not with `42`? The secret is in the dots. When a JavaScript engine parses your source code, it has to interpret just what a dot means in a given context.

In the case of `42.toString()` the dot is potentially ambiguous. Does it mean the decimal separator? Or does it mean object member access? JavaScript opts to interpret all integers followed by a dot as representing part of a floating point number. But as there is no such number `42.toString()`, it raises a SyntaxError.

In the case of `9.75.toString()` the first dot was unambiguously part of a floating point number, and the second dot was unambiguously object member access.

So the way to avoid problems with integers like `42` is to make sure that the parser can't mistake your dot notation for a decimal point. There are at least three ways to do that.

```
var meaningOfLife = 42..toString();
```

The double-dot approach above is syntactically equivalent to typing `42.0.toString()`. And since JavaScript doesn't have separate types for integers and floating point numbers, the internal value of the number is the same as well.

```
var meaningOfLife = 42 .toString();
```

This one is confusing at first glance. There is only one dot here, so how is it working as member access? The difference is the space between `42` and the the dot. JavaScript numbers cannot have spaces between the integer portion and the decimal point portion. Because of this, the dot is interpreted as member access. However, because of its poor readability and the tendency other developers will have to "fix" it, this is a bad choice for production code.

```
var meaningOfLife = (42).toString();
```

Having looked at the examples above, you'll understand why this one works. The parentheses ensure that the parser doesn't get confused. This is also by far my favorite approach for use in production code. It is highly readable, and the use of parentheses clearly communicates that `toString` is being called on the value of `42`.

Date: November 10, 2013

Finding Array Elements with Array#indexOf

If you are a JavaScript developer you are probably quite familiar with the `indexOf` method available on strings. The basic functionality of `Array`'s `indexOf` is very similar. But we will also explore the complexities introduced due to arrays being composed of disparate types of values.

Suppose that you have an array which represents an ordered rank of elements and you need to find out just where a given element falls in the ranking. Without `indexOf` we would need to do something like this.

```
var power = [
    "Superman",
    "Wonder Woman",
    "Batman"
];

for (var i = 0; i < power.length && power[i] !== "Wonder Woman"; i++) {
    // No internal logic is necessary.
}

var rank = i + 1;

// Outputs: "Wonder Woman's rank is 2"
console.log("Wonder Woman's rank is " + rank);
```

While the `for` loop isn't too complicated, with `indexOf` we no longer have to keep track of the state of our counters and array elements. Instead we can simply do this:

```
var power = [
    "Superman",
    "Wonder Woman",
    "Batman"
];

var index = power.indexOf("Wonder Woman");

var rank = index + 1;

// Outputs: "Wonder Woman's rank is 2"
console.log("Wonder Woman's rank is " + rank);
```

The `indexOf` method will return the index of the first element which matches your search. Or if it does not find a match it will return `-1`.

It also gives you the ability to start your search at an index other than `0`. This can help speed things up if you already know you can rule out some portion at the beginning of the array. You can use it like this:

```
var power = [
    "Superman",
    "Wonder Woman",
    "Batman"
];

var index = power.indexOf("Wonder Woman", 1);

var rank = index + 1;

// Outputs: "Wonder Woman's rank is 2"
console.log("Wonder Woman's rank is " + rank);
```

While merely skipping one element is a trivial gain, if you are dealing with thousands of elements you may be able to obtain a significant boost in performance.

There is one final aspect to consider: equality. The `indexOf` method uses strict equality (===) to determine if an element matches your search. This works great for primitives like strings and numbers. But you may have difficulty using `indexOf` to search for a given object or array. For example:

```
var basilicas = [
    {
        name: "St. Peter's",
        city: "Rome"
    },
    {
        name: "St. John Lateran",
        city: "Rome"
    }
];

var lateranIndex = basilicas.indexOf({
    name: "St. John Lateran",
    city: "Rome"
});

// Outputs: -1
console.log(lateranIndex);
```

What's going on here? Why isn't `indexOf` finding the index of St. John Lateran? That's a consequence of how object equality works in JavaScript. Two objects can have exactly the same properties and values, but unless they are the exact same object instance they are not equal. If you need to match on an object or array with `indexOf`, you'll need to retain a reference to the original object instance, like so:

```
var peters = {
    name: "St. Peter's",
    city: "Rome"
};

var lateran = {
    name: "St. John Lateran",
    city: "Rome"
};

var basilicas = [peters, lateran];

var lateranIndex = basilicas.indexOf(lateran);

// Outputs: 1
console.log(lateranIndex);
```

If you need a more robust solution that allows you to match on complex objects without retaining a reference, look into combining the findIndex[53] and isEqual[54] methods from Lo-Dash.

It is worth pointing out that `Array`'s `indexOf` method is part of the ECMAScript 5 specification. As such it is not available in IE8 and below. If you need this functionality in older browsers, you can use ES5 Shim[55] to backport it, or the equivalent `indexOf` method in the Underscore[56] or Lo-Dash[57] libraries.

[53] http://lodash.com/docs#findIndex
[54] http://lodash.com/docs#isEqual
[55] https://github.com/kriskowal/es5-shim
[56] http://underscorejs.org/#indexOf
[57] http://lodash.com/docs#indexOf

The Problem with Testing for NaN in JavaScript

In JavaScript, the special value `NaN` (meaning "not a number") is used to represent the result of a mathematical calculation that cannot be represented as a meaningful number. For example:

```
var divisionByZod = 42 / "General Zod";

// Outputs: NaN
console.log(divisionByZod);
```

You are also likely run into `NaN` when using `parseInt` or `parseFloat` to extract a number from a string, as in this example:

```
var doomedParse = parseInt("Doomsday", 10);

// Outputs: NaN
console.log(doomedParse);
```

As a result, JavaScript developers often need to test a result variable to see whether it contains `NaN` or whether it is a meaningful value instead. There are several different ways of checking if a value is `NaN`, but unfortunately most of them are unreliable. Let's walk through some of them and see how they fail.

The logical thing to do is check whether the result is equal to `NaN`.

```
var divisionByZod = 42 / "General Zod";

var equalsNaN = (divisionByZod === NaN);

// Outputs: false
console.log(equalsNaN);
```

What's going on here? In JavaScript, `NaN` has the distinction of being the only value that is not equal to itself. That means we can't find out whether a value is `NaN` by checking equality to `NaN` because the answer will always be no.

What else could we try? How about the built in `isNaN` function?

```
var divisionByZod = 42 / "General Zod";

var valueIsNaN = isNaN(divisionByZod);

// Outputs: true
console.log(valueIsNaN);
```

Awesome! This one seems to work. And in fact it does work under some circumstances. However, despite the name, the purpose of the `isNaN` function isn't to check whether a value is `NaN`. Instead, the purpose is to check whether a value cannot be coerced to a number. Because of this, it may return false positives. Consider the following:

```
var isJorElNaN = isNaN("Jor El");

// Outputs: true
console.log(isJorElNaN);
```

Unless you are certain that you will be dealing only with numbers or `NaN`, the `isNaN` function isn't quite fit for this purpose.

However, it turns out that ECMAScript 6 creates a new function for the specific purpose of checking whether a value is `NaN`. Confusingly, this function is also called `isNaN`, though it is attached to the `Number` object rather than directly to the global object. It can be used like this:

```
var divisionByZod = 42 / "General Zod";

var valueIsNaN = Number.isNaN(divisionByZod);

// Outputs: true
console.log(valueIsNaN);
```

Excellent! We've got a method that really works. Unfortunately, `Number.isNaN` is only available in newer versions of FireFox and Chrome, and isn't yet supported in IE or other major browsers. This may be a viable approach in the future, but for right now it can only be used in very narrow circumstances.

Maybe we can check `NaN`'s type instead?

```
// Outputs: "Number"
console.log(typeof NaN);
```

Yes, you're reading that correctly. In JavaScript, the value `NaN` is of type `number`. That's not going to be very helpful.

So what can we do to get something that works correctly, but is also available cross-browser? Remember how I mentioned that `NaN` is the only value in JavaScript that is not equal to itself? We can take advantage of that fact.

```
var divisionByZod = 42 / "General Zod";

// This can only be true if the value is NaN
var valueIsNaN = (divisionByZod !== divisionByZod);

// Outputs: true
console.log(valueIsNaN);
```

Currently the best way to check whether a value is `NaN` is to check whether it is not equal to itself. If so, then you know that it is `NaN`.

Of course, using this technique isn't especially readable, so it is common to wrap it up into a utility method which communicates the intent more clearly. Both Underscore[58] and Lo-Dash[59] provide implementations.

I hope this has helped you get a better grasp on the quirks of `NaN`.

[58] http://underscorejs.org/#isNaN
[59] http://lodash.com/docs#isNaN

Understanding the Module Pattern in JavaScript

Of all the design patterns you are likely to encounter in JavaScript, the module pattern is probably the most pervasive. But it can also look a little strange to developers coming from other languages.

Let's walk through an example to see how it works. Suppose that we have a library of utility functions that looks something like this:

```
var batman = {
    identity: "Bruce Wayne",

    fightCrime: function () {
        console.log("Cleaning up Gotham.");
    },

    goCivilian: function () {
        console.log("Attend social events as " + this.identity);
    }
};
```

This version of `batman` is perfectly serviceable. It can fight crime when you call upon it. However, it has a serious weakness. This `batman`'s `identity` property is publicly accessible.

Any code in your application could potentially overwrite it and cause `batman` to malfunction. For example:

```
// Some joker put this in your codebase
batman.identity = "a raving lunatic";

// Outputs: "Attend social events as a raving lunatic"
batman.goCivilian();
```

To avoid these sorts of situations we need a way to keep certain pieces of data private. Fortunately, JavaScript gives us just such a tool. We've even talked about it before: the immediately invoked function expression (IIFE).

A standard IIFE looks like this:

```
(function () {
    // Code goes here
})();
```

The advantage of the IIFE is that any `var`s declared inside it are inaccessible to the outside world. So how does that help us? The key is that an IIFE can have a return value just like any other function.

```
var batman = (function () {
    var identity = "Bruce Wayne";

    return {
        fightCrime: function () {
            console.log("Cleaning up Gotham.");
        },

        goCivilian: function () {
            console.log("Attend social events as " + identity);
        }
    };
})();

// Outputs: undefined
console.log(batman.identity);

// Outputs: "Attend social events as Bruce Wayne"
batman.goCivilian();
```

As you can see, we were able to use the IFFE's return value to make `batman`'s utility functions publicly accessible. And at the same time we were able to ensure that `batman`'s `identity` remains a secret from any clowns who want to mess with it.

You might be wondering when using the module pattern is a good idea. The answer is that it works well for situations like the one illustrated here. If you need to both enforce privacy for some of your data and provide a public interface, then the module pattern is probably a good fit.

It is worth considering, though whether you really need to enforce data privacy, or whether using a naming convention to indicate private data is a better approach. The answer to that question will vary on a case by case basis. But now you're equipped to enforce data privacy if necessary.

The Virtue of JavaScript Linting

One of the great benefits of being a developer is the extent to which we can automate away many of the necessary but tedious parts of our jobs. And sometimes the value of automating something is actually greater than doing it by hand. This is particularly true of linting.

A linter is a program that can analyze source code in a particular programming language and flag potential problems like syntax errors, deviations from a prescribed coding style, or using constructs known to be unsafe. For example a JavaScript linter would flag the first use of `parseInt` below as unsafe:

```
1  // Unsafe without a radix argument
2  var count1 = parseInt(countString);
3
4  // Safe because a radix is specified
5  var count2 = parseInt(countString, 10);
```

Of course, you could always examine all of your code by hand to determine whether an error was introduced, but that requires a great deal of expertise. And it quickly becomes impractical as the size of your codebase increases, requiring ever greater amounts of caffeine to allow a human being to maintain sufficient attention to read it. (You should probably stop well before reaching a lethal dose[60].) A linter can check the entire codebase in seconds and help you keep the number of bugs to a minimum.

The two best known linters for JavaScript are JSLint[61] and JSHint[62]. JSLint was originally authored by Douglas Crockford, but it was sometimes seen as too inflexible and dogmatic. Because of this concern, JSHint was born as a more flexible linting tool which gives users more control over what sort of issues it should flag as potential problems.

Getting started with linting is easy. Both JSHint and JSLint have web-based tools that will let you paste in code to check. But you can also use their command-line tools to do project-wide linting.

Let's walk through what it takes to set that up with JSHint. To get started, you'll want to install Node.js[63], which will give you the ability to execute JavaScript programs (like JSHint) from the command-line.

After you have Node installed, you can install JSHint by entering `npm install -g jshint` at the command-line. Depending on your permissions and operating system you may need to preface that command with `sudo`.

[60] http://www.energyfiend.com/death-by-caffeine
[61] http://jslint.com/
[62] http://www.jshint.com/
[63] http://nodejs.org/

Once you have JSHint installed, navigate to a directory which contains some JavaScript files and enter `jshint *.js`. Chances are you will soon have a list of warnings and errors printing to your terminal. It should look something like this:

```
example.js: line 1, col 34, Missing radix parameter.
example.js: line 5, col 14, Missing semicolon.
example.js: line 7, col 17, Use '!==' to compare with 'null'.

3 errors
```

Of course, you don't necessarily have to use JSHint's default settings. Suppose that you typically prefer to use === but in a particular project you want to allow the use of `x != null` (as mentioned in a previous drip).

Fortunately, JSHint supports a project settings file to specify how your project should be linted. Simply create a file named `.jshintrc` in the top directory of your project. The options should be specified as properly formatted JSON like the following:

```
{
    "eqnull": true
}
```

From now on, any time JSHint is run anywhere inside your project directory, it will use the options you specified. For a list of the available options, see the JSHint documentation[64].

Using a tool like JSHint or JSLint is a great way to save time and improve the quality of your code all at once.

[64] http://www.jshint.com/docs/options/

Emulating Block Scope in JavaScript

While there are many issues that trip up developers coming from other languages, variable scoping may be number one. The fundamental problem is that many expect variables to be scoped to a particular block (like a `for` loop), but in JavaScript variables declared with `var` are scoped to the nearest parent function.

First let's take a look at how this can go wrong.

```
var avatar = "Ang";
var element = "Air";

var elements = [
    "Air",
    "Earth",
    "Fire",
    "Water"
];

for (var i = 0; i < elements.length; i++) {
    var element = elements[i];
    console.log(avatar + " has mastered " + element);
}

// Outputs: "Ang's primary element is Water"
console.log(avatar + "'s primary element is " + element);
```

A developer used to a language with block scoping might not see anything wrong with the code above, and would expect `"Ang's primary element is Air"` instead of the actual result.

This issue is easily avoided once you become aware of it. Avoiding variable declarations within blocks tends to prevent any confusion.

But suppose that we really wanted to use block scoping in JavaScript. How would we go about it? We could do something like this.

```
var avatar = "Ang";
var element = "Air";

var elements = [
    "Air",
    "Earth",
    "Fire",
    "Water"
];

for (var i = 0; i < elements.length; i++) {
    (function() {
        var element = elements[i];
        console.log(avatar + " has mastered " + element);
    })();
}

// Outputs: "Ang's primary element is Air"
console.log(avatar + "'s primary element is " + element);
```

This solution uses an IIFE to emulate block scoping. Since functions are JavaScript's scoping mechanism, we define and immediately invoke a new function on each pass through the loop, therefore approximating the behavior of a block scope.

While this isn't idiomatic JavaScript, it does help give you an idea of just how flexible JavaScript can be.

It is also worth noting that drafts of the ECMAScript 6 specification (the next version of JavaScript) include a `let` keyword which is used to define block-scoped variables. If you are interested in playing around with this proposed keyword, it is available in FireFox[65].

[65] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

Truthy and Falsy Values in JavaScript

Longtime JavaScript developers often toss around the terms “truthy” and “falsy”, but for those who are newer to JavaScript these terms can be a bit mystifying.

When we say that a value is “truthy” in JavaScript, we don’t just mean that the value is `true`. Rather, what we mean is that the value coerces to `true` when evaluated in a boolean context. Let’s look at what that means.

```
function logTruthiness (val) {
    if (val) {
        console.log("Truthy!");
    } else {
        console.log("Falsy.");
    }
}
```

This function takes the `val` parameter and evaluates it in a boolean context (the condition of the `if` statement.) So let’s try it out on some values.

```
// Outputs: "Truthy!"
logTruthiness(true);

// Outputs: "Truthy!"
logTruthiness({});

// Outputs: "Truthy!"
logTruthiness([]);

// Outputs: "Truthy!"
logTruthiness("some string");

// Outputs: "Truthy!"
logTruthiness(3.14);

// Outputs: "Truthy!"
logTruthiness(new Date());
```

As you can see, there are a lot of truthy values in JavaScript. And there are many more than could be listed here. On the other side, though, there are only six falsy values. In fact, because the list of falsy values is so short, memorizing that list is the easiest way to tell whether a value is truthy or falsy. Here is the list:

```
// Outputs: "Falsy."
logTruthiness(false);

// Outputs: "Falsy."
logTruthiness(null);

// Outputs: "Falsy."
logTruthiness(undefined);

// Outputs: "Falsy."
logTruthiness(NaN);

// Outputs: "Falsy."
logTruthiness(0);

// Outputs: "Falsy."
logTruthiness("");
```

That's a pretty straightforward list. But how can we actually use truthiness? Let's look at an example.

```
function reportAttitude (person) {
    if (person.skepticism) {
        console.log(person.name +
                    " is skeptical about " +
                    person.skepticism);
    } else {
        console.log(person.name + " wants to believe.");
    }
}

var mulder = {
    name: "Fox Mulder"
};

var scully = {
    name: "Dana Scully",
    skepticism: "UFOs & conspiracy theories"
```

```
};

var frohikey = {
    name: "Melvin Frohikey",
    skepticism: ""
};

// Outputs: "Fox Mulder wants to believe."
reportAttitude(mulder);

// Outputs: "Dana Scully is skeptical about UFOs and conspiracy theories."
reportAttitude(scully);

// Outputs: "Melvin Frohikey wants to believe."
reportAttitude(frohikey);
```

Taking advantage of truthiness can make your code a little bit more concise. We don't need to explicitly check for `undefined`, `""`, etc. Instead we can just check whether `person.skepticism` is truthy. However, there some caveats to keep in mind.

Firstly, this approach only works if **all** of the falsy values should be excluded (or included.) For instance, if a value of `0` or `""` was meaningful, then the `if` above would not function correctly.

Secondly, it is important to keep in mind that truthiness is not the same as `== true`. The algorithm for loose equality is much more complicated than for truthiness. We'll go into more detail about that in a future drip.

I hope this introduction to truthiness and falsiness helps you to write clearer, more concise JavaScript.

Object Equality in JavaScript

Equality is one of the most initially confusing aspects of JavaScript. The behavior of == versus ===, the order of type coercions, etc. all serve to complicate the subject. Today we'll be looking at another facet: how object equality works.

You might suppose that if two objects have the same properties and all of their properties have the same value, they would be considered equal. Let's take a look and see what happens.

```
var jangoFett = {
    occupation: "Bounty Hunter",
    genetics: "superb"
};

var bobaFett = {
    occupation: "Bounty Hunter",
    genetics: "superb"
};

// Outputs: false
console.log(bobaFett === jangoFett);
```

The properties of `bobaFett` and `jangoFett` are identical, yet the objects themselves aren't considered equal. Perhaps it's because we're using triple equals? Let's test that theory.

```
// Outputs: false
console.log(bobaFett == jangoFett);
```

The reason for this is that internally JavaScript actually has two different approaches for testing equality. Primitives like strings and numbers are compared by their value, while objects like arrays, dates, and plain objects are compared by their reference. That comparison by reference basically checks to see if the objects given refer to the same location in memory. Here is an example of how that works.

```
var jangoFett = {
    occupation: "Bounty Hunter",
    genetics: "superb"
};

var bobaFett = {
    occupation: "Bounty Hunter",
    genetics: "superb"
};

var callMeJango = jangoFett;

// Outputs: false
console.log(bobaFett === jangoFett);

// Outputs: true
console.log(callMeJango === jangoFett);
```

On the one hand, the variables `jangoFett` and `bobaFett` refer to two objects with identical properties, but they are each distinct instances. On the other hand `jangoFett` and `callMeJango` both refer to the same instance.

Because of this, when you are trying to check for object equality you need to have a clear idea about what sort of equality you are interested in. Do you want to check that these two things are the exact same instance? Then you can use JavaScript's built-in equality operators. Or do you want to check that these two objects are the "same value?" If that's the case, then you'll need to do a bit more work.

Here is a very basic approach to checking an object's "value equality".

```
function isEquivalent(a, b) {
    // Create arrays of property names
    var aProps = Object.getOwnPropertyNames(a);
    var bProps = Object.getOwnPropertyNames(b);

    // If number of properties is different,
    // objects are not equivalent
    if (aProps.length != bProps.length) {
        return false;
    }

    for (var i = 0; i < aProps.length; i++) {
        var propName = aProps[i];

```

```
        // If values of same property are not equal,
        // objects are not equivalent
        if (a[propName] !== b[propName]) {
            return false;
        }
    }

    // If we made it this far, objects
    // are considered equivalent
    return true;
}

// Outputs: true
console.log(isEquivalent(bobaFett, jangoFett));
```

As you can see, to check the objects' "value equality" we have essentially have to iterate over every property in the objects to see whether they are equal. And while this simple implementation works for our example, there are a lot of cases that it doesn't handle. For instance:

- What if one of the property values is itself an object?
- What if one of the property values is `NaN` (the only value in JavaScript that is not equal to itself?)
- What if `a` has a property with value `undefined`, while `b` doesn't have this property (which thus evaluates to `undefined`?)

For a robust method of checking objects' "value equality" it is better to rely on a well-tested library that covers the various edge cases. Both Underscore[66] and Lo-Dash[67] have implementations named `_.isEqual` which handle deep object comparisons well. You can use them like this:

```
// Outputs: true
console.log(_.isEqual(bobaFett, jangoFett));
```

I hope that this drip of JavaScript has helped you get a better grasp of how object equality works.

[66] http://underscorejs.org/#isEqual
[67] http://lodash.com/docs#isEqual

Building Up Arrays with Array#concat

Working with arrays is the bread and butter of being a JavaScript developer. And among the most common tasks is building up a new array out of smaller ones. Let's take a look at one way to do this.

Suppose that you are implementing a registry that holds a list of metahumans. We want to be able to add to the list by giving it a list of new metahumans. A quick and simple way to do it might look like this.

```
1  // Metahuman Registry
2  var mhr = [];
3
4  var heroes = ["Captain Marvel", "Aquaman"];
5  var villains = ["Black Adam", "Ocean Master"];
6
7  mhr = mhr.concat(heroes);
8
9  // Outputs: ["Captain Marvel", "Aquaman"]
10 console.log(mhr);
11
12 mhr = mhr.concat(villains);
13
14 // Outputs: [
15 //     "Captain Marvel",
16 //     "Aquaman",
17 //     "Black Adam",
18 //     "Ocean Master"
19 // ]
20 console.log(mhr);
```

As you can see, when we pass in an array `concat` creates a new array which consists of the elements of both `mhr` and whatever we passed in, maintaining the order of the elements. This is useful on its own, but it turns out that `concat` can actually accept multiple arguments. So we could rewrite the example above as follows:

```
// Metahuman Registry
var mhr = [];

var heroes = ["Captain Marvel", "Aquaman"];
var villains = ["Black Adam", "Ocean Master"];

mhr = mhr.concat(heroes, villains);

// Outputs: [
//     "Captain Marvel",
//     "Aquaman",
//     "Black Adam",
//     "Ocean Master"
// ]
console.log(mhr);
```

But `concat` can handle more than just arrays. If `concat` is given a non-array value, it will just drop that value into the new array at the appropriate location.

```
// Metahuman Registry
var mhr = [];

var heroes = ["Captain Marvel", "Aquaman"];
var villains = ["Black Adam", "Ocean Master"];

mhr = mhr.concat(heroes, villains, "Death");

// Outputs: [
//     "Captain Marvel",
//     "Aquaman",
//     "Black Adam",
//     "Ocean Master",
//     "Death"
// ]
console.log(mhr);
```

So far I've been glossing over an important detail. The `concat` method doesn't modify the array that it is called on. Instead it creates a brand new array. That's why in the examples above I've been assigning the result to `mhr`.

The fact that it is a new array can be quite useful. Suppose that you want to be able to check whether `mhr` has been modified since the last time you checked on it. You could do something like this:

```
// Metahuman Registry
var mhr = [];

var original = mhr;

// Outputs: true
console.log(mhr === original);

var heroes = ["Captain Marvel", "Aquaman"];
mhr = mhr.concat(heroes);

// Outputs: false
console.log(mhr === original);
```

I hope this look at `concat` gave you a deeper understanding of how it works.

Ditching jQuery with `querySelectorAll`

For many developers, jQuery serves as their first introduction to JavaScript. And jQuery does a great job of easing the learning curve by hiding away browser inconsistencies and using an intuitive chaining syntax. But probably the most distinctive feature of jQuery is its use of CSS selectors to choose which elements to manipulate.

For example, consider an HTML page that looks like this:

```
<!doctype html>
<html>
    <head>
        <title>Programming Languages</title>
    </head>
    <body>
        <h1>Programming Languages</h1>

        <ul>
            <li>JavaScript</li>
            <li>CoffeeScript</li>
            <li>Ruby</li>
            <li>Python</li>
        </ul>

        <script src="example.js"></script>
    </body>
</html>
```

If you want to use jQuery to add a class to the first and last `li`, that's as simple as:

```
$("li:first-child, li:last-child").addClass("list-end");
```

What many people don't realize is that modern web browsers actually support a native DOM method that can use CSS selectors in exactly the same way. Let's see how that works.

```
var selector = "li:first-child, li:last-child";
var listEnds = document.querySelectorAll(selector);

var listEndsArr = [].slice.call(listEnds);

listEndsArr.forEach(function (el) {
    el.className += "list-end";
});
```

As you can see, `querySelectorAll` will return a list of matching elements. However, the list that it returns is a `NodeList`, one of those "array-like" objects that isn't really an array. That means that we can't directly use some array methods like `forEach`.

To get around that problem, we use `slice.call` to create a real array of the matching elements. And once we have a real array we use `forEach` to add the class to each matching element.

There are some issues to keep in mind when using `querySelectorAll`. First, the CSS selectors you can use will be limited by the selectors that the browser supports. Second, while it enjoys good support among modern browsers, `querySelectorAll` is not available in IE7 and below.

Of course, `querySelectorAll` isn't a full replacement for jQuery, but it does get you one step closer to jQuery independence.

Storing Metadata on Arrays in JavaScript

We've talked before about the fact that in JavaScript, even arrays are objects. But one thing we haven't really talked about is the sort of flexibility that this implies.

Suppose that we have a system which we can query for records, but we want to be able to capture the time at which those records were returned. We could do something like this:

```
var digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

function filterDigits (filterFn) {
    return {
        result: digits.filter(filterFn),
        timestamp: new Date()
    };
}

var filterObj = filterDigits(function(x) {
    return (x > 8);
});

// Outputs: [9]
console.log(filterObj.result);

// Outputs: Mon Nov 04 2013 13:34:09 GMT-0500 (EST)
console.log(filterObj.timestamp);
```

This works okay. But if you think about it, it is a little odd having to create an entirely new object just to store metadata. Wouldn't it be nice if you could just store that metadata directly on the the resulting array? It turns out that you can.

```
var digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

function filterDigits (filterFn) {
    var result = digits.filter(filterFn);
    result.timestamp = new Date();

    return result;
}

var filtered = filterDigits(function(x) {
    return (x > 8);
});

// Outputs: [9]
// Visual output may vary depending on your console.
console.log(filtered);

// Outputs: Mon Nov 04 2013 13:34:09 GMT-0500 (EST)
console.log(filtered.timestamp);
```

Because an array is an object, you can assign arbitrary properties to it. Using this approach means that our code maintains focus on the central concern (the results array) while still carrying around the metadata for those places which need it.

Looking at the changes in the variable names can help illuminate why it is an improvement. On the one hand, the name `filterObj` essentially means "a collection of disparate values." The name `filtered` on the other hand, conveys a single concept.

It's a little improvement, but little improvements add up over time.

JavaScript's Primitive Wrapper Objects

We've talked before about how in JavaScript most things behave like objects even when they aren't objects. For instance, consider how we can call methods on a string even though it is a primitive:

```
// Outputs: "FRED FLINTSTONE"
console.log("Fred Flintstone".toUpperCase());
```

How does that work, though? Initially you might think that strings are really objects in disguise and try assigning properties to them.

```
var fred = "Fred Flintstone";
fred.favoriteFood = "Brontosaurus Steak";

// Outputs: undefined
console.log(fred.favoriteFood);
```

But that doesn't work. And even more strangely, it doesn't trigger an error. It turns out that in order to allow you to call methods on a primitive, JavaScript does a little bit of trickery which we'll get to shortly.

Apart from `null` and `undefined`, each primitive type (string, number and boolean) has a corresponding object equivalent which you can create by invoking its constructor. For instance:

```
var barney = new String("Barney Rubble");

// Outputs: "Barney Rubble"
console.log(barney);

barney.favoriteFood = "Pterodactyl Eggs";

// Outputs: "Pterodactyl Eggs"
console.log(barney.favoriteFood);

// Outputs: "object"
console.log(barney);
```

As you can see, though, the string object can have properties assigned to it, and it reports itself to be of type "object."

The trickery I mentioned before is that any time you attempt to access a property on a primitive, JavaScript will implicitly create a temporary wrapper object. We can verify this by doing the following:

```
String.prototype.reportType = function () {
    return typeof this;
};

var fred = "Fred Flintstone";

// Outputs: "string"
console.log(typeof fred);

// Outputs: "object"
console.log(fred.reportType());
```

When we directly check the type of a string primitive we get `"string"` as expected, but when we check the type of `this` in a method executed on a string primitive we get `"object"`.

The JavaScript engine doesn't keep this wrapper object around, though. As soon as the work of the method (or other property) is done, it is disposed of.

This explains why trying to assign properties to a primitive doesn't work, but also doesn't throw an error. Assigning the property succeeds, but the property is set on a wrapper object which is immediately destroyed. So when you go to look up the property later, there is nothing there anymore.

I hope this has helped you better learn the quirks of JavaScript primitives.

Basic Inheritance with JavaScript Constructors

We've looked before at using JavaScript's constructors to create our own custom object types. But what we didn't look at was how we can create an inheritance hierarchy.

It's important to note that even though constructors are often referred to as "classes," they really aren't the same thing as classes in other languages. In JavaScript, a constructor is just a function invoked by the `new` operator which builds a new object.

Here's a little refresher:

```
function SuperHuman (name, superPower) {
    this.name = name;
    this.superPower = superPower;
}

SuperHuman.prototype.usePower = function () {
    console.log(this.superPower + "!");
};

var banshee = new SuperHuman("Silver Banshee", "sonic wail");

// Outputs: "sonic wail!"
banshee.usePower();
```

The `SuperHuman` constructor contains our initialization logic, while `SuperHuman.prototype` contains the methods that are shared across all `SuperHuman` instances.

But suppose that we want to create a new type which inherits from `SuperHuman` while adding its own functionality? What would that look like?

```
function SuperHero (name, superPower) {
    this.name = name;
    this.superPower = superPower;
    this.allegiance = "Good";
}

SuperHero.prototype.saveTheDay = function () {
    console.log(this.name + " saved the day!");
};

var marvel = new SuperHero("Captain Marvel", "magic");

// Outputs: "Captain Marvel saved the day!"
marvel.saveTheDay();
```

While this gets us started, there are a couple of problems. First of all, the `SuperHero` constructor is repeating some of the logic of the `SuperHuman` constructor. And more importantly, at this point instances of `SuperHero` don't have access to `SuperHuman` methods. For example:

```
// TypeError: Object <#SuperHero> has no method 'usePower'
marvel.usePower();
```

Let's fix those couple of issues.

```
function SuperHero (name, superPower) {
    // Reuse SuperHuman initialization
    SuperHuman.call(this, name, superPower);

    this.allegiance = "Good";
}

SuperHero.prototype = new SuperHuman();

SuperHero.prototype.saveTheDay = function () {
    console.log(this.name + " saved the day!");
};

var marvel = new SuperHero("Captain Marvel", "magic");

// Outputs: "Captain Marvel saved the day!"
marvel.saveTheDay();

```

```
// Outputs: "magic!"
marvel.usePower();
```

We've managed to eliminate the repeated constructor logic by calling `SuperHuman` with `SuperHero`'s `this` object and passing along the necessary arguments. That ensures that `SuperHuman`'s initialization logic will act on the new `SuperHero` object. And then we tack on the additional logic that is specific to `SuperHero`.

But where the inheritance comes in is on `SuperHero.prototype`. In order to ensure that it inherits the methods from `SuperHuman.prototype`, we actually make it an instance of `SuperHuman` with `new SuperHuman()`.

This basic inheritance pattern won't always work, particularly if the parent constructor is complex, but it will handle simple situations quite well.

In future issues we'll take a look at more sophisticated ways of doing inheritance.

The `delete` Operator in JavaScript

The `delete` operator is one of the less frequently used aspects of the JavaScript language. But there are times when you need `delete` and nothing else will do. In this drip, we'll dive into how to use it and how it works.

The purpose of `delete`, as you might imagine, is to delete things. More specifically, it will delete object properties. For example:

```
var multiverse = {
    earth1: "Silver Age",
    earth2: "Golden Age"
};

delete multiverse.earth2;

// Outputs: { earth1: "Silver Age" }
console.log(multiverse);
```

The `delete` operator will not delete ordinary variables.

```
var alex = "Alexander Luthor";

delete alex;

// Outputs: "Alexander Luthor"
console.log(alex);
```

However, it will delete "global variables," since they are actually properties of the global object (`window` in the browser).

```
// Because var isn't used, this is a property of window
classicFlash = "Jay Garrick";

delete window.classicFlash;

// ReferenceError: classicFlash is not defined
console.log(classicFlash);
```

The delete operator also has a return value. If it succeeds in deleting a property, it will return true. If it fails to delete a property because the property is unwritable it will return false, or if in strict mode it will throw an error.

```
var multiverse = {
    earth1: "Silver Age",
    earth2: "Golden Age"
};

var earth2Deleted = delete multiverse.earth2;

// Outputs: true
console.log(earth2Deleted);
```

You are probably wondering under what circumstance you'd want to use delete. The answer is whenever you actually want to remove a property from an object.

Sometimes rather than delete a property, JavaScript developers will just give it a value of null, like so:

```
var multiverse = {
    earth1: "Silver Age",
    earth2: "Golden Age"
};

multiverse.earth2 = null;
```

While this effectively severs the property from the original value, the property itself still exists on the object, as you can see below:

```
// Outputs: {
//    earth1: "Silver Age",
//    earth2: null
// }
console.log(multiverse.earth2)
```

And some operators like `in` and the `for in` loop will still report the presence of the `null` property. If you are passing around an object that might be inspected using those methods, you probably want to make sure that you really delete any unwanted properties.

Finally, you should keep in mind that `delete` doesn't actually destroy the property's value, just the property itself. For example:

```
var earth3 = "The Crime Syndicate";
multiverse.earth3 = earth3;

delete multiverse.earth3;

// Outputs: "The Crime Syndicate";
console.log(earth3);
```

Here, both `earth3` and `multiverse.earth3` referred to the same value. And as you can see, deleting `multiverse.earth3` doesn't affect `earth3` at all.

That's it for our overview of `delete`.

Retrieving Property Names with `Object.getOwnPropertyNames` and `Object.keys`

In past versions of JavaScript it was fairly painful to figure out what properties an object possessed. Essentially you would need to manually iterate over the object and filter out inherited properties, like so:

```
var charactersBooks = {
    Frodo: "Lord of the Rings",
    Aslan: "Chronicles of Narnia",
};

var characters = [ ];

for (var prop in charactersBooks) {
    if (charactersBooks.hasOwnProperty(prop)) {
        characters.push(prop);
    }
}

// Outputs: ["Frodo", "Aslan"]
console.log(characters);
```

But with ECMAScript 5 we get access to `Object.keys` which eliminates this tedious boilerplate.

```
var charactersBooks = {
    Frodo: "Lord of the Rings",
    Aslan: "Chronicles of Narnia",
};

var characters = Object.keys(charactersBooks);

// Outputs: ["Frodo", "Aslan"]
console.log(characters);
```

As might be expected, `Object.keys` only retrieves the names of properties that are declared directly on the object. It doesn't get the names of inherited properties. In addition, it only retrieves the names of enumerable properties. Non-enumerable property names are omitted.

But ES5 also gave us another method called `Object.getOwnPropertyNames` which is less strict about which property names it will retrieve. Like `keys`, `getOwnPropertyNames` will only retrieve "own" properties. However, it will retrieve the names of non-enumerable properties.

Not sure what enumerable means in this context? Non-enumerable properties are essentially properties that shouldn't be "counted" or iterated over for some reason. (More discussion to come in future drips.) The simplest example is probably an array's `length` property. Let's see how our functions treat it.

```
1  var authors = ["Tolkien", "Lewis"];
2
3  // Outputs: ["0", "1"]
4  console.log(Object.keys(authors));
5
6  // Outputs: ["0", "1", "length"]
7  console.log(Object.getOwnPropertyNames(authors));
```

As you can see, `Object.keys` skipped over the non-enumerable `length` property while `Object.getOwnPropertyNames` did not.

How do you know when to use one or the other? My personal rule is to always default to using `Object.keys` unless I specifically need the name of non-enumerable properties.

Because both of these functions are part of the ES5 spec, they are not available in older browsers like IE8. And unfortunately, it is impossible to backport `Object.getOwnPropertyNames`. However, `Object.keys` can be backported with something like the ES5 Shim library[68].

[68] https://github.com/kriskowal/es5-shim

Immutable Objects with `Object.freeze`

One of the more common techniques in JavaScript is the use of an object to hold configuration values. The object might be accessed as a global or passed around as an argument. For example:

```
var artist = {
    name: "Johnny Cash",
    latestAlbum: "American V"
};

function announce (artist) {
    if (artist.name == "Johnny Cash") {
        console.log("The Man in Black");
    } else {
        console.log(artist.name);
    }
}

// Outputs: "The Man in Black"
announce(artist);

// Outputs: {
//     name: "Johnny Cash",
//     latestAlbum: "American V"
// }
console.log(artist);
```

But in either sort of situation, there is a problem. Functions that have access to the configuration object can modify the object, whether intentionally or accidentally. Suppose that you had a coworker modify the `announce` function above to highlight Elvis rather than Cash, but they mistyped the comparison.

```
var artist = {
    name: "Johnny Cash",
    latestAlbum: "American V"
};

function announce (artist) {
    // Whoops! Assigning the name rather than testing equality!
    if (artist.name = "Elvis Presley") {
        console.log("The King");
    } else {
        console.log(artist.name);
    }
}

// Outputs: "The King"
announce(artist);

// Outputs: {
//     name: "Elvis Presley",
//     latestAlbum: "American V"
// }
console.log(artist);
```

Hmm. I'm pretty sure that Elvis didn't record American V.

Some of you may be thinking that this what something like JSHint is for, and you'd be right. But as part of a defence in depth[69] strategy, it would be awfully nice to make sure that our configuration objects don't get changed around after they're created. Fortunately, JavaScript gives us a way to do exactly that.

```
var artist = {
    name: "Johnny Cash",
    latestAlbum: "American V"
};

Object.freeze(artist);

function announce (artist) {
    // Whoops! Assigning the name rather than testing equality!
    if (artist.name = "Elvis Presley") {
        console.log("The King");
```

[69] http://en.wikipedia.org/wiki/Defence_in_depth#Non-military_examples

```
    } else {
        console.log(artist.name);
    }
}

// Outputs: "The King"
announce(artist);

// Outputs: {
//     name: "Johnny Cash",
//     latestAlbum: "American V"
// }
console.log(artist);
```

The `Object.freeze` method takes an object and renders it effectively immutable. Its existing properties may not be modified and new properties may not be added. In the example above this means that even though the logical error is still there, our `artist` object remains safe from modification and available for later use.

While in normal mode attempts to modify the object will fail silently, if you use strict mode, an error will be thrown.

```
var artist = {
    name: "Johnny Cash",
    latestAlbum: "American V"
};

Object.freeze(artist);

(function() {
    "use strict";

    // TypeError: Can't add property firstAlbum, object is not extensible
    artist.firstAlbum = "A Hard Days Night";
})();
```

Of course, we don't want to throw errors all over the place, so JavaScript also gives us a method to detect whether an object is frozen. Appropriately, it is named `Object.isFrozen`.

```
var artist = {
    name: "Johnny Cash",
    latestAlbum: "American V"
};

Object.freeze(artist);

// Outputs: "Frozen Artist!"
if (Object.isFrozen(artist)) {
    console.log("Frozen artist!");
}
```

I hope this quick introduction to `Object.freeze` helps you write more robust code.

Sealing JavaScript Objects with `Object.seal`

In the last drip we talked about making an object completely immutable. But suppose you need something a little less than full immutability? Then you're in luck. `Object.freeze` has a little brother named `Object.seal`.

Let's walk through how it works.

```
var rectangle = {
    height: 5,
    width: 10
};

Object.seal(rectangle);

rectangle.depth = 15;

rectangle.width = 7;

// Outputs: {
//     height: 5,
//     width: 7
// }
console.log(rectangle);
```

As you can see, once the object is sealed, new properties can't be added, but existing properties can still be modified.

In addition to preventing the addition of new properties, a sealed object can't have properties removed via `delete`. For example:

```
delete rectangle.width;

// Outputs: {
//     height: 5,
//     width: 7
// }
console.log(rectangle);
```

`Object.seal` has one final effect. It makes all object properties non-configurable, preventing you from configuring them into a different state with `Object.defineProperty` and similar methods.

```
Object.defineProperty(rectangle, "height", {
    writable: false
});

rectangle.height = 22;

// Outputs: 22
console.log(rectangle.height);
```

In this example, despite attempting to configure the writability to `false`, the `height` property remains writable. (For a refresher on `Object.defineProperty`, see drip #30.)

Attempting to make any of these forbidden modifications to a sealed object will either fail silently or (in strict mode) throw an error.

Fortunately, we also have a method to detect whether an object is sealed.

```
// Outputs: true
console.log(Object.isSealed(rectangle));
```

Like `Object.freeze`, `Object.seal` is part of the ECMAScript 5 specification, which means it isn't available in older browsers like IE8 and below. If you need to support those browsers, then you'll need to either avoid `Object.seal` or use feature detection to use it only in the browsers that support it.

Preventing Object Extensions in JavaScript

In the last two drips, we've discussed safeguarding your objects with `Object.seal` and `Object.freeze`. In this issue we look at our final object protection mechanism: `Object.preventExtensions`.

Suppose that you have an object recording the populations of the continents. You probably don't want new continents added willy-nilly.

```
var continentPopulations = {
    africa: 1100000000,
    antarctica: 5000,
    asia: 4300000000,
    australia: 36000000,
    europe: 739000000,
    northAmerica: 529000000,
    southAmerica: 387000000
};

Object.preventExtensions(continentPopulations);

continentPopulations.atlantis = 50000;

// Outputs: undefined
console.log(continentPopulations.atlantis);
```

As you can see, the effect of `Object.preventExtensions` is to prevent the adding of new properties to an object. If you are using strict mode it will throw an error, but in normal mode it will fail silently.

It is important to remember that `Object.preventExtensions` does not prevent manipulating the values of existing properties.

```
// The antarctic population drops during winter
continentPopulations.antarctica = 500;

// Outputs: 500
console.log(continentPopulations.antarctica);
```

Nor does it prevent deleting existing properties.

```
// In the year 2248 Antarctica was destroyed
// by spacefaring dinosaurs from another dimension
delete continentPopulations.antarctica;

// Outputs: undefined
console.log(continentPopulations.antarctica);
```

To detect whether an object will accept new properties, use the corresponding `Object.isExtensible` method like so:

```
// Outputs: false
console.log(Object.isExtensible(continentPopulations));
```

`Object.preventExtensions` and `Object.isExtensible` are both part of the ECMAScript 5 specification, so older browsers like IE8 don't support them. And due to the limitations of previous versions of JavaScript, these features cannot be polyfilled.

That's it for our series on JavaScript's built-in object protection methods.

Avoiding Problems with Decimal Math in JavaScript

One of the more unintuitive aspects of JavaScript, particularly for new developers, is the fact that decimal math doesn't always work as you'd expect it to. Suppose that Worf has exactly $600.90 of 21st century American dollars to trade for a wine of excellent vintage.

```
var worfsMoney = 600.90;
var bloodWinePrice = 200.30;
var worfsTotal = bloodWinePrice * 3;

// Outputs: false
console.log(worfsMoney >= worfsTotal);

// Outputs: 600.9000000000001
console.log(worfsTotal);
```

As you can see, simply checking whether Worf has enough money fails because his total doesn't come to exactly $600.90. But why is that?

In JavaScript all numbers are IEEE 754 floating point numbers[70]. Due to the binary nature of their encoding, some decimal numbers cannot be represented with perfect accuracy. This is analagous to how the fraction 1/3 cannot be accurately represented with a decimal number with a finite number of digits. Once you hit the limit of your storage you'll need to round the last digit up or down.

Your first thought might be to try rounding to the second decimal place. Unfortunately, JavaScript's built-in rounding functions only round to the nearest integer.

Your second thought might be to do something like this:

```
// Outputs: true
console.log(worfsMoney.toFixed(2) >= worfsTotal.toFixed(2));
```

But even though we get the correct result, using `toFixed` means that we are just comparing formatted strings rather than actual numbers. And what we really want is a way to get an accurate numeric representation of our numbers.

Fortunately there is an easy way to do that in JavaScript. Instead of representing our money's value as decimal numbers based on dollars, we can just use whole integers of cents.

[70] http://en.wikipedia.org/wiki/IEEE_754

```
var worfsMoney = 60090;
var bloodWinePrice = 20030;
var worfsTotal = bloodWinePrice * 3;

// Outputs: true
console.log(worfsMoney >= worfsTotal);

// Outputs: 60090
console.log(worfsTotal);
```

Because integers can be represented perfectly accurately, just shifting our perspective to treat cents as the basic unit of currency means that we can often avoid the problems of comparisons in floating point arithmetic.

Of course, this doesn't solve all problems. Division of integers and multiplication by decimals may still result in inexact values, but basic integer addition, subtraction, and multiplication will be accurate. (At least until you hit JavaScript's upper limit for numbers.)

Thanks for reading!

Josh Clanton

Basic Inheritance with Object.create

A few issues back we looked at how to implement basic inheritance with constructors. In this issue, we'll look at how to do the same with the newer `Object.create`.

When using constructors for inheritance, we attach properties to the constructor's `prototype` property like so:

Here's a little refresher:

```
function SuperHuman (name, superPower) {
    this.name = name;
    this.superPower = superPower;
}

SuperHuman.prototype.usePower = function () {
    console.log(this.superPower + "!");
};

var banshee = new SuperHuman("Silver Banshee", "sonic wail");

// Outputs: "sonic wail!"
banshee.usePower();
```

The `SuperHuman` constructor contains our initialization logic, while `SuperHuman.prototype` contains the methods that are shared across all `SuperHuman` instances.

If we were to implement the same basic logic using `Object.create`, it would look a bit different:

```
var superHuman = {
    usePower: function () {
        console.log(this.superPower + "!");
    }
};

var banshee = Object.create(superHuman, {
    name: { value: "Silver Banshee" },
    superPower: { value: "sonic wail" }
});

// Outputs: "sonic wail!"
banshee.usePower();
```

In this case we first define the prototype object `superHuman`, and then we use `Object.create` to make a new object which inherits from `superHuman`. That second argument might look a little strange to you, but it's just a simple property descriptor object, like we use with `Object.defineProperty` to fine-tune an object's properties.

Now, what if we want to create a new type which inherits from `superHuman` while adding its own functionality? What would that look like?

```
var superHero = Object.create(superHuman, {
    allegiance: { value: "Good" },
    saveTheDay: {
        value: function () {
            console.log(this.name + " saved the day!");
        }
    }
});

var marvel = Object.create(superHero, {
    name: { value: "Captain Marvel" },
    superPower: { value: "magic" }
});

// Outputs: "Captain Marvel saved the day!"
marvel.saveTheDay();
```

So far so good. But does Captain Marvel have access to the `superHuman` prototype methods?

```
// Outputs: "magic!"
marvel.usePower();
```

Yes, she does!

Using `Object.create` makes setting up inheritance chains simple because any object can be used as a prototype. However, inheritance managed by `Object.create` can't be detected by `instanceof`. Instead you'll need to use the `isPrototypeOf` method, like so:

```
// Outputs: true
console.log(superHero.isPrototypeOf(marvel));

// Outputs: true
console.log(superHuman.isPrototypeOf(marvel));
```

Because both `superHero` and `superHuman` are part of `marvel`'s prototype chain, their `isPrototypeOf` calls each return true.

As with other JavaScript features we've reviewed, `Object.create` is a feature of ECMAScript 5 and is not available in older browsers like IE8.

That's our brief introduction to using `Object.create`. Thanks for reading!

Joshua Clanton

Creating Objects Without Prototypes

Last drip we talked about inheritance using prototypes and Object.create. But one point that sometimes surprises new JavaScript developers is that even ordinary "empty" objects are already part of an inheritance chain. Consider the following:

```
var empty = {};

// Outputs: Function Object()
console.log(empty.constructor);
```

Every time you create a new object via an object literal (the `{}`), behind the scenes JavaScript invokes the `Object` constructor to create the object, just as if you'd used `new Object()`. This is what allows your new object to inherit properties from `Object.prototype`.

But sometimes it would be convenient to create an object that doesn't inherit from a prototype at all. For instance, if you'd like to use an object as a hash/map of arbitrary keys to values.

```
var dictionary = {
    destructor: "A destructive element"
};

function getDefinition(word) {
    return dictionary[word];
}

// Outputs: "A destructive element"
console.log(getDefinition("destructor"));

// Outputs: Function Object()
console.log(getDefinition("constructor"));
```

As you can see, our `getDefinition` function works perfectly for words that we have explicitly defined. However, it also returns inherited properties like `constructor`.

One way of solving this would be to introduce a `hasOwnProperty` check to make sure that the properties aren't inherited. But another way is just to ensure that our `dictionary` never inherits any properties to begin with.

Fortunately, `Object.create` makes that rather easy.

```
var dictionary = Object.create(null, {
    destructor: { value: "A destructive element" }
});

function getDefinition(word) {
    return dictionary[word];
}

// Outputs: "A destructive element"
console.log(getDefinition("destructor"));

// Outputs: undefined
console.log(getDefinition("constructor"));
```

As you can see, in this case `getDefinition` returned our intended result.

The trick here is in our first argument to `Object.create`. Normally this is where we would pass in the object that we want to serve as the prototype. But if we pass in `null` instead, our new object doesn't inherit from a prototype at all.

Of course, `Object.create` isn't available in IE8 and older, so you should only use this technique in modern browsers.

I hope you found this week's drip informative!

Joshua Clanton

Using ECMAScript 6 Maps

In last week's drip I discussed using `Object.create(null)` in order to simplify creating key-value maps in JavaScript. But even with that approach, objects aren't a substitute for a full-fledged map data type. For example:

```
var heroesNemesis = Object.create(null);

var greenLantern = { name: "Green Lantern" };
var sinestro = { name: "Sinestro" };

heroesNemesis[greenLantern] = sinestro;

// Outputs: { name: "Sinestro" }
console.log(heroesNemesis[greenLantern]);

var wonderWoman = { name: "Wonder Woman" };

// Outputs: { name: "Sinestro" }
console.log(heroesNemesis[wonderWoman]);
```

Why is our object reporting Wonder Woman's nemesis as Sinestro? In JavaScript, an object's keys can only be strings. If you try to supply something else, the JavaScript engine will try to coerce it into a string with `toString`.

And the default `toString` implementation for an object returns `"[object Object]"` regardless of what properties the object possesses. So unless we decide to start monkeying around with `Object.prototype` (generally a bad idea) our "map" object can't distinguish between different objects as keys.

Fortunately, ECMAScript 6 (the next version of JavaScript) includes a new `Map` data type which allows us to use any valid JavaScript value as a key.

```
var heroesNemesis = new Map();

var greenLantern = { name: "Green Lantern" };
var sinestro = { name: "Sinestro" };

heroesNemesis.set(greenLantern, sinestro);

// Outputs: { name: "Sinestro" }
console.log(heroesNemesis.get(greenLantern));

var wonderWoman = { name: "Wonder Woman" };

// Outputs: undefined
console.log(heroesNemesis.get(wonderWoman));
```

As you can see, `Map` gives us the result we'd expect. To use `Map` our main interfaces are `set` which accepts a key and associated value to store, and `get` which accepts a key and returns the associated value.

When checking to see whether a given key matches, `Map` uses the "same value" algorithm. So even if two objects look identical, only the original can be used to retrieve it's associated value. For example:

```
var clonedLantern = { name: "Green Lantern" };

// Outputs: undefined
console.log(heroesNemesis.get(clonedLantern));
```

And you can determine how many elements have been stored in a `Map` by checking its `size` property.

```
// Outputs: 1
console.log(heroesNemesis.size);
```

This basic set of functionality for `Map` is supported in the latest versions of Chrome and FireFox, as well as IE11. In future drips we'll look into more advanced features of `Map`.

Thanks for reading!

Joshua Clanton

Negating Predicate Functions in JavaScript

While you may not have heard the term, chances are you've used predicate functions before. A predicate is essentially a function that determines whether something is `true` or `false` based on its arguments. It is common (though not necessary) for predicates to be named "isX", such as `isEven` or `isNumber`.

Suppose that we have a program which deals with cataloging comic book heroes and villains represented as simple objects:

```
var superman = {
    name: "Superman",
    strength: "Super",
    heroism: true
};
```

And as part of that program we have a number of predicates that might look something like this:

```
function isSuperStrong (character) {
    return character.strength === "Super";
}

function isNotSuperStrong (character) {
    return character.strength !== "Super";
}

function isHeroic (character) {
    return character.heroism === true;
}

function isNotHeroic (character) {
    return character.heroism !== true;
}

// Outputs: false
console.log(isNotSuperStrong(superman));

```

```
// Outputs: false
console.log(isNotHeroic(superman));
```

As you can see, this is a bit repetitive. But the problem isn't that the code is longer. Rather, the problem is that for each pair of predicates (the "is" and "isNot") we are defining our core logic twice. Having that logic repeated means we are more likely to make mistakes like updating only one of the predicates when our logic changes.

What can we do to fix that problem? Our first thought might be to do something like this:

```
function isSuperStrong (character) {
    return character.strength === "Super";
}

function isNotSuperStrong (character) {
    return !isSuperStrong(character);
}

function isHeroic (character) {
    return character.heroism === true;
}

function isNotHeroic (character) {
    return !isHeroic(character);
}

// Outputs: false
console.log(isNotSuperStrong(superman));

// Outputs: false
console.log(isNotHeroic(superman));
```

While this is certainly an improvement, we still have repetition of a different kind. Both of our "isNot" predicates share a piece of core logic. They both reverse the sense of the predicates that they are based upon.

Wouldn't it be nice if we could abstract that away into something clearer and more maintainable? Fortunately, we can.

```
function negate (predicateFunc) {
    return function () {
        return !predicateFunc.apply(this, arguments);
    };
}
```

This is another example of treating functions as first-class values in JavaScript. The `negate` function accepts a predicate as an argument, and returns a function whose sense is the opposite of the original predicate.

(If the usage of `apply` is confusing, you might want to read the previous drip on call and apply.)

Let's use `negate` on our original problem.

```
function isSuperStrong (character) {
    return character.strength === "Super";
}

var isNotSuperStrong = negate(isSuperStrong);

function isHeroic (character) {
    return character.heroism === true;
}

var isNotHeroic = negate(isHeroic);

// Outputs: false
console.log(isNotSuperStrong(superman));

// Outputs: false
console.log(isNotHeroic(superman));
```

Everything continues to work as expected. But now we've pulled the shared logic of the "isNot" predicates into one location. And it is much easier to tell at a glance that the definitions of the "isNot" predicates are derived from the "is" predicates.

Applied to only a couple of functions, this refactoring may not look like much. But applied to dozens of functions scattered throughout a complex system, `negate` can help to make things much more maintainable.

Both Underscore[71] and Lo-Dash[72] will include `negate` in future versions. It is also available right now as part of Underscore-contrib[73] under the name `complement`.

[71] https://github.com/jashkenas/underscore/blob/a315e9f4473005a8310540b16d565519a9556106/underscore.js#L785

[72] https://github.com/lodash/lodash/blob/6a839967b4beeaf4be0601d75ff4272a18cb5bec/lodash.js#L5259

[73] https://github.com/documentcloud/underscore-contrib/blob/4ae487956e054f0e0d8de1a85f47e8e8d2e77f06/underscore.function.combinators.js#L107

Thanks for reading!

Josh Clanton

Finding an Object's Size in JavaScript

When working in JavaScript, we inevitably end up using objects as if they were hash/map data structures. While there are new data structures coming, for now we need to just work our way around the hash/map features that JavaScript objects don't have.

One of those little features is a convenient way to tell just how big an object is. Suppose we are doing some analysis of library data and have a set of books and authors that looks like this:

```
var bookAuthors = {
    "Farmer Giles of Ham": "J.R.R. Tolkien",
    "Out of the Silent Planet": "C.S. Lewis",
    "The Place of the Lion": "Charles Williams",
    "Poetic Diction": "Owen Barfield"
};
```

As part of our analysis, we send the book authors data to an API. Unfortunately, the API will only accept an object with up to a hundred books at a time. So we need a good way to tell how many books are in an object before trying to send them.

Our first inclination might be to try something like this:

```
function countProperties (obj) {
    var count = 0;

    for (var property in obj) {
        if (Object.prototype.hasOwnProperty.call(obj, property)) {
            count++;
        }
    }

    return count;
}

var bookCount = countProperties(bookAuthors);

// Outputs: 4
console.log(bookCount);
```

While this certainly works, it would be nice to avoid using a hand-rolled method just to count properties. Surely JavaScript has something better available? Fortunately, it does.

While it's not quite as simple as finding the length of an array, it is awfully close. In modern browsers, we can do this:

```
var bookCount = Object.keys(bookAuthors).length;

// Outputs: 4
console.log(bookCount);
```

The `Object.keys` method returns an array of all the object's own property keys. And since we are interested in knowing just how many properties there are, we can just check the array's length.

It's a little indirect, but now we can go about our business and make sure that we only call the API when our book count is below the acceptable threshold.

As I mentioned above, the `keys` method is only available in modern browsers, and is not available in IE8 and below. If you want to use it in those older browsers you can use es5-shim[74] to make it available.

Thanks for reading!

Josh Clanton

[74] https://github.com/es-shims/es5-shim

Detecting Arrays vs. Objects in JavaScript

When writing JavaScript, it is often necessary to detect whether a certain variable is an array or an ordinary object so that you can perform a different set of actions. For instance, consider a function that can be called with an object representing a marathon, or an array of objects representing multiple marathons:

```
function getListOfMarathonNames (marathons) {
    if (marathons instanceof Object) {
        // Return an array containing the name of the only
        // marathon given.
        return [marathons.name];
    } else if (marathons instanceof Array) {
        // Return an array containing all marathon names.
        return marathons.map(function(race) {
            return race.name;
        });
    }
}
```

At a glance, this seems like perfectly reasonable code. Unfortunately, it is hiding a major bug. Let's test it out to see how it behaves.

```
var londonMarathon = {
    name: "London Marathon",
    date: "April 13, 2014"
};

console.log(getListOfMarathonNames(londonMarathon));
// -> ["London Marathon"]
```

So far so good.

```
var moreMarathons = [
    {
        name: "New York City Marathon",
        date: "November 2, 2014"
    },
    {
        name: "Chicago Marathon",
        date: "October 12, 2014"
    }
];

console.log(getListOfMarathonNames(moreMarathons));
// -> [undefined]
```

What's going on here? The problem is that our function's array detection logic is reversed. Because in JavaScript arrays are just a special type of object, it is impossible for our `else if` to ever be triggered. And since the array we passed in doesn't have a `name` property, we end up returning an array containing only an `undefined` element.

In order to correct this logic we need to work in the other direction, first checking whether the input is of the `Array` subtype before proceeding to check whether it is part of the broader `Object` type.

```
function getListOfMarathonNames (marathons) {
    if (marathons instanceof Array) {
        // Return an array containing all marathon names.
        return marathons.map(function(race) {
            return race.name;
        });
    } else if (marathons instanceof Object) {
        // Return an array containing the name of the only
        // marathon given.
        return [marathons.name];
    }
}

console.log(getListOfMarathonNames(londonMarathon));
// -> ["London Marathon"]

console.log(getListOfMarathonNames(moreMarathons));
// -> ["New York City Marathon", "Chicago Marathon"]
```

And now we have everything working as it should.

Of course, this type of error can also occur when trying to detect other types of objects as well, like `Date`, `RegExp`, etc. The general rule is to check for the subtype first, only handling `Object` afterwards.

Thanks for reading!

Josh Clanton

Checking Date Equality in JavaScript

When working with dates, one of a programmer's most common needs is to check whether one date matches another. And if you're like most programmers, the first way that you thought of doing it is something like this:

```
function isChristmas (dateToTest) {
    var christmas = new Date("12/25/2014");
    return (dateToTest === christmas);
}
```

Unfortunately, this function will never return `true`.

```
console.log(isChristmas(new Date("12/25/2014")));
// => false
```

The reason it will never return `true` is because of the way object equality works in JavaScript. Object equality isn't tested by the internal value of the object, but by identity. In other words, if it isn't the exact same copy of the `Date` object, it isn't considered equal.

To make our `isChristmas` function work, we need to check equality in a different way.

```
function isChristmas (dateToTest) {
    var christmas = new Date("12/25/2014");
    return (dateToTest.getTime() === christmas.getTime());
}
```

Here we compare the return values of `getTime`. The `getTime` method returns an integer representing the number of milliseconds since midnight of January 1, 1970 (the beginning of the Unix epoch).

And we can see that we now get the correct result.

```
console.log(isChristmas(new Date("12/25/2014")));
// => true
```

But if we happen to compare against a `Date` object that is the same day, but a different hour, we'll have trouble again.

```
console.log(isChristmas(new Date("12/25/2014 12:00")));
// => false
```

In order to take different times on the same day into account we might try checking only the year, month, and date of the month.

```
function isChristmas (dateToTest) {
    return (dateToTest.getFullYear() === 2014) &&
           // getMonth is 0-indexed
           (dateToTest.getMonth() === 11) &&
           (dateToTest.getDate() == 25);
}

console.log(isChristmas(new Date("12/25/2014 12:00")));
// => true
```

Despite the "gotcha" of `getMonth` returning a 0-indexed number for months, this function now works. However, it doesn't take into account the complexities of timezones and working with local time versus UTC/GMT time.

Because of these complexities, it is generally better to lean on a robust and well-tested library like Moment.js[75] to do things like date comparisons.

But most importantly, now you know not to count on JavaScript's equality operators when comparing dates.

Thanks for reading!

Josh Clanton

[75] http://momentjs.com/

The Perils of Non-Local Mutation

One of the most important things to understand about JavaScript is how to deal with the mutable nature of objects. Unlike primitives (strings, numbers, booleans, etc.) objects are subject to modification.

Consider this simple example:

```
var solarSystem = {
    planets: 9,
    inhabited: "Earth"
};

var homeSystem = solarSystem;

// Scientists reclassify Pluto
homeSystem.planets = 8;

console.log(solarSystem);
// => { planets: 8, inhabited: "Earth" }
```

In this example, both `solarSystem` and `homeSystem` point to the same underlying object. And when the number of planets is modified, that is reflected in both variables.

Ordinarily there isn't much point to explicitly declaring two variables for the same object. However, it is incredibly common to do so **implicitly** when defining a function.

```
var solarSystem = {
    planets: 9,
    inhabited: "Earth"
};

function reclassifyPluto (system) {
    system.planets = 8;
}

reclassifyPluto(solarSystem);

console.log(solarSystem);
// => { planets: 8, inhabited: "Earth" }
```

Here `system` becomes a reference to `solarSystem`, which the `reclassifyPluto` function modifies directly. That may not seem like such a big deal. But it can have far-reaching consequences.

```
var solarSystem = {
    planets: 9,
    inhabited: "Earth"
};

function simulateMarsSettlement (system) {
    system.inhabited = "Earth & Mars";

    return system;
}

function reclassifyPluto (system) {
    system.planets = 8;
}

var simulatedSystem = simulateMarsSettlement(solarSystem);

console.log(simulatedSystem);
// => { planets: 9, inhabited: "Earth & Mars" }

reclassifyPluto(solarSystem);

console.log(solarSystem);
// => { planets: 8, inhabited: "Earth & Mars" }
```

Why does `solarSystem` think that both Earth and Mars are inhabited? Because `simulateMarsSettlement` directly modified it. Imagine how difficult it would be to debug if there were dozens of functions depending on the `solarSystem` object which suddenly changed.

We could code defensively by freezing[76] `solarSystem` so that it can't be modified. But this would only solve part of our problem. The underlying issue is that the functions are modifying an object which doesn't "belong" to them.

In general it is better to avoid modifying objects which are passed in as function arguments, because you may not know what else is depending on them. If you need a modified version of argument, then creating a copy is probably your best solution.

[76] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze

```
var solarSystem = {
    planets: 9,
    inhabited: "Earth"
};

function simulateMarsSettlement (system) {
    var simulation = {
        planets: system.planets,
        inhabited: system.inhabited
    };

    simulation.inhabited = "Earth & Mars";

    return simulation;
}

function reclassifyPluto (system) {
    var reclassified = {
        planets: system.planets,
        inhabited: system.inhabited
    };

    reclassified.planets = 8;

    return reclassified;
}

var simulatedSystem = simulateMarsSettlement(solarSystem);

console.log(simulatedSystem);
// => { planets: 9, inhabited: "Earth & Mars" }

// Replace solarSystem with updated version
var solarSystem = reclassifyPluto(solarSystem);

console.log(solarSystem);
// => { planets: 8, inhabited: "Earth" }
```

Note that in this solution we do end up modifying the `solarSystem` variable. But we do it "locally," not hidden away inside a function. This keeps our code simple and easy to reason about.

Thanks for reading!

Josh Clanton

Measuring JavaScript Performance with console.time

At one point or another almost every JavaScript developer has to deal with optimizing performance-critical code. But how exactly do you go about it? This drip will show you how to use `console.time` as a low-cost performance testing tool.

First, let's suppose that we have a scientific application that makes frequent use of the factorial function.

If you're not familiar with the factorial in mathematics, it basically works like this: Any input to the factorial function must be a non-negative integer. The factorial of X is the product of every positive integer less than or equal to X. In other words, the factorial of 3 is 1 * 2 * 3, the factorial of 4 is 1 * 2 * 3 * 4, and so on.

Here we have a simple implementation of `factorial`.

```
function factorial (num) {
    if (num < 0) {
        throw new Error("Number cannot be negative.");
    }

    if (num % 1 !== 0) {
        throw new Error("Number must be an integer.");
    }

    // The base case
    if (num === 0 || num === 1) {
        return 1;
    }

    // The general case
    return num * factorial(num - 1);
}

console.log(factorial(4));
// => 24
```

In order to optimize `factorial`, we need to measure its current performance so that we have something to compare against.

```
console.time("factorial test");

for (var i = 1; i < 100000; i++) {
    factorial(20);
}

console.timeEnd("factorial test");
// => 512.46ms
```

The `console.time` method starts a timer with the name that you give it. In this case we called our timer `"factorial test"`. Correspondingly, when the `console.timeEnd` method is called with the same name, it will halt the timer and log how many milliseconds have elapsed.

You'll note that in order to get roughly accurate performance measurements, I reran the factorial calculation 100,000 times and it still only took about half a second. Modern JavaScript engines running on modern hardware are **fast**.

Now let's try optimizing `factorial`.

```
function factorial (num) {
    if (num < 0) {
        throw new Error("Number cannot be negative.");
    }

    if (num % 1 !== 0) {
        throw new Error("Number must be an integer.");
    }

    function factorialEquation (num) {
        // The base case
        if (num === 0 || num === 1) {
            return 1;
        }

        // The general case
        return num * factorialEquation(num - 1);
    }

    return factorialEquation(num);
}

console.time("factorial test");

```

```
25 for (var i = 1; i < 100000; i++) {
26     factorial(20);
27 }
28
29 console.timeEnd("factorial test");
30 // => 81.39ms
```

The optimization in this case is quite simple. Rather than checking `factorial`'s inputs every time the equation is recursively called, we have separated out the equation logic from the input checking logic and ensured that the input checks are only run once. The performance improvements are pretty substantial. Our test case goes from half a second to less than one tenth of a second.

Keep in mind that the performance characteristics of a given piece of code vary by JavaScript engine, so for real performance tests you should check in each browser you support.

It is also important to note that the vast majority of JavaScript code doesn't need to be heavily optimized. In the general case, readability and maintainability should trump optimization. Generally you will only need to optimize JavaScript that is part of your program's critical path, and then only if it has been found to be too slow for your use case.

The `console.time` and `console.timeEnd` methods are available in all modern browsers, including IE 11.

Now you have one more tool in your belt to help you put together excellent JavaScript applications.

Thanks for reading!

Josh Clanton

The Problems with for...in and JavaScript Arrays

We've talked in the past about different ways of iterating over arrays. But in this drip we'll take a look at one way **not** to do it.

JavaScript's `for...in` loop iterates over the enumerable properties of an object like so:

```
var tMinus = {
    two: "Two",
    one: "One",
    zero: "Blast off!"
};

var countdown = "";

for (var step in tMinus) {
    countdown += tMinus[step] + "\n";
}

console.log(countdown);
// => "Two
//    One
//    Blast Off!
//    "
```

Because `for...in` operates on any JavaScript object, it is also possible to use it with arrays. For example:

```
var tMinus = [
    "Two",
    "One",
    "Blast off!"
];

var countdown = "";

for (var step in tMinus) {
    countdown += tMinus[step] + "\n";
}

console.log(countdown);
// => "Two
//    One
//    Blast Off!
//    "
```

However, there are three problems with using this approach on arrays. First, the `for...in` also iterates over an object's prototype properties if those properties are enumerable. For example:

```
Array.prototype.voice = "James Earl Jones";

var tMinus = [
    "Two",
    "One",
    "Blast off!"
];

var countdown = "";

for (var step in tMinus) {
    countdown += tMinus[step] + "\n";
}

console.log(countdown);
// => "Two
//    One
//    Blast Off!
//    James Earl Jones
//    "
```

That can be solved by using `hasOwnProperty` to exclude prototype properties.

Array.prototype.voice = “James Earl Jones”;

```
Array.prototype.voice = "James Earl Jones";

var tMinus = [
    "Two",
    "One",
    "Blast off!"
];

var countdown = "";

for (var step in tMinus) {
    if (tMinus.hasOwnProperty(step)) {
        countdown += tMinus[step] + "\n";
    }
}

console.log(countdown);
// => "Two
//    One
//    Blast Off!
//    "
```

Second, according to the specification[77] `for...in` loops may iterate over an object’s values in an **arbitrary order**.

That’s not really a problem for an ordinary object whose values are inherently unordered anyway. But you probably don’t want your JavaScript engine handing back array values in a random order, because you could get unexpected results like this:

```
console.log(countdown);
// => "Blast Off!
//    One
//    Two
//    "
```

The third problem is that `for...in` iterates over **all enumerable properties**, not just the array’s elements. As we’ve discussed before, it is possible to store additional properties on an array. This can also lead to unexpected results.

[77] http://www.ecma-international.org/ecma-262/5.1/#sec-12.6.4

```
var tMinus = [
    "Two",
    "One",
    "Blast off!"
];

tMinus.announcer = "Morgan Freeman";

var countdown = "";

for (var step in tMinus) {
    if (tMinus.hasOwnProperty(step)) {
        countdown += tMinus[step] + "\n";
    }
}

console.log(countdown);
// => "Two
//    One
//    Blast Off!
//    Morgan Freeman
//    "
```

Because of these problems you will almost never want iterate over arrays with `for...in` loops. Instead, use an ordinary `for` loop, or one of the built-in array iteration methods like `forEach` or `map`.

Now you know one more quirk to avoid as you write safe and clean JavaScript.

Thanks for reading!

Josh Clanton

Safely Referencing Undeclared Global Variables

JavaScript has its fair share of quirks, but one of the more subtle ones is the fact that there are both safe and unsafe ways to reference undeclared global variables.

Suppose that we are expecting a global array called `wishList`, but it isn't declared. When we attempt to reference it, we'll get a `ReferenceError`, like so:

```
function addToWishList (item) {
    wishList.push(item);
}

addToWishList("The Complete Calvin & Hobbes");
// => ReferenceError: wishList is not defined
```

That's as expected. But suppose that we want to fail silently rather than throwing an error. How would we do that?

We might try checking to see whether `wishList` exists before using it.

```
function addToWishList (item) {
    if (wishList) {
       wishList.push(item);
    }
}

addToWishList("The Complete Calvin & Hobbes");
// => ReferenceError: wishList is not defined
```

But as you can see, we still get a `ReferenceError`. The problem here is that the `if` itself attempts to reference `wishList`. So how can we make this work?

The trick is to remember that in JavaScript global variables are also properties of the global object (`window` in the browser, or `global` in Node.js.)

```
function addToWishList (item) {
    if (window.wishList) {
       wishList.push(item);
    }
}

addToWishList("The Complete Calvin & Hobbes");
```

Now our function won't throw errors when `wishList` hasn't yet been declared.

The reason this works is that the semantics of referencing a variable and attempting to access an object are different. When we attempt to access `window.wishList`, the JavaScript engine sees that there is no such property and hands us back the value `undefined`. But when attempting to directly reference a variable that hasn't been declared, the JavaScript engine will throw a `ReferenceError`.

Of course, most of the time you won't want your code to just fail silently, but when you are depending on a global variable that may or may not have been declared yet, this little trick can make your life a lot easier.

Thanks for reading!

Josh Clanton

Faster Websites - Minifying Your JavaScript with Uglify

Programmers always have to be mindful of their code's performance. (Should we really be doing this expensive function call in a `for` loop? Probably not.) But writing client-side JavaScript brings its own special set of performance considerations.

The browser can't display your handy-dandy JavaScript widget until it downloads your JavaScript. That makes it critical to get the code to the browser as quickly as possible so your site doesn't feel laggy.

The most important thing you can do to get your JavaScript downloading faster is to minify it. Minification is a form of file compression that takes ordinary JavaScript:

```
(function() {
    var process = "minification";
    var tool = "Uglify";

    function logProcess () {
        console.log(process);
    }

    logProcess();
    // => "minification"
})();
```

It then transforms it into something nearly unreadable like this:

```
!function(){function n(){console.log(i)}var i="minification";n()}();
```

While this may look very different to the human eye, it tells the JavaScript engine the exact same thing. And it manages to do so in one-third the space of the original by employing techniques like:

- Eliminating whitespace.
- Eliminating comments.
- Changing inner variable and function names to something really short.
- Using a more concise IIFE syntax.

- Dropping unused variables. (See `tool` above.)

There are several good minification tools available, but one of the easiest to use is UglifyJS[78]. To get started:

1. Make sure you have Node.js[79] installed.
2. Run `npm install -g uglify-js` on the command line.

This will give you access to the `uglifyjs` command, which you can use as follows:

```
uglifyjs myfile.js --output myfile.min.js
```

By default, UglifyJS will only perform basic minification techniques like eliminating whitespace and comments. I recommend enabling two additional flags for better results:

- `--mangle` will enable variable and function renaming.
- `--compress` will enable optimizations like dropping unused variables.

Using those flags the command would look like this:

```
uglifyjs myfile.js --output myfile.min.js --mangle --compress
```

In addition to compressing individual files, UglifyJS also supports minifying directories of files and joining them together in a single minified output file:

```
uglifyjs myproject/*.js --output myproject.min.js --mangle --compress
```

While these comprise the main use cases, UglifyJS has a whole host of options[80] available which may help you achieve even higher rates of compression.

Now that you're armed with a solid minification tool, it's time to go build some well-optimized web applications.

Thanks for reading!

Josh Clanton

[78] https://github.com/mishoo/UglifyJS2
[79] http://nodejs.org/
[80] https://github.com/mishoo/UglifyJS2#usage

Don’t Blow Your Stack - Recursion and Trampolines in JavaScript

For some types of programming problems (particularly mathematical problems) the most elegant solution is to use a recursive function, because it directly translates a mathematical definition. Unfortunately, recursive functions in JavaScript can run into trouble quickly.

Suppose that we want to define a recursive function that tells us whether a number is even. (Yes, there are easier ways to determine evenness, we’re just exploring recursion.) We might write something like this:

```
function isEvenNaive (num) {
    if (num === 0) {
        return true;
    }

    if (num === 1) {
        return false;
    }

    return isEvenNaive(Math.abs(num) - 2);
}

isEvenNaive(10);
// => true

isEvenNaive(9);
// => false
```

The code looks pretty reasonable. But what happens if we use it on a large number, say `99999`?

```
isEvenNaive(99999);
// => InternalError: too much recursion
```

The exact error varies depending on the JavaScript engine, but at large numbers like these you will eventually hit the call stack depth limit[81] of your engine and have a stack overflow[82].

[81] http://stackoverflow.com/questions/7826992/browser-javascript-stack-size-limit

[82] http://en.wikipedia.org/wiki/Stack_overflow

The call stack[83] is essentially the in-memory list of all the unresolved function calls, their associated variables, and related information. Each time `isEvenNaive` is invoked with a value greater than `1`, it calls itself, adding to the stack and consuming more resources.

Only when we eventually reach the base cases `0` or `1` do the recursive function calls resolve, emptying the stack and freeing up resources. Without a limit on recursion depth (or tail call optimization[84]), a deeply recursive function could consume all of your computer's resources.

But recursive functions are really useful. Is there a way that we can write recursive functions that don't overflow the stack? It turns out that there is.

```
function isEvenInner (num) {
    if (num === 0) {
        return true;
    }

    if (num === 1) {
        return false;
    }

    return function() {
        return isEvenInner(Math.abs(num) - 2);
    };
}

isEvenInner(8);
// => function() {
//        return isEvenInner(Math.abs(num) - 2);
//    };

isEvenInner(8)()()()();
// => true
```

The first thing to notice about our `isEvenInner` function is that instead of directly calling itself again, it returns an anonymous function. That means each call to `isEvenInner` gets resolved immediately, **and doesn't increase the size of the stack**. It also means that we need a way to automatically invoke all of those anonymous functions that will get returned along the way. That's where `trampoline` comes in.

[83] http://en.wikipedia.org/wiki/Call_stack

[84] http://stackoverflow.com/questions/310974/what-is-tail-call-optimization

```
function trampoline (func, arg) {
    var value = func(arg);

    while(typeof value === "function") {
        value = value();
    }

    return value;
}

trampoline(isEvenInner, 99999);
// => false

trampoline(isEvenInner, 99998);
// => true
```

The `trampoline` function effectively turns this recursive algorithm into something that is executed by a `while` loop. As long as `isEvenInner` keeps returning functions, `trampoline` will keep executing them. When we finally reach a non-function value, `trampoline` will return the result.

Now we can avoid blowing the stack, but calling `trampoline(isEvenInner, 3)` isn't that nice. Let's add a little bit of partial application with bind.

```
var isEven = trampoline.bind(null, isEvenInner);

isEven(99999);
// => false
```

It's important to note that while the principles illustrated are widely applicable, this particular implementation of `trampoline` has some limitations.

- It assumes that you are only passing one argument to the recursive function.
- It assumes that the final returned value will not be a function.
- In some older JavaScript engines `typeof function` is inaccurate.

For a more robust implementation, see underscore-contrib[85].

I hope that this has helped give you a stronger understanding of how recursion works in JavaScript.

Several people have mentioned that they are interested in more intermediate/advanced topics, so I'm working to add more of those into the mix. If you have a topic to suggest, just reply to this email. I love getting feedback.

Thanks for reading!

Josh Clanton

[85] http://documentcloud.github.io/underscore-contrib/#trampoline

Lists of Unique Values - Using ES6 Sets in JavaScript

Up until recently, one of the major weaknesses of JavaScript as a language was that it had a poor set of basic collection types. We were essentially limited to arrays and objects. With the official standardization of ECMAScript 6 (the next version of JavaScript) on the horizon, that's changing. We've already looked at ES6 Maps. In this drip we'll take a look at ES6 Sets.

Suppose that we are writing a program for the bat-computer to keep track of which superheroes to call in case of a city-threatening emergency. Your first instinct might be to use an array:

```
var heroCallList = [];

heroCallList.push("Robin");
heroCallList.push("Batgirl");
heroCallList.push("Nightwing");

console.log(heroCallList);
// => ["Robin", "Batgirl", "Nightwing"]
```

That looks reasonable enough. But what if Alfred doesn't notice that Batgirl is already on the list and adds her again?

```
heroCallList.push("Batgirl");

console.log(heroCallList);
// => ["Robin", "Batgirl", "Nightwing", "Batgirl"]
```

That's not good. The last thing we want to do in an emergency is distract our heroes with extra calls.

We could just write a function which checks whether a value is already in the array before adding it:

```
var heroCallList = [];

function addToCallList (hero) {
    var exists = false;

    for (var i = 0; i < heroCallList.length && !exists; i++) {
        exists = heroCallList[i] === hero;
    }

    if (!exists) {
        heroCallList.push(hero);
    }
}

addToCallList("Batgirl");
addToCallList("Batgirl");

console.log(heroCallList);
// => ["Batgirl"]
```

This will certainly work. However, each time we try to add a hero, it potentially has to search the entire array. It is probably sufficient for Gotham-scale. But what if Batman wants to reuse the same program on the Justice League computer to keep a call list of all the superheroes in the multiverse? It would become painfully slow.

Now, we could solve the problem by using an object, like so:

```
var heroCallList = Object.create(null);

function addToCallList (hero) {
    heroCallList[hero] = true;
}

addToCallList("Batgirl");
addToCallList("Batgirl");

console.log(heroCallList);
// => { Batgirl: true }
```

However, this is a bit unintuitive, and also means that you can't reliably keep a list of anything but strings.

Fortunately, ES6 has us covered with the new `Set` collection type. Using it to solve our problem looks like this:

```
var heroCallList = new Set();

heroCallList.add("Batgirl");
heroCallList.add("Batgirl");

console.log(heroCallList);
// => Set ["Batgirl"]
```

The fantastic thing is that instances of `Set` will only store a single instance of a value, no matter how many times you try to add it. You can even initialize the `Set` with the contents of an array and it will automatically remove the duplicate values.

```
var heroCallList = new Set(["Batgirl", "Batgirl"]);

console.log(heroCallList);
// => Set ["Batgirl"]
```

Sets can store any sort of value, including objects, arrays, etc. But you need to be careful when storing non-primitives, because `Set` checks for identity (aka ===) on complex types like objects. For instance:

```
var batgirl1 = { name: "Batgirl" };
var batgirl2 = { name: "Batgirl" };

var heroCallList = new Set([batgirl1, batgirl2]);

console.log(heroCallList);
// => Set [{ name: "Batgirl" }, { name: "Batgirl" }]
```

Even though these two objects look the same, because they are different instances, `Set` considers them two different values.

So far we've only talked about adding values, but what about deleting them? Fortunately, that's just as easy with the `delete` and `clear` methods.

```
{: lang="javascript" }
var heroCallList = new Set(["Batgirl", "Robin", "Nightwing"]);

// Poor Damian. :-(
heroCallList.delete("Robin");

console.log(heroCallList);
// => Set ["Batgirl", "Nightwing"]

heroCallList.clear();

console.log(heroCallList);
// => Set []
```

The `delete` method allows us to remove an individual value from the set, while `clear` will empty the set of all values.

We can use the `has` method to determine if a particular value is in the set.

```
var heroCallList = new Set(["Batgirl"]);

console.log(heroCallList.has("Batgirl"));
// => true

console.log(heroCallList.has("Robin"));
// => false
```

And in order to iterate over the list we can use the `forEach` method.

```
var heroCallList = new Set(["Batgirl"]);

heroCallList.forEach(function(val, val2, fullSet) {
    console.log(val);
    // => "Batgirl"

    console.log(val2);
    // => "Batgirl"

    console.log(fullSet);
    // => Set ["Batgirl"]
});
```

One tricky thing to note is that the value appears twice in the `forEach` callback. It is defined this way in order to maintain the same function signature as `Array.prototype.forEach`.

The last tricky thing to keep in mind is that if you need to check how many values are in a set, you need to check the `size` property rather than the `length` property.

```
1 var heroCallList = new Set(["Batgirl"]);
2
3 console.log(heroCallList.size);
4 // => 1
5
6 console.log(heroCallList.length);
7 // => undefined
```

Browser support for `Set` is a little spotty at this point, but all the features mentioned above work in IE 11 and the latest versions of FireFox. Unfortunately, it is not yet available in Chrome unless you "Enable Experimental JavaScript" via `chrome://flags/#enable-javascript-harmony`.

But don't let that stop you from using `Set`. You can start using it today by relying on a decent polyfill[86].

Thanks for reading. Now go have fun with your fancy new collection type!

Josh Clanton

[86] https://github.com/paulmillr/es6-shim/

Automating Your Way to Better JavaScript with Grunt

Writing high-quality JavaScript can be a difficult process. Especially as applications get larger, the number of things to keep track of and verify can grow and grow.

In a previous Drip, we discussed how to use JSHint to flag possible errors in your code. But manually running JSHint is prone to error. If you forget to run it before committing, you might let a problem slip into the codebase. In addition, running it manually means that the feedback you get is dependent on how often you run it. What if you could get feedback every time you saved instead?

> Civilization advances by extending the number of important operations which we can perform without thinking about them. - *Alfred North Whitehead*

It turns out that there is a nice easy way to get faster feedback by using Grunt[87]. Grunt is a JavaScript task runner which gives you the ability to perform tasks (like running JSHint) on demand, or even on every save, depending on your configuration.

There are other JavaScript build systems and task runners, but Grunt is especially easy to get started with.

Before you can use Grunt, you'll need to make sure that you have a recent version of Node.js[88] installed. Once that's done, open up a command line/terminal and run `npm install -g grunt-cli`. You'll see your computer download and install the Grunt command-line interface. Once it is installed, type `grunt --version`, and you should see something like `grunt-cli v0.1.13`.

Next, let's create a directory for our project, calling it `test-project`. Inside that directory, we'll create another directory to hold our project's source files (the JavaScript we want JSHint to examine). Let's call it `src`. And inside `src`, let's create a JavaScript file called `project.js`.

At this point you should be looking at a directory structure like this:

```
+ test-project
  + src
    - project.js
```

Now, back in our `test-project` directory, we'll create a `package.json` file which will keep track of the Node.js modules we are using. Initially our file will look like this:

[87] http://gruntjs.com/
[88] http://nodejs.org/

```
{
  "name": "test-project",
  "version": "0.0.0"
}
```

Now let's install a couple of things. First make sure you're in the `test-project` directory and install Grunt itself by typing `npm install grunt --save-dev` at the command line. Then install the jshint plugin[89] for Grunt by typing `npm install grunt-contrib-jshint --save-dev`.

You may be wondering why we are installing Grunt again. It turns out that Grunt has two components, the `grunt` command that you type, as well as the Grunt task runner itself. The `grunt` command has to be installed on your system only once. But each project has to have its own instance of the Grunt task runner.

If we look back at our `package.json` file, you'll see that it now looks something like this:

```
{
  "name": "test-project",
  "version": "0.0.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0"
  }
}
```

Using `npm`'s `--save-dev` flag automatically updated `package.json` with the information that we have these two development dependencies. In addition, you'll see a folder called `node_modules` which actually contains the dependencies.

Now let's create a new file in `test-project` called `Gruntfile.js`. It should look like this.

```
module.exports = function(grunt) {
    // Load the jshint plugin
    grunt.loadNpmTasks('grunt-contrib-jshint');

    // Project configuration.
    grunt.initConfig({
        jshint: {
            options: {
                eqeqeq: true
            },
            all: ["src/**/*.js"]
```

[89] https://github.com/gruntjs/grunt-contrib-jshint

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

The `jshint` object contains our configuration for the `jshint` task. The `options` object specifies how we would like JSHint to lint our code. And the `all` property creates a subtask that will lint all the JavaScript files in the `src` directory.

To run our task we can type `grunt jshint:all` or `grunt jshint` within our project folder. (Using `grunt jshint` will run every subtask of `jshint`, but in this case we have only one.) You should see something like this:

```
Running "jshint:all" (jshint) task
>> 1 file lint free
```

That's all well and good, but at this point it's not really any better than just using JSHint without Grunt. Let's set up the watch plugin[90].

First, run `npm install grunt-contrib-watch --save-dev`. Then modify `Gruntfile.js` to look like this:

```
module.exports = function(grunt) {
    // Load the jshint plugin
    grunt.loadNpmTasks('grunt-contrib-jshint');

    // Load the watch plugin
    grunt.loadNpmTasks('grunt-contrib-watch');

    // Project configuration.
    grunt.initConfig({
        jshint: {
            options: {
                eqeqeq: true
            },
            all: ["src/**/*.js"]
        },

        watch: {
            jshint: {
                tasks: ["jshint:all"],
                files: ["src/**/*.js"]
```

[90] https://github.com/gruntjs/grunt-contrib-watch

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

The `watch` task will "watch" the files we specify and automatically run the tasks we specify when the files change. Let's see it in action. On the command line run `grunt watch`. You should see something like this:

```
Running "watch" task
Waiting...
```

Now let's open up `project.js` and enter in some JavaScript that will fail JSHint.

```
var onesEqual = (1 == 1);
```

As soon as we save, we'll see the following:

```
>> File "src/project.js" changed.
Running "jshint:all" (jshint) task

    src/project.js
       1 |var onesEqual = (1 == 1);
                                ^ Expected '===' and instead saw '=='

>> 1 error in 1 file
Warning: Task "jshint:all" failed. Use --force to continue.

Aborted due to warnings.
Completed in 0.851s at Sat Aug 09 2014 1109:15 GMT-0500
 - Waiting...
```

And as soon as we correct the error from == to ===, we see this:

```
>> File "src/project.js" changed
Running "jshint:all" (jshint) task
>> 1 file lint free
```

Having this kind of nearly instant linting feedback can make a tremendous difference in the quality of our code, and the speed with which we develop, by helping us catch errors as soon as they occur. We could extend this example by adding feedback from unit tests as well.

But this isn't about Grunt or JSHint, linting or unit tests. The point is that as developers we need to constantly seek out ways of getting better and faster feedback on our code.

Fast feedback from linting is one component. Fast feedback from unit tests is another. But we should also be seeking out fast feedback on test coverage, code complexity, and any other sort of information that encourages us to write better code.

The faster we get useful information about our code, the faster we can improve it, and the faster we can improve our own skills.

Thanks for reading!

Josh Clanton

Creating Private Properties with ES6 Symbols

One important aspect of creating reusable code is the ability to hide internal implementation details from those who will use our code. Hiding those details allows us to revise internal code while still providing the same interface to our consumers. We've talked before about maintaining private data using the module pattern.

However, managing private properties of individual object instances has historically been somewhat clunky. Let's take a look at a couple of approaches we could use.

Suppose that we are dealing with superheroes who need to keep their secret identities, well, secret. One common approach is *privacy by convention*.

```
function Superhero (name, identity) {
    this.name = name;
    this._identity = identity;
}

Superhero.prototype.goCivilian = function () {
    console.log("Attend social events as " + this._identity);
};

var captainMarvel = new Superhero ("Captain Marvel", "Billy Batson");

captainMarvel.goCivilian();
// => "Attend social events as Billy Batson"
```

The problem here is that `_identity` isn't really private. Anyone who happens to look at `_identity` can find out that Captain Marvel is Billy Batson. In addition, this property will show up when iterating over properties using a `for..in` loop.

We could also try using function scope to enforce privacy.

```
function Superhero (name, identity) {
    this.name = name;
    this.goCivilian = function () {
        console.log("Attend social events as " + identity);
    };
}

var captainMarvel = new Superhero ("Captain Marvel", "Billy Batson");

captainMarvel.goCivilian();
// => "Attend social events as Billy Batson"
```

In this case we **do** have real privacy. But in order to achieve it we have to make `goCivilian` a property of the object instance rather than a property of `Superhero.prototype`. That means that each superhero will have a separate copy of `goCivilian`. This is better than just using convention, but it's still not ideal.

With ECMAScript 6 (the next version of JavaScript) we have a new way to create private properties using symbols.

```
var Superhero = (function () {
    var secretIdentity = Symbol("secret identity");

    function Superhero (name, identity) {
        this.name = name;
        this[secretIdentity] = identity;
    }

    Superhero.prototype.goCivilian = function () {
        console.log("Attend social events as " + this[secretIdentity]);
    }

    return Superhero;
})();

var captainMarvel = new Superhero ("Captain Marvel", "Billy Batson");

captainMarvel.goCivilian();
// => "Attend social events as Billy Batson"
```

So far so good, but will this really keep things private? Let's try accessing `captainMarvel`'s secret identity.

```
captainMarvel[secretIdentity];
// => ReferenceError: secretIdentity is not defined
```

What if we loop over properties using `for..in`?

```
for (var key in captainMarvel) {
    console.log(captainMarvel[key]);
}
// => "Captain Marvel"
```

From the outside it looks like Captain Marvel's secret identity doesn't even exist. How does this work?

In ES6, symbols are a new type of primitive value that can be created by using the `Symbol` function. But the really new thing here is that symbols can be used as object keys, as we see above with `this[secretIdentity]`.

When creating the symbol we can optionally pass in a description string, but all symbols created by `Symbol` are unique and cannot equal one another. Because they are unique, a user can't just use `Symbol("secret identity")` to get access to Captain Marvel's secrets.

```
var externalSecretIdentity = Symbol("secret identity");

captainMarvel[externalSecretIdentity];
// => undefined
```

And as we saw above, object properties whose keys are symbols don't show up using the normal object inspection approaches. This combination means that as long as we keep our symbols in a private scope (perhaps using the module pattern) the property itself will remain private as well.

There is, however, one caveat. ES6 also includes a new method called `Object.getOwnPropertySymbols` which allows users to discover what symbols are being used as an object's keys, and potentially access the properties. For example:

```
Object.getOwnPropertySymbols(captainMarvel);
// => [Symbol(secret identity)]

var symbols = Object.getOwnPropertySymbols(captainMarvel);

var secretIdentSymbol = symbols[0];

captainMarvel[secretIdentSymbol];
// => "Billy Batson"
```

So the privacy of these properties isn't absolute, but they also can't be read or modified accidentally. Anyone using `getOwnPropertySymbols` to manipulate private properties knows that what they are doing is dangerous and can lead to unpredictable results if it isn't used carefully.

ES6 symbols aren't yet ready for production use, but will be hitting mainstream browsers soon. They are available in Chrome 38 (currently in beta) and FireFox 33 (also in beta).

Now is the time to start playing with symbols so you'll be familiar with them when they are ready for prime time.

Thanks for reading!

Josh Clanton

Holy Bat-Signal, Batman! Implementing the Observer Pattern in JavaScript

When trying to write clean JavaScript, there is a common problem that you can run into. You may have managed to separate your program's concerns into separate objects, yet still need object A to know if something happens to object B. Let's take a look at an example.

```
var jimGordon = {
    arrest: function (criminal) {
        console.log("Jim Gordon arrests " + criminal.name);
    },
    respondToCrimeBy: function (criminal) {
        if (!criminal.superVillain) {
            this.arrest(criminal);
        } else {
            console.log("Jim Gordon can't stop " + criminal.name);
        }
    }
};

(function () {
    var batman = {
        defeatSupervillain: function (villain) {
            console.log("Batman handily defeats " + villain.name);
        }
    };
})();

var joeShmoe = {
    name: "Joe Shmoe",
    supervillain: false
};

jimGordon.respondToCrimeBy(joeShmoe);
// => "Jim Gordon arrests Joe Shmoe"

```

```
var mrFreeze = {
    name: "Mr. Freeze",
    superVillain: true
};

jimGordon.respondToCrimeBy(mrFreeze);
// => "Jim Gordon can't stop Mr. Freeze"
```

In this example, Jim Gordon is responsible for responding to crimes, and he can handle arresting your typical petty crooks, but he can't handle supervillains on his own. That responsibility belongs to Batman. But Batman doesn't respond to ordinary crimes directly. Gordon needs some way to notify Batman when he has a supervillain issue.

The problem is that while Batman knows how to get hold of Gordon, Gordon doesn't know how to get hold of Batman. That's where the observer pattern[91] comes in.

```
var jimGordon = {
    arrest: function (criminal) {
        console.log("Jim Gordon arrests " + criminal.name);
    },
    respondToCrimeBy: function (criminal) {
        if (!criminal.superVillain) {
            this.arrest(criminal);
        } else {
            this.notifyObservers(criminal);
        }
    },
    observers: [ ],
    registerObserver: function (observerFn) {
        this.observers.push(observerFn);
    },
    notifyObservers: function (data) {
        this.observers.forEach(function(observerFn) {
            observerFn(data);
        });
    }
};

(function () {
    var batman = {
        defeatSupervillain: function (villain) {
```

[91] http://en.wikipedia.org/wiki/Observer_pattern

```
            console.log("Batman handily defeats " + villain.name);
        },
        batSignal: function (villain) {
            batman.defeatSupervillain(villain);
        }
    };

    jimGordon.registerObserver(batman.batSignal);
})();

var mrFreeze = {
    name: "Mr. Freeze",
    superVillain: true
};

jimGordon.respondToCrimeBy(mrFreeze);
// => "Batman handily defeats Mr. Freeze"
```

Let's walk through how this works. By adding the `observers`, `registerObserver`, and `notifyObserver` properties to Gordon, we've made him "observable," which means that if someone like Batman wants to know about what's happening in Gotham, they just need to give Gordon a way to let them know about it. That's what `registerObserver` does. It takes a function that someone gives to `jimGordon` and keeps it around in the `observers` array. When something of interest happens, Gordon can `notifyObservers` and pass along the information that he has.

That way, even though `jimGordon` doesn't have direct access to `batman`, Gordon can use the `batSignal` method to let Batman know what is going on.

One of the important aspects of the observer pattern is that we can register multiple observers.

```
(function () {
    var robin = {
        defeatSupervillain: function (villain) {
            console.log("Robin helps defeat " + villain.name);
        },
        radio: function (villain) {
            robin.defeatSupervillain(villain);
        }
    };

    jimGordon.registerObserver(robin.radio);
})();

```

```
14 var penguin = {
15     name: "Penguin",
16     superVillain: true
17 };
18
19 jimGordon.respondToCrimeBy(penguin);
20 // => "Batman handily defeats Penguin"
21 // => "Robin helps defeat Penguin"
```

Of course, there are other ways of implementing the observer pattern as well. In particular, the property names don't have to be the exact ones I chose here: `observers`, `registerObserver` and `notifyObserver`. The important thing is the overall pattern allowing you to register and notify observers.

Now you have one more tool to keep your object design clean and well-factored.

Thanks for reading!

Josh Clanton

Even Stricter Equality with Object.is

One of the most common recommendations in JavaScript style guides is to always[92] (or almost always[93]) make comparisons using the === strict equality operator, because it avoids a lot of ambiguities in the ordinary == equality operator. However, === does have two oddities of its own.

First, the special value `NaN` is **not** strictly equal to itself.

```
console.log(NaN === NaN);
// => false
```

Second, the values `0` (positive 0) and `-0` **are** strictly equal, even though they are distinct values in JavaScript. (The reasons for why they are distinct values is a bit complicated, but Dr. Axel Rauschmayer has a great article on the topic[94].)

```
console.log(0 === -0);
// => true
```

Of course, there are ways to work around these ambiguities. As I've mentioned before, we can detect `NaN` by using the fact that it is the only value which is not equal to itself.

```
function isNaNValue (val) {
    return val !== val;
}

console.log(isNaNValue(NaN));
// => true
```

And we can detect `0` versus `-0` by using the fact that JavaScript also distinguishes between `Infinity` and `-Infinity`.

[92] https://github.com/airbnb/javascript#conditional-expressions--equality
[93] http://contribute.jquery.org/style-guide/js/#equality
[94] http://www.2ality.com/2012/03/signedzero.html

```
function isNegativeZero(val) {
    if (val !== 0) {
        return false;
    }

    var thisInfinity = 1 / val;

    return (thisInfinity < 0);
}

isNegativeZero(-0);
// => true
```

However, even though we can detect these things, it would be nice if there was something built into the language that would do it for us. That's where `Object.is` comes in.

ECMAScript 6 adds the new `is` method to the global `Object` object. The entire reason for `Object.is` to exist is resolving these ambiguities.

For `Object.is`, `NaN` equals itself:

```
console.log(Object.is(NaN, NaN));
// => true
```

`Object.is` also treats `0` and `-0` as distinct values:

```
console.log(Object.is(0, -0));
// => false
```

So, when ES6 finally comes around will you be better off using `Object.is` instead of `===` to compare values? Probably not. While `Object.is` resolves some ambiguities in `===`, encountering those ambiguities is relatively rare. You should keep `Object.is` in your JavaScript toolbox, but it's not an everyday tool for most people.

Because it is an ES6 method, `Object.is` isn't available everywhere yet. But it is available in FireFox and Chrome for the past few versions. And if you'd like use to it in older browsers, you can polyfill it using Paul Miller's es6-shim[95].

Thanks for reading!

Josh Clanton

[95] https://github.com/paulmillr/es6-shim