JavaScript & DOM

Tips, Tricks, and Techniques

No. 3

Louis Lazaris

Tips and Tutorials for all Levels of JavaScript Developers

JavaScript & DOM Tips, Tricks, and Techniques (Volume 3)

Quick Tips and Tutorials for all Levels of JavaScript and Front-end Development

Louis Lazaris

This book is for sale at http://leanpub.com/javascriptdom3

This version was published on 2019-04-01

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.

Contents

Preface 1

String.prototype.link() 2

innerHTML and the DOM 3

Attaching Properties to Functions 4

Using the delete Operator 6

Arguments Aliasing Removed in Strict Mode 8

parentNode 9

The form Property on Input Elements 10

getComputedStyle() for Fonts 11

HTML Comments in JavaScript 13

Optional Parameters for Window.setTimeout() 14

history.back with iframes 15

Understanding the Callstack 16

Tips on Using `const` 18

The `for...of` Loop 20

ES6 String Methods 22

String.repeat() 24

Nested Template Literals 26

Using function.length with the Rest Operator 27

SEO-Friendly Ajax States 28

Objects in ES6 30

The Comma Operator 32

ES6 Object Destructuring 34

Destructuring Assignment 36

ES6 Object Destructuring with Nested Objects 37

ES6 Array Destructuring 39

Array Destructuring Swap 40

Destructuring in Function Arguments 42

ES5 Multi-line Strings 44

Computed Object Properties in ES6 46

ES6 Sets 48

More Tips on Using ES6 Sets 49

Array.of() in ES6 51

Array.from() in ES6 53

Array.from() Map Function 54

Array.from() and the Optional this Value 55

Array.find() and Array.findIndex() 56

ESx Naming vs. ES20xx Naming 57

A Simple Introduction to Closures in JavaScript 58

HTML Collections 60

DOM Scripting on Malformed or Invalid HTML 61

A Little-known Fact About Hoisting 63

Getting the Value of a Selected Radio Button 65

The querySelector() Single-Byte Trick 66

Triggering a Click Event 67

The focusin and focusout Events **68**

Intro to ES6 Modules **70**

More on ES6 Modules **72**

Importing a Full Module **74**

Renaming Module Exports **75**

JavaScript Errors and the Error Object **76**

Some Facts on ES6 Arrow Functions **77**

Default Exports in ES6 Modules **79**

An Intro to the Page Visibility API **81**

Using document.domain for Cross-origin Requests **82**

What is JSON? An Introduction and Guide for Beginners **84**
- JSON Defined 84
- A Brief History of JSON 85
- How is JSON Different from a JavaScript Object? 85
- How Should JSON be Stored? 87
- Using JSON.stringify() 87
- Using JSON.parse() 89
- Using JavaScript to Manipulate JSON 90
- Is JSON Better Than XML? 91
- What is JSONP? 92
- How Does JSONP Work? 93
- JSON Tools 94
- Conclusion 95

Using Default Parameters in ES6 . . . . . **96**
- Default Parameters in ES5 and Earlier 96
- Default Parameters in ES6 97
- Dealing With Omitted Values 98
- Default Parameter Values and the arguments Object 99
- Expressions as Default Parameters 101
- Conclusion 103

An Introduction and Guide to the CSS Object Model (CSSOM) **104**
- What is the CSSOM? 104
- Inline Styles via element.style 105
- Getting Computed Styles 106

Getting Computed Styles of Pseudo-Elements . . . 107
The CSSStyleDeclaration API . . . 109
setProperty(), getPropertyValue(), and item() . . . 109
Using removeProperty() . . . 110
Getting and Setting a Property's Priority . . . 110
The CSSStyleSheet Interface . . . 112
Working with a Stylesheet Object . . . 113
Accessing @media Rules with the CSSOM . . . 115
Accessing @keyframes Rules with the CSSOM . . . 116
Adding and Removing CSS Declarations . . . 119
Revisiting the CSSStyleDeclaration API . . . 120
The CSS Typed Object Model… The Future? . . . 122
Final Words . . . 123

More Weekly Tips! . . . **124**

About the Author . . . **125**

Preface

Every week in my newsletter, Web Tools Weekly[1], I publish a categorized list of tools, scripts, plugins, and other cool stuff geared towards front-end and full-stack developers. In addition to the weekly list of tools, I also start the issue with a brief tutorial or tip, usually something focused on JavaScript and the DOM. This book contains tips just like the ones in my newsletter.

Special thanks and kudos to Nicholas Zakas, David Flanagan, Douglas Crockford, and Cody Lindley for their published works on JavaScript and the DOM that have been a huge inspiration to my own research and writing. Thanks also to Chris Coyier of CSS-Tricks who allowed me to republish a few of the articles I wrote for his website.

I'll continue to publish similar tips in the newsletter in the weeks ahead, so be sure to subscribe if you like what you read here.

Enjoy the book!

[1] https://webtoolsweekly.com/

String.prototype.link()

I just came across one of these obscure JavaScript features that maybe you've seen before but it's a first for me. It's a way to create an HTML link from a String object, and it's part of the ECMAScript specification (i.e. it's not a DOM method). Best way to explain it is to see an example:

```
let linkText = 'Web Tools Weekly',
    url = 'https://webtoolsweekly.com';

document.querySelector('output').innerHTML = `Visit the ${linkText.link(url)} websit\
e.`;
```

You can view this example on CodePen[2].

As the code above demonstrates, in order to generate a link from the link text and the URL, I simply call the `link()` method on the string that holds the link text. The `link()` method takes one argument: the URL that will become the `href` value of the generated link.

As you can see from the code and comments in the example below, the link method returns a String object, not an HTML element.

```
console.log(typeof linkText.link('example'));
// "string"

console.log(linkText.link('example'));
// "<a href=\"example\">Web Tools Weekly</a>"
```

Once you have the returned result, it's trivial to insert the returned value into the DOM using something like `innerHTML()`. By contrast, if I was to use the `textContent()` property to inject the string, then I'd get the HTML characters as text on the page, as seen in this demo[3]. Likely not what I want, but it helps to understand what gets returned.

You can see this feature in the latest ECMAScript spec[4], and on MDN[5].

[2] https://codepen.io/impressivewebs/pen/ZZzoxr?editors=0011
[3] https://jsbin.com/gozocax/edit?html,js,output
[4] https://tc39.github.io/ecma262/#sec-string.prototype.link
[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/link

innerHTML and the DOM

While fiddling around with some JavaScript, I came across something that had me stumped. If you're relatively new to JavaScript, see if you can figure out what the problem is. Take a look at this CodePen demo[6].

If you're experienced with JavaScript and the DOM, there's a good chance you'll understand immediately what's happening. Why doesn't the second button allow you to append the message multiple times? Before reading further, try and figure it out, then come back.

I posed the problem on Twitter[7] and Phil Walton was kind enough to explain. This was definitely something that I felt like I should have been able to understand because I'm sure I've read something about this in the past. Anyhow, here's a quick summary of the problem and the explanation:

- When using an inline `onclick` handler on a button, I'm able to append a message to `document.body` multiple times using `innerHTML`.
- When using `addEventListener` or `btn.onclick` to attach the event, the button would only add the message once (subsequent clicks did nothing).

The explanation:

- The line that uses `innerHTML+=` overwrites the event handlers because it destroys and recreates the DOM nodes.
- Declaring the event handler inline apparently avoids this problem (I'm still not clear on the logic of it in this case, because to me it appears to be doing the same thing).

The same basic problem is discussed in this Stack Overflow thread[8] where it's suggested to use `appendChild()` instead. If you do a search on this subject, you'll notice some really old articles that discuss the problem, as well as some similar Q&A threads.

In summary, if you use `innerHTML` to add content to an element, remember that the events attached to those elements will not be reattached, even when you're appending to existing content.

[6] https://codepen.io/impressivewebs/pen/QPLxmV?editors=0010
[7] https://twitter.com/ImpressiveWebs/status/832057322981695488
[8] http://stackoverflow.com/questions/595808/is-it-possible-to-append-to-innerhtml-without-destroying-descendants-event-list

Attaching Properties to Functions

In the book Secrets of the JavaScript Ninja[9], the authors discuss a technique called *memoization*, in which they make use of a JavaScript feature I wasn't previously aware of.

If you want a good intro to memoization, this article by Addy Osmani[10] is a good start or you can grab a copy of the aforementioned book. The specific feature I'm referring to, however, is used in the book and in Addy's article.

Briefly, memoization is a way to allow a function to remember previously computed results, so as to avoid repeating highly-intensive computations unnecessarily. So a memoization function will cache results and this can be done by taking advantage of the fact that functions are objects on which you can attach properties.

Below is a rudimentary example of the basic principle of using function properties to cache values so you'll know if the function has already been run. Note that this is not a proper memoization technique, this is just to demonstrate the properties-on-functions technique:

```
let id = 1;

function checkItOut (fn) {
  if (fn.done) {
    fn.done = id++;
    console.log(`Function run # ${fn.done}`);
  } else {
    fn.done = id++;
    console.log(`First function run ('${fn.done})`);
  }
}

function doSomething () {
  // nothing needed here...
}

checkItOut(doSomething); // "First function run (1)"
checkItOut(doSomething); // "Function run #2"
checkItOut(doSomething); // "Function run #3"
```

View on CodePen[11]

[9] https://webtoolsweekly.com/reading/?view=2JqYwea
[10] https://addyosmani.com/blog/faster-javascript-memoization/
[11] https://codepen.io/impressivewebs/pen/gyYKZo?editors=0011

In this code, the `doSomething()` function (paradoxically) does nothing. But each time I call it, I'm attaching the incrementing value of the `id` variable to that function's `done` property. This gives the function a sort of memory of itself, so I can tell how many times it's been called. In the demo you can see that each log has a new value.

As mentioned, this is not proper memoization, this is just a simple example to demonstrate that you can attach properties to functions, and how this might be useful. The book[12] discusses this in greater detail and there are lots of sources online, including the aforementioned article, for a better understanding of memoization. You might also want to check out this memoization tool[13].

[12] https://webtoolsweekly.com/reading/?view=2JqYwea
[13] https://github.com/caiogondim/fast-memoize.js

Using the delete Operator

JavaScript's `delete` operator is a handy one to understand and be cautious about, so here's a quick summary of it along with some of its quirks.

The `delete` operator allows you to remove a property from an object. In the following code I'm removing the `one` property:

```
let myObj = { one: "one", two: true, three: 3 };

delete myObj.one;

console.log(myObj);
/* [object Object] { three: 3, two: true } */
```

View on CodePen[14]

The `delete` operator has a return value, a Boolean representing whether the property was removed. So, from the previous code, if I logged the `delete` line, it would return "true".

Another quirk is the fact that `delete` will return `true` even if you try to remove a non-existent property. I guess this is kind of like it's saying "item is not there" even if the action wasn't performed at that moment.

```
let myObj = {
  one: "one",
  two: true,
  three: 3
};

console.log(delete myObj.four); // still logs true
```

View on CodePen[15]

Technically, variables and functions defined with `var`, `let`, and `const` are properties of the global `window` object (assuming JavaScript in the browser). However, using `delete` on these is not allowed. In non-strict mode, such an operation will return `false` and the item won't be deleted. In strict mode, you'll get a syntax error. This happens because these properties are considered non-configurable.

The last point I'll mention here is demonstrated in the next code example, when using `delete` with arrays:

[14] https://codepen.io/impressivewebs/pen/wZwXOx?editors=0011
[15] https://codepen.io/impressivewebs/pen/JVPZqK?editors=0011

```
let myArrayOne = ['one', 'two', 'three', 'four'],
    myArrayTwo = ['one', 'two', 'three', 'four'];

// remove Array element from each array
// using 2 different techniques
myArrayOne[2] = undefined;
delete myArrayTwo[2];

// same result, both have item as undefined
console.log(myArrayOne, myArrayTwo);

// length = 4 for both
console.log(
  myArrayOne.length,
  myArrayTwo.length
);

// But the item doesn't exist in one array
console.log(2 in myArrayOne); // true
console.log(2 in myArrayTwo); // false
```

View on CodePen[16]

As you can see, although the `delete` operator can be used to remove an item from an array, the result is slightly different from setting an array item to `undefined`. There's lots more info on `delete` in MDN's article[17].

[16] https://codepen.io/impressivewebs/pen/LvPrKx?editors=0010
[17] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete

Arguments Aliasing Removed in Strict Mode

Most JavaScript developers are using strict mode, a feature introduced in ES5 to opt in to a restricted variant of JavaScript. One of the changes in strict mode is the fact that the `arguments` object is no longer an alias for function parameters.

For example:

```
function myFunc (a, b) {
  arguments[0] = 30;
  return a + b;
}

// returns 33 in non-strict mode
console.log(myFunc(2,3));
```

Try on CodePen[18]

Notice that before returning the result, I've redefined the first parameter in the arguments object. Before ES5, the return value in this example would be `33` instead of `5`. You'll also notice that some linters will give a warning when you do this.

Here's the same code using strict mode inside the function:

```
function myFunc (a, b) {
  'use strict';
  arguments[0] = 30;
  console.log( arguments[0] ); // 30
  return a + b;
}

// returns 5 in strict mode
console.log( myFunc(2,3) );
```

Although I've still attempted to redefine the first parameter, the return value is what's expected. Notice that I'm still able to redefine the first element in the `arguments` object, it's just not mapped to the parameters anymore.

And note that the reverse was also true in non-strict mode. That is, you could redefine one of the parameters and this would again be aliased to the `arguments` object. But not in strict mode.

[18] https://codepen.io/impressivewebs/pen/PgYBYe?editors=0011

parentNode

The `parentNode` property is one of the simplest and most commonly used DOM features. It's probably one you're well familiar with, but if not, then definitely have it ready to go in your DOM toolbox.

It's very handy when manipulating the DOM to be able to quickly reference the parent of a specified node in the tree. So although this one is an easy one to use, I just wanted to use this opportunity to say "Hey, `parentNode`, you rock, and I like having you at my finger tips." Here are some things you can do with it.

With `parentNode`, you can easily remove the first child of the current element's parent:

```
child.parentNode.removeChild(child.parentNode.firstChild);
```

You can check the existence of a class (or maybe a `data-*` attribute) on the current element's parent (for example to check the context of the current element):

```
if (el.parentNode.classList.contains('abc')) {
  // do something...
}
```

You can chain as many parentNode's as you need to in order to check on something up the DOM tree:

```
console.log(child.parentNode.parentNode.nodeName); // BODY
```

You can insert a new element next to a specific element:

```
let el = document.createElement('div');
child.parentNode.insertBefore(el, child.nextSibling);
```

It has so many uses, that's just a small sampling. Of course, the manipulations could probably be done in other ways too, but sometimes `parentNode` is the simplest and easiest to maintain.

And one more good-to-know fact about `parentNode`: `Document`, `DocumentFragment`, and newly created nodes not yet inserted in the DOM do not have a `parentNode` so the property will return `null` for those.

The form Property on Input Elements

There are lots of things you can do with form elements with DOM scripting, a few of which I've discussed in previous tips. One that you may not be aware of is the `form` property that you can access on any form element.

This property allows you to easily and unambiguously get a reference to the containing form for the targeted element. As an example, let's assume I have two separate forms on a page, each with a submit button:

```
let sOne = document.querySelector('input[type=submit]'),
    sTwo = document.querySelectorAll('input[type=submit]')[1],
    op = document.querySelector('output');

op.innerHTML = `${sOne.id} belongs to ${sOne.form.className}<br>
                ${sTwo.id} belongs to ${sTwo.form.className}`;
```

Try it on CodePen[19]

In the code above, I'm getting a reference to the two different submit buttons, then I'm printing the class of each form to the page. You can do this with any form element. Just get a reference to the element and then check the `form` property of that element.

The `form` property is a read-only property that's part of the HTMLInputElement interface[20], which has lots of other methods and properties, some more commonly known than others.

[19] https://codepen.io/impressivewebs/pen/gyOrgg?editors=0010
[20] https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement

getComputedStyle() for Fonts

You're likely familiar with `window.getComputedStyle()`, which is an easy way to get the computed value of a specific CSS property. That method, however, has its flaws.

For example, if you try to get the computed value of the `font-family` property for a specific element, you'll just get whatever font stack is defined.

```
console.log(
  window.getComputedStyle(document.body)
        .getPropertyValue('font-family')
);

// "Arial, Helvetica, sans-serif"
```

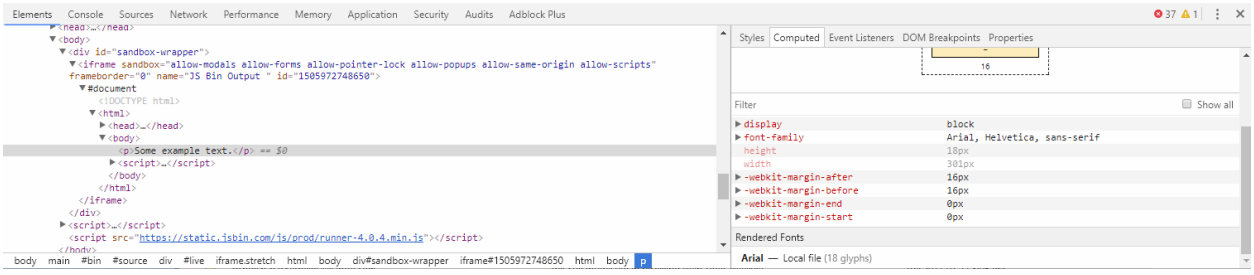
That's not bad, but this doesn't actually tell you which font is being rendered on the page.

If you want to know the font for debugging purposes, you can see what font is rendered by using your browser's developer tools.

In Chrome:

- Right-click the text on the page that you want to check, then choose "Inspect"
- In the right pane select the "Computed" tab
- Scroll down to the bottom of the "Computed" tab (you might have to drill down into the element to select the text node itself)

You should see the "Rendered Fonts" section of the "Computed" tab. See the following screenshot:



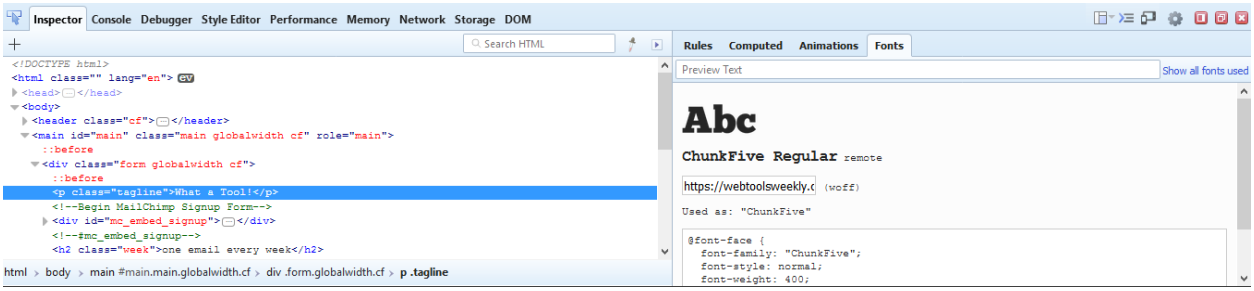
Rendered Fonts in the Computed Tab in Chrome

In Firefox, it's almost the same:

- Right click text, choose "Inspect Element"

- Choose the “Fonts” tab (instead of “Computed”)

See this screenshot:



Rendered Fonts in the Fonts Tab in Firefox

Again, this is useful for debugging purposes and for examining fonts on other websites, but as far as reading the rendered font with JavaScript, I couldn’t find an easy solution.

There’s a utility called DetectFont[21], which I haven’t tried yet but might do the job. There’s also the possibility to use the Canvas API to check for the size of the rendered text and then determine it that way. But from the code I’ve seen, it looks like that’s a pretty complex solution. This Chrome extension[22] actually uses Canvas to do this, so it seems like Canvas is the only way.

[21] https://github.com/Wildhoney/DetectFont
[22] http://www.chengyinliu.com/whatfont.html

HTML Comments in JavaScript

If you've been coding JavaScript since the early days of the web, then you might recall writing inline scripts like this in your HTML:

```
<script language="javascript">
<!--
  if (document.all) {
    // do something...
  }
// -->
</script>
```

Notice the HTML comments around the JavaScript. This was done to ensure that browsers that didn't understand JavaScript would view this as nothing but an HTML comment. You could say it was a very early version of graceful degradation.

But maybe you didn't know that the ES6 specification has actually standardized HTML-like comments, for the purpose of ensuring backwards compatibility with older pages that might contain this kind of code. You can see this described in the spec here[23]. You can also read about it in the HTML5 spec in a sub-section covering scripting[24].

So technically it's valid to include HTML comments in your JavaScript. You can try it out in this CodePen[25].

Notice a couple of lines in the JavaScript with HTML-like comments. I don't see any kind of error in the console in the latest Chrome, Firefox, or Edge so it seems to work fine as long as the comments are single-line comments.

As mentioned, browsers already supported this behaviour and the spec added it to ensure compatibility with older pages, which it explains at the start of the section where this feature is discussed:

> "All of the language features and behaviours specified in this annex have one or more undesirable characteristics and in the absence of legacy usage would be removed from this specification. However, the usage of these features by large numbers of existing web pages means that web browsers must continue to support them."

It's unlikely you'll ever need to use HTML comments in your JavaScript, and you probably shouldn't use them – but this is definitely a nice thing to know in case you weren't familiar with old-school scripting that used the aforementioned technique.

[23] https://tc39.github.io/ecma262/#sec-html-like-comments
[24] https://html.spec.whatwg.org/#restrictions-for-contents-of-script-elements
[25] https://codepen.io/impressivewebs/pen/eoYZWb?editors=0011

Optional Parameters for Window.setTimeout()

JavaScript's `window.setTimeout()` method is commonly used to create a delay before executing another bit of code. The first two parameters it takes are the function to be called when the delay is complete and the length of the delay in milliseconds:

```
// 3-second delay before executing doSomething()
window.setTimeout(doSomething, 3000);
```

Fairly straightforward and many of you have probably used that before. You can also use an anonymous function in there instead of pointing to a named one. And of course, you can use `clearTimeout()` to cancel a timeout that's in progress.

A lesser-known feature available with `setTimeout()` is the fact that it takes an unlimited number of optional parameters after the delay value. These optional parameters, supported in all browsers including IE10+, allow you to pass values into the called function:

```
let btn = document.querySelector('button'),
    op = document.querySelector('output'),
    a = 5,
    b = 7;

btn.addEventListener('click', () => {
  op.innerHTML = 'Calculating...';
  setTimeout(doSomething, 2000, a, b);
});
```

Try it on CodePen[26]. Notice the `a` and `b` variables that are passed as parameters inside `setTimeout()`, which are then used inside the `doSomething()` function.

Of course, this wasn't impossible to do before (e.g. you can use an anonymous function, which then calls another function that passes in arguments), but this is a much cleaner way to accomplish the same thing.

[26] https://codepen.io/impressivewebs/pen/PgoNEj?editors=0010

history.back with iframes

If you've dealt with browser history in your apps, then maybe you've come across an interesting behavior with regards to how history works in combination with iframes.

Go to the following demo page[27]. Click to go to the next page using the link provided. You'll notice there's an iframe on that new page, along with a link above the iframe.

The link has an inline click handler that forces the page to go back in the current browsing session:

```
history.back();
```

I could also do the same thing using the `go()` method, and you could of course mimic this by simply using your browser's back button:

```
history.go(-1);
```

The default behavior is that the browser will treat the page visit inside the iframe as part of the browsing history. So, although your focus might be on the full parent page, hitting the back button (or using history scripting) will cause the iframe to go back in history while the parent page stays the same. You'll have to hit the back button again to go to the intended original page.

There is a workaround for this that might be acceptable in certain circumstances. When navigating inside the iframe, you can use the `location.replace()` method to change the iframe's URL, which doesn't add to browsing history:

```
location.replace('source2.html');
```

Here's another demo[28] where the problem is fixed. Try it with the back link as well as your browser's back button. And note that this wouldn't work with other ways of changing the URL via JavaScript (e.g. using `location.assign()` or `location.href`). MDN explains[29]:

> "...after using replace() the current page will not be saved in session History, meaning the user won't be able to use the back button to navigate to it."

[27] https://www.impressivewebs.com/demo-files/history-back-js/
[28] https://www.impressivewebs.com/demo-files/history-back-js-2/
[29] https://developer.mozilla.org/en-US/docs/Web/API/Location/replace

Understanding the Callstack

If you're new to JavaScript or if you've had trouble understanding how JavaScript code executes in comparison to other programming languages, it might help to get familiar with how the call stack works.

The book Secrets of the JavaScript Ninja[30] has a good section covering this, but here is a quick summary along with a test page that you can use to demonstrate how the call stack works.

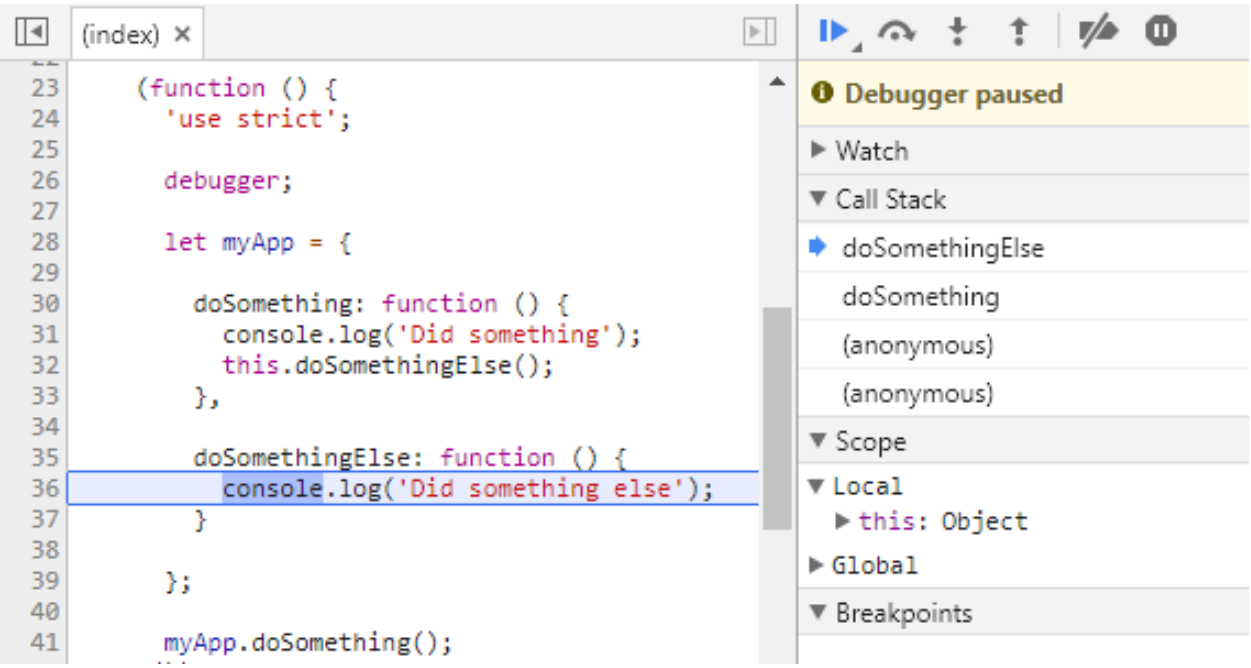
When code is being executed, each statement is executed in one of two types of execution contexts: *global* and *function*. When a function is invoked, the current execution context is stopped and a new context is created. When that function has performed its task, that context is discarded and the previous context is restored.

These different contexts are tracked by the JavaScript engine using a call stack. To steal the book's beautiful illustration, the stack is like a pile of trays in a cafeteria; new trays are added to the top and trays taken away from the stack are removed from the top. Execution contexts work the same way, always being added or removed from the top of the stack.

To make this easy to see in action, I've set up a demo page[31] that runs a simple bit of JavaScript that nicely illustrates this. In order to see how this works, open the page in a private window (to disable add-ons and whatnot) then open your browser's developer tools (F12). The developer tools have a way to view the call stack as it changes, helped out by using JavaScript's `debugger` statement.

Using either the "Sources" tab or the "Debugger" tab (depending on the browser), refresh the page and then use the "Step into…" button to step through the different parts of the code.

[30] https://webtoolsweekly.com/reading/?view=2JqYwea
[31] https://www.impressivewebs.com/demo-files/js-call-stack/



The Call Stack in Chrome's DevTools

As shown in the above screenshot, you can see the "Call Stack" in the right pane, along with the "Scope" and the value of the `this` keyword at the current point in the code. I'm using strict mode, so the value of `this` will occasionally differ from non-strict mode.

There's much more to say about this subject (here's an article](https://medium.com/@gaurav.pandvia/understandin
javascript-function-executions-tasks-event-loop-call-stack-more-part-1-5683dea1f5ec) that might help), but this should serve as a rudimentary introduction to the concept of the call stack, and the demo with the help of dev tools should be a decent albeit simple way to visualize how it works.

Tips on Using `const`

I'm sure by now you've seen posts and examples of code that use ES6's new `const` statement, for declaring constants. For a succinct summary, here is a run-through of the basics regarding `const`.

A `const` cannot be changed after it's initialized. In other words, this will throw an error:

```
const example = 'Hello';
example = 'Bonjour';
// Error: Assignment to constant variable.
```

Also, you can't repurpose a `let` or `var` variable as a `const`, so you'd get an error in that case too (i.e. if the `example` variable was defined using `var`, then defined again using `const`).

You cannot initialize a `const` without defining a value, so this will also throw an error:

```
const example;
// Error: Missing initializer in const declaration
```

Constants, like `let` declarations, are block scoped. This means they aren't accessible outside of the block in which they're declared. This is one of the ways that `let` statements improve on `var`, and this is identical with `const`.

```
if (true) {
  var example = "Hi";
  const example2 = "Hello";
}

console.log(example); // "Hi"
console.log(example2); // Error: example2 is not defined
```

As shown in the example above, the `const` declaration is not "hoisted" so it's not accessible outside the `if` block. And note that you can use `const` to initialize a variable in a `for` loop as long as the loop doesn't modify the value of the `const`.

Although a `const` declaration can't be changed after it's initialized, an object declared as a `const` can be modified. For example, this is valid:

```
const example = {
  one: 'Hello'
};

console.log(example.one); // "Hello"

example.one = 'Hi';
console.log(example.one); // "Hi"
```

Using the above code, it's the value of "example" that can't be altered, not the values of properties within the object itself.

Finally, as a best practice recommendation: Use `const` as the default, unless you know that the value is intended to change; then use `let`. This is different from what most developers might have assumed, which is to use `let` to replace `var`. You should use `const` to replace `var` as long as you know the value is not intended to change.

More info on const on MDN[32] and credit to Nicolas Zakas' Understanding ECMAScript 6[33] for his great coverage of this and other ES6 topics.

[32] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
[33] https://webtoolsweekly.com/reading/?view=2FYjM94

The `for...of` Loop

JavaScript has had a `for...in` loop since the earliest version of the spec. ES6 however, adds a `for...of` loop, which has good support across all modern browsers (but no IE11 support, only Edge).

Due to the syntax being so similar, the difference between the two can be tough to remember, but they are quite different. Simply put, the difference is:

- `for...in` iterates over the enumerable properties of an object
- `for...of` iterates over the property values of objects

You can see how this works by using each type of loop to iterate over the same string:

```
let myString = "test";

// This loop logs: 0, 1, 2, 3
for (let i in myString) {
  console.log(i);
}

// This loop logs: t, e, s, t
for (let i of myString) {
  console.log(i);
}
```

View on CodePen[34]

In this case, `for...in` iterates over the properties (not the values), thus logging the indexes because the object is a string, while `for...of` logs each character in the string (i.e. the values).

In other words, in the case of `for...in`, the variable (`i`) is assigned a different property name on each iteration, whereas with `for...of`, the variable is assigned a different property value.

Some notes and tips to keep in mind for both types of loops:

- `for...in` should not be used if the index order of the items is important because there is no guarantee the items will be given in the intended order
- If you only want to iterate over an object's own properties, use `Object.getOwnPropertyNames()` or something similar, instead of `for...in`.

[34] https://codepen.io/impressivewebs/pen/QPWNBB?editors=0011

- You can use `const` as the variable for either loop, as long as you're not trying to change the value of the variable (thus obeying the rules of `const`)
- You can use statements like `break` or `return` to terminate either of these loops before they complete

More info on MDN: * for...in[35] * for...of[36]

[35] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in
[36] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of

ES6 String Methods

With so much hoopla surrounding arrow functions, template literals, `let` and `const`, and other more well-known ES6 features, there are a few minor additions to the language that have gone relatively unnoticed.

Three new string-related methods would fall under this category:

- `String.includes()` - Check if a string contains a specified string.
- `String.startsWith()` - Check if a string starts with a specified string
- `String.endsWith()` - Check if a string ends with a specified string

For the most part, these are more or less convenience upgrades to the often cumbersome-to-use `indexOf()` and `lastIndexOf()` methods. All three of the new methods return a Boolean, which simplifies their use compared to those other options that return `-1` in place of `false` and the string's position in place of `true`.

Example code:

```
let myString = 'Some example text.';

// Does myString contain given string
console.log(myString.includes('some')); // false
console.log(myString.includes('Some')); // true

// Does myString begin with given string
console.log(myString.startsWith('some')); // false
console.log(myString.startsWith('Some')); // true

// Does myString end with given string
console.log(myString.endsWith('text')); // false
console.log(myString.endsWith('text.')); // true
```

As shown in the code, the search is case sensitive, which you'd expect with string comparisons.

The `includes()` and `startsWith()` methods offer an optional parameter that defines where to start searching (default is 0). It's a little odd to use `startsWith()` in this way, but here's some code to show you how those work:

```
let myString = 'Some example text.';

// Start searching at specified position
console.log(myString.includes('Some', 1)); // false
console.log(myString.includes('Some', 0)); // true

// Start searching at position 5
console.log(myString.startsWith('ex', 5)); // true
```

Pretty simple to use. But note that in the final log, I start searching at position `5`, which returns `true` because the string starts with the specified string at that position (i.e. not at the actual start of the string).

Finally, the `endsWith()` method takes an optional second parameter that redefines the length of the string for the purpose of this search. For example:

```
let myString = 'Some example text.';

console.log(myString.endsWith('text.')); // true
console.log(myString.endsWith('ample')); // false
console.log(myString.endsWith('ample', 12)); // true
```

The first two logs are clear: The string ends with "text.", so that returns `true`; but it doesn't end with "ample" so it returns `false`. However, by telling the method that the string is `12` characters long, I effectively force it to think it ends with "ample", thus returning `true` on the final line.

All the above examples can be found in this CodePen[37].

These methods have strong modern browser support and MDN's articles include a polyfill for each one (includes()[38], startsWith()[39], endsWith()[40]), if you need deeper browser support.

[37] https://codepen.io/impressivewebs/pen/rbNeqE?editors=0011
[38] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/includes
[39] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/startsWith
[40] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith

String.repeat()

Of all the JavaScript and DOM coding tips I've discussed, I guarantee you this one is the greatest of all. It will blow your mind.

Let's say I have a string and for whatever reason I want to repeat that string a specified amount of times. Maybe there's some user input that defines how many times to repeat it. Well, instead of doing some messy string concatenation gymnastics, I can do this:

```
let myString = "Na ",
myNewString = `${myString.repeat(16)}Batman!`;

console.log(myNewString);
```

Try it on CodePen[41]

As shown in the code and the demo, ES6 has added the `String.prototype.repeat()` method to your string manipulation tool box, so you can easily repeat a string a specified amount by passing in a non-negative integer value as the lone argument.

(Alright, I apologize for the hyperbole in that first paragraph, but I think it's nice to appreciate the little things in ES6!)

In his book Understanding ES6[42], Nicholas Zakas offers a nice use case: A code formatting utility that defines indentation level. Here's a hacky example:

```
function doCodeIndent (character = '  ', units) {
  let indent = character.repeat(units),
      indentLevel = 0,
      codeBlock = document.querySelector('pre code');

  codeBlock.innerHTML = "function myFunc () {\n";

  ++indentLevel;
  codeBlock.innerHTML += indent + "if (something) {\n";

  ++indentLevel;
  codeBlock.innerHTML += indent.repeat(indentLevel) + "return true;\n";

```

[41] https://codepen.io/impressivewebs/pen/bJGpZK?editors=0011
[42] https://webtoolsweekly.com/reading/?view=2FYjM94

```
    --indentLevel;
    codeBlock.innerHTML += indent.repeat(indentLevel) + "}\n";

    --indentLevel;
    codeBlock.innerHTML += indent.repeat(indentLevel) + "}\n";
}

// Takes 2 arguments: The character used for indenting and the number of characters \
for each indent.
doCodeIndent(' ', 2);
```

View on CodePen[43]

In that example, which builds a mock JavaScript code block and displays it on the page, you can change the indent character and level by adjusting the arguments in the function call. Everything else will just work based on incrementing and decrementing the `indentLevel` variable.

The only desktop browser that doesn't support `String.repeat()` is IE11, but MDN has a polyfill[44] if you want to use this today with full support.

[43] https://codepen.io/impressivewebs/pen/KYKMKE?editors=0010
[44] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/repeat

Nested Template Literals

Here's a quick tip on using ES6 template literals. As you probably know, template literals allow you to insert a JavaScript expression using `${}` to include a variable within a string. These placeholders (also referred to as substitutions) need not be merely variable references, but can be any JavaScript expression (a calculation, function call, etc).

On top of that, since a template literal placeholder itself is a JavaScript expression, you can nest a placeholder within a placeholder. Here's an example, similar to one used in MDN's article[45] on the topic:

```
// A couple of mock functions
function isLargeScreen() {
  return false;
}

function isCollapsed() {
  return true;
}

// Nested template literals
let classes = `header ${ isLargeScreen() ? '' : icon-${(isCollapsed() ? 'expander' :\
 'collapser')}` }`;

console.log(classes);
```

View on CodePen[46]

The key portion of the code is the line that defines the `classes` variable. Notice the ternary expression that begins inside the first placeholder. If `isLargeScreen()` evaluates to `false`, another ternary expression is evaluated using `isCollapsed()`, which itself nests another placeholder inside the original.

This is just another way that template literals make dealing with strings much easier in JavaScript. A technique like this was certainly possible in ES5, but code becomes much more elegant and easier to maintain using template literals.

[45] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals
[46] https://codepen.io/impressivewebs/pen/GLRqpQ?editors=0011

Using function.length with the Rest Operator

Here's a good-to-know tip related to the rest operator, which is a new feature in ES6. As I discussed in another tip, every function has a `length` property that allows you to expose the number of arguments that a function is expecting. For example:

```
function myFunc (a, b, c, d) { }
console.log(myFunc.length) // 4
```

The `length` property of this function has a value of `4` because, as shown in the code, the function is expecting four arguments. Simple enough, right? But what if you're using rest parameter syntax?

If you're familiar with rest syntax, this feature allows you to use three dots in front of the final argument to represent an indefinite number of arguments as an array. Here are a few examples that use the `length` property on the function while incorporating rest syntax:

```
function myFunc1 (a, ...b) { }
function myFunc2 (a, b, c, ...d) { }
function myFunc3 (...d) { }

console.log(myFunc1.length); // 1
console.log(myFunc2.length); // 3
console.log(myFunc3.length); // 0
```

CodePen demo[47]

As you can see, using rest syntax doesn't add to the function's `length` value. Whatever arguments are expected are the only ones counted, so even if a rest argument is the only argument, the `length` property has a value of zero.

Just something to keep in mind if you happen to get unexpected results when checking a function's `length` while incorporating rest syntax.

[47] https://codepen.io/impressivewebs/pen/WWNxqN?editors=0011

SEO-Friendly Ajax States

If you're concerned about SEO for single page apps, here's something I discovered the hard way. I have an info-site called CSS Click Chart[48] that I recently overhauled and updated that suddenly lost its search engine position for sub-pages. So only the home page was indexed and nothing else.

Prior to the code overhaul, this wasn't the case. All sub-pages (which are Ajax-driven "states", rather than actual pages) were indexed individually. I used a PHP fall-back with query string URLs (e.g. css3clickchart.com/?prop=gradients) for non-JavaScript users, while the majority of users saw URLs with hashes (e.g. css3clickchart.com/#gradients).

Again, for years, this was not a problem. Google indexed the pages just fine. But after re-coding the site, I noticed this was no longer the case. So I asked about it[49] on Google's Webmaster Central Help Forum and did some further research myself. Two key findings:

- Using hash symbols for URL fragments to navigate within the same page is fine and won't affect SEO (some people were confused about this)
- The hash symbol should not be used to represent app state to load new content. As shown by my experience, Google doesn't differentiate between css3clickchart.com and css3clickchart.com/#gradients.

Sources for the above points:

- Google: Stop Using URL Fragments[50]
- Twitter conversation with Google's John Mueller[51]
- Google+ post[52] where Mueller clarifies some misconceptions in the comments section.

What's interesting is that Google's John Mueller claims that this is nothing new. Google has always done it this way. But in my case, I suspect that because all the in-page URLs were pointing to query string based states, Google was using those to provide indexing. It seems that their algorithm changed over the past year or so, and now they'll ignore those states if they're redirected to hash URLs (which is what my site was doing).

Related to this is the use of the hash-bang URL (e.g. css3clickchart.com/#!gradients), which apparently was proposed as a potential solution to this problem (you might recall Twitter using this in the past) but that's now not recommended either, as indicated by the feature deprecation message on this Google Developers page[53].

[48] https://css3clickchart.com/
[49] https://productforums.google.com/forum/#!msg/webmasters/nzxH1L862Ec/NJhioqt5BQAJ
[50] https://www.seroundtable.com/google-no-url-fragments-23449.html
[51] https://twitter.com/JohnMu/status/834362743763431424
[52] https://plus.google.com/+TerryVanHorne/posts/Fb66pPrQW2J
[53] https://developers.google.com/webmasters/ajax-crawling/docs/getting-started

I guess it makes sense that fragments aren't differentiated, and I probably should have figured that out. But it certainly wasn't helping that my pages had no problem being indexed in the past while using this technique. My site is now using query strings for the Ajax states and the PHP fall back, and the sub-pages are now fully indexed[54].

[54] https://www.google.com/search?q=site:css3clickchart.com

Objects in ES6

There have been improvements in ES6 in how we can write object literal syntax. These changes make things a little more succinct and less redundant. Here I'll describe two of those changes.

First, when plugging local variables into values for object literal properties, you would normally do something like this in ES5:

```
var one = 1,
    two = 2;

var myObject = {
  one: one,
  two: two
};
```

This isn't the worst code in the world, but there is some redundancy there, repeating the variable names. In ES6, you can do the following as a shortcut:

```
let one = 1,
    two = 2;

let myObject = {
  one,
  two
};
```

In this case, the property name alone is a shortcut for the usual `property: value` syntax, saving a few keystrokes and making the whole thing a little cleaner. When the JavaScript engine sees a property in an object literal without a value following it, it will look for the same name in a local variable in the containing scope and assign that value to it. If you try the same syntax in an ES5 environment in a non-supporting browser, you'll get an error.

Another similar improvement is when including a function (that is, a method) in an object literal. In ES5 you would do this:

```
var myObject = {
  one: 'one',
  two: 'two',
  three: function () {
    // do something...
  }
};
```

But now you have the option to remove the colon and `function` keyword in favor of something called *concise method* syntax:

```
let myObject = {
  one: 'one',
  two: 'two',
  three() {
    // do something...
  }
};
```

Again, if you try this in an ES5 environment, you'll get an error indicating that the JavaScript engine is looking for a colon in the object literal property/value pair.

In both these instances you can still use the old syntax, which you might end up doing out of habit. But if you prefer shorter, cleaner code, you can start to incorporate these into your code if you haven't done so already.

You can view all these examples in this CodePen demo[55].

[55] https://codepen.io/impressivewebs/pen/KYKgpz?editors=0011

The Comma Operator

The comma in JavaScript is commonly used in a number of contexts:

- Function parameters and arguments: `(param1, param2)`
- Array elements: `[1, 2, 3]`
- Object property/value pairs: `obj = { one: 1, two: 2 }`
- Multiple variables: `let one=1, two=2, three=3`

But technically the comma can also be used as an operator, just as +, =, *, and so forth. The comma operator is usually put in a category all by itself, rather than falling under assignment operators or similar.

As an operator, the comma allows the execution of more than one expression in a single statement. The last item in a statement with multiple comma-separated expressions will always be the one returned.

Take note of the logs for the following example:

```
function myFunc () {
  return i=1, j=2, k=3;
}

console.log(myFunc()); // 3

let num = (3, 4, 5, 6);
console.log(num); // 6
```

The `myFunc()` function returns `3` because that's the value of the right-most expression in the line using the comma operators. The other items are also evaluated but don't return anything so they're rendered useless in that context.

In the second example, the `num` variable is assigned the return value of the expression in parentheses. Again, only the last item is returned, so the return value is `6`.

Similarly, the following looks confusing at first, but makes sense when you understand how the comma operator works:

```
let a = (b = 3, c = 4);

console.log(a); // 4 (right-most expression)
console.log(b); // 3
console.log(c); // 4
```

Outside of the ability to separate multiple variables in a single line, probably the most practical use of the comma operator is in a `for` loop that has multiple loop variables:

```
for (let i=0, j=10; i < j; i++, j--) {
  console.log(i, j);
}
```

View all the above examples in this CodePen demo[56].

Along with a little extra info on MDN's article[57], that should be enough to help you understand how the comma works as an operator.

[56] https://codepen.io/impressivewebs/pen/ROwGWv?editors=0011
[57] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator

ES6 Object Destructuring

Another useful feature added in ES6 is object and array destructuring. Even if you aren't familiar with the technique, you've likely heard the phrase thrown around in ES6 conversations.

In brief, destructuring allows you to extract data from objects and arrays using a simpler and less redundant syntax. Prior to ES6, if you wanted to assign an object's properties to variables, you might do something like this:

```
var robot = {
  name: 'Meepzorp',
  job: 'Guard',
  shell: 'Aluminum',
  cpu: 'Premium'
};

var name = robot.name,
    job = robot.job,
    shell = robot.shell,
    cpu = robot.cpu;

// "Meepzorp" "Guard" "Aluminum" "Premium"
console.log(name, job, shell, cpu);
```

The above works fine for grabbing these values, but the code is too repetitive. With destructuring, the same thing can be accomplished with the following brief and elegant code:

```
let animal = {
  type: 'Domestic',
  ability: 'Stealth',
  skill: 'Flight',
  coat: 'Double'
};

let { type, ability, skill, coat } = animal;

// "Domestic" "Stealth" "Flight" "Double"
console.log(type, ability, skill, coat);
```

View on CodePen[58]

Notice in this latter example that only a singl eline is needed to drop all the object's values into a group of variables.

That's the basics of object destructuring, but there's much more to be said about the concept, which I'll cover in later tips.

[58] https://codepen.io/impressivewebs/pen/yrLjeJ?editors=0011

Destructuring Assignment

Previously, I introduced the concept of object destructuring, a new feature in ES6, using an example that incorporated variable declarations. In addition to that technique, I also have the option to use destructuring in an assignment expression.

So instead of this:

```
let { dairy, grains, vegetables, fruits } = groceries;
```

I would do this:

```
({ dairy, grains, vegetables, fruits } = groceries);
```

Assuming, of course, that `groceries` represents an object that has the number of properties expected (in this case, four).

In the latter example, I'm using four variables that have already been given values in another part of the code (not shown above), but they will be updated to reflect the values taken from the destructured `groceries` object.

Notice that the syntax is slightly different in the assignment statement: there are parentheses placed around the entire statement. This is because the opening curly brace looks like a block statement to the JavaScript parser and a block statement cannot appear on the left side of an assignment statement.

This feature opens up lots of possibilities because I can put a destructuring assignment expression anywhere I would normally put a value. So I can even use it as an argument in a function call:

```
function doSomething (foods) {
  console.log(foods.fruits); // "apples"
  console.log(grains, vegetables, fruits);
  // "bread", "celery", "apples"
}

doSomething({ grains, vegetables, fruits } = groceries);
```

Again, the above assumes I've already defined a `groceries` object. Here's a full demo[59].

[59] https://codepen.io/impressivewebs/pen/EJxLyw?editors=0011

ES6 Object Destructuring with Nested Objects

Continuing on the topic of ES6 destructuring, the power of destructuring is evident in the ability to navigate into nested object structures to get specific bits of information as needed.

For example, let's say I have the following object:

```
let myObj = {
  name: 'Pat',
  location: 'Earth',

  status: {
    initial: {
      start: 1,
      end: 2
    },

    other: {
      start: 10,
      end: 20
    }
  }

};
```

Using destructuring, I can assign deeply nested values from this object to variables for use later in my code:

```
let { status: { initial }} = myObj,
    { status: { other }} = myObj;

console.log(initial.start); // 1
console.log(other.end); // 20
```

View on CodePen[60]

[60] https://codepen.io/impressivewebs/pen/QPWrGy?editors=0011

In previous destructuring examples, the element(s) inside the first curly brace represented the variable(s) that would be used to hold the destructured value(s). In this case, however, the variable is inside another nested set of curly braces. The first thing before the colon here indicates the object where I want to start searching.

Each nested set of curly braces allows me to go deeper into the object. I can then use dot notation to access the property I want, as I did in the code above when logging the values.

If I want to use a different variable name for the object I'm accessing, I can do this:

```
let { status: { initial: initialValue }} = myObj;
console.log(initialValue.end); // 2
```

View on CodePen[61]

In this case, I'm assigning the `status.initial` object to a variable called `initialValue`. This adds a little extra complexity to the process, and might make the code a little harder to maintain, but it's an option if that's something you need to do.

[61] https://codepen.io/impressivewebs/pen/NmWMpq?editors=0011

ES6 Array Destructuring

Previous tips focused on object destructuring in ES6. Let's now look at Array destructuring, which is based on the same principles with a few syntax differences and, of course, is used on arrays instead of objects.

```
1  let fruits = [ 'apple', 'orange', 'pear'],
2      [ fruitOne, fruitTwo ] = fruits;
3
4  console.log(fruitOne, fruitTwo);
5  // "apple"
6  // "orange"
```

The syntax (as shown on line two above) is identical to object destructuring except for the use of square brackets in place of the curly braces that are used with object destructuring. And similarly, the variable names can be whatever you want.

The array that's destructured is not affected or changed in any way so, just like objects, this is simply an easy way to extract data from different parts of the array.

With array destructuring, you also have the option to omit items so you can get to only the ones you want. Here's how that would look using the same array as in the previous code block:

```
1  let fruits = [ 'apple', 'orange', 'pear'];
2  let [ , , fruitThree ] = fruits;
3
4  console.log(fruitThree); // "pear"
```

Notice in order to extract the third item from the array, I've included two empty placeholders using two commas to "skip" the first two and get to the third item, which I've then logged.

Array Destructuring Swap

Continuing the theme of ES6 destructuring, here's a quick tip that allows you to use array destructuring to swap the values of two variables.

Before ES6, to do this you'd normally have to incorporate a temporary variable, something like the following:

```
let one = 'Dogs',
    two = 'Alligators',
    temp;

temp = one;
one = two;
two = temp;

console.log(one); // "Alligators"
console.log(two); // "Dogs"
```

That code is fine. It's not difficult to understand and it does the job. But with array destructuring, look how much cleaner the same swap can be done:

```
let three = 'Birds',
    four = 'Crocodiles';

[ three, four ] = [ four, three ];

console.log(three); // "Crocodiles"
console.log(four); // "Birds"
```

Try it on CodePen[62]

The third line above uses an array destructuring assignment. The main difference here from the previous tip on array destructuring is the fact that the right side of that line is using an array literal. And of course, the left side is the array destructuring pattern where the variables that will absorb the array are named.

I could have used different variable names on the left side of the equal sign (e.g. `[ five, six ]`), but that would defeat the purpose of the swap I want to achieve. Also, I'd have to ensure that I

[62] https://codepen.io/impressivewebs/pen/qwBQjR?editors=0011

initialized the variables properly with `let` or `const`. The right side, however, has to be an object to be destructured, so I'm simply dropping the variables into it to build the array literal.

As mentioned, either technique is fine, but the ES6 method is just a little bit cleaner.

Destructuring in Function Arguments

Here's yet another useful technique associated with ES6 destructuring. You may have seen a pattern like the following:

```
function doStuff(one, two, options) {
  options = options || {};

  let a = options.a,
      b = options.b,
      c = options.c,
      d = options.d;

// more code here...
}

doStuff('example1', 'example2', {
  a: 'opt1',
  b: 'opt2',
  c: 'opt3',
  d: 'opt4'
});
```

The idea here is that the `doStuff()` function takes two mandatory parameters (`one` and `two`) along with a number of optional values as a final parameter in the form of an object (`options`). When you glance at the definition of the `doStuff()` function, however, you might not be able to tell right away that multiple values are possible here.

You can use destructuring to make this more clear:

```
function doStuff(one, two, { a, b, c, d }) {
  console.log(one, two);
  console.log(a, b, c, d);
}

doStuff('example1', 'example2', {
  a: 'opt1',
  b: 'opt2',
  c: 'opt3',
  d: 'opt4'
});
```

Try it on CodePen[63]

How much nicer is that? Not only is the code shorter, but you can tell immediately by looking at the head of the function definition that the final argument is an object of options.

Of course, if you want the final object parameter to truly be optional, you have to provide a default value for it, otherwise the code will throw an error if the third object is not passed in. So you can do this:

```
function doStuff(one, two, { a, b, c, d } = {}) {
  console.log(a, b, c, d);
}
```

Now the browser won't throw an error because I've declared a default empty object value just in case a third argument isn't passed in.

[63] https://codepen.io/impressivewebs/pen/pBoQaM?editors=0011

ES5 Multi-line Strings

Before ES6's template literals were introduced, doing multi-line strings in JavaScript was a little messy, but possible. Oddly, for the longest time, I always thought the following was perfectly valid JavaScript:

```
var message = 'This is a \
               multi-line string.';
```

The idea here is that, for whatever reason, I want my string to break onto multiple lines without closing the quotes. Inserting the backslash at the end of the line allows me to get away with this in pre-ES6 code.

But the fact that all browsers basically handled this the same way (i.e. allowing multi-line strings to work) was not a language feature, but a bug. And many style guides and linters, as I'm sure many of you have learned, will warn about this. For example, see this JSFiddle[64] that gives a JSHint warning: "Bad escaping of EOL. Use option multistr if needed."

So if you don't want to have a warning like that in your linting workflow when using pre-ES6 multi-line strings, you can do the following:

```
/*jshint multistr: true */
var message = 'This is a \
               multi-line string.';
```

Try on JSFiddle[65]

Of course, this doesn't solve the problem, but it prevents the warning and will likely make you feel superficially warm and fuzzy inside. You can also do something similar in linters with config options, rather than as a JavaScript comment.

Google's JavaScript Style Guide advises against this backslash trick[66] and recommends using regular string concatenation to mimic the same idea:

```
let message = 'This is a ' +
              'multi-line string.';
```

[64] https://jsfiddle.net/ImpressiveWebs/p865danw/
[65] https://jsfiddle.net/ImpressiveWebs/wk3188ys/
[66] https://google.github.io/styleguide/javascriptguide.xml#Multiline_string_literals

One final point that you may not have known with regards to these old-school multi-line strings: If you put a space after the backslash, the code will fail. Google's style guide warns specifically about this.

There are other old solutions to this problem, which you can read about in this old and lengthy Stack Overflow thread[67]. Some of the answers are quite creative (e.g. using an array to mimic it) and you also get a pretty good history lesson on bugs and browser support in relation to this topic.

As mentioned, template literals make this problem basically obsolete except for in rare legacy-based environments.

[67] https://stackoverflow.com/questions/805107/creating-multiline-strings-in-javascript

Computed Object Properties in ES6

As you might be aware, properties of objects can be defined using variables or even as strings with spaces in the property names, as long as you use square brackets, as in the following:

```
var myObj = {},
    myProp = 'example property';

myObj['other prop'] = 'example value';
myObj[myProp] = 'another value';

console.log(myObj);
// {other prop: "example value",
//  example property: "another value"}
```

View on CoddPen[68]

The first property of `myObj` contains a space and the second property is defined using an existing variable. But the problem here is you can't access these property names using dot notation. This is because of the space in the property name.

You can also use a space for the property name in an object literal, as follows:

```
var myObj = {
  'example prop': 'example value'
};
```

But if you try to drop a variable on the left side of the colon, it won't work. You won't get the value of the variable, you'll just have a property on that object equal to that variable's name. See the same CodePen[69] for an example.

ES6 now allows you to include computed property names in object literals. So the following is now valid (notice the square brackets):

[68] https://codepen.io/impressivewebs/pen/EJxOJW?editors=0011
[69] https://codepen.io/impressivewebs/pen/EJxOJW?editors=0011

```
let myVar = 'example prop';

let myObj = {
  [myVar]: 'example value'
};

console.log(myObj);
// {example prop: "example value"}
```

View on CodePen[70]

Notice, in contrast to the previous demo, the `myVar` property name now computes to the value it holds. This means you can also use string concatenation within the square brackets to define the property name.

Yet another minor but convenient new feature that makes JavaScript just a little easier to work with.

[70] https://codepen.io/impressivewebs/pen/mgdadQ?editors=0011

ES6 Sets

While arrays in JavaScript are useful (as they are in any language), they require a little wrangling if I don't want an array to contain duplicates. That's why ES6 introduced sets, a new type of collection that cannot contain duplicates.

I can create a set in ES6 by passing an array into the `Set()` constructor:

```
let set = new Set([1, 2, 3, 3, 4, 5, 5, 5, 6]);
console.log(set.size); // 6
```

Notice the array I passed in contains duplicates. But the set essentially strips them out and I'm left with a collection of 6 unique items (instead of the original 9 including duplicates), which is much more useful in certain situations.

I don't have to pass in an array to create a set, but that's one way to do it. I also have access to the `add()` method, so I can do this:

```
let set = new Set();

set.add(1);
set.add('two');

console.log(set.size); // 2
```

Finally, there's the `has()` method, which is probably what you'll use the most. This method allows you to check if an item exists. And because sets can't contain duplicates, it's a simple check, then you can move on to other things without any complexities:

```
console.log(set.has(1)); // true
console.log(set.has('two')); // true
console.log(set.has(3)); // false
```

That's ES6 sets in a nutshell. You can view all the above examples in this CodePen[71]. The feature has strong browser support, which you can review at the bottom of MDN's article[72] on the subject.

[71] https://codepen.io/impressivewebs/pen/rbNoxy?editors=0011
[72] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

More Tips on Using ES6 Sets

In the previous tip, I introduced ES6's new sets feature, a more practical alternative to arrays. With that introduction in mind, here are some other things worth noting when using sets.

Sets don't coerce values when making comparisons as sometimes happens in other parts of the language. For example:

```
let set = new Set([1, '1', 2, 2, 3, 3]);

console.log(set.size); // 4
console.log(set.has('1')); // true
console.log(set.has('2')); // false
```

Keeping in mind that a set cannot contain duplicates, notice in the above code I've included `1` and `'1'` as separate entries in the array. But in the final set, they are considered different, so the string `'1'` is not coerced into a number. The second `2` on the other hand is a duplicate so it's removed.

Another point to take note of is what happens with duplicate objects:

```
let set = new Set(),
    obj1 = {},
    obj2 = {};

set.add(obj1);
set.add(obj2);

console.log(set.size); // 2
```

Try the two above code examples in this CodePen[73].

Notice here I'm using the `add()` method to add two objects as the only items in the set. Although the objects are exactly the same, they are not considered duplicates, so both are kept. If the objects were converted to strings, however, then they would be considered duplicates and one would be stripped out.

Finally, in addition to the methods I've already mentioned, I also have access to the `delete()` method, which removes a single item from a set, and the `clear()` method, which will clear all items from the set:

[73] https://codepen.io/impressivewebs/pen/WWNLKx?editors=0011

```
let set = new Set([1, 2, 3, 4, 5]);

console.log(set.size); // 5

set.delete(3);
console.log(set.size); // 4
console.log(set.has(3)); // false

set.clear();
console.log(set.size); // 0
```

Try it CodePen[74]

As you can see, the third item is removed using `delete()` (confirmed using the `size` property and the `has()` method). Then the entire set is cleared, leaving it with zero items.

[74] https://codepen.io/impressivewebs/pen/xexmQa?editors=0011

Array.of() in ES6

Using arrays in JavaScript has improved greatly since ES5, and that includes some new things added in ES6. One method that improves slightly on array creation is `Array.of()`, which I'll briefly introduce here.

The main use for `Array.of()` is to replace use of the Array constructor, which has some odd behavior. For example:

```
let myArray = new Array(3);
console.log(myArray.length); // 3

let myArray2 = new Array("3");
console.log(myArray2.length); // 1
```

Try it on CodePen[75]

This is not a major problem, but as you can see, passing a single item into the constructor can produce unexpected results. Should the single item be one of the items in the array? Or is it supposed to declare the number of items? Of course, this gets worse in the stream of a large codebase where you don't know what data is coming in.

`Array.of()` makes this easy because it always creates an array made up of whatever items are passed in, regardless of the number of items or the types of data:

```
let myArray = Array.of(1);
console.log(myArray.length); // 1

let myArray2 = Array.of("1");
console.log(myArray2.length); // 1

let myArray3 = Array.of(1, "2");
console.log(myArray3.length); // 2
console.log(myArray3); // [1, "2"]
```

Try on CodePen[76]

In most cases you'll use array literal syntax, which allows you to create the array directly using square brackets. However, in his book Understanding ES6[77], Nicholas Zakas advises:

[75] https://codepen.io/impressivewebs/pen/GLRPag?editors=0011
[76] https://codepen.io/impressivewebs/pen/MRWLWQ?editors=0011
[77] https://webtoolsweekly.com/reading/?view=2FYjM94

> “But if you ever need to pass the Array constructor into a function, you might want to pass `Array.of()` instead to ensure consistent behavior.”

Array.from() in ES6

One of the most welcome additions to JavaScript in recent years is a feature that allows you to convert an array-like object to an array. This is now possible using the `Array.from()` method, added in ES6.

Array-like objects are quite common in front-end applications. Here's a common example to demonstrate how this feature works:

```
function doSomething (a, b, c) {
  console.log(Array.isArray(arguments)); // false
  let args = Array.from(arguments);
  console.log(Array.isArray(args)); // true
}

doSomething(1, 2, 3);
```

View on CodePen[78]

The above code uses one of the most common array-like objects – the `arguments` object accessible inside a function body. You can see that the object is initially not an array, but after using `Array.from()`, the collection becomes a valid array. From there, I then have the option to use any array method to manipulate it.

`Array.from()` can convert any array-like object or any iterable object to an array. One example is a DOM collection. If you want to convert a DOM collection to an array, you can use spread syntax, another ES6 feature: {lang="js"} let myLinks = document.body.childNodes;

```
console.log(Array.isArray(myLinks)); // false

myLinks = [... myLinks];
console.log(Array.isArray(myLinks)); // true
console.log(myLinks.length); // 4
```

Try on CodePen[79]

Notice the collection of child nodes becomes a valid array with a length of four items in the above demo page. If you open your browser's developer tools, you can log out the entire object to see what exactly it contains.

In my example I'm using `childNodes` to create a DOM collection, but I could also use `document.querySelectorAll()`, `getElementsByTagName()`, or any other method that collects DOM elements.

[78] https://codepen.io/impressivewebs/pen/OGJdNR?editors=0011
[79] https://codepen.io/impressivewebs/pen/WWNPpZ?editors=0011

Array.from() Map Function

The ES6 `Array.from()` method that I introduced previously allows you to pass in an optional map function as a second argument. The map function gets called on every element in the array-like object that you're converting to an array, so you can manipulate the items in some way as you convert them. Here's an example:

```
function doNums () {
  return Array.from(arguments, (item) => item + 10);
}

let myNums = doNums(20, 30, 40, 50);

console.log(myNums); // [30, 40, 50, 60]
```

Try it on CodePen[80]

Here the `doNums()` function returns the object being converted to an array (again, it's the `arguments` object). Only this time, as I'm converting the items to an array, I'm altering each of the items using the second argument passed into `Array.from()`. In this example, I'm using ES6's arrow function, but I could use a regular anonymous function or function reference instead.

So the result for the above example produces an array from an array-like object, and the numbers that were passed in were changed – in this case, by adding 10 to each one.

As you can see, `Array.from()` is a simple but powerful feature that allows you to manipulate collections using clean syntax rather than a bunch of complex utility functions that can get messy.

[80] https://codepen.io/impressivewebs/pen/OGJqVQ?editors=0011

Array.from() and the Optional this Value

Previously I showed you how to include a map function as the second argument when using the `Array.from()` method. The map function lets you manipulate a collection as you convert it to an array.

One final optional argument you are permitted to include is the value of `this` when the map function is executed. This is similar to using `bind()`, `call()`, and `apply()`.

In this case, if the map function is an object, defining the value of `this` can come in handy:

```
let myObj = {
  someValue: 5,

  doSomething(a) {
    return a + this.someValue;
  }
}

function doAnotherThing() {
  return Array.from(arguments, myObj.doSomething, myObj);
}

let myArray = doAnotherThing(5, 10, 20);

console.log(myArray); // [10, 15, 25]
```

Try on CodePen[81]

Two things to note above:

- The `doSomething()` function is a method of `myObj`, and it references `this`
- The `doAnotherThing()` function is where `Array.from()` is used. In order to ensure `myObj.someValue` is easily accessible inside the map function, `myObj` is specified as the value of `this`.

If I didn't define `myObj` as the third argument in `Array.from()`, then the value of `this` would either be the `Window` object or `undefined`, depending on whether the code is in strict mode or not. Otherwise, I'd have to refer to it directly using `myObj.someValue`, which is fine but maybe not ideal in certain larger code examples.

[81] https://codepen.io/impressivewebs/pen/xexBrO?editors=0011

Array.find() and Array.findIndex()

The `indexOf()` and `lastIndexOf()` methods have been available on strings since the first edition of ECMAScript, allowing you to find a string within a string. In ES5, the same methods were added for searching inside arrays.

ES6 now conveniently adds the `find()` and `findIndex()` methods, which add similar but more advanced functionality to searching within arrays. Here is some example code that demonstrates both:

```
let myArray = [10, 20, 30, 41, 53];

console.log(myArray.find(function(a) {
  return a < 25;
})); // => 10

let myArray2 = [3, 44, 19, 89, 2];

console.log(myArray2.findIndex(b => b > 75)); // 3
```

Try it on CodePen[82]

The first example uses `Array.find()`, which is passed a callback function to execute on each value in the array until it finds the first one that meets the criteria provided by the function. The log outputs "10" because 10 is the first value that meets the criteria of "less than 25". The only required argument to the callback is the current element being processed (I've used `a` as the variable name).

In the second example I'm using ES6's arrow function syntax to simplify the use of the callback. This one uses `Array.findIndex()`, which will return the location (instead of the value itself) of the first item that meets the given criteria. In this instance, I'm looking for the first item that's greater than the value 75, which is the item at index 3.

That's `Array.find()` and `Array.findIndex()` in a nutshell — two array methods that I'm sure you'll find useful.

[82] https://codepen.io/impressivewebs/pen/gyOEoK?editors=0011

ESx Naming vs. ES20xx Naming

If you're confused when articles, tutorials, and social media posts use the terms ES6, ES2015, ES2016, ES7, ES.Next, etc. then you're not alone. I'm sure many have wondered why there seem to be overlapping terms for all the recent additions to the ECMAScript specification.

Here's a quick summary of what these terms mean, and what terms we generally should be using today.

ES6 was the successor to ES5, so the name for that was obvious. The problem was that ES6 took four years to develop and it seemed impractical to have to wait that long for new features to become official standards and start being added to browsers.

To improve on this, starting with post-ES6 releases, it was determined that the releases would be yearly and would be named after the year in which they were developed. This is why we now have ES2016, ES2017, ES2018, and so on. Even ES6 has been renamed to ES2015 (although in this case, I think it's fine to just say ES6, because so many books and blog posts use the name). Generally speaking, the spec uses both names (e.g. ES2018[83] is also called "ECMA-262, 9th edition"), so you'd be forgiven if you just used the names ES7, ES8, etc.

As for ES.Next, the Wikipedia article on ECMAScript[84] says:

> "ES.Next is a dynamic name that refers to whatever the next version is at time of writing. ES.Next features are more correctly called proposals, because, by definition, the specification has not been finalized yet."

There's no source currently attached to that quote on Wikipedia, so take that with a grain of salt (as you should anything not sourced on Wikipedia). But I think that does sum it up fairly well.

In my opinion, it's probably best to use the terms ES5, ES6, ES2016, ES2017, ES2018, and ES.Next. Although there's a lack of naming consistency there, it's a matter of doing what's most familiar, which I think is more practical for everyone.

[83] http://www.ecma-international.org/ecma-262/9.0/index.html
[84] https://en.wikipedia.org/wiki/ECMAScript

A Simple Introduction to Closures in JavaScript

You've probably seen many articles online discussing the concept of JavaScript closures. The funny thing about closures is you're almost better off not knowing anything about them. And I'd argue that developers for whom JavaScript is their only programming language will benefit from using closures more so than developers who come from a language that doesn't have the feature.

Here is some code that demonstrates the benefits of a closure:

```
let outside = 'This is outside the function.',
    funcRef;

function parentFunction () {
  let inside = 'This is inside the function.';
  function nestedFunction () {
    console.log(outside);
    console.log(inside);
  }

  funcRef = nestedFunction;
}

parentFunction();
funcRef();
```

View on CodePen[85]

For most JavaScript developers, there's really nothing all that special in this code. What this does is so common that developers use it without even thinking and probably take for granted the effect.

The closure in the code helps the variable called `inside` to be accessible even after `parentFunction()` has finished executing. If you try to log the `inside` variable outside `parentFunction()`, you'll get an error (which is just basic JavaScript scoping).

In line with this, in the book Secrets of the JavaScript Ninja[86], the authors define closures succinctly:

> "A closure allows a function to access and manipulate variables that are external to that function. Closures allow a function to access all the variables, as well as other functions, that are in scope when the function itself is defined."

[85] https://codepen.io/impressivewebs/pen/XQWGyK?editors=0011
[86] https://webtoolsweekly.com/reading/?view=2JqYwea

Again, if JavaScript is the only language you've used extensively, then this is probably just redundant information that doesn't help you a whole lot because you've likely been taking advantage of this the whole time you've been using the language. In this Stack Overflow thread[87], the top answer adds this useful bit of info:

> "In C and most other common languages, after a function returns, all the local variables are no longer accessible because the stack-frame is destroyed."

I think that part helps to clarify the contrast. Some of us are so accustomed to using this feature that we don't realize how different it is when closures aren't a feature.

[87] https://stackoverflow.com/questions/111102/how-do-javascript-closures-work

HTML Collections

When you use a DOM method like `querySelectorAll()` or `childNodes()`, you get what's called a `NodeList` object, which is an array-like collection. Depending on the method used, the collection could contain nodes other than element nodes. In the past, these collections were referred to as *HTML collections*, which is basically now a legacy term.

The DOM spec says[88]:

> "HTMLCollection is a historical artifact we cannot rid the web of. While developers are of course welcome to keep using it, new API standard designers ought not to use it."

As far as I can tell, the only three DOM methods that return an HTML collection are `getElementsByTagName()`, `getElementsByTagNameNS()`, and `getElementsByClassName()`. There are properties that return an HTML collection (e.g. `document.forms` or the `element.children` property), but these seem to be the only methods that do so.

None of this is to suggest that you shouldn't use these methods or properties. They'll always be supported and they work fairly well for collecting elements. In addition, HTML collections have two little-known methods you can use, as shown in this code:

```
let myLists = document.getElementsByTagName('ul'),
myListItems = document.getElementsByTagName('li');

console.log(myLists.namedItem('myList').tagName); // UL
console.log(myListItems.item(2).innerText); // "Example item three"
```

Try on CodePen[89]

The methods are:

- `HTMLCollection.item(index)` - Returns the node at the given zero-based index.
- `HTMLCollection.namedItem(id)` - Returns the node that has an ID attribute that matches the string passed in as a parameter. It falls back to the `name` attribute if an ID isn't found and if the element supports the `name` attribute.

Both these methods are in the spec, and browser support is good across the board (including IE11 and Edge).

[88] https://dom.spec.whatwg.org/#htmlcollection
[89] https://codepen.io/impressivewebs/pen/YMzMyj?editors=0011

DOM Scripting on Malformed or Invalid HTML

What happens when you are using JavaScript to access DOM elements that are part of mangled or invalid HTML? Well, due to the prevalence of good text editors and IDEs, and the fact that most developers will validate and lint their code, this usually will not be a problem.

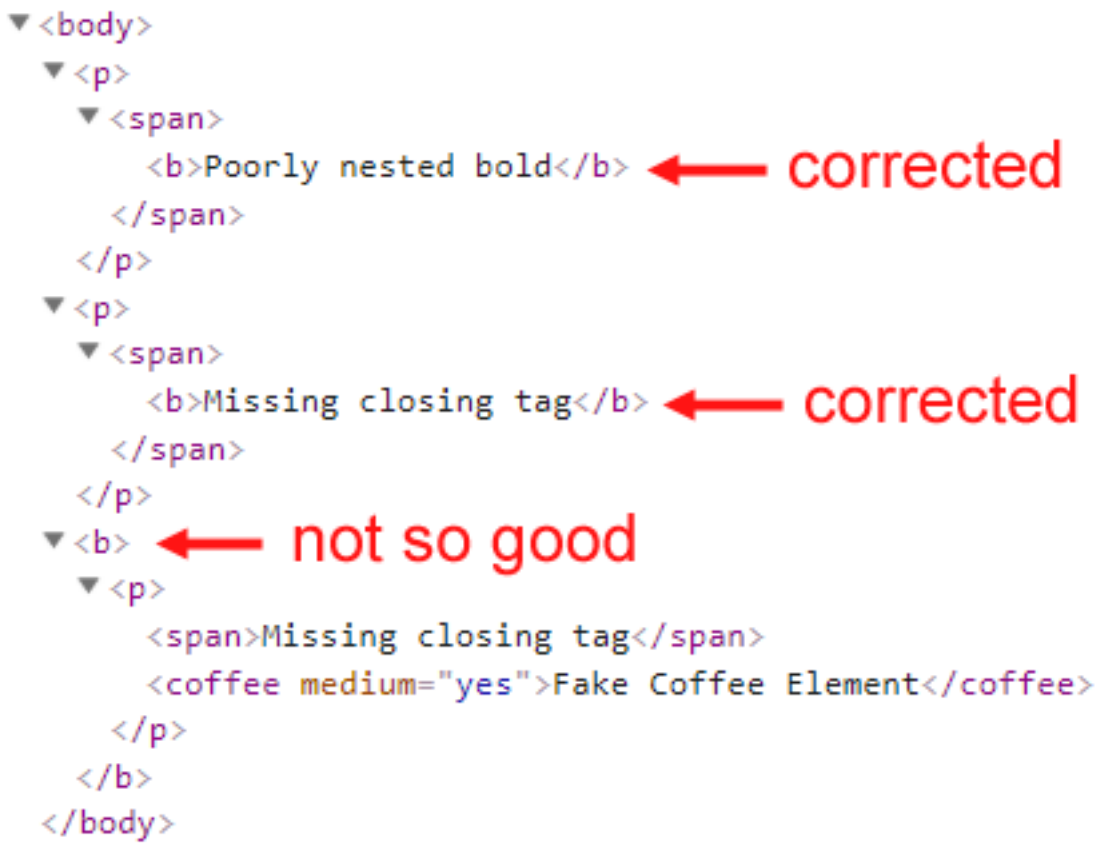
But if you're curious, let's consider the following HTML:

```
<p><span><b>Poorly nested bold</span></p></b>
<p><span><b>Missing closing tag</span></p>
<p><span>Missing closing tag</span>
<coffee medium="yes">Fake Coffee Element</coffee>
```

If you look closely, this example has five issues:

- A really late closed `<b>` element
- A missing `</b>` closing tag
- A missing closing paragraph tag (which is not invalid, but might have an undesired result)
- A made up tag name (`<coffee></coffee>`)
- A made up attribute name (`medium`)

First, as you can see from the following screenshot of the rendered DOM, the browser will attempt to correct most of these problems to a reasonable degree, which makes the elements accessible as expected:



The browser will correct malformed HTML as well as possible

You can see that the only major problem is the fact that a third `<b>` element was added in an attempt to correct the issue. Now let's try to access these with some DOM scripting:

```
let mybold = document.querySelectorAll('b'),
    myCoffee = document.getElementsByTagName('coffee');

console.log(mybold.length); // 3
console.log(mybold[0].tagName); // "B"
console.log(myCoffee[0].tagName); // "COFFEE"
console.log(myCoffee[0].nodeType); // 1
console.log(myCoffee[0].innerText); // "Fake Coffee Element"
```

Again, the only issue is the fact that there are three "b" elements. Even the "coffee" element is accessible, including its fake attribute. Here's a demo[90] that has the above code plus some extra code that manipulates the attribute.

So the DOM is pretty forgiving when it comes to mistyped or made-up HTML, though I wouldn't recommend any of this in a real project.

[90] https://codepen.io/impressivewebs/pen/OGJGQJ?editors=0011

A Little-known Fact About Hoisting

At some point when reading about scope in JavaScript, you may have come across the term *hoisting*. It's often used in the context of variable or function definitions.

For example, consider the following code:

```
doSomething();

for (i = 0; i <= 10; i++) {
  console.log(i);
}

function doSomething() {
  console.log('function executed');
}
```

Although the `doSomething()` function is called before it's defined, the code still works. In fact, the `for` loop isn't started until the JavaScript engine figures out that `doSomething()` is defined and can execute the call.

In many blog posts and books over the years, this is what has been referred to as "hoisting". For example, Doug Crockford's popular book *JavaScript: The Good Parts*, says:

> "function statements are subject to hoisting. This means that regardless of where a function is placed, it is moved to the top of the scope in which it is defined."

Similarly, David Flanagan's *JavaScript: The Definitive Guide* says:

> "This feature ... is informally known as hoisting: JavaScript code behaves as if all variable declarations in a function ... are 'hoisted' to the top of the function."

Flanagan gets it more right than Crockford because he correctly points out that "hoisting" is an informal term and uses the phrase "as if" to indicate that this isn't what's actually happening.

For all practical purposes, most explanations about hoisting are fine and serve their purpose in helping understand how things work. But here's what the authors of Secrets of the JavaScript Ninja[91] say about hoisting:

[91] https://webtoolsweekly.com/reading/?view=2JqYwea

> "Variables and function declarations are technically not 'moved' anywhere. They're visited and registered in lexical environments before any code is executed."

This is what Flanagan was more or less alluding to but which the above quote makes more explicit. That latter book has a fairly extensive section within one of the chapters that discusses lexical environments, so be sure to grab a copy if you want to expand your understanding of that subject.

Getting the Value of a Selected Radio Button

If you're working with an HTML form that includes multiple groups of radio buttons, there's a quick way to access the selected (or "checked") value of a particular group of radio buttons with JavaScript.

Let's assume I have a page with three sets of radio buttons (which are distinguished using the `name` property in the HTML). Here's some DOM scripting that will give me the string value of each selected radio option.

```
let myForm = document.forms[0],
    myRadio1 = myForm.elements['group1'],
    myRadio2 = myForm.elements['group2'],
    myRadio3 = myForm.elements['group3'],
    op = document.querySelector('output'),
    btn = document.querySelector('button');

myRadio3[2].checked = false;

btn.addEventListener('click', function () {
  op.innerHTML = `<p>1st group selected is: ${myRadio1.value}</p>
                  <p>2nd group selected is: ${myRadio2.value}</p>
                  <p>3rd group selected is: ${myRadio3.value}</p>`;
}, false);
```

You can view a full demo, including the HTML, in this CodePen[92].

Notice after I grab each group of radio buttons, I'm using the `value` property to find the `checked` option. In other words, each radio group has a `value` property that's equal to the value of the selected option.

Additionally, in this case I'm also setting the selected radio button in the third set to `false` (i.e. "unchecked"). In that instance, the third group would return an empty string because by that point in the code the third group has no selection (showing that the collection is a "live" collection, not a static one).

And as a side point, the `elements` property is available on a form object to help you get access to a radio group or an individual form input. This property only retrieves non-image form elements and will helpfully ignore all other elements.

[92] https://codepen.io/impressivewebs/pen/bJGyeq?editors=0010

The querySelector() Single-Byte Trick

Here's an interesting little tidbit that I found on Twitter[93] from someone named Gabriel Lima: If you use `querySelector()` or `querySelectorAll()` to access an element via an attribute selector, you can leave off the closing square bracket and it will still work.

Take the following HTML for example:

```
<section class="example"></section>
```

I can access that element using a line of JavaScript that uses the attribute selector:

```
console.log(document.querySelector('section[class').className); // "example"
```

Here I'm just logging out the class name but, as you can see, the attribute selector inside the `querySelector()` call is missing the closing square bracket. Here is a CodePen demo[94] so you can try it out. This works in all browsers I tested in but, according to Gabriel, isn't working in Safari (though I didn't test that).

This would seem to be a bug, but according to a recent answer in this Stack Overflow thread[95] and a Chrome bug report[96], this is expected behavior. One answer in the bug report thread (marked as WontFix) says in part:

> "The fact that the square-bracket block was unclosed is lost. You can think of this as the parser 'auto-closing all unclosed blocks'."

So if you're interested in saving a single character in a line of JavaScript that uses an attribute selector, you've struck gold. Well, maybe code golf would be the only place this might come in handy – though I have a feeling using an attribute selector in a `querySelector()` call is not something you'd see often in a code golf competition.

[93] https://twitter.com/gabriHellmateus/status/1018443795564818434
[94] https://codepen.io/impressivewebs/pen/pBomWV?editors=1011
[95] https://stackoverflow.com/questions/29120822/how-why-is-attribute-string-a-valid-queryselector-js-bug
[96] https://bugs.chromium.org/p/chromium/issues/detail?id=460399

Triggering a Click Event

This Stack Overflow thread[97] discusses different ways to trigger a click event on an element even when the click doesn't actually occur via user input. If you've used jQuery, then you might be familiar with `trigger()`, which allows you to do this.

The thread, however, points out an old DOM method called `click()` that can be used similarly. What's interesting is that the old spec[98] (which is the only one MDN's brief article references for this method) says this method will:

> "Simulate a mouse-click. For INPUT elements whose type attribute has one of the following values: 'button', 'checkbox', 'radio', 'reset', or 'submit'."

But despite that this seemed to be originally limited to clicks on certain types of elements, you can apparently do this on any element. For example, assuming this HTML:

```
<div class="btn">Toggle Background</div>
```

You can do something like this:

```
let btn = document.querySelector('.btn');

btn.addEventListener('click', function () {
  document.body.classList.toggle('bg');
}, false);

btn.click();
```

Try it on CodePen[99]

Here I'm triggering the click on a `div` element, and it works. And this is in agreement with the current HTML spec[100] (not the DOM spec), which includes `click()`. There it runs through some specific steps the browser should take when this method is used. The `click()` method is also referenced in the DOM spec, but only as a side point while discussing the `isTrusted` attribute[101].

So if you ever need to simulate a click on some element, this is an easy and mostly cross-browser way to accomplish it.

[97] https://stackoverflow.com/questions/2381572/how-can-i-trigger-a-javascript-event-click
[98] https://www.w3.org/TR/DOM-Level-2-HTML/html.html#ID-2651361
[99] https://codepen.io/impressivewebs/pen/VNwORj?editors=0010
[100] https://html.spec.whatwg.org/multipage/interaction.html#dom-click
[101] https://dom.spec.whatwg.org/#dom-event-istrusted

The focusin and focusout Events

As you might be aware, when an element on a web page receives focus, the `focus` event fires on that element. But the `focus` event doesn't bubble. That is, it isn't detected up the DOM tree like a `click` event.

There is, however, a useful pair of events you'll want to make a mental note of should you want a `focus` event that bubbles: The `focusin` and `focusout` events, both of which have excellent browser support.

The spec[102] describes `focusin`:

> "A user agent MUST dispatch this event when an event target is about to receive focus."

Or in the case of `focusout`, "is about to lose focus." This is accomplished via event bubbling, which doesn't happen with the `focus` event.

When any element is focused, the `focusin` event will bubble up as high as it can go in the DOM tree. It thus becomes useful when you listen for it on an element that contains multiple focusable elements.

Let's consider the following HTML:

```
<div class="one" tabindex="0">
  <button>Button 1</button>
  <button>Button 2</button>
</div>

<div class="two" tabindex="0">
  <button>Button 3</button>
  <button>Button 4</button>
</div>
```

Notice I have two buttons inside each `div` element. I've also added a `tabindex` value to each `div` to make each `div` focusable too. I don't need to do this, but it's an option.

I want to detect focus on any element inside `div` "one" and I also want to detect when an element inside `div` "one" loses focus. Assuming "one" is a reference to that `div`, I can use these as with any events:

[102] https://www.w3.org/TR/uievents/#focusin

```
one.addEventListener('focusin', function (e) {
  // do something here...
}, false);

one.addEventListener('focusout', function (e) {
  // do something here...
}, false);
```

With that in place, if either button inside the first `div` receives focus, or even if the `div` itself receives focus, the `focusin` event will fire. And if an element inside div "one" loses focus, the `focusout` event fires. As with `focusin`, `focusout` bubbles, which again is what allows this to work.

View a full demo here[103]

Try clicking any of the buttons or using the keyboard to move from one element to the next on the page and you'll see the output describing what's happening with each tab movement or click.

While the `focus` event is specific only to the element that receives focus, `focusin` and `focusout` can come in handy if you want to detect focus on any element in a group of elements.

[103] https://codepen.io/impressivewebs/pen/wZvLJx?editors=0010

Intro to ES6 Modules

For those of you who still haven't looked into using JavaScript modules, introduced in ES6 and now with strong browser support, I'll introduce this feature here. Among other things, ES6 modules help with errors and problems that can occur due to naming collisions in the global scope. They can also keep your code clean, organized, and easier to maintain – especially on a large project in a team environment.

Let's look at a basic example to see how you can use ES6 modules in a JavaScript codebase running in the browser.

I'm going to have an HTML page that includes the following script at the bottom of the page:

```
<body>
  ...

  <script src="js/main.js" type="module"></script>
</body>
```

Notice the `type` attribute set to "module". This is required if I want this script to be able to use ES6's module-based features. This tells the browser that I want this file to load as a *module*, rather than a *script*.

In addition to `main.js`, I'm also going to create a `secondary.js` file in the same directory as my `main.js` file. But I'm not going to include that as a script in the HTML – the `secondary.js` file will hold code that I'm going to import into `main.js`.

Here is my `secondary.js` file:

```
export let color = 'blue';

export function add(n1, n2) {
  return n1 + n2;
}
```

In this file I'm exporting two specific parts of the code: the `color` variable and the `add()` function. With these export commands in place, I can now import them to my `main.js` file:

```
import { color, add } from '../js/secondary.js';

console.log(color); // "blue"
console.log(add(10, 20)); // 30
```

The `import` keyword and the curly braces specify the bindings, separated by commas, that I want to import. As you saw in the previous code block, I identified via the `export` keyword which ones were available inside that module. If there's something inside `secondary.js` that's not exported, it won't be available via `import` in another module.

There's lots more to discuss on this topic, but this should give you a quick start to use this elegant feature introduced in ES6.

More on ES6 Modules

Continuing on the topic of ES6 modules, here are a few more notes and tips to keep in mind when using this feature in your projects.

Normally, the value of `this` at the top level of any script in the browser will be equal to the `Window` object. Inside of a module, however, the value of `this` is `undefined`:

```
import { one, two } from '../js/module.js';
console.log(this); // undefined
```

Another thing you'll want to take note of is that HTML-style comments are not allowed inside modules. HTML-style comments inside JavaSript is a concept I discussed in a previous tip so check that out if you want to know why. But just know you can't use them in modules.

Another thing that separates modules from scripts is the fact that all modules run in strict mode with no way to opt out. Also, when functions and variables are defined inside a module, they are available only in the top-level scope of that module. Each module has its own encapsulated scope and anything defined in a module won't pollute the global scope like top-level variables and functions in regular scripts. Only when a module's variables are exported will these be available outside the module.

Another tip: You don't have to export something at the time it's defined. Consider the following code:

```
function add(n1, n2) {
  return n1 + n2;
}

export { add };
```

In this module, I'm not exporting the `add()` function when it's defined, but instead I export it as a reference. Notice the curly braces around the reference to the `add()` function after the `export` keyword.

The last point I'll mention here is that when you import a function or variable from a module, it will act as if it was defined using `const`. For example, if my module file contains the following:

```
export let color = 'yellow';
```

Notice what happens if I try to redefine the `color` variable after it's imported:

```
import { color } from '../js/secondary.js';

console.log(color); // yellow
color = 'pink';

// Chrome: Uncaught TypeError: Assignment to constant variable.
// Firefox: TypeError: "color" is read-only
```

In the code comment, I've printed the errors that Chrome and Firefox display when I attempted to do this with the `color` variable.

Again, even though the `color` variable was defined using `let`, it's treated as if it was defined using `const`, because it's imported from a module. Imported functions and classes will likewise be read-only.

Importing a Full Module

As previous tips have shown, ES6 modules are a great way to isolate specific bindings (i.e. variables, functions, or classes) and import them as needed into the current module.

But what if you want to import an entire module, with all its variables and functions? There's a way to achieve this all in one shot. Let's say my `secondary.js` file, which holds my exports, looks like this:

```
export function add(n1, n2) {
  return n1 + n2;
}

export let amt1 = 10,
           amt2 = 25,
           amt3 = 30;
```

Here I'm exporting an `add()` function along with three 'amt' variables. I'm going to import this entire module so I can use these bindings inside `main.js`:

```
import * as mod from '../js/secondary.js';

console.log(mod.add(3, 4)); // 7
console.log(mod.add(mod.amt1, mod.amt2)); // 35
console.log(mod.amt3); // 30
```

On the first line is where the import takes place. I'm using the asterisk character to indicate that I want all bindings imported. I do this by loading them into an object that I've named `mod`.

Once this is done, I can access the `add()` function and the three variables as properties of the `mod` object. I can't access any of those bindings without referencing them as properties of `mod`. And to reiterate: "mod" is just a name I chose. This could be any name, but obviously you'd want to avoid using a reserved word (e.g. "continue" or "function" would both throw an error if you tried to use one of those as the object name).

As you can see, ES6 modules are extremely flexible. You can import an entire module or any individual bindings that are exported.

Renaming Module Exports

I've discussed ES6 modules in multiple tips, but there's one small detail I still didn't cover. When exporting or importing bindings from modules, you have the choice to use different names for the bindings than originally used when they are defined.

For example, let's assume my `secondary.js` file (the one I'm using for exports to `main.js`) is exporting two bindings, a variable and a function. Here's what I can do when exporting:

```
function add(n1, n2) {
  return n1 + n2;
}

let mynum = 10;

export { add as addFunc };
export { mynum as myNumber };
```

Notice two things here. First, I'm not exporting when the function and variable are defined; I'm exporting references to those bindings, which I discussed previously.

But more significantly, notice the "as" keyword used inside the curly braces, which is what allows me to give these exported items new names. This is similar syntax to when I previously namespaced my references by attaching them to a specific object on export (see previous tip).

Now when I import them into `main.js`, I would have to use the new names:

```
import { addFunc, myNumber } from '../js/secondary.js';

console.log(addFunc(5, myNumber)); // 15
```

The original names for these bindings ("add" and "mynum") would not work. They have to be referenced using the exported names.

And one final note here: When I exported the `add()` function and the `mynum` variable with their new names, I could have done this with a single line, like this:

```
export { add as addFunc, mynum as myNumber };
```

That's similar to what I'm doing in `main.js` when I import the items. Something to keep in mind if you're exporting multiple references.

JavaScript Errors and the Error Object

I recently came across this page on MDN[104] that is a JavaScript error reference. Amazingly, that page lists more than 70 different errors that you could see in your JavaScript apps. I'd guess that I've come across maybe 20 of those over the years.

The errors themselves might not be that interesting or useful at first glance, but you'll notice that each error in that list links to a specific page that discusses the error in detail. From there, you can get some valuable info on specific JavaScript features that you may not be familiar with.

Another thing that page mentions and links to is the global `Error` object. An `Error` object can be created via the `Error` constructor. An instance of the `Error` object is thrown every time you see a JavaScript error in your browser or developer tools console.

Whatever the error is, you can suppress the error from appearing to the user by means of a `try...catch` statement. Inside the `catch` block, you have the option to read the exact error message:

```
try {
  const myVar = 5;
  myVar = 10;
} catch(err) {
  console.log(err.name + ': ' + err.message);
}

// Chrome: "TypeError: Assignment to constant variable."
// Firefox: "TypeError: invalid assignment to const `myVar'"
// Edge "TypeError: Assignment to const"
```

Try it on CodePen[105]

As you can see, the `Error` object exposes a `name` property and `message` property that allow you to read the exact error message that the browser throws. In this case, it's the fact that I'm trying to change a constant. Each browser expresses the error a little differently. So any one of the 70+ errors can be caught this way and the message logged.

I suppose something like this might be useful because although you're keeping errors from showing up to users, you're able to log those errors somewhere for debugging purposes which might help you understand where your code is throwing exceptions without you realizing it.

[104] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors
[105] https://codepen.io/impressivewebs/pen/BEaggb?editors=0011

Some Facts on ES6 Arrow Functions

If you still haven't used ES6's arrow functions much, then you might not be familiar with some of this feature's quirks and differences from regular functions. Besides the syntax, there are some things to keep in mind when using arrow functions.

First, unlike regular functions, the `arguments` object is not available inside the body of an arrow function:

```
let myFunc = (a, b) => {
  console.log(arguments);
  return a + b;
}

// Error: arguments is not defined
console.log(myFunc(1, 2));
```

Second, you might be aware that if your arrow function has only a single line, the return value is implied without the use of the `return` keyword. But if you want to return an object literal, you have to wrap the literal in parentheses:

```
// This won't work
let myFunc = a => { one: a, two: 'b' };
console.log(myFunc(6));
```

In the above code I'm attempting to return the object literal, but the browser will throw an error because it thinks the first curly brace is the beginning of the function body. Here's the corrected version with the `return` value shown:

```
// This works
let myFunc = a => ({ one: a, two: 'b' });
console.log(myFunc(6));

/* Return value:
[object Object] {
  one: 6,
  two: "b"
}
*/
```

Try it on CodePen[106]

The final quirk I'll mention here, which is intended to reduce confusion surrounding the use of the `this` keyword, is the fact that `this` is not defined inside an arrow function. This might trip you up if you are accustomed to referencing `this` inside an event handler. For example:

```
btn.addEventListener('click', function () {
  console.log(this); // references the button
}, false);

btn.addEventListener('click', () => {
  console.log(this); // undefined
}, false);
```

I do this all the time, so it's certainly something that will likely prevent me from using arrow functions with event handlers. But if you're intending to do this sort of thing, just know that you can't use `this` inside an arrow function as part of an `addEventListener()` call if you expect the value of `this` to be the object on which the event was triggered.

[106] https://codepen.io/impressivewebs/pen/VNwoaB?editors=0011

Default Exports in ES6 Modules

One more thing it's good to know about ES6 modules is the fact that you are permitted to define a default export for each module. This pattern is taken from pre-ES6 module systems like CommonJS.

Here's one way a module can define its default value:

```
export default function (n1, n2) {
  return n1 + n2;
}
```

Here I'm exporting a function as the default. I could also do this by naming my function and then referencing it elsewhere in the file as the default:

```
function addFunc (n1, n2) {
  return n1 + n2;
}

export default addFunc;
```

So in my `main.js` file I can import this function using any name I want:

```
import myAddFunc from '../js/secondary.js';
console.log(myAddFunc(5, 7)); // 12
```

Notice I've named the function `myAddFunc` even though it's exported as `addFunc`. This is permitted because it's the default.

A module can have only one default value and it can also export non-default bindings. For example, if my `secondary.js` file exported the `addFunc()` function as the default along with two other variables, I could import them all like this (notice the use of the comma and the curly braces):

```
import myAddFunc, { num1, num2 } from '../js/secondary.js';
console.log(myAddFunc(num1, num2));
```

When it comes to best practices for use of defaults, I feel there's some disagreement in the community about what's best. Nicolás Bevacqua recommends[107] their use exclusively, using a single default at

[107] https://ponyfoo.com/articles/es6-modules-in-depth#best-practices-and-export

the bottom of your module to basically export your entire API. But Ben McCormick says[108] named exports are better simply because naming stuff is better.

If you've never used pre-ES6 modules, you might also find this 2015 article by Kent C. Dodds[109] useful because he clears up much of the confusion around the purpose of modules, which includes a discussion of default values.

[108] https://benmccormick.org/2018/06/05/es6-the-bad-parts/
[109] https://blog.kentcdodds.com/misunderstanding-es6-modules-upgrading-babel-tears-and-a-solution-ad2d5ab93ce0

An Intro to the Page Visibility API

The Page Visibility API is one that I had forgotten about until I just stumbled on it again. This simple set of features allows developers to handle specific actions when a page becomes visible or hidden.

A page is considered "hidden" when it is no longer "visible" to the user (e.g. user has opened a new tab page, or the entire browser has been minimized). Here's a simple way to log the current page's status whenever the `visibiiltychange` event is fired:

```
let d = document;

d.addEventListener('visibilitychange' , function () {
  console.log(d.hidden);
}, false);
```

Try it on CodePen[110]

You can test it using the demo by simply visiting the page, then switching to another tab, and switching back. The log will update each time you view or stop viewing the page.

Use cases for this API include pretty much any situation where you want some kind of feature on the page to pause or stop running in order to save resources. This could be an auto-running carousel that stops running, a video that stops playing, audio that pauses, animations that stop running, and so on.

As you've probably noticed, many browsers will do this sort of thing automatically. For example, `requestAnimationFrame()` and `setTimeOut()` call backs will stop when a page is "hidden". Audio, however, keeps playing since it's considered a desired feature (which makes sense). You can mimic this behavior with the Page Visibility API.

One thing I'd like to point out as a best practice here is with regards to video. MDN's article[111] on this subject links to this video page[112], which demonstrates the API. Try playing the video, then moving away from the page and back again. Notice the video will start and stop based on page visibility.

The only problem is if you manually pause the video, then leave the page and come back, the video starts playing again. This is not desired behavior. If the user has manually paused the video, the page should have a means to detect this and keep it paused until it's manually played again – regardless of the page's visibility status. Of course, that page is just a simple demo, so it's not a big deal in that case. But if you're going to use this API, be sure to implement it in an intuitive way that takes the user's potential actions and desires into consideration.

[110] https://codepen.io/impressivewebs/pen/pBoMdN?editors=0011
[111] https://developer.mozilla.org/en-US/docs/Web/API/Page_Visibility_API
[112] http://daniemon.com/tech/webapps/page-visibility/

Using document.domain for Cross-origin Requests

If I have a page on impressivewebs.com that loads an iframe with its `src` set to `testing.impressivewebs.com`, I can't access the iframe's DOM from the parent page. This is for security reasons. Even though both resources are on the same domain, the subdomain is considered a different origin.

There are various ways to overcome cross-origin restrictions, but one easy way to do this, assuming you have access to both origins, is using the `document.domain` property in JavaScript.

Let's say my primary page is loading an iframe:

```
<iframe src="https://testing.impressivewebs.com/"></iframe>
```

Below the iframe, I'm loading the following script:

```
document.domain = 'impressivewebs.com';
let myFrame = document.getElementsByTagName('iframe')[0];

window.addEventListener('load', function() {
  console.log(myFrame.contentWindow.document.body.tagName);
}, false);
```

Also, inside the iframe's page, I'm doing this:

```
document.domain = 'impressivewebs.com';
console.log(parent.document.body.id);
```

You can try it out at this demo page[113]. Open up the console and you'll see the two logs that are printed.

Notice two things happening in the each document:

- The `document.domain` property is set to `impressivewebs.com`
- The DOM of the other document is accessible

[113] https://www.impressivewebs.com/demo-files/document-domain/

Neither document would be able to access the other's DOM unless the `document.domain` property is set to the same value (or if you use some other method to overcome the restriction).

The `document.domain` property can only be set to the current domain and can't be changed to another resource (more info on MDN[114]). This is just a quick way to relax these restrictions. You can read more about cross-orgin resource sharing in this MDN article covering CORS[115], which has long been a standard.

[114] https://developer.mozilla.org/en-US/docs/Web/API/Document/domain
[115] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

What is JSON? An Introduction and Guide for Beginners

If you're new to web development and have some basic knowledge of HTML, CSS, and possibly a little bit of JavaScript, a practical area in which to expand your front-end skills is JSON.

But even if you already have a basic understanding of what JSON is and have used it in some of your projects, there might be a few things you weren't aware of. So in this JSON tutorial and guide, I'm attempting to provide a fairly comprehensive discussion of JSON, its history, and its usefulness. I'll close with a list of some practical JSON tools that might come in handy in future projects.

JSON Defined

JSON stands for JavaScript Object Notation and it's a data format. That is, it's a way to hold bits of information, similar to a database. Although JSON originated outside the ECMAScript specification, it is now closely related to JavaScript with the spec now including a JSON object[116] and many developers incorporating it as a quasi-subset of the language.

Here's an example of JSON syntax:

```
{
  "species": "Dog",
  "breed": "Labrador Retriever",
  "color": "Yellow",
  "age": 6
}
```

As you can see, JSON is a data format that consists of name/value pairs (AKA key/value pairs) in the form of strings. The name/value pairs are separated by a colon and each pair is separated by a comma.

Although JSON originated and is usually most closely associated with JavaScript, many (if not all?) programming languages can generate and read the JSON format. The fact that it's universal in this way is what has made it so popular as a way to store, parse, read, and share information in web apps and web services.

[116] https://tc39.github.io/ecma262/#sec-json-object

A Brief History of JSON

As mentioned, JSON originated in association with JavaScript and client-side scripting. Douglas Crockford is the inventor of JSON and he maintains the official JSON.org website[117] where it's discussed in great technical detail.

The earliest official specification for JSON is the ECMA-404 standard[118] from 2013. But JSON goes back much further than that.

The JSON website was officially launched in 2002[119] (where it was originally redirecting to Crockford's website). Yahoo and Google began using JSON as early as 2005 and 2006, respectively, and JSON took off in the years following. Wikipedia's article has more details on the history[120], if you want to go into that a little more.

How is JSON Different from a JavaScript Object?

As you can tell from the name, JSON is more or less a JavaScript object. But there are differences. First of all, as the spec explains, "JSON is a text format that facilitates structured data interchange between all programming languages." So it's universal, it's not just a JavaScript thing. In fact, it's not part of JavaScript at all, it's just derived from the way JavaScript objects are written.

In terms of the syntax itself, there are a few major differences. First, all names (keys) are represented as strings (i.e. they need to be inside quotes). So the following is not valid JSON:

```
// not valid JSON, but valid as JS object
{
  foo: "bar"
}
```

The correct way to write that as JSON is:

```
// valid JSON
{
  "foo": "bar"
}
```

Note the use of quotes around the name, or key. Also, note the use of double quotes. While single quotes are fine for JavaScript objects, JSON requires double quotes.

Another area where JSON differs from JavaScript objects is the types of values that JSON can store. According to the spec, a JSON value can be one of the following:

[117] http://json.org/
[118] http://www.ecma-international.org/flat/publications/files/ECMA-ST/ECMA-404.pdf
[119] http://web.archive.org/web/20030228034147/http://www.crockford.com/JSON/index.html
[120] https://en.wikipedia.org/wiki/JSON#History

- Object
- Array
- Number
- String
- true
- false
- null

That's quite similar to the stuff you can put into JavaScript objects, but since JSON is represented as text, that means you can't feed it stuff like functions or dynamic date values using `Date()`. So there are no methods or other functionality in JSON, it's just text. And this is a good thing because that's what makes it a universal data interchange format.

It's also important to note that an entire bit of JSON is technically itself a single valid JSON object, and the object type itself is a way to nest JSON objects as values, similar to how this is done with objects in JavaScript. For example, here is a chunk of JSON with an object nested as one of the values:

```
// Valid JSON
{
  "species": "Dog",
  "breed": "Labrador Retriever",
  "age": 6,
  "traits": {
    "eyeColor": "brown",
    "coatColor": "yellow",
    "weight": "137lbs"
  }
}
```

Here the root JSON object has four key/value pairs ("species", "breed", "age", and "traits"); but the fourth value ("traits") has three more key/value pairs nested as a single value (this is an example of the object value type). And as you probably guessed, you can nest objects like this infinitely. Of course, you should only do this to a reasonable degree.

By contrast, a JavaScript object with a nested object might look like this:

```
// JS object; not valid JSON
let myAnimal = {
  species: "dog",
  breed: "Labrador Retriever",
  age: 6,
  traits: {
    eyeColor: "brown",
    coatColor: "yellow",
    weight: "137lbs"
  }
}
```

From that, you should be able to see the differences between JSON and JavaScript objects. The keys are not quoted in the JavaScript version, and in order to make the object useful in JavaScript, the whole thing is placed as a value of a variable.

How Should JSON be Stored?

As you probably guessed, since JSON is basically just text, you can pretty much store it however you want. You can store it in a database, in a separate text file, in client storage (like cookies or localStorage) or even using its own file format using the .json file extension (which is basically just a text file with a .json extension). I don't know of any limitation on how you can store JSON, but if you do, feel free to mention it in the comments.

Once you have JSON content stored, it's just a matter of retrieving it and parsing it appropriately. There are different ways to retrieve and parse JSON in different languages, but this being a front-end development blog, I'm going to discuss how this can be done using JavaScript.

Using JSON.stringify()

As mentioned, depending on what language you're using to read the JSON data, you'll have different tools and built-in functions at your disposal to get access to the JSON data in a convenient way. JavaScript, usefully, has two built-in methods that are part of the ECMAScript specification that allow you to achieve two specific tasks.

Let's say your application is building or populating JSON data in some way. In order to save the data somewhere, it needs to be converted to a valid JSON string. You can do this using `JSON.stringify()`:

```
let myJSON = {
  species: "Dog",
  breed: "Labrador Retriever",
  age: 6,
  traits: {
    eyeColor: "brown",
    coatColor: "yellow",
    weight: "137lbs"
  }
};

let myNewJSON = JSON.stringify(myJSON, null, '\t');

/* output of myNewJSON:
{
  "species": "Dog",
  "breed": "Labrador Retriever",
  "age": 6,
  "traits": {
    "eyeColor": "brown",
    "coatColor": "yellow",
    "weight": "137lbs"
  }
}
*/
```

View on CodePen[121]

The `JSON.stringify()` method takes one mandatory parameter (the JSON you want to convert to a string) and two optional arguments. The first optional argument is a replacer function that you can use to filter out some of the name/value pairs that you don't want to include. I didn't filter anything out in my example, so I set the replacer function as `null`. I wouldn't normally use `null`, but I wanted to use the third parameter and it's required to have the second parameter if you want to use the third.

The third parameter is referred to as a space value. This parameter helps you format the stringified JSON so it's more readable (using indenting, line breaks, etc). In my case, I used a tab character (represented by `\t`). Whatever character is used will be inserted once for each indent level. If you use a number for the third argument, this will represent the number of spaces to use for each indent.

[121] https://codepen.io/impressivewebs/pen/dLyxwa?editors=0011

Using JSON.parse()

By contrast, if you receive JSON in string format, you can do the reverse using the `JSON.parse()` method:

```
let myJSON = '{"species":"Dog","breed":"Labrador Retriever","age":6,"traits":{"eyeCo\
lor":"brown","coatColor":"yellow","weight":"137lbs"}}';

let myNewJSON = JSON.parse(myJSON);

// logs a JavaScript object, not a string
console.log(myNewJSON);
```

View on CodePen[122]

In the example above, I'm creating a JSON string manually, which I store in a variable. This is just for demonstration purposes, but in a real-world scenario the JSON may be coming from a third-party source or a separate JSON file.

As you can see, the `JSON.parse()` method converts the JSON string into a proper object that I can then manipulate and otherwise deal with in my JavaScript. The string passed into `JSON.parse()` is the only mandatory argument, but you have can also add an optional second argument, which is referred to as a reviver function. Here's an example that assumes the same string is being used as in the previous code:

```
let myNewJSON = JSON.parse(myJSON, function (name, value) {
  if (name === "species") {
    value = "Cat";
  }
  return value;
});
```

View on CodePen[123]

That's just a simple example that alters one of the values for one of the names. For more info on the use of the reviver function, see MDN[124] or this article on Dynamic Web Coding[125].

[122] https://codepen.io/impressivewebs/pen/vMYoPq?editors=0011
[123] https://codepen.io/impressivewebs/pen/vMEBYB?editors=0011
[124] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse#Using_the_reviver_parameter
[125] http://www.dyn-web.com/tutorials/php-js/json/reviver.php

Using JavaScript to Manipulate JSON

As you might have figured out, once you've converted string-based JSON data into a JavaScript object, you then have access to that object in the same way you would any JavaScript object. For those already familiar with using JavaScript objects, this will be familiar.

Let's assume you've parsed a JSON string and the `myNewJSON` variable holds the result, as shown in the previous section. You can now use what's often called dot notation[126] to access the different parts of the JSON data:

```
console.log(myNewJSON.species); // "Dog"
console.log(myNewJSON.breed); // "Labrador Retriever"
console.log(myNewJSON.age); // 6
```

I can also access the nested object and the values inside the nested object by going deeper with dot notation:

```
console.log(myNewJSON.traits);
/*
[object Object] {
  coatColor: "yellow",
  eyeColor: "brown",
  weight: "137lbs"
}
*/

console.log(myNewJSON.traits.coatColor); // "yellow"
console.log(myNewJSON.traits.eyeColor); // "brown"
console.log(myNewJSON.traits.weight); // "137lbs"
```

View on CodePen[127]

You can do whatever you want with this data. You can add new values, change current values, or even delete entire name/value pairs:

[126] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Property_accessors
[127] https://codepen.io/impressivewebs/pen/xebKwm?editors=0011

```
myNewJSON.age = 7;
delete myNewJSON.traits.weight;
myNewJSON.traits.cuddly = true;

console.log(myNewJSON);

/*
[object Object] {
  age: 7,
  species: "Dog",
  breed: "Labrador Retriever",
  traits: [object Object] {
    coatColor: "yellow",
    cuddly: true,
    eyeColor: "brown"
  }
}
*/
```

View on CodePen[128]

In the above code, I changed the `age` value, deleted the `weight` property of the `traits` object, and added a new `cuddly` property to the `traits` object.

I can then use `JSON.stringify()` to convert this new data back into the correct format so I can store it wherever I want as valid JSON data.

Is JSON Better Than XML?

Although XML is certainly not a defunct data format, JSON has largely taken over in popularity. Douglas Crockford explains many of the advantages of JSON over XML[129], and I'll quote part of his document here:

> XML is not well suited to data-interchange, much as a wrench is not well-suited to driving nails. It carries a lot of baggage, and it doesn't match the data model of most programming languages. When most programmers saw XML for the first time, they were shocked at how ugly and inefficient it was. It turns out that that first reaction was the correct one. There is another text notation that has all of the advantages of XML, but is much better suited to data-interchange. That notation is JavaScript Object Notation (JSON)

He goes on to discuss the claimed advantages and benefits of XML while showing why JSON equals and betters those advantages.

[128] https://codepen.io/impressivewebs/pen/dLPbXx?editors=0011
[129] http://www.json.org/xml.html

What is JSONP?

JSONP (meaning "JSON with Padding") is a solution that attempts to overcome the cross-origin restrictions and limitations that come with fetching data from another server. You may have run into this when attempting to use Ajax to request data from an external source. Simply put, you can't do this, for security reasons.

As a solution to this problem, Bob Ippolito first proposed JSONP back in 2005[130] shortly after George Jempty proposed something similar called JSON++[131].

JSONP takes advantage of the fact that `<script>` elements are not bound by cross-origin limitations. That's why we're able to link to scripts on remote CDNs[132] with no problems.

Here's an example that accesses JSONP data using plain JavaScript:

```
function doJSONP(result) {
  console.log(result.data);
}

let script = document.createElement('script');
script.src = 'https://api.github.com/users/impressivewebs?callback=doJSONP'

document.getElementsByTagName('body')[0].appendChild(script);
```

View on CodePen[133]

This code creates a `<script>` element, adds a specified URL as the `src` attribute, then appends the script to the document. In this case, I'm accessing a specific user's data on GitHub (my own data, actually), using GitHub's API. If you open up the demo[134] you'll see the display of the JSON data in the console.

If you're using jQuery, you can make a similar request using jQuery's `$.getJSON()` method:

```
$.getJSON('https://api.github.com/users/impressivewebs', function (result) {
  console.log(result);
});
```

View on CodePen[135]

Note that in this case, the data I want is exactly what's returned. In the vanilla JavaScript version, however, I have to drill down into the `data` property to get the JSON data I want.

[130] http://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/
[131] https://web.archive.org/web/20060212113746/http://htmatters.net/htm/1/2005/07/evaling-JSON.cfm
[132] https://cdnjs.com/
[133] https://codepen.io/impressivewebs/pen/JVoPbV?editors=0011
[134] https://codepen.io/impressivewebs/pen/JVoPbV?editors=0011
[135] https://codepen.io/impressivewebs/pen/gybYWM?editors=0011

In either case, after getting the resulting data, I can use the dot notation discussed previously to access the different name/value pairs. For example, I can get my follower count on GitHub, my Avatar URL, number of public repos, and lots more.

How Does JSONP Work?

To understand how JSONP works, you first have to understand that a plain JSON file cannot be used in this way unless the server sets up the data to respond correctly. In other words, you can't do this with any external .json file you find online. So how does the JSONP technique resolve this?

In the vanilla JavaScript example in the previous section, you probably noticed the GitHub API URL had a parameter attached to it: `callback=doJSONP`. This is a callback function that acts as a wrapper (or "padding") that allows access to the data, because normally when you inject JSON data using a `<script>` element, it will do nothing because the contents aren't stored in a variable that can be accessed in your JavaScript.

With JSONP, instead of this:

```
{
  "one": "value_1",
  "two": "value_2"
}
```

You're actually getting this:

```
callback({
  "one": "value_1",
  "two": "value_2"
});
```

Where `callback` would be the name of the callback. In my example it would be `doJSONP`.

So when you set up the JSON to be accessed on the remote server, you have to ensure that it's sent with the padding (i.e. the wrapper function). You can see an example in this CodePen[136] where I'm trying to access this game data from Major League Baseball[137] but the browser console says "Uncaught SyntaxError: Unexpected token :". This is because Major League Baseball hasn't set up that data to be JSONP accessible.

A detailed discussion of how to set up a JSONP service is beyond the scope of this article, but a simple solution is explained in this Stack Overflow thread[138].

There are some security concerns associated with JSONP, so be sure to do some research if you are planning to set something up. For more info on JSONP, see the links below:

[136] http://codepen.io/impressivewebs/pen/mWeNzN
[137] http://gd.mlb.com/components/game/mlb/year_2016/month_10/day_04/gid_2016_10_04_balmlb_tormlb_1/plays.json
[138] http://stackoverflow.com/questions/9519209/how-do-i-set-up-jsonp

- JSONP on Wikipedia[139]
- JSONP in layman's terms Question on Stack Overflow[140]
- Using JSONP Safely[141]

JSON Tools

There are lots of useful tools for doing different things with JSON data. Below I've compiled a list of many of the ones I've come across.

- JSONLint[142] – A validator for JSON data. This is a good tool for learning the basic syntax and how it differs from JavaScript object syntax.
- json.browse[143] – Online to that lets you browse, prettify, and manipulate JSON from an external source or JSON pasted in. A neat feature is the ability to filter the data based on a keyword.
- JSONedit[144] – A visual JSON builder that makes it simple to build complex JSON structures with different data types.
- JSON Schema[145] – A vocabulary that allows you to annotate and validate JSON documents.
- JSON API[146] – A specification for building APIs in JSON.
- CSVJSON[147] – CSV and SQL to JSON converter and beautifier.
- JSON Formatter[148] – Online tool to validate, beautify, minify, and convert JSON data.
- excelJSON[149] – Online tool to convert CSV or TSV to JSON, and JSON back to CSV or TSV.
- Myjson[150] – A simple JSON store for your web or mobile app.
- jsonbin.org[151] – A project from Remy Sharp, "a personal key/value JSON store as a service. Protected behind authentication and API key requests, data is stored as JSON and can be deep linked."
- Kinto[152] – A generic JSON document store with sharing and synchronisation capabilities.
- JSON Generator[153] – Online tool to generate random JSON data.
- Hjson[154] – A syntax extension for JSON, to help make it easier to read and edit for humans, before it's passed to a machine.
- JSON Selector Generator[155] – Paste in some JSON, then click on any part of the data and this tool will tell you the "selector" to use in your JavaScript to access that bit of data.

[139] https://en.wikipedia.org/wiki/JSONP
[140] http://stackoverflow.com/questions/3839966/can-anyone-explain-what-jsonp-is-in-layman-terms
[141] https://www.metaltoad.com/blog/using-jsonp-safely
[142] http://jsonlint.com/
[143] https://jsonbrowse.com/
[144] http://mb21.github.io/JSONedit/
[145] http://json-schema.org/
[146] http://jsonapi.org/
[147] http://www.csvjson.com/
[148] http://jsonformatter.org/
[149] http://exceljson.com/
[150] http://myjson.com/
[151] https://jsonbin.org/
[152] https://www.kinto-storage.org/
[153] http://json-generator.appspot.com/
[154] http://hjson.org/
[155] http://jsonselector.com/

Conclusion

If JSON was a relatively new concept to you, then I hope this post has given you a decent overview of what it's all about. JSON is a solid technology that's easy to use and powerful because of it being universal.

Using Default Parameters in ES6

I've recently begun doing more research into what's new in JavaScript, catching up on a lot of the new features and syntax improvements that have been included in ES6 (i.e. ES2015 and later).

You've likely heard about and started using the usual stuff: arrow functions, let and const, rest and spread operators, and so on. One feature, however, that caught my attention is the use of default parameters in functions, which is now an official ES6+ feature. This is the ability to have your functions initialize parameters with default values even if the function call doesn't include them.

The feature itself is pretty straightforward in its simplest form, but there are quite a few subtleties and gotchas that you'll want to note, which I'll try to make clear in this post with some code examples and demos.

Default Parameters in ES5 and Earlier

A function that automatically provides default values for undeclared parameters can be a beneficial safeguard for your programs, and this is nothing new.

Prior to ES6, you may have seen or used a pattern like this one:

```
function getInfo (name, year, color) {
  year = (typeof year !== 'undefined') ? year : 2018;
  color = (typeof color !== 'undefined') ? color : 'Blue';
  // remainder of the function...
}
```

In this instance, the `getInfo()` function has only one mandatory parameter: `name`. The `year` and `color` parameters are optional, so if they're not provided as arguments when `getInfo()` is called, they'll be assigned default values:

```
getInfo('Chevy', 1957, 'Green');
getInfo('Benz', 1965); // default for color is "Blue"
getInfo('Honda'); // defaults are 2018 and "Blue"
```

Try it on CodePen[156]

Without this kind of check and safeguard in place, any uninitiated parameters would default to a value of `undefined`, which is usually not desired.

You could also use a truthy/falsy pattern to check for parameters that don't have values:

[156] https://codepen.io/impressivewebs/pen/jZEYYP?editors=0011

```
function getInfo (name, year, color) {
  year = year || 2018;
  color = color || 'Blue';
  // remainder of the function...
}
```

But this may cause problems in some cases. In the above example, if you pass in a value of `0` for the year, the default 2018 will override it because `0` evaluates as falsy. In this specific example, it's unlikely you'd be concerned about that, but there are many cases where your app might want to accept a value of `0` as a valid number rather than a falsy value.

Try it on CodePen[157]

Of course, even with the `typeof` pattern, you may have to do further checks to have a truly bulletproof solution. For example, you might expect an optional callback function[158] as a parameter. In that case, checking against `undefined` alone wouldn't suffice. You'd also have to check if the passed-in value is a valid function.

So that's a bit of a summary covering how we handled default parameters prior to ES6. Let's look at a much better way.

Default Parameters in ES6

If your app requires that you use pre-ES6 features for legacy reasons or because of browser support, then you might have to do something similar to what I've described above. But ES6 has made this much easier. Here's how to define default parameter values in ES6 and beyond:

```
function getInfo (name, year = 2018, color = 'blue') {
  // function body here...
}
```

Try it on CodePen[159]

It's that simple.

If `year` and `color` values are passed into the function call, the values passed in as arguments will supersede the ones defined as parameters in the function definition. This works exactly the same way as with the ES5 patterns, but without all that extra code. Much easier to maintain, and much easier to read.

This feature can be used for any of the parameters in the function head, so you could set a default for the first parameter along with two other expected values that don't have defaults:

[157] https://codepen.io/impressivewebs/pen/rJapgz?editors=0011
[158] https://www.impressivewebs.com/callback-functions-javascript/
[159] https://codepen.io/impressivewebs/pen/OQPzKJ?editors=0011

```
function getInfo (name = 'Pat', year, color) {
  // function body here...
}
```

Dealing With Omitted Values

Note that—in a case like the one above—if you wanted to omit the optional `name` argument (thus using the default) while including a `year` and `color`, you'd have to pass in `undefined` as a placeholder for the first argument:

```
getInfo(undefined, 1995, 'Orange');
```

If you don't do this, then logically the first value will always be assumed to be `name`.

The same would apply if you wanted to omit the `year` argument (the second one) while including the other two (assuming, of course, the second parameter is optional):

```
getInfo('Charlie', undefined, 'Pink');
```

I should also note that the following may produce unexpected results:

```
function getInfo (name, year = 1965, color = 'blue') {
  console.log(year); // null
}
getInfo('Frankie', null, 'Purple');
```

Try it on CodePen[160]

In this case, I've passed in the second argument as `null`, which might lead some to believe the `year` value inside the function should be `1965`, which is the default. But this doesn't happen, because `null` is considered a valid value. And this makes sense because, according to the spec[161], `null` is viewed by the JavaScript engine as the intentional absence of an object's value, whereas `undefined` is viewed as something that happens incidentally (e.g. when a function doesn't have a `return` value it returns `undefined`).

So make sure to use `undefined` and not `null` when you want the default value to be used. Of course, there might be cases where you want to use `null` and then deal with the `null` value within the function body, but you should be familiar with this distinction.

[160] https://codepen.io/impressivewebs/pen/rJaJNY?editors=0011
[161] https://tc39.github.io/ecma262/#sec-undefined-value

Default Parameter Values and the arguments Object

Another point worth mentioning here is in relation to the `arguments` object. The `arguments` object is an array-like object, accessible inside a function's body, that represents the arguments passed to a function.

In non-strict mode, the `arguments` object reflects any changes made to the argument values inside the function body. For example:

```
function getInfo (name, year, color) {
  console.log(arguments);
  /*
  [object Arguments] {
    0: "Frankie",
    1: 1987,
    2: "Red"
  }
  */

  name = 'Jimmie';
  year = 1995;
  color = 'Orange';

  console.log(arguments);
  /*
  [object Arguments] {
    0: "Jimmie",
    1: 1995,
    2: "Orange"
  }
  */
}

getInfo('Frankie', 1987, 'Red');
```

Try it on CodePen[162]

Notice in the above example, if I change the values of the function's parameters, those changes are reflected in the `arguments` object. This feature was viewed as more problematic than beneficial, so in strict mode the behavior is different:

[162] https://codepen.io/impressivewebs/pen/RQNQPm?editors=0011

```
function getInfo (name, year, color) {
  'use strict';

  name = 'Jimmie';
  year = 1995;
  color = 'Orange';

  console.log(arguments);
  /*
  [object Arguments] {
    0: "Frankie",
    1: 1987,
    2: "Red"
  }
  */
}

getInfo('Frankie', 1987, 'Red');
```

Try it on CodePen[163]

As shown in the demo, in strict mode the `arguments` object retains its original values for the parameters.

That brings us to the use of default parameters. How does the `arguments` object behave when the default parameters feature is used? Take a look at the following code:

```
function getInfo (name, year = 1992, color = 'Blue') {
  console.log(arguments.length); // 1

  console.log(year, color);
  // 1992
  // "Blue"

  year = 1995;
  color = 'Orange';

  console.log(arguments.length); // Still 1
  console.log(arguments);

  /*
  [object Arguments] {
    0: "Frankie"
```

[163] https://codepen.io/impressivewebs/pen/vdEdNw?editors=0011

```
  }
  */

  console.log(year, color);
  // 1995
  // "Orange"
}

getInfo('Frankie');
```

Try it on CodePen[164]

There are a few things to note in this example.

First, the inclusion of default parameters doesn't change the `arguments` object. So, as in this case, if I pass only one argument in the functional call, the `arguments` object will hold a single item—even with the default parameters present for the optional arguments.

Second, when default parameters are present, the `arguments` object will always behave the same way in strict mode and non-strict mode. The above example is in non-strict mode, which usually allows the `arguments` object to be modified. But this doesn't happen. As you can see, the length of `arguments` remains the same after modifying the values. Also, when the object itself is logged, the `name` value is the only one present.

Expressions as Default Parameters

The default parameters feature is not limited to static values but can include an expression to be evaluated to determine the default value. Here's an example to demonstrate a few things that are possible:

```
function getAmount() {
  return 100;
}

function getInfo (name, amount = getAmount(), color = name) {
  console.log(name, amount, color)
}

getInfo('Scarlet');
// "Scarlet"
// 100
// "Scarlet"
```

[164] https://codepen.io/impressivewebs/pen/NyPyrr?editors=0011

```
13
14 getInfo('Scarlet', 200);
15 // "Scarlet"
16 // 200
17 // "Scarlet"
18
19 getInfo('Scarlet', 200, 'Pink');
20 // "Scarlet"
21 // 200
22 // "Pink"
```

Try it on CodePen[165]

There are a few things to take note of in the code above. First, I'm allowing the second parameter, when it's not included in the function call, to be evaluated by means of the `getAmount()` function. This function will be called only if a second argument is not passed in. This is evident in the second `getInfo()` call and the subsequent log.

The next key point is that I can use a previous parameter as the default for another parameter. I'm not entirely sure how useful this would be, but it's good to know it's possible. As you can see in the above code, the `getInfo()` function sets the third parameter (`color`) to equal the first parameter's value (`name`), if the third parameter is not included.

And of course, since it's possible to use functions to determine default parameters, you can also pass an existing parameter into a function used as a later parameter, as in the following example:

```
1  function getFullPrice(price) {
2    return (price * 1.13);
3  }
4
5  function getValue (price, pricePlusTax = getFullPrice(price)) {
6    console.log(price.toFixed(2), pricePlusTax.toFixed(2))
7  }
8
9  getValue(25);
10 // "25.00"
11 // "28.25"
12
13 getValue(25, 30);
14 // "25.00"
15 // "30.00"
```

Try it on CodePen[166]

[165] https://codepen.io/impressivewebs/pen/GQgQrR?editors=0011
[166] https://codepen.io/impressivewebs/pen/PQwQjP?editors=0011

In the above example, I'm doing a rudimentary tax calculation in the `getFullPrice()` function. When this function is called, it uses the existing `price` parameter as part of the `pricePlusTax` evaluation. As mentioned earlier, the `getFullPrice()` function is not called if a second argument is passed into `getValue()` (as demonstrated in the second `getValue()` call).

Two things to keep in mind with regards to the above. First, the function call in the default parameter expression needs to include the parentheses, otherwise you'll receive a function reference rather than an evaluation of the function call.

Second, you can only reference previous parameters with default parameters. In other words, you can't reference the second parameter as an argument in a function to determine the default of the first parameter:

```
// this won't work
function getValue (pricePlusTax = getFullPrice(price), price) {
  console.log(price.toFixed(2), pricePlusTax.toFixed(2))
}

getValue(25); // throws an error
```

Try it on CodePen[167]

Similarly, as you would expect, you can't access a variable defined inside the function body from a function parameter.

Conclusion

That should cover just about everything you'll need to know to get the most out of using default parameters in your functions in ES6 and above. The feature itself is quite easy to use in its simplest form but, as I've discussed here, there are quite a few details worth understanding.

If you'd like to read more on this topic, here are some sources:

- Understanding ECMAScript 6[168] by Nicholas Zakas. This was my primary source for this article. Nicholas is definitely my favorite JavaScript author.
- Arguments object[169] on MDN
- Default Parameters[170] on MDN

[167] https://codepen.io/impressivewebs/pen/NyPywd?editors=0011
[168] https://leanpub.com/understandinges6/
[169] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments
[170] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters

An Introduction and Guide to the CSS Object Model (CSSOM)

If you've been writing JavaScript for some time now, it's almost certain you've written some scripts dealing with the Document Object Model (DOM). DOM scripting takes advantage of the fact that a web page opens up a set of APIs (or interfaces) so you can manipulate and otherwise deal with elements on a page.

But there's another object model you might want to become more familiar with: The CSS Object Model (CSSOM). Likely you've already used it but didn't necessarily realize it.

In this guide, I'm going to go through many of the most important features of the CSSOM, starting with stuff that's more commonly known, then moving on to some more obscure, but practical, features.

What is the CSSOM?

According to MDN:[171]

> The CSS Object Model is a set of APIs allowing the manipulation of CSS from JavaScript. It is much like the DOM, but for the CSS rather than the HTML. It allows users to read and modify CSS style dynamically.

MDN's info is based on the official W3C CSSOM specification[172]. That W3C document is a somewhat decent way to get familiar with what's possible with the CSSOM, but it's a complete disaster for anyone looking for practical coding examples that put the CSSOM APIs into action.

MDN is much better, but still largely lacking in certain areas. So for this post, I've tried to do my best to create useful code examples and demos of these interfaces in use, so you can see the possibilities and mess around with the live code.

As mentioned, the post starts with stuff that's already familiar to most front-end developers. These common features are usually lumped in with DOM scripting, but they are technically part of the larger group of interfaces available via the CSSOM (though they do cross over into the DOM as well).

[171] https://developer.mozilla.org/en-US/docs/Web/API/CSS_Object_Model
[172] https://drafts.csswg.org/cssom/

Inline Styles via element.style

The most basic way you can manipulate or access CSS properties and values using JavaScript is via the `style` object, or property, which is available on all HTML elements. Here's an example:

```
document.body.style.background = 'lightblue';
```

Most of you have probably seen or used that syntax before. I can add to or change the CSS for any object on the page using that same format: `element.style.propertyName`.

In that example, I'm changing the value of the `background` property to `lightblue`. Of course, `background` is shorthand. What if I want to change the `background-color` property? For any hyphenated property, just convert the property name to camel case:

```
document.body.style.backgroundColor = 'lightblue';
```

In most cases, a single-word property would be accessed in this way by the single equivalent word in lowercase, while hyphenated properties are represented in camel case. The one exception to this is when using the `float` property. Because `float` is a reserved word in JavaScript, you need to use `cssFloat` (or `styleFloat` if you're supporting IE8 and earlier). This is similar to the HTML `for` attribute being referenced as `htmlFor` when using something like `getAttribute()`.

Here's a demo that uses the style property to allow the user to change the background color of the current page:

View on CodePen[173]

So that's an easy way to define a CSS property and value using JavaScript. But there's one huge caveat to using the `style` property in this way: This will only apply to inline styles on the element.

This becomes clear when you use the `style` property to read CSS:

```
document.body.style.backgroundColor = 'lightblue';
console.log(document.body.style.backgroundColor);
// "lightblue"
```

In the example above, I'm defining an inline style on the `<body>` element, then I'm logging that same style to the console. That's fine. But if I try to read another property on that element, it will return nothing — unless I've previously defined an inline style for that element in my CSS or elsewhere in my JavaScript. For example:

[173] https://codepen.io/impressivewebs/pen/mQbqGR

```
console.log(document.body.style.color);
// Returns nothing if inline style doesn't exist
```

This would return nothing even if there was an external stylesheet that defined the `color` property on the `<body>` element, as in the following CodePen:

View on CodePen[174]

Using `element.style` is the simplest and most common way to add styles to elements via JavaScript. But as you can see, this clearly has some significant limitations, so let's look at some more useful techniques for reading and manipulating styles with JavaScript.

Getting Computed Styles

You can read the computed CSS value for any CSS property on an element by using the `window.getComputedStyle()` method:

```
window.getComputedStyle(document.body).background;
// "rgba(0, 0, 0, 0) none repeat scroll 0% 0% / auto padding-box border-box"
```

Well, that's an interesting result. In a way, `window.getComputedStyle()` is the `style` property's overly-benevolent twin. While the `style` property gives you far too little information about the actual styles on an element, `window.getComputedStyle()` can sometimes give you too much.

Try it on CodePen[175]

In the example above, the `background` property of the `<body>` element was defined using a single value. But the `getComputedStyle()` method returns all values contained in `background` shorthand. The ones not explicitly defined in the CSS will return the initial (or default) values for those properties.

This means, for any shorthand property, `window.getComputedStyle()` will return all the initial values, even if none of them is defined in the CSS:

Try it on CodePen[176]

Similarly, for properties like `width` and `height`, it will reveal the computed dimensions of the element, regardless of whether those values were specifically defined anywhere in the CSS, as the following interactive demo shows:

Try it on CodePen[177]

[174] https://codepen.io/impressivewebs/pen/LXPewe
[175] https://codepen.io/impressivewebs/pen/XyWKJE
[176] https://codepen.io/impressivewebs/pen/OaJXjR
[177] https://codepen.io/impressivewebs/pen/mQdOzg

Try resizing the parent element in the above demo to see the results. This is somewhat comparable to reading the value of `window.innerWidth`, except this is the computed CSS for the specified property on the specified element and not just a general window or viewport measurement.

There are a few different ways to access properties using `window.getComputedStyle()`. I've already demonstrated one way, which uses dot-notation to add the camel-cased property name to the end of the method. You can see three different ways to do it in the following code:

```
// dot notation, same as above
window.getComputedStyle(el).backgroundColor;

// square bracket notation
window.getComputedStyle(el)['background-color'];

// using getPropertyValue()
window.getComputedStyle(el).getPropertyValue('background-color');
```

The first line uses the same format as in the previous demo. The second line is using square bracket notation, a common JavaScript alternative to dot notation. This format is not recommended and code linters will warn about it. The third example uses the `getPropertyValue()` method.

The first example requires the use of camel casing (although in this case both `float` and `cssFloat` would work) while the next two access the property via the same syntax as that used in CSS (with hyphens, often called "kebab case").

Here's the same demo as the previous, but this time using `getPropertyValue()` to access the widths of the two elements:

Try it on CodePen[178]

Getting Computed Styles of Pseudo-Elements

One little-known tidbit about `window.getComputedStyle()` is the fact that it allows you to retrieve style information on pseudo-elements. You'll often see a `window.getComputedStyle()` declaration like this:

```
window.getComputedStyle(document.body, null).width;
```

Notice the second argument, `null`, passed into the method. Firefox prior to version 4 required a second argument, which is why you might see it used in legacy code or by those accustomed to including it. But it's not required in any browser currently in use.

That second optional parameter is what allows me to specify that I'm accessing the computed CSS of a pseudo-element. Consider the following CSS:

[178] https://codepen.io/impressivewebs/pen/pQoPqN

```
.box::before {
  content: 'Example';
  display: block;
  width: 50px;
}
```

Here I'm adding a `::before` pseudo-element inside the `.box` element. With the following JavaScript, I can access the computed styles for that pseudo-element:

```
let box = document.querySelector('.box');
window.getComputedStyle(box, '::before').width;
// "50px"
```

Try it on CodePen[179]

You can also do this for other pseudo-elements like `::first-line`, as in the following code and demo:

```
let p = document.querySelector('.box p');
window.getComputedStyle(p, '::first-line').color;
```

Try it on CodePen[180]

And here's another example using the `::placeholder` pseudo-element, which apples to `<input>` elements:

```
let input = document.querySelector('input');
window.getComputedStyle(input, '::placeholder').color
```

Try it on CodePen[181]

Note: The above works in the latest Firefox, but not in Chrome or Edge (I've filed a bug report[182] for Chrome).

It should also be noted that browsers have different results when trying to access styles for a non-existent (but valid) pseudo-element compared to a pseudo-element that the browser doesn't support at all (like a made up `::banana` pseudo-element). You can try this out in various browsers using the following demo:

Try it on CodePen[183]

As a side point to this section, there is a Firefox-only method called getDefaultComputedStyle()[184] that is not part of the spec and likely never will be.

[179] https://codepen.io/impressivewebs/pen/YRXzdm
[180] https://codepen.io/impressivewebs/pen/rQVajQ
[181] https://codepen.io/impressivewebs/pen/ZmGGXG
[182] https://bugs.chromium.org/p/chromium/issues/detail?id=850744
[183] https://codepen.io/impressivewebs/pen/VVLLgx
[184] https://developer.mozilla.org/en-US/docs/Web/API/Window/getDefaultComputedStyle

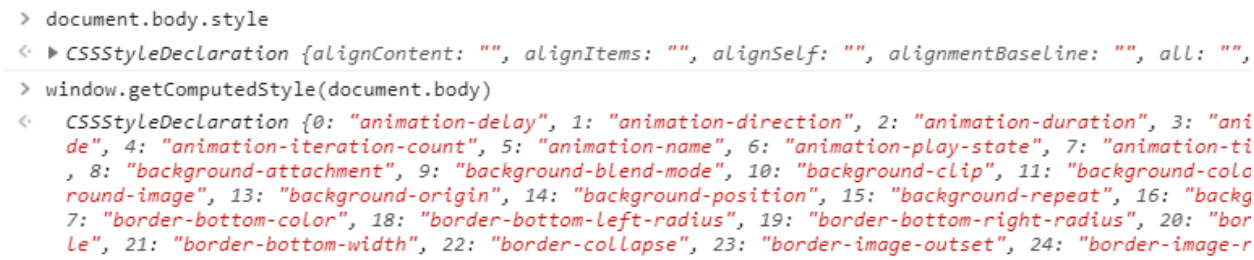
The CSSStyleDeclaration API

Earlier when I showed you how to access properties via the style object or using `getComputedStyle()`, in both cases those techniques were exposing the `CSSStyleDeclaration` interface.

In other words, both of the following lines will return a `CSSStyleDeclaration` object on the document's `body` element:

```
1 document.body.style;
2 window.getComputedStyle(document.body);
```

In the following screenshot you can see what the console produces for each of these lines:



The CSSStyleDeclaration API in the DevTools console

In the case of `getComputedStyle()`, the values are read-only. In the case of `element.style`, getting and setting the values is possible but, as mentioned earlier, these will only affect the document's inline styles.

setProperty(), getPropertyValue(), and item()

Once you've exposed a `CSSStyleDeclaration` object in one of the above ways, you have access to a number of useful methods to read or manipulate the values. Again, the values are read-only in the case of `getComputedStyle()`, but when used via the `style` property, some methods are available for both getting and setting.

Consider the following code and demo:

```
1 let box = document.querySelector('.box');
2
3 box.style.setProperty('color', 'orange');
4 box.style.setProperty('font-family', 'Georgia, serif');
5 op.innerHTML = box.style.getPropertyValue('color');
6 op2.innerHTML = `${box.style.item(0)}, ${box.style.item(1)}`;
```

Try it on CodePen[185]

In this example, I'm using three different methods of the `style` object:

- The `setProperty()` method. This takes two arguments, each a string: The property (in regular CSS notation) and the value you wish to assign to the property.
- The `getPropertyValue()` method. This takes a single argument: The property whose value you want to obtain. This method was used in a previous example using `getComputedStyle()`, which, as mentioned, likewise exposes a `CSSStyleDeclaration` object.
- The `item()` method. This takes a single argument, which is a positive integer representing the index of the property you want to access. The return value is the property name at that index.

Keep in mind that in my simple example above, there are only two styles added to the element's inline CSS. This means that if I were to access `item(2)`, the return value would be an empty string. I'd get the same result if I used `getPropertyValue()` to access a property that isn't set in that element's inline styles.

Using removeProperty()

In addition to the three methods mentioned above, there are two others exposed on a `CSSStyleDeclaration` object. In the following code and demo, I'm using the `removeProperty()` method:

```
box.style.setProperty('font-size', '1.5em');
box.style.item(0) // "font-size"

document.body.style.removeProperty('font-size');
document.body.style.item(0); // ""
```

Try it on CodePen[186]

In this case, after I set `font-size` using `setProperty()`, I log the property name to ensure it's there. The demo then includes a button that, when clicked, will remove the property using `removeProperty()`.

Note: In the case of `setProperty()` and `removeProperty()`, the property name that you pass in is hyphenated (the same format as in your stylesheet), rather than camel-cased. This might seem confusing at first, but the value passed in is a string in this example, so it makes sense.

Getting and Setting a Property's Priority

Finally, here's an interesting feature that I discovered while researching this article: The `getPropertyPriority()` method, demonstrated with the code and CodePen below:

[185] https://codepen.io/impressivewebs/pen/vQOKxb
[186] https://codepen.io/impressivewebs/pen/dQoBZq

```
box.style.setProperty('font-family', 'Georgia, serif', 'important');
box.style.setProperty('font-size', '1.5em');

box.style.getPropertyPriority('font-family'); // important
op2.innerHTML = box.style.getPropertyPriority('font-size'); // ""
```

Try it on CodePen[187]

In the first line of that code, you can see I'm using the `setProperty()` method, as I did before. However, notice I've included a third argument. The third argument is an optional string that defines whether you want the property to have the `!important` keyword attached to it.

After I set the property with `!important`, I use the `getPropertyPriority()` method to check that property's priority. If you want the property to not have importance, you can omit the third argument, use the keyword `undefined`, or include the third argument as an empty string.

And I should emphasize here that these methods would work in conjunction with any inline styles already placed directly in the HTML on an element's `style` attribute.

So if I had the following HTML:

```
<div class="box" style="border: solid 1px red !important;">
```

I could use any of the methods discussed in this section to read or otherwise manipulate that style. And it should be noted here that since I used a shorthand property for this inline style and set it to `!important`, all of the longhand properties that make up that shorthand will return a priority of `important` when using `getPropertyPriority()`. See the code and demo below:

```
// These all return "important"
box.style.getPropertyPriority('border'));
box.style.getPropertyPriority('border-top-width'));
box.style.getPropertyPriority('border-bottom-width'));
box.style.getPropertyPriority('border-color'));
box.style.getPropertyPriority('border-style'));
```

Try it on CodePen[188]

In the demo, even though I explicitly set only the `border` property in the `style` attribute, all the associated longhand properties that make up `border` will also return a value of `important`.

[187] https://codepen.io/impressivewebs/pen/EOjqKd
[188] https://codepen.io/impressivewebs/pen/EOVKeK

The CSSStyleSheet Interface

So far, much of what I've considered deals with inline styles (which often aren't that useful) and computed styles (which are useful, but are often too specific).

A much more useful API that allows you to retrieve a stylesheet that has readable and writable values, and not just for inline styles, is the `CSSStyleSheet` API. The simplest way to access information from a document's stylesheets is using the `styleSheets` property of the current document. This exposes the `CSSStyleSheet` interface.

For example, the line below uses the `length` property to see how many stylesheets the current document has:

```
1  document.styleSheets.length; // 1
```

I can reference any of the document's stylesheets using zero-based indexing:

```
1  document.styleSheets[0];
```

If I log that stylesheet to my console, I can view the methods and properties available:

The CSSStyleDeclaration API in the DevTools console

The one that will prove useful is the `cssRules` property. This property provides a list of all CSS rules (including declaration blocks, at-rules, media rules, etc.) contained in that stylesheet. In the following sections, I'll detail how to utilize this API to manipulate and read styles from an external stylesheet.

Working with a Stylesheet Object

For the purpose of simplicity, let's work with a sample stylesheet that has only a handful of rules in it. This will allow me to demonstrate how to use the CSSOM to access the different parts of a stylesheet in a similar way to accessing elements via DOM scripting.

Here is the stylesheet I'll be working with:

```
* {
  box-sizing: border-box;
}

body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 2em;
  line-height: 1.4;
}

main {
  width: 1024px;
  margin: 0 auto !important;
}

.component {
  float: right;
  border-left: solid 1px #444;
  margin-left: 20px;
}

@media (max-width: 800px) {
  body {
    line-height: 1.2;
  }

  .component {
    float: none;
    margin: 0;
  }
}

a:hover {
  color: lightgreen;
}
```

```
36
37 @keyframes exampleAnimation {
38   from {
39     color: blue;
40   }
41
42   20% {
43     color: orange;
44   }
45
46   to {
47     color: green;
48   }
49 }
50
51 code {
52   color: firebrick;
53 }
```

There's a number of different things I can attempt with this example stylesheet and I'll demonstrate a few of those here. First, I'm going to loop through all the style rules in the stylesheet and log the selector text for each one:

```
1 let myRules = document.styleSheets[0].cssRules,
2     p = document.querySelector('p');
3
4 for (i of myRules) {
5   if (i.type === 1) {
6     p.innerHTML += `<code>${i.selectorText}</code><br>`;
7   }
8 }
```

Try it on CodePen[189]

A couple of things to take note of in the above code and demo. First, I cache a reference to the `cssRules` object for my stylesheet. Then I loop over all the rules in that object, checking to see what type each one is.

In this case, I want rules that are type `1`, which represents the `STYLE_RULE` constant. Other constants include `IMPORT_RULE` (3), `MEDIA_RULE` (4), `KEYFRAMES_RULE` (7), etc. You can view a full table of these constants in this MDN article[190].

[189] https://codepen.io/impressivewebs/pen/VVemNb
[190] https://developer.mozilla.org/en-US/docs/Web/API/CSSRule#Type_constants

When I confirm that a rule is a style rule, I print the `selectorText` property for each of those style rules. This will produce the following lines for the specified stylesheet:

```
*
body
main
.component
a:hover
code
```

The `selectorText` property is a string representation of the selector used on that rule. This is a writable property, so if I want I can change the selector for a specific rule inside my original `for` loop with the following code:

```
if (i.selectorText === 'a:hover') {
  i.selectorText = 'a:hover, a:active';
}
```

Try it on CodePen[191]

In this example, I'm looking for a selector that defines `:hover` styles on my links and expanding the selector to apply the same styles to elements in the `:active` state. Alternatively, I could use some kind of string method or even a regular expression to look for all instances of `:hover`, and then do something from there. But this should be enough to demonstrate how it works.

Accessing @media Rules with the CSSOM

You'll notice my stylesheet also includes a media query rule and a keyframes at-rule block. Both of those were skipped when I searched for style rules (type 1). Let's now find all `@media` rules:

```
let myRules = document.styleSheets[0].cssRules,
    p = document.querySelector('.output');

for (i of myRules) {
  if (i.type === 4) {
    for (j of i.cssRules) {
      p.innerHTML += `<code>${j.selectorText}</code><br>`;
    }
  }
}
```

Based on the given stylesheet, the above will produce:

[191] https://codepen.io/impressivewebs/pen/oQbYKZ

```
body
.component
```

Try it on CodePen[192]

As you can see, after I loop through all the rules to see if any `@media` rules exist (type 4), I then loop through the `cssRules` object for each `@media` rule (in this case, there's only one) and log the selector text for each rule inside that media rule.

So the interface that's exposed on a `@media` rule is similar to the interface exposed on a stylesheet. The `@media` rule, however, also includes a `conditionText` property, as shown in the following snippet and demo:

```
let myRules = document.styleSheets[0].cssRules,
    p = document.querySelector('.output');

for (i of myRules) {
  if (i.type === 4) {
    p.innerHTML += `<code>${i.conditionText}</code><br>`;
    // (max-width: 800px)
  }
}
```

Try it on CodePen[193]

This code loops through all media query rules and logs the text that determines when that rule is applicable (i.e. the condition). There's also a `mediaText` property that returns the same value. According to the spec, you can get or set either of these.

Accessing @keyframes Rules with the CSSOM

Now that I've demonstrated how to read information from a @media rule, let's consider how to access a @keyframes rule. Here's some code to get started:

[192] https://codepen.io/impressivewebs/pen/MzyeWd
[193] https://codepen.io/impressivewebs/pen/OaNXgo

```
let myRules = document.styleSheets[0].cssRules,
    p = document.querySelector('.output');

for (i of myRules) {
  if (i.type === 7) {
    for (j of i.cssRules) {
     p.innerHTML += `<code>${j.keyText}</code><br>`;
    }
  }
}
```

Try it on CodePen[194]

In this example, I'm looking for rules that have a type of 7 (i.e. `@keyframes` rules). When one is found, I loop through all of that rule's `cssRules` and log the `keyText` property for each. The log in this case will be:

```
"0%"
"20%"
"100%"
```

You'll notice my original CSS uses `from` and `to` as the first and last keyframes, but the `keyText` property computes these to `0%` and `100%`. The value of `keyText` can also be set. In my example stylesheet, I could hard code it like this:

```
// Read the current value (0%)
document.styleSheets[0].cssRules[6].cssRules[0].keyText;

// Change the value to 10%
document.styleSheets[0].cssRules[6].cssRules[0].keyText = '10%'

// Read the new value (10%)
document.styleSheets[0].cssRules[6].cssRules[0].keyText;
```

Try it on CodePen[195]

Using this, we can dynamically alter an animation's keyframes in the flow of a web app or possibly in response to a user action.

Another property available when accessing a `@keyframes` rule is name:

[194] https://codepen.io/impressivewebs/pen/MzybxL
[195] https://codepen.io/impressivewebs/pen/bQpByM

```
let myRules = document.styleSheets[0].cssRules,
    p = document.querySelector('.output');

for (i of myRules) {
  if (i.type === 7) {
    p.innerHTML += `<code>${i.name}</code><br>`;
  }
}
```

Try it on CodePen[196]

Recall that in the CSS, the `@keyframes` rule looks like this:

```
@keyframes exampleAnimation {
  from {
    color: blue;
  }

  20% {
    color: orange;
  }

  to {
    color: green;
  }
}
```

Thus, the `name` property allows me to read the custom name chosen for that `@keyframes` rule. This is the same name that would be used in the `animation-name` property when enabling the animation on a specific element.

One final thing I'll mention here is the ability to grab specific styles that are inside a single keyframe. Here's some example code with a demo:

[196] https://codepen.io/impressivewebs/pen/oQxWGg

```
let myRules = document.styleSheets[0].cssRules,
    p = document.querySelector('.output');

for (i of myRules) {
  if (i.type === 7) {
    for (j of i.cssRules) {
      p.innerHTML += `<code>${j.style.color}</code><br>`;
    }
  }
}
```

In this example, after I find the `@keyframes` rule, I loop through each of the rules in the keyframe (e.g. the "from" rule, the "20%" rule, etc). Then, within each of those rules, I access an individual `style` property. In this case, since I know `color` is the only property defined for each, I'm merely logging out the color values.

The main takeaway in this instance is the use of the `style` property, or object. Earlier I showed how this property can be used to access inline styles. But in this case, I'm using it to access the individual properties inside of a single keyframe.

You can probably see how this opens up some possibilities. This allows you to modify an individual keyframe's properties on the fly, which could happen as a result of some user action or something else taking place in an app or possibly a web-based game.

Adding and Removing CSS Declarations

The `CSSStyleSheet` interface has access to two methods that allow you to add or remove an entire rule from a stylesheet. The methods are: `insertRule()` and `deleteRule()`. Let's see both of them in action manipulating our example stylesheet:

```
let myStylesheet = document.styleSheets[0];
console.log(myStylesheet.cssRules.length); // 8

document.styleSheets[0].insertRule('article { line-height: 1.5; font-size: 1.5em; }'\
, myStylesheet.cssRules.length);
console.log(document.styleSheets[0].cssRules.length); // 9
```

Try it on CodePen[197]

In this case, I'm logging the length of the `cssRules` property (showing that the stylesheet originally has 8 rules in it), then I add the following CSS as an individual rule using the `insertRule()` method:

[197] https://codepen.io/impressivewebs/pen/QJNMgN

```
article {
  line-height: 1.5;
  font-size: 1.5em;
}
```

I log the length of the cssRules property again to confirm that the rule was added.

The `insertRule()` method takes a string as the first parameter (which is mandatory), comprising the full style rule that you want to insert (including selector, curly braces, etc). If you're inserting an at-rule, then the full at-rule, including the individual rules nested inside the at-rule can be included in this string.

The second argument is optional. This is an integer that represents the position, or index, where you want the rule inserted. If this isn't included, it defaults to `0` (meaning the rule will be inserted at the beginning of the rules collection). If the index happens to be larger than the length of the rules object, it will throw an error.

The `deleteRule()` method is much simpler to use:

```
let myStylesheet = document.styleSheets[0];
console.log(myStylesheet.cssRules.length); // 8

myStylesheet.deleteRule(3);
console.log(myStylesheet.cssRules.length); // 7
```

Try it on CodePen[198]

In this case, the method accepts a single argument that represents the index of the rule I want to remove.

With either method, because of zero-based indexing, the selected index passed in as an argument has to be less than the length of the `cssRules` object, otherwise it will throw an error.

Revisiting the CSSStyleDeclaration API

Earlier I explained how to access individual properties and values declared as inline styles. This was done via `element.style`, exposing the `CSSStyleDeclaration` interface.

The `CSSStyleDeclaration` API, however, can also be exposed on an individual style rule as a subset of the `CSSStyleSheet` API. I already alluded to this when I showed you how to access properties inside a `@keyframes` rule. To understand how this works, compare the following two code snippets:

[198] https://codepen.io/impressivewebs/pen/OaNjxL

```
<div style="color: lightblue; width: 100px; font-size: 1.3em !important;"></div>
```

```
.box {
  color: lightblue;
  width: 100px;
  font-size: 1.3em !important;
}
```

The first example is a set of inline styles that can be accessed as follows:

```
document.querySelector('div').style
```

This exposes the `CSSStyleDeclaration` API, which is what allows me to do stuff like `element.style.color`, `element.style.width`, etc.

But I can expose the exact same API on an individual style rule in an external stylesheet. This means I'm combining my use of the `style` property with the `CSSStyleSheet` interface.

So the CSS in the second example above, which uses the exact same styles as the inline version, can be accessed like this:

```
document.styleSheets[0].cssRules[0].style
```

This opens up a single `CSSStyleDeclaration` object on the one style rule in the stylesheet. If there were multiple style rules, each could be accessed using `cssRules[1]`, `cssRules[2]`, `cssRules[3]`, and so on.

So within an external stylesheet, inside of a single style rule that is of type `1`, I have access to all the methods and properties mentioned earlier. This includes `setProperty()`, `getPropertyValue()`, `item()`, `removeProperty()`, and `getPropertyPriority()`. In addition to this, those same features are available on an individual style rule inside of a `@keyframes` or `@media` rule.

Here's a code snippet and demo that demonstrates how these methods would be used on an individual style rule in our sample stylesheet:

```
// Grab the style rules for the body and main elements
let myBodyRule = document.styleSheets[0].cssRules[1].style,
    myMainRule = document.styleSheets[0].cssRules[2].style;

// Set the bg color on the body
myBodyRule.setProperty('background-color', 'peachpuff');

// Get the font size of the body
myBodyRule.getPropertyValue('font-size');

// Get the 5th item in the body's style rule
myBodyRule.item(5);

// Log the current length of the body style rule (8)
myBodyRule.length;

// Remove the line height
myBodyRule.removeProperty('line-height');

// log the length again (7)
myBodyRule.length;

// Check priority of font-family (empty string)
myBodyRule.getPropertyPriority('font-family');

// Check priority of margin in the "main" style rule (!important)
myMainRule.getPropertyPriority('margin');
```

Try it on CodePen[199]

The CSS Typed Object Model... The Future?

After everything I've considered here, it would seem odd that I'd have to break the news that it's possible that one day the CSSOM as we know it will be mostly obsolete.

That's because of something called the CSS Typed OM[200] which is part of the Houdini Project[201]. Although some people have noted that the new Typed OM is more verbose compared to the current CSSOM, the benefits, as outlined in this article by Eric Bidelman[202], include:

- Fewer bugs

[199] https://codepen.io/impressivewebs/pen/aQZvXB
[200] https://drafts.css-houdini.org/css-typed-om/
[201] https://drafts.css-houdini.org/
[202] https://developers.google.com/web/updates/2018/03/cssom

- Arithmetic operations and unit conversion
- Better performance
- Error handling
- CSS property names are always strings

For full details on those features and a glimpse into the syntax, be sure to check out the full article[203].

As of this writing, CSS Typed OM is supported only in Chrome. You can see the progress of browser support in this document[204].

Final Words

Manipulating stylesheets via JavaScript certainly isn't something you're going to do in every project. And some of the complex interactions made possible with the methods and properties I've introduced here have some very specific use cases.

If you've built some kind of tool that uses any of these APIs I'd love to hear about it[205]. My research has only scratched the surface of what's possible, but I'd love to see how any of this can be used in real-world examples.

I've put all the demos from this article into a CodePen collection[206], so you can feel free to mess around with those as you like.

[203] https://developers.google.com/web/updates/2018/03/cssom
[204] https://www.chromestatus.com/feature/5682491075592192
[205] https://webtoolsweekly.com/?view=suggest
[206] https://codepen.io/collection/DEmrVv/

More Weekly Tips!

If you liked this e-book, be sure to subscribe to my weekly newsletter Web Tools Weekly[207]. Almost every issue features a JavaScript quick tip like the ones you've read in this e-book, followed by a categorized list of tools for front-end developers. The tools cover a wide range of categories including CSS, JavaScript, React, Vue, Responsive Web Design, Mobile Development, Workflow automatiion, Media, and lots more.

[207] http://webtoolsweekly.com/

About the Author

Unless otherwise indicated, all content in this e-book belongs to Louis Lazaris. Louis is a front-end developer who's been involved in web development since 2000. He writes about front-end development at Impressive Webs[208], speaks at conferences, and does editing and writing for various web and print publications.

[208] http://www.impressivewebs.com/