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.

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

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.

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[13] 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.

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