**W3 JavaScript Dom and Bom**

**Use => way**

**Traversing the DOM with filter(), map(), and arrow functions**

Today I was trying to iterate through all the DOM elements with a certain class on the page, pull their attributes into an object, and compile these objects into an array. Sounds pretty simple, right? Apparently not. I ran into some trouble, so I wanted to write this to help others in similar situations.

**The naïve (wrong ) way**

var elements = document.getElementsByClassName("bgflag");   
var BgFlags = [] //Array of objects of info about the elements  
for (i in elements){ //assemble our info objects  
 var flag = {  
 height: elements[i].offsetTop,  
 bgsrc: elements[i].dataset.bgsrc,  
 bgcolor: elements[i].dataset.bgcolor,  
 size: elements[i].dataset.size,  
 name: elements[i].id,  
 image: parseInt(elements[i].dataset.image)  
 }  
 BgFlags.push(flag);   
 if(flag.name == "defaultFlag"){  
 defaltFlag = flag //global variable  
 }  
}

It turns out that [getElementsByClassName()](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementsByClassName) returns a [HTMLCollection](https://developer.mozilla.org/en-US/docs/Web/API/HTMLCollection), not an array.

This presented a challenge, as it worked fine in Chrome, but when I tested it in Edge, i took on the ID of the element, not the index like it should have. Hmm…

I linked the two relevant MDN docs above. If you read them, you’ll note that they don’t say anything about using the getElementsByClassName() method or the HTMLCollection interface in for loops. The HTMLCollection doc does, however reference the [NodeList](https://developer.mozilla.org/en-US/docs/Web/API/NodeList) doc, which does mention iteration:

Don’t be tempted to use for…in or for each…in to enumerate the items in the list, since that will also enumerate the length and item properties of the NodeList and cause errors if your script assumes it only has to deal with element objects. Also, for..in is not guaranteed to visit the properties in any particular order.

So it turns out that this is undefined behavior, meaning there’s no specification in the language about how to handle a using for…in on an HTMLCollection so browsers can just do whatever. Not fun.

**The boring way**

Per the NodeList doc linked above:

It’s possible to loop over the items in a NodeList using:

for (var i = 0; i < myNodeList.length; ++i) {  
 var item = myNodeList[i]; // Calling myNodeList.item(i) isn’t necessary in JavaScript  
}

for…of loops will loop over NodeList objects correctly:

var list = document.querySelectorAll( ‘input[type=checkbox]’ );  
for (var item of list) {  
 item.checked = true;  
}

Recent browsers also support iterator methods, forEach(), as well as entries(), values(), and keys()

That was an enormously helpful piece of information, but after searching online I noticed that it seemed like everyone on StackOverflow was using forEach() . That’s fine, but it’s not very fun.

**The fun(ctional) way**

In college, I took a class that taught functional programming. It was the most fun I’ve ever had programming.

Javascript arrays have a few methods that are known in functional programming as ***high order*** functions. In a nutshell, this generally means they take functions as arguments.

Wait a minute, that sounds familiar. While passing functions around is a bit unusual for most object oriented languages, it’s extremely common in JavaScript. ECMAScript 5.1 added a few very useful ones.

* [Array.prototype.map(fn)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) calls fn on each item in an array and returns a new array with the values that fn returned.
* [Array.prototype.filter(fn)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter) calls fn on each item in the array amd assembles a new array of only the elements from the original array for which fn returned true.

Do you see where I’m going with this? Before we get there, we need two more things:

* [***Arrow Functions***](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions): ES6 added this nifty little feature in 2015. These are different from normal functions in a number of ways, but they’re really cool because of how they handle returning (and they’re really terse)

var timesTwo = num => num \* 2;  
// if you have more than one argument, you need () around them  
// equivalent to  
var timesTwo = num => { return num \* 2 }   
//equivalent to  
var double = function(num){ return num \* 2;}

To recap:

* Arrow functions are always anonymous
* The body of an arrow function can be an expression or a block
* If the body is an expression, the result of the expression will be returned
* If the body is a block, it’s basically more compact syntax for an anonymous function\*
* \*It’s not relevant here, but arrow functions handle context and this *very* differently. Read the doc if it matters to you.

The last thing we need is

* [***Array.from()***](http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from)*:* Our whole problem at the start of this journey was that we couldn’t treat our HTMLCollection like an array. Array.from() will take an iterable object and return an array of it.

You may think to yourself “why didn’t we just do that and keep our original code?” I would reply “where’s the fun in that? We’re learning!” Also, there’s actually a good reason not to convert it to an array which I will get to later.

We’re finally here. Let’s see what we can do.

var elements = document.getElementsByClassName("bgflag");  
elements = Array.from(elements); //convert to array  
BgFlags = elements.map(element =>  
 ({  
 height: element.offsetTop,  
 bgsrc: element.dataset.bgsrc,  
 bgcolor: element.dataset.bgcolor,  
 size: element.dataset.size,  
 name: element.id,  
 image: parseInt(element.dataset.image)  
 })  
 );  
defaultFlag = BgFlags.filter(flag => flag.name == "defaultFlag")[0]; //we need the [0] because filter() returns an array

Don’t act like you’re not impressed. Sure it looks similar, but we don’t even need to worry about loops anymore! We just say “here’s the relationship between my data and what I want, make it happen.” There’s just a small detail I need to point out: the parenthesis around the curly braces for the object literal are required for this to work. Recall that arrow functions only return implicitly when they’re given an expression. Without the parenthesis, the arrow function would see the braces that define the object literal as the beginning and end of a block and throw a syntax error. Parenthesis in Javascript in general say “do this first and consider the result,” thus giving us an object.

My code actual code looks just a bit different:

var elements = document.getElementsByClassName("bgflag");  
BgFlags = Array.prototype.map.call(elements,  
 element =>  
 ({  
 height: element.offsetTop,  
 bgsrc: element.dataset.bgsrc,  
 bgcolor: element.dataset.bgcolor,  
 size: element.dataset.size,  
 name: element.id,  
 image: parseInt(element.dataset.image)  
 })  
 );  
defaultFlag = BgFlags.filter(flag => flag.name == "defaultFlag")[0]; //we need the [0] because filter() returns an array

I removed the step of converting elements to an array. Why bother if you don’t need it? It’s just a waste of an instruction. *Even though map() is a function of Array.prototype, it works on any array-like iterable*. We just need a way to call it on our HTMLCollection . Because elements is not an array, it doesn’t have the methods in Array.prototype. But Array.prototype does. We can use it and the call(thisArg, ...args) function. Every function has the call() method (functions are objects in JavaScript which means they have methods, remember? ). The call() method allows you to call a function and override the value of this. Normally, this in a method refers to the object that owns the method. That’s fine when we .map() and array, but we don’t want to map the array prototype (that wouldn’t even work), we want to map the HTMLCollection . So, we pass it elements as the first argument. The rest of the arguments will be passed through to the function it calls, so we pass it our anonymous arrow function, which would normally be the only argument we passed to map().