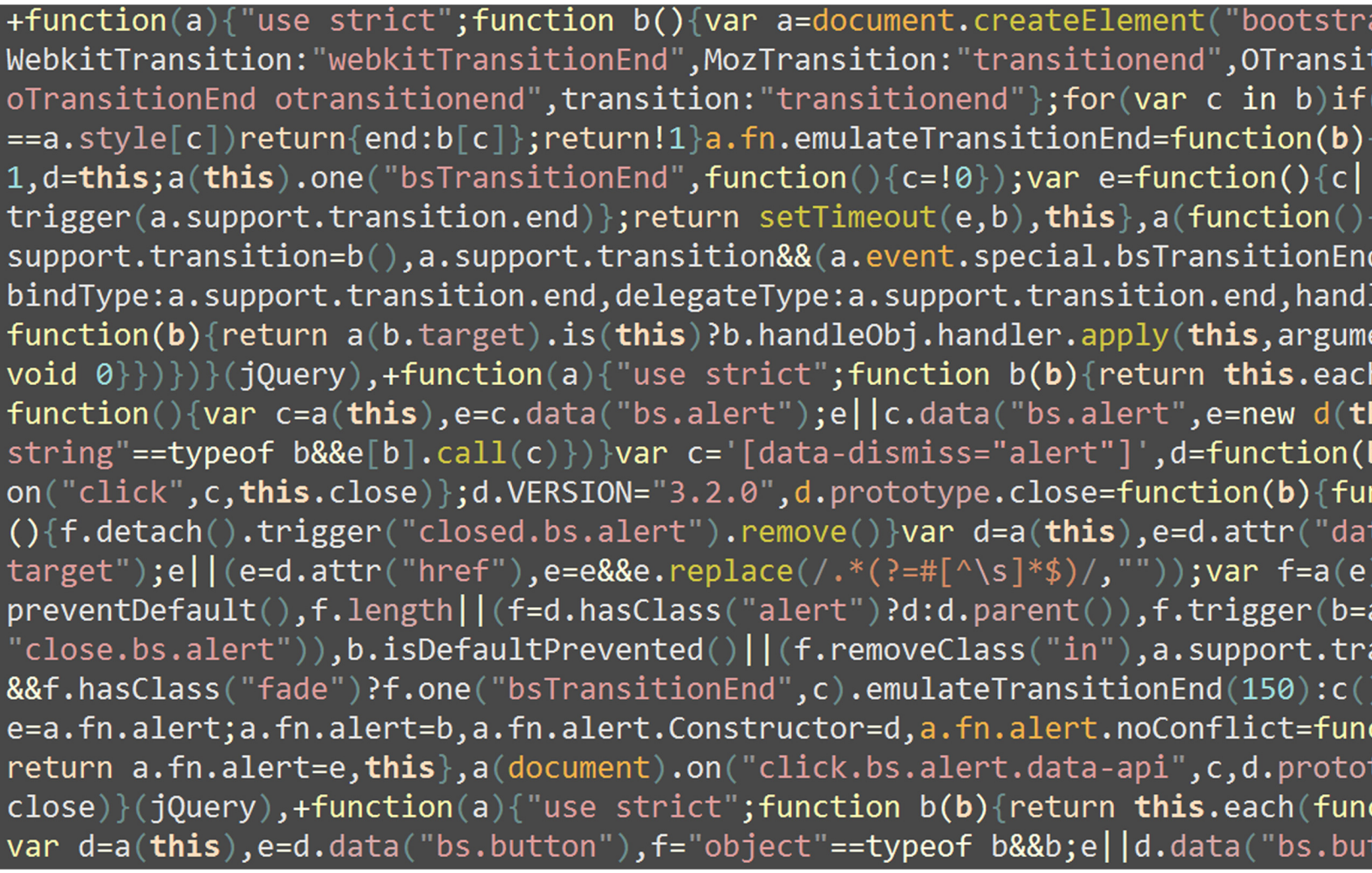
# JavaScript & DOM

## Tips, Tricks, and Techniques

No. 1





## Louis Lazaris


Quick Tips for all Levels of JavaScript Developers

# JavaScript & DOM Tips, Tricks, and Techniques (Volume 1)

## Quick Tips for All Levels of Developers

Louis Lazaris

This book is for sale at http://leanpub.com/javascriptdom1

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.

*For people who like simple, no fluff coding tips.*

# Contents

# Preface

Every week in my weekly newsletter, Web Tools Weekly, I publish a categorized list of tools, scripts, plugins, and other cool stuff geared towards front-end 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.

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 Derick Bailey for his three contributions to this volume.

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

Enjoy the book!

# JavaScript Naming Conventions

I recently read Nicholas Zakas' Maintainable JavaScript, which I highly recommend for anyone developing JavaScript apps on a large team. Near the start of the book, he talks about naming conventions. Below I summarize his suggestions, showing both good and bad examples (indicated by comments in the code).

First, he recommends using camel casing for variables and functions, which stays consistent with the fact that JavaScript's core is written using camel casing:

```
// good examples
let hereIsOneExample;
let hereIsAnotherExample;

// bad examples
let hereisoneexample;
let here_is_another_example;
```

Next, he recommends that variable names begin with a noun, to help differentiate them from functions:

```
// good examples
let count = 100;
let userName = 'Mark Twain';

// bad examples; "get" and "retrieve" are not nouns
let getCount = 100;
let retrieveName = 'Mark Twain';
```

And naturally, function names should begin with a verb:

```
// good example
function getName() {

}

// bad example; "user" is a noun, not a verb
function userName() {

}
```

He also gives some suggestions on what type of verb, depending on what the function returns. For example, if a function returns a Boolean value, it should start with “is” or “has” or another verb that implies true/false. If it’s a non-Boolean, you would use something like “get” or “set”.

Next, he recommends the C programming convention for constants, which is all uppercase with underscores:

```
// good examples
let NEWS_TITLE = 'Web Tools Weekly';
let NEWS_LOCATION = 'http://webtoolsweekly.com';
```

And finally, for constructors, use Pascal case, which is the same as camel case except the first letter is uppercase, and using a noun:

```
function NewsletterArchive() {
  // good example
}
```

Of course, the most important thing is to be consistent, but I think these are some good suggestions that can help the readability of your code, especially on large teams working on complex projects.

# insertAdjacentHTML()

Every once in a while I stumble across a native JavaScript feature that I haven't seen before. I recently came across the previously-unknown-to-me `insertAdjacentHTML()` method and thought I would share a quick summary of it.

In short, this method lets you insert an HTML fragment adjacent to a specified element. To show you how it works, suppose you have the following HTML:

```
<div id="one">
  <p>example text one.</p>
</div>
<div id="two">
  <p>example text two.</p>
</div>
```

And suppose you want to inject an HTML fragment in between element "one" and element "two". (The fragment doesn't have to be HTML, it could just be text with no actual tags.) Using `insertAdjacentHTML()` you can insert some HTML quite easily:

```
let two = document.getElementById('two');
two.insertAdjacentHTML('beforebegin',
                       '<div><p>example inserted text.</p></div>');
```

And the generated DOM will be:

```
<div id="one">
  <p>example text one.</p>
</div>
<div><p>example inserted text.</p></div>
<div id="two">
  <p>example text two.</p>
</div>
```

The method takes two parameters. The first is the position of the inserted HTML, which is relative to the element targeted (in this case the "two" element). The second parameter is the actual HTML you want to insert. The position parameter has to be one of the following strings, inside quotes:

*'beforebegin'* (places the HTML immediately before the targeted element) *'afterbegin'* (immediately before the targeted element's content) *'beforeend'* (immediately after the targeted element's content) *'afterend'* (immediately after the targeted element)

Here's a CodePen demo that interactively demonstrates how each of the values works.

According to MDN, using `insertAdjacentHTML()` has performance advantages over other methods like `innerHTML`.

And if you're wondering how this method flew under your radar for so long, it's probably because it started out as a proprietary IE feature created by Microsoft, and didn't get cross-browser support until Firefox `8` added it. As it stands, this works in all current browsers and is now an HTML5 standard. The only browsers that don't support it have now fallen out of general use.

For further reading on `insertAdjacentHTML()`, check out this article on Mozilla Hacks and this one by John Resig.

# getBoundingClientRect()

Using a little-known JavaScript method, you can get the exact coordinates (or dimensions) of any element on the page.

Let's say you have a section of HTML on your page and you want to find out exactly where that element is positioned. Maybe you want to overlay something on the element, or maybe due to user interaction, the element has moved or has been resized, and you want to reposition it or do something else with that info. Here's how you'd get the coordinates:

```
let wrapper = document.getElementById('wrapper'),
    left, top, w;

left = wrapper.getBoundingClientRect().left;
top = wrapper.getBoundingClientRect().top;
if (wrapper.getBoundingClientRect().width) {
  // for modern browsers
  w = wrapper.getBoundingClientRect().width;
} else {
  // for oldIE
  w = wrapper.offsetWidth;
}

console.log(left, top, w);
```

Notice I'm using `getBoundingClientRect()` and accessing three of its properties: `left`, `top`, and `width`. The returned object has eight read-only properties: the ones already mentioned, along with `height`, `right`, `bottom`, `x`, and `y`. The values returned would be numbers representing those properties in pixels.

Notice the code also has an if/else branch. Because IE6-8 don't support the `width` property on `getBoundingClientRect()`, I'm instead using `offsetWidth` to get the same value for those browsers.

All in-use browsers support `getBoundingClientRect()`, so as long as you account for the lack of support for `width` in old IE, you can use this safely in any project. In fact, this is yet another feature that was introduced first in Internet Explorer as far back as version 4.

And if you're wondering, the resulting coordinates will take any CSS transform values into consideration when returning the various values in the `getBoundingClientRect()` object, so that's pretty cool.

For more info see `getBoundingClientRect()` on MDN. And if you want to fiddle with this method, here's a CodePen demo that interactively displays the different values for an element on the page that you can resize then get the values again to see how they change.

# textContent()

If you want to grab the content of any element on the page, the first property you'll try is probably `innerHTML`. That's fine if you want everything in the element, including the HTML. If, however, you know that the element only contains text, or if you want to grab only the text content with the HTML stripped out, it's more appropriate to use `textContent`.

Let's say you have the following HTML:

```
<div id="module">
  <p>
    This is some <b>example</b> text
    <span>with some random markup.</span>
  </p>
</div>
```

If you want to grab all the text content inside of `#module`, you can do this:

```
let module = document.getElementById('module'),
    moduleText = module.textContent;

console.log(moduleText);
// result:
// "This is some example text with some random markup."
```

The variable `moduleText` (displayed in the console) now holds the text shown in the comment on the last line. Notice that the `<p>`, `<b>`, and `<span>` tags have been stripped out and only the text content remains.

You can see this in action in this CodePen demo. Click the button on the demo page to strip all the text out of the `#module` element and have it displayed in the console.

All browsers support `textContent` except (wait for it) IE6-8. If you need old IE support, you can use the proprietary `innerText` in a feature-detect branch that targets those browsers. MDN has more info on the differences between `textContent` and `innerText`. That article even mentions that, for performance reasons, you should use `textContent` instead of `innerHTML` when you are only manipulating HTML-less text.

# scrollIntoView()

When dealing with content that scrolls, in the past many apps and websites have incorporated animated scrolling. This might involve using jQuery's `scrollTop()` combined with `animate()`, or maybe a library or plugin like this one or this one.

But what if you want to scroll without animation to a specific part of the page without using URL hashes and without incorporating an entire library? JavaScript has a native method called `scrollIntoView()` that scrolls the page until a targeted element is 'in view'.

Here's how it works:

```
let el = document.getElementById('el');
el.scrollIntoView();
```

Using the above example, the page will scroll until the targeted element is scrolled into view, placing it at the top of the page. You also have the option to pass an optional Boolean parameter called `alignWithTop`. The default sets `alignWithTop` to `true`, thus aligning the element with the top of the scroll area.

Or it can be set to `false`, like this:

```
let el = document.getElementById('el');
el.scrollIntoView(false);
```

Then the targeted element will align to the bottom of the scroll area. You can view this method in action in this CodePen demo. You'll notice that page has a whole slew of ipsum content to allow scrolling, and one of the paragraphs is ID'd and targeted as the scroll-to point. Use the button on the page to scroll to that element.

Browser support for this method is excellent (IE7+ and everywhere else that's relevant). What's interesting is that there are a few other related methods that are not part of the spec and that have spotty support. These include `scrollIntoViewIfNeeded()` (which centers the element in the viewport), `scrollByLines()` (which lets you scroll the window by a set number of lines), and `scrollByPages()` (which scrolls by pages, where a single page is the visible vertical space on the page).

More info:

- scrollIntoView
- scrollByLines
- scrollIntoViewIfNeeded (polyfill)

# The <table> API

Most of you probably know that using core DOM methods in JavaScript you can create any HTML element and add it to the document wherever you like. While you can do the same thing when creating an HTML table, you also have the option to use the `HTMLTableElement` API.

This API has quite a few methods and properties. Below is a chunk of code that will create a table with 10 rows, 3 cells per row, along with a table caption and contents for each cell:

```
let table = document.createElement('table'),
    tbody = document.createElement('tbody'),
    i, rowcount;

table.appendChild(tbody);

for (i = 0; i <= 9; i++) {
  rowcount = i + 1;
  tbody.insertRow(i);
  tbody.rows[i].insertCell(0);
  tbody.rows[i].insertCell(1);
  tbody.rows[i].insertCell(2);
  tbody.rows[i].cells[0].appendChild(
        document.createTextNode('Row ' + rowcount + ', Cell 1')
  );
  tbody.rows[i].cells[1].appendChild(
        document.createTextNode('Row 1, Cell 2')
  );
  tbody.rows[i].cells[2].appendChild(
        document.createTextNode('Row 1, Cell 3')
  );
}

table.createCaption();
table.caption.appendChild(
  document.createTextNode('A DOM-generated Table')
);

document.body.appendChild(table);
```

Notice the use of the methods `insertRow()`, `insertCell()`, and `createCaption()`, as well as the `rows`

and `cells` properties. But that's just the tip of the iceberg. Here are a few resources to check out for more on this API:

- HTMLTableElement on MDN
- The <table> Element on WHATWG

Are there any performance issues or other drawbacks to using this API instead of some of the core methods? I'm not sure, but I do know that the code above would be much more complex if we used core DOM methods to create a table. As for browser support, from my quick research, this API seems to work everywhere, including old IE.

As always, if you want to mess around with the code above, I've put up a CodePen demo that has no HTML but generates a full 10-row, 3-column table using this API.

# Array Manipulation Tips

You've probably dealt with arrays in JavaScript quite a bit. Below I summarize a few methods and tips for manipulating arrays.

You're likely familiar with the `length` property to read the length of a string or array. When using this property on an array, it can be used to define a new length. For example:

```
let myArray = ['html', 'css', 'javascript', 'php'];
console.log(myArray.length);
// 4

myArray.length = 2;
console.log(myArray);
// ["html", "css"]

console.log(myArray.length);
// 2
```

Notice here, after checking the length of the array (confirming it's "4"), I then set the length to "2" which removes the last two items. If I set the array to a length that is higher than the actual array's length, the extra values will be set to `undefined`.

The `length` property also comes in handy if you simply want to add an item to the end of an array. For example:

```
let myArray = ['html', 'css', 'javascript', 'php'];
myArray[myArray.length] = 'ruby';
// ["html", "css", "javascript", "php", "ruby"]

myArray[myArray.length] = 'python';
// ["html", "css", "javascript", "php", "ruby", "python"]
```

Notice here that twice I've added a new item to the end of the array, and both times I used `myArray.length` to do this. The reason this works is because of zero-based indexing. The length value of an array will always be the array's index + 1 (e.g. the original array had 4 items with indexes of 0, 1, 2, and 3). Of course, this is equivalent to simply using `Array.push()`, the difference being that `Array.push()` will always return the array's new length value, whereas the other method will return the newly added array item.

Finally, although you can sort an array alphabetically using the `sort()` method, this won't sort numbers numerically. For example:

```
let myArray = ['html', 'css', 'javascript', 'php'];
// ["html", "css", "javascript", "php"]
myArray = myArray.sort();
// ["css", "html", "javascript", "php"]

let numericArray = [100, 6, 89, 1276];
// [100, 6, 89, 1276]
numericArray = numericArray.sort();
// [100, 1276, 6, 89]
```

Notice the first array was sorted fine, but the numeric array was sorted lexicographically. To resolve this, you can do the following:

```
numericArray = [100, 6, 89, 1276];
// [100, 6, 89, 1276]

let numericArray = numericArray.sort(function (a,b) {
  return a - b;
});
// [6, 89, 100, 1276]
```

As demonstrated here, the `sort()` method allows you to pass in a compare function that defines how you want the elements sorted.

Here's a CodePen demo that includes all these tips so you can fiddle with them.

That's just a few quick tips on using arrays. Here are a couple of resources for more:

JavaScript Array Methods Reference Array on MDN

# Pattern Matching

In addition to the fairly common `replace()` method, JavaScript has three other methods for doing pattern matching when dealing with strings. First, there's the `match()` method:

```
let myString = 'The loon moon arose at noon',
    myPattern = /.oon/,
    myResult = myString.match(myPattern);

console.log(myResult); // ["loon"]
```

In this example, I'm calling `match()` on `myString` and passing in a regular expression as the only argument. (Note: If you don't use a RegExp, the argument will be converted to a RegExp.) The result returned is an array with the successfully matched part of the string.

To match all results, I can add the global flag (g) to the RegExp:

```
let myString = 'The cat sat on the mat.',
    myPattern = /.at/g,
    myResult = myString.match(myPattern);

console.log(myResult); // ["cat", "sat", "mat"]
```

Now the array returns three objects. This method additionally exposes a few properties:

```
let myString = 'The loon moon arose at noon',
    myPattern = /.oon/,
    myResult = myString.match(myPattern);

console.log(myResult); // ["loon"]
console.log(myResult.index); // 4
console.log(myResult.input); // "The loon moon arose at noon"
```

The `index` property gives the position of the start of the match while the `input` property shows the original, unaffected string. The `match()` method has similar results as the `exec()` method, the difference being that `exec()` is called on the RegExp object, not the string, while the string is passed in:

```
let myString = 'The loon moon arose at noon',
    myPattern = /.oon/,
    myResult = myPattern.exec(myString);

console.log(myResult); // ["loon"]
console.log(myResult.index); // 4
console.log(myResult.input); // "The loon moon arose at noon"
```

Notice the results are the same. Finally, one of the simplest methods to pattern match is using `search()` on a string (which works very much like `test()` on a RegExp):

```
let myString = 'Rumpelstiltskin.',
    myPattern = /tilt/,
    myResult = myString.search(myPattern);

console.log(myResult); // 7
```

The `search()` method returns the position of the match, if found, otherwise it returns `-1`.

This is just a brief overview of these features. You can view these examples in action in this CodePen demo.

Check out the links below from MDN for more details:

- RegExp.exec
- String.prototype.search
- String.match

# Strict Mode

If you haven't started using strict mode in your JavaScript programs, or don't fully understand it, here's a quick overview of the hows and whys of it all.

You declare strict mode as if you were defining a string without assigning it to a variable, placing this at the top of your code:

```
1  'use strict';
```

Old browsers that don't support strict mode will see this as exactly that: a string in limbo, so they won't throw an error. But browsers that support it (IE10+ and everywhere else) will treat this as a command to switch into strict mode, which is basically a "better" version of JavaScript.

Strict mode, to put it briefly, makes it easier for you to catch errors and poor practices in your code.

Some of the improvements that strict mode introduces include:

- Makes the `with` statement illegal.
- Prevents accidental global variables.
- Makes using `eval()` safer.
- Will throw an error when trying to modify non-modifiable objects (whereas non-strict code will fail silently).

If you're going to use strict mode (and most developers now recommend that you do), here are two simple rules of thumb:

- Don't use it in the global scope, especially if you're concatenating strict and non-strict files.
- Use it inside individual functions or else use one declaration and wrap everything in a self-invoking function.

In other words, this is the wrong way to use it:

```
'use strict';

function doSomething () {
  // stuff here
}

function doAnother () {
  // other stuff
}
```

And this is the right way:

```
(function () {

  'use strict';

  function doSomething () {
    // stuff here
  }

  function doAnother () {
    // other stuff
  }

}());
```

To summarize why this is important, MDN says: "Browsers not supporting strict mode will run strict mode code with different behavior from browsers that do."

All of that being said, you should be careful when concatenating strict mode files with non-strict mode files, because a single strict mode declaration will make an entire scope run in strict mode.

For more on using strict mode, see the following resources:

- It's time to start using JavaScript strict mode by Zakas
- Comment by Paul Irish on Zakas' post, regarding production vs. development
- Strict Mode on MDN

# The + Operator

Take a look at the following code and the two resulting lines displayed in the console (shown in the code comments):

```
let date_one = new Date(),
    date_two = +new Date();

console.log(date_one);
// Mon Oct 21 2013 21:09:28 GMT-0400 (Eastern Daylight Time)

console.log(date_two);
// 1382404168241
```

With just a simple addition of the + operator (2nd line), the `date_two` variable displays a completely different format for the date compared to `date_one`. In the first instance, the format is the full date and current time (according to system settings). Naturally, you'll see different results depending on where/when you try it, but you'll see the same format.

The second variable, on the other hand, displays a really large number. Notice the following:

```
let date_one = new Date();

console.log(date_one);
// Mon Oct 21 2013 21:09:28 GMT-0400 (Eastern Daylight Time)

let date_two = Number(date_one),
    date_three = date_one.valueOf(),
    date_four = date_one.getTime(),
    date_five = Date.now();
// All produce this format: 1382404168241
```

As you can see, each of the final four variables produces the date in the same format as that of `+new Date()`. I won't get into the details on these latter four methods, so I'll just focus on the + operator.

To really oversimplify what this does: Using + in front of `new Date()` tells the browser to convert the date into a number. So, as explained on MDN's reference, the return value is "the number of milliseconds since midnight 01 January, 1970 UTC."

You can also use the + operator for a couple of other tricks:

```
+'0030768'; // becomes: 30768
+true; // becomes: 1
+false; // becomes: 0
+'0xfecd3'; // becomes: 1043667
```

View all these examples in this CodePen demo.

It might be rare that you need to do some of these conversions but they might come in handy. For example, the first line in the last code block might help with user-entered data, converting the string to a valid number with no leading zeroes. And the last example is a hex value converted to its decimal equivalent.

And yes, all these examples could be done using the admittedly more readable `Number()` constructor. Just another quirk in the JavaScript language that might save you a few bytes.

# contains()

Although we commonly think of Internet Explorer when we consider old browsers that prevent us from using certain features, this was, to some extent, also true of older versions of Firefox, before that browser began auto-updating.

Because of this, many JavaScript/DOM features that weren't available in (for example) FF3.6 have gone somewhat unnoticed. I've mentioned a few of these in previous tips, so here's another one: `contains()`.

What this does is simple: It checks to see if any given node is a descendant of another specified node. Let's look at an example.

Here's our HTML:

```
<body>

  <div id="parentBox">
    <div id="childBox">
      <div id="grandchildBox">
      </div>
    </div>
  </div>

  <div id="otherBox">
  </div>

</body>
```

Notice there are four elements inside the body. Three are nested one inside the other, then I have another sole element as a sibling of the outermost of the nested group.

Now I can use `contains()` to do various checks, based on the above structure:

```
let body = document.getElementsByTagName('body')[0],
    pb = document.getElementById('parentBox'),
    cb = document.getElementById('childBox'),
    gcb = document.getElementById('grandchildBox'),
    ob = document.getElementById('otherBox');

console.log(body.contains(pb)); // true
console.log(body.contains(gcb)); // true
console.log(pb.contains(cb)); // true
console.log(pb.contains(gcb)); // true
console.log(cb.contains(gcb)); // true
console.log(cb.contains(pb)); // false
console.log(body.contains(ob)); // true
console.log(pb.contains(ob)); // false
```

And here is a CodePen demo with the above code so you can play around if you want.

As you can see, `contains()` will return either `true` or `false`, so it's simple to use. Unfortunately, one seeming flaw is the fact that it will return `true` if the same node is checked and applied to. In other words, it will say `<body>` is contained inside of `<body>`. MDN's reference on the subject has a simple function that gets around this.

And the best part of `contains()`? It's cross-browser, including old IE back to version 5. As already mentioned, probably the only reason it hasn't gotten a lot of coverage is because it was invented by Microsoft and was not supported in Firefox until version 9.

# Document Fragments

If you've read anything about JavaScript-related performance, then you've probably heard that you should try to touch the DOM as little as possible. One specific DOM feature that helps with this is `DocumentFragment`.

`DocumentFragment` lets you create a sort of imaginary object that can hold any type of node that you could insert into a regular document. You use it as a temporary placeholder, then "empty" it by inserting its contents into the DOM all at once. This has an advantage over `createElement()` because the root element disappears as soon as you empty it.

Let's consider a typical example of its use: Adding elements to a list.

Here's the HTML:

```
<ul id="list">
  <!-- inserted nodes will end up here -->
</ul>
```

To add items to this empty list, I could do so one by one, interacting with the DOM each time. But I'll do it more efficiently with a document fragment:

```
let myList = document.getElementById('list'),
    tempFrag = document.createDocumentFragment(),
    i, listItem;

for (i = 0; i < 5; i += 1) {
  listItem = document.createElement('li');
  listItem.innerHTML = 'List item #' + (+i + 1);
  tempFrag.appendChild(listItem);
}

myList.appendChild(tempFrag);
```

Notice that the interaction inside the loop appends items to the document fragment (represented by the `tempFrag` variable). When the fragment has the full collection of nodes for the list, the whole thing is added in a single line at the end.

Here's a demo

Many experienced developers are likely already familiar with this technique and its benefits. But you might not be aware of some of the new additions coming to the DocumentFragment API, including:

- `DocumentFragment.children`
- `DocumentFragment.firstElementChild`
- `DocumentFragment.lastElementChild`
- `DocumentFragment.childElementCount`
- `new DocumentFragment()` constructor

While `DocumentFragment` in general has excellent basic support everywhere, the new features shown above are listed as "experimental" and have little or no support. Nonetheless, they give us a good preview of what's to come when using document fragments. In the meantime, we can utilize this useful technique today using its standard DOM methods.

For more details on this or the above new features, here are a few links:

- DocumentFragment on MDN
- DocumentFragment vs. createElement

# querySelector() and querySelectorAll()

By now you are likely familiar with `querySelector()` and `querySelectorAll()`, but maybe haven't made the jump to using them regularly. Here is a quick summary of these two features:

The `querySelector()` method returns the first element that matches the selector query that it contains:

```
1  let el = document.querySelector('.example');
```

This is similar to what you might have done with jQuery's `$()` wrapper when passing in a selector. In the example above, since I'm using `querySelector`, and not `querySelectorAll()`, only the first element with a class of "example" will be selected. If no element matches, it will return `null`.

Next there's `querySelectorAll()`:

```
1  let els = document.querySelectorAll('.example, .other');
```

In this case, because I'm using `querySelectorAll()`, the query will return a NodeList object that contains all the elements in the document that have a class of either "example" or "other".

In both cases, the selector value must be a valid CSS selector that the browser recognizes. Also, I can apply both methods directly to another base element rather than to the whole document:

```
1  let baseEl = document.getElementById('base'),
2  el = baseEl.querySelector('.example'),
3  els = baseEl.querySelectorAll('.example, .other');
```

In this case, line 2 will return the first element with a class of "example" that is inside an element with an ID of "base". The last line will return all elements with a class of "example" or "other" that are inside the same "base" element.

The great thing about `querySelector()` and `querySelectorAll()` is the fact that they're supported in all modern browsers even going back to IE8. The only catch with regards to IE8 is the fact that IE8 does not recognize most CSS selectors after CSS2.1. This means you can use both methods safely in IE8 as long as you don't use any new CSS selectors that IE8 doesn't support.

# Implied eval()

You might be familiar with the fact that using `eval()` is regarded as bad practice. If you use a linting tool like JSLint or JSHint, you'll be warned not to use `eval()` unnecessarily. In addition, these tools will also warn you of syntax that uses *implied* `eval()`. Here are three such examples:

## 1. The Function() constructor

You'll get a warning in many tools and linters about this syntax, which looks like this:

```
let myFunc = new Function("x", "y", "return x + y");
myFunc(3,4); // 7
```

To remedy this, you should use a function expression or function declaration.

## 2. Passing a string literal to a `setTimeout()` or `setInterval()` call

This is another form of `eval()`. Here's an example:

```
setTimeout("alert('Hello World!');", 3000);
```

The correct way to do this, to avoid eval-like behavior, is to pass a function instead:

```
setTimeout(function() {
  alert("Hello World!");
}, 3000);
```

## 3. Using document.write()

You've most certainly seen this or used it at some point:

```
1 document.write('Example text.');
```

When using `document.write()` and linting your code you'll get a warning such as "document.write can be a form of eval", or something similar.

In almost all instances, `document.write()` is unnecessary and there are perfectly valid alternatives.

For more on `eval()` and the things I mentioned above, here are some links:

- The function constructor is a form of eval (JSLint Errors)
- Implied eval, consider passing a function (JSLint Errors) eval() (MDN)
- Function constructor (MDN)
- setTimeout and setInterval (MDN)
- document.write can be a form of eval (James Wiseman)

# selectionStart and selectionEnd

When using scripting to manipulate HTML forms, you're able to use the `select()` method to select text inside of a form input or textarea element when that element receives focus (and this happens automatically when tabbing to a field).

While the `select()` method (and the related `select` event) can be useful for knowing when text is selected, you also have the option to use the `selectionStart` and `selectionEnd` properties to find out exactly what text has been selected.

Here is how that might look:

```
let txtname = document.getElementById('txtname'),
    output = document.getElementById('output');

function getSelectedText(txt) {
  return txt.value.substring(
    txt.selectionStart, txt.selectionEnd
  );
}

txtname.onselect = function () {
  output.innerHTML = getSelectedText(txtname);
};
```

And here is a demo. Notice when you select a portion of the text in the input, the selected text is displayed below the input. The selected text will also be displayed if you tab into the field.

The `selectionStart` and `selectionEnd` properties return zero-based numbers indicating where in the text the selection begins and ends. In the code above, inside the `getSelectedText()` function, I'm using the `substring()` method, which requires zero-based indicators for its arguments, so those work well together.

In my testing, these two properties worked in Chrome, Firefox, and IE11. According to Nicholas Zakas' book Professional JavaScript for Web Developers, these properties were created by Mozilla, so I don't believe they have support in older versions of IE, and Zakas mentions a workaround for old IE. Here is a demo with the old IE workaround (untested in IE).

More info on these in the WHATWG spec.

# activeElement

You can easily get the current element that is 'selected' by using a little-known DOM property called `document.activeElement`. Let's see this in action.

Suppose I have an HTML form with three form fields and a submit button. I'll also add a generic 'output' element to the page, to display the results. In addition, I've added unique `data-*` attribute values to each element, so I can use those for the output. I've also set up a few random box elements with `tabindex` attributes, so they too can be 'selected' when clicked.

Here's the JavaScript:

```
let doc = document,
    output = doc.getElementById('op'),
    activeEl;

doc.addEventListener('click', function () {
  activeEl = doc.activeElement;
  output.innerHTML = activeEl.getAttribute('data-input');
}, false);
```

And here's a CodePen demo. Click any element to see the active element's info displayed on the page.

Things to note:

- MDN explains that the 'active' element is the element that is focused and ready to receive text input. As you can see from my demo, any element can be 'active', not just those with text input capabilities.
- If you want to be able to track elements that receive focus by tabbing, you'd have to write a different script that doesn't use 'click' as the event.
- Notice my demo includes two generic boxes and a paragraph that can also be clicked, which also display their `data-*` attribute values when 'active'.
- If I don't use the `data-*` attributes, then I can output a string showing the element object itself, so the output might say something like `[object HTMLInputElement]`.

MDN's article has its own demo to view, and you can also view the WHATWG spec on the subject.

According to a few sources, this feature works everywhere, even back to IE4! Not sure if all versions of IE have the same behaviour, but the feature is present in all of them.

# removeEventListener()

You've likely used `addEventListener()` to assign event handlers to elements. You can also remove events using `removeEventListener()`. Consider the following code:

```
let btn = document.querySelector('button'),
    output = document.querySelector('output');

btn.addEventListener('click', doStuff, false);
btn.removeEventListener('click', doStuff, false);

function doStuff() {
  // do something here...
}
```

In this example, the `doStuff()` function will never fire, because I'm removing the event immediately after I add it. Normally, however, the event would be removed as a result of some condition being met. This is just to illustrate the removal.

This CodePen demo allows you to interactively remove an event listener using a button on the page.

But notice that you can't remove an event that uses an anonymous function as the second argument:

```
btn.addEventListener('click', function () {
  // do something here...
}, false);

btn.removeEventListener('click', function {
  // do something here...
}, false);
```

View on CodePen)

This doesn't work because the browser interprets these functions as different functions, both being anonymous. So in order to remove an event, you have to use a pointer of some sort that you can identify correctly in `removeEventListener()`.

Some basic notes on `removeEventListener()`:

- You can't use this method to do a general 'remove all listeners'; you need to have a specific event and result targeted.

- Interestingly, you can do something crazy like clone the body element, which removes all event listeners. But this might have other undesirable consquences.
- As is the case with `addEventListener`, no support in IE8 and below.

More info:

- `removeEventListener` on MDN
- Question on removing all listeners
- `cloneNode` on MDN

# window.matchMedia()

Using CSS media queries, you can design your website to change styles based on device size, orientation, or other media features. You can also do something similar in JavaScript using `window.matchMedia()`.

`matchMedia()` is kind of like an amalgamation of `querySelector()` and CSS media queries. Here's a simple example, followed by an explanation:

```
let MM = window.matchMedia('(max-width: 700px)'),
    op = document.querySelector('.op'),
    b = document.body;

function doMediaCheck () {
  if (MM.matches) {
    op.innerHTML = 'Viewport is less than 700px wide.';
    b.className = 'orange';
  } else {
    op.innerHTML = 'Viewport is more than 700px wide.';
    b.className = '';
  }
}

window.onresize = function () {
  doMediaCheck();
};

doMediaCheck();
```

View on CodePen

The `matchMedia()` method accepts a string value that represents the media query you want to test for, and it will return a `MediaQueryList` object. Properties on the returned object include whether the media query matches (which I'm using above) and the string representing the original query (accessed via "media"; so in the above example it would be `MM.media`).

If you resize the preview window in the demo, you'll see the background color change and the text on the page tells you whether the width is more or less than 700px. This is determined based on whether the supplied 'medium' is 'matched'.

So if you need to write some scripts or load resources in response to certain media features (window size, landscape vs. portrait, etc.), `matchMedia()` could come in handy. `matchMedia()` has pretty good

support everywhere except IE9 and lower. So basically, it's safe it use it if you're not concerned about old IE.

Below are some links for further reading along with a polyfill:

- Using window.matchMedia on Mozilla Hacks
- window.matchMedia on MDN
- matchMedia polyfill

# Try, Catch, Throw

You can use the try-catch construct as a way to handle exceptions, or errors, in JavaScript. It looks like this:

```
try {
  // code that may cause an error
} catch (error) {
  // do this when the error occurs
}
```

If an error occurs in the `try` section, code execution immediately exits and continues in `catch`, which receives an object that has info about the error. This prevents errors from being displayed by the browser.

You can also use the the optional `finally` clause:

```
try {
  // code that may cause an error
} catch (error) {
  // do this when the error occurs
}  finally {
  // this will always run, no matter what
}
```

`finally` isn't always useful because it always runs, no matter what — whether there's an error or not. So it should be used with care, especially if you're returning values in any of the code (e.g. a `return` statement in `finally` will cause a `return` statement in `try` to be overridden, even with no error).

You also have the option to 'throw' errors using the `throw` operator. This lets you 'throw' a custom error message whenever you want. For example, if a function requires that an argument is an array, you can check for this and 'throw' a custom error to help you debug:

```
function needsArray (arr) {
  if (!(arr instanceof Array)) {
    throw new Error('needsArray(): Object not an array.');
  }
  // remainder of function here...
}
```

In this case, the error thrown is creating a new instance of an existing error type, but you can throw any type of value.

Nicholas Zakas, in his book Professional JavaScript for Web Developers, explains nicely when to use try-catch and throw:

> "You should only catch errors if you know exactly what to do next. The purpose of catching an error is to prevent the browser from responding in its default manner; the purpose of throwing an error is to provide information about why an error occurred."

More info:

- try...catch on MDN
- throw on MDN
- Error Types on MDN

# setSelectionRange()

Previously I talked about the `selectionStart` and `selectionEnd` properties to help you recognize what range of text a user has selected inside of a text field.

A method related to those properties is `setSelectionRange()`. This method lets you define a range of text to select. Suppose I have the following HTML:

```
<input value="JavaScript is Awesome" id="textField" readonly>

<button id="btn">Select Text</button>
```

This will display a text field with the words "JavaScript is Awesome" in it. I can add some JavaScript so that when the user clicks the button, the word "Script" will be selected:

```
let tf  = document.getElementById('textField'),
    btn = document.getElementById('btn');

btn.onclick = function () {
  // change the numbers to select a different range
  tf.focus();
  tf.setSelectionRange(4, 10);
};
```

Here is a demo.

This is a simple method that has two required parameters: `selectionStart` and `selectionEnd`. `selectionStart` is the position in the string to begin the selection (using zero-based indexing); `selectionEnd` is the position after which you want to end the selection. In this case, the word "Script" starts at position "4" and ends at position "9", thus we use "4, 10" to target that word with the selection.

In Firefox, the `focus()` method is required prior to the `setSelectionRange()` call, but Chrome and IE11 don't seem to require this.

There is also a third optional parameter, `selectionDirection`. This defines the direction of the selection, which indicates if `selectionStart` should begin at the end of the string. It takes a value of "forward", "backward", or "none".

`setSelectionRange()` was a Mozilla creation and is now supported in all browsers, including IE9+. More info on this method on MDN and WHATWG.

# classList

Traditionally, reading and manipulating the `class` attribute on an element has been fairly unintuitive. Older versions of IE support the `className` property, but it's ugly to work with if you're dealing with multiple classes on the same element.

Enter `classList` – a much better feature for handling classes that's supported everywhere except IE9 and below. Let's look at a simple example. Suppose my page's `<body>` tag looks like this:

```
<body class="main main-about extend extend-new">
```

I'll use this to demonstrate the different things you can do with `classList`:

```
let bodyClasses = document.querySelector('body').classList;

// log classList object
console.log(bodyClasses);

bodyClasses.add('new-class');
bodyClasses.remove('main-about');

// check if 'extend' class exists
console.log(bodyClasses.contains('extend')); // true

bodyClasses.toggle('extend-new');
console.log(bodyClasses);
```

View on CodePen with a few extra console.logs so you can see the progression.

Instead of having to do complex string manipulations, you can easily add, remove, toggle, and check for the existence of any class using this simple API. Very useful for web apps that rely on CSS to change the look of something depending on different circumstances.

Things to note about `classList`:

- Technically, `classList` is read-only, but it can be altered with the `add()` and `remove()` methods.
- The `toggle()` method has an optional argument that either adds or removes the specified class depending on the truthiness or falsiness of the argument.

More info and polyfills:

- Element.classList on MDN
- classList.js by Eli Grey
- classList.js by Remy Sharp

# stopPropagation()

Likely in the past you've used `return false` to intercept clicks on links, or cancel form submissions. The standard method for achieving this in modern browsers (specifically, since IE9) is `preventDefault()`, which is now pretty well known.

But it's possible you haven't heard of or used `stopPropagation()`, which we might refer to as the handsome cousin of `preventDefault()`.

In short, `stopPropagation()` prevents further propagation (or spreading) of the current event. Let's see how it can be used.

If you have a parent box with a smaller box inside of it, you can add a click event handler to the parent box, which will also apply to the child box. But if you use the `stopPropagation()` method, you can disable the event on the child while letting it remain on the parent:

```
let parent = document.getElementById('parent'),
    child  = document.getElementById('child'),
    op     = document.querySelector('.op'),
    op2    = document.querySelector('.op2');

parent.addEventListener('click', function () {
  op.innerHTML += 'Click registered<br>';
}, false);

function stopEvent (e) {
  e.stopPropagation();
}

child.addEventListener('click', stopEvent, false);
```

View on CodePen)

Here I have a click event on the parent, which will append a message in an output element. I also have a click event on the child element.

When clicking on the child element, the browser would by default also register a click on the parent element. But since clicking the child triggers the `stopPropagation()` method, the child click doesn't propagate to the parent like it normally would. You can test this by commenting out the line with the `stopPropagation()` method.

If my explanation doesn't suffice, here's more info:

- Event Propagation on MDN
- Turning off event bubbling on QuirksMode.org

# The location Object

The `location` object has a whole slew of properties as well as some interesting methods, including `assign()`, `replace()`, and `reload()`. These three are similar, so let's briefly look at each.

The `assign()` method causes the browser to navigate to the specified URL, and it looks like this:

```
if (condition) {
  location.assign('example.html');
}
```

The `replace()` method has the same syntax, appearing to do the same thing:

```
if (condition) {
  location.replace('example.html');
}
```

(Note that this assumes you're in the browser, which means `window` is the global object.)

The only difference between these two methods is that `assign()` keeps the browser history intact, while `replace()` will replace the current page, removing it from the browser's history.

Using `assign()` is the same as defining `location.href` or changing the location directly:

```
if (condition) {
  location = 'example.html';
}
```

Those latter two are kind of like a short cut for `location.assign()`, so you'll usually not see `assign()` much in code.

As for `replace()`, in David Flanagan's JavaScript: The Definitive Guide, he says you might use `replace()` "to load a static HTML version of your web page if you detected that the user's browser did not have the features required to display the full-featured version." Using `replace()` here would be preferable to the other methods because in this instance you wouldn't want the previous page to exist in the browser's history.

Finally, the `location.reload()` method doesn't seem to have many use cases. It just reloads the current page:

```
if (condition) {
  location.reload(true);
}
```

If you use it, you have to make sure it executes only once, otherwise it will just reload the page over and over indefinitely. It also takes an optional Boolean parameter that is false by default, meaning the page will attempt to be loaded from the cache. A value of `true` forces a reload from the server.

As for browser support, MDN says all three methods are supported in all browsers.

# normalize() and splitText()

In most cases, a text node is counted as a single node in the DOM tree. But if you add a text node to an element as a sibling element (e.g. using `appendChild()`), the element will then have more than one text node:

```
let el = document.getElementById('el');
el.appendChild(document.createTextNode('extra text'));
console.log(el.childNodes.length); // 2
```

The log in that code would produce a length value of "2" after the text node is appended.

If you want to merge these text nodes, you can use a little-known method called `normalize()`:

```
el.normalize();
console.log(el.childNodes.length); // 1
```

As shown, this would produce a length value of 1, as was the case prior to adding the second text node.

You can also use a method that is the opposite of `normalize()`, and that's the `splitText()` method:

```
el.firstChild.splitText(3);
console.log(el.childNodes.length); // 2
console.log(el.firstChild.nodeValue + el.lastChild.nodeValue);
```

`splitText()` takes an argument specifying the offset to define where to do the split.

To see these two methods in action, check out this CodePen demo. Pay close attention to the way the HTML starts out, and how the output progresses (see the comments in the JavaScript panel).

As for browser support, one book I own says these methods are supported in IE6-8 but other sources say they aren't. At the very least, they're supported everywhere including IE9+.

More info on normalize() and splitText() on MDN.

# Pseudo-Elements in JavaScript

Likely you've heard of or used the `getComputedStyle()` method to get style information of a specific element. For example, the following code gets you the color for the specified element:

```
let elColor = window.getComputedStyle(myEl).color;
```

The CSS property you want to read is written in camel-case, so `background-color` becomes `backgroundColor`, `z-index` becomes `zIndex`, and so on.

One lesser-known aspect of `getComputedStyle()` is the fact that you can pass a second parameter that lets you target the style information of a pseudo-element:

```
let elColor = window.getComputedStyle(myEl, ':before').color;
```

This will get the color value of a pseudo-element that's applied to the `myEl` element. The second parameter is a string value that represents the pseudo-element. So you could also use ':after'.

Here are some notes to keep in mind when using this:

- This should also work on other pseudo-elements like `:first-letter` and `:first-line` but be sure to test to ensure it works in all browsers.
- You can use single or double colon syntax, with the same results, assuming browsers have support for both syntaxes in CSS.
- If you leave off the CSS property, it returns an object representing the full style information. Browsers seem to have different ways of handling what info is returned.
- If the pseudo-element doesn't exist, the style returned for the specified property will be the initial value for that property. Some properties return different results in different browsers. e.g. Firefox returns an empty string for `background` and `border` shorthand, but Chrome will return all the shorthand values.
- IE11 seems to be the first IE browser to support this feature and `getComputedStyle()` in general is only supported since IE9 (polyfill here).

You can check out this CodePen demo to see this how this feature works.

This is fairly useful because, generally speaking, pseudo-elements can be a pain to deal with in JavaScript because they're not really part of the DOM. So with this you can do some interesting things to detect if styles exist, which could come in handy in some instances.

# lastIndexOf()

You most certainly know about JavaScript's `indexOf()` method, which searches a given string for the first instance of a given substring, returning the position of the start of the substring. If the substring is not found, it returns -1.

There's also a method called `lastIndexOf()`, which you don't see as often. This method returns the position of the last instance of the substring, also returning -1 if nothing is found. And just like `indexOf()`, `lastIndexOf()` lets you pass in an optional second argument that defines where in the string to start the search.

```
let string = 'How much wood could a wood chuck chuck if a wood chuck could chuck woo\
d?',
  findWood = string.lastIndexOf('wood'),
 findWood2 = string.lastIndexOf('wood chuck');

console.log(findWood, findWood2); // 67, 44
```

The difference, however, with `lastIndexOf()` when using the optional parameter is the fact that the position is counted from the start, but the search goes in reverse from the defined position. So with the same string from the previous example, the following would return 22:

```
findCandy3 = stringOne.lastIndexOf('wood chuck', 32); // 22
```

This is because everything after character 32 is disregarded, since the search starts at character 32 and works backwards, looking for the substring "wood chuck".

View these examples on CodePen

As a side point, keep in mind that both these methods are case sensitive, so changing the substring to "Wood chuck" would return -1 in all cases.

# The continue Statement

The `break` statement is probably pretty familiar to most of you. It lets you "break" out of a loop at a specified point, depending on the existence of a condition.

But maybe you haven't had much exposure to the related `continue` statement and its optional `label` parameter. Look at the following example:

```
let i = 1,
    j;

for (j = 1; j <= 5; j += 1) {

  if (j === 2) {
    continue;
  }

  i += 1;
}

console.log(j); // 6
console.log(i); // 5
```

View on CodePen

Here I'm looping through and checking the value of the `j` variable. At the end of each loop iteration, I increment the `i` variable. But what does that conditional with the `continue` statement do?

Well, `continue` tells the code not only to break out of the loop at that point, but also to continue the next iteration of the loop but without executing the rest of the code in the loop. So in this example, on iteration #2 (when `j` is equal to "2"), the loop doesn't get to the point where `i` is incremented. This causes `i` and `j` to have different values when the loop is done all its iterations.

Now here's an example with the optional `label` parameter used with `continue`:

```
let num = 0,
    i, j;

outside:
for (i = 0; i < 10; i += 1) {

  for (j = 0; j < 10; j += 1) {

    if (i === 5 && j === 5) {
      continue outside;
    }

    num += 1;
  }
}

console.log(num); // 95
```

View on CodePen

Notice the label called `outside`, which is within the conditional. Also notice the `outside` label identifying the outer loop. This label in the conditional tells the script to break out of the inner `for` loop, which means the `num` variable won't increment when that condition is met. It then "continues" with the outer `for` loop.

Without the `continue` statement, this code would output a value of "100" for `num` (10*10), but because the `if` condition breaks out of the inside loop when `j` gets to "5", the inside loop will miss 5 iterations, thus num ends up as "95".

It's worth fiddling around with labels because it can get confusing. Nicholas Zakas recommends: "Always use descriptive labels and try not to nest more than a few loops", in order to avoid confusion. (Professional JavaScript for Web Developers, 2nd Edition)

More info:

- continue on MDN
- label on MDN

# event.button

The `click` event might be the most used event in JavaScript. There's a very similar event you've also likely used called `mousedown`, and the related `mouseup`.

Here are the distinct traits of all three of these events, after which I'll describe a related property you may not have used before:

- A `click` event will fire when the mouse button is clicked then released, so it's kind of like a combination of `mousedown` and `mouseup` (i.e. both must occur on the same element).
- `mousedown` will fire when the primary mouse button is pushed down, technically prior to the button being released.
- `mouseup` will occur when the mouse button is released, but the `mousedown` action can occur on a different element. For example, you can `mousedown` on the background, then drag the cursor onto an element expecting the `mouseup` event, release the mouse, and the event will fire.

One of the main differences between `click` and `mousedown` (or `mouseup`), however, is the fact that the `mousedown` and `mouseup` events give you the ability to detect which of three mouse buttons was pressed. Look at the following code:

```
1  btn.onmousedown = function (event) {
2    console.log(event.button);
3  };
```

In the above code, when the `btn` element is clicked, the console will display the `button` property of the `event` object. If the primary mouse button is pressed (usually the left button), the console will display "0". If the right mouse button is pressed (or left for a left-handed mouse), the console will display "2". If the middle (scroll-wheel) button is pressed, the result will be "1". The `mouseup` event has the same results.

Unfortunately, there are some significant browser differences that need to be noted:

- Chrome and IE11 let you detect left and middle buttons using `onclick`, but Firefox and pre-WebKit Opera won't.
- IE6-8 have a completely different system for the output numbers.
- Pre-WebKit Opera requires the user to hold `CTRL` in order to register a middle mouse button click, but not using the `click` event.
- Chrome on Windows 7 recognizes right click on `mouseup`, but Chrome on OSX has trouble with it. A few fast right clicks is the only way to get it to register.

There are tons more things I could write about mouse events but if you want to fiddle with these properties, here's a CodePen demo.

# The keypress and keydown Events

The difference between the `keydown` and `keyup` events is pretty clear: One is fired when a key on the keyboard is pressed down, and the other is fired when the key is released. But what about the `keypress` event? How is it different from `keydown`?

You can see the difference by testing various keys using the following code:

```
document.addEventListener('keydown', function () {
  console.log('KEYDOWN');
}, false);

document.addEventListener('keypress', function () {
  console.log('KEYPRESS');
}, false);
```

Here's a CodePen demo that displays the results in the console. Try pressing different keys on your keyboard. So what exactly is the difference?

You'll notice that for most keys, both events get fired, so both messages will display on the page. But for the other keys (like `ESC`, `F2`, `SHIFT`, `ALT`, etc.) only the `keydown` event is fired.

So the difference is whether or not the key has the ability to produce an output of some sort (i.e. the "H" key can produce the letter "h" whereas the `ESC` key can't produce anything visible). Another difference is the character code number that's read by the `keyCode` property when the event fires. `keypress` will display a different character code for uppercase "H" vs. lowercase "h", whereas `keydown` will display the same for both.

So, knowing this, `keypress` is always the better choice when you want to read exactly what character was entered whereas `keydown` can be used as a generic event to read whether something on the keyboard was pushed, be it an actual character or even just the `SHIFT` key.

When expecting visible keyboard output, MDN's reference suggests the use of the `input` event instead of `keypress` (used on inputs, textareas, etc. and supported in IE9+), but if you want oldIE support, you'll have to use `keypress`. Also, `keypress` exposes many properties that `input` doesn't.

- keypress on MDN
- keydown on MDN
- input on MDN

# Invoking Functions

Early in their book Secrets of the JavaScript Ninja, the authors include a chapter covering functions. I highly recommend the book for all JavaScript developers, and I thought I would summarize here part of their section on invoking functions in JavaScript.

They describe four ways to invoke a function, all of which are practical in some way. Below I briefly describe each technique.

## As a function

This is the most straightforward manner to invoke a function:

```
function doSomething () {
  // stuff here
}

doSomething();

let doSomethingElse = function () {
  // stuff here
}

doSomethingElse();
```

When invoked like this in non-strict mode, the function context is the global context, which is the `window` object in the browser. The two invocations above are equivalent to `window.doSomething()` or `window.doSomethingElse()`.

## As a method

This is when a function is assigned to a property of an object:

```
let obj = {};

obj.doSomething = function () {
  // stuff here
};

obj.doSomething();
```

This allows the parent object of the method to be referenced inside the method using the `this` keyword (similarly, "this" would refer to the `window` object in the previous example, assuming non-strict mode).

## As a constructor

This uses the `new` keyword:

```
function Constructor () {
  // stuff here
}

new Constructor();
```

When a constructor is invoked, a new empty object is created, the object gets passed to the constructor as the `this` parameter, and the new object is returned as the constructor's value.

## Using apply() and call()

These native methods allow you to change the function context that normally occurs with the other invocation methods mentioned above. These are discussed in detail later in this book.

# Mouse Coordinates

You can get the user's mouse coordinates in three distinct ways in JavaScript: Relative to the browser window, relative to the page, and relative to the entire screen. Let's see how we can do this in each case.

First, here is how to get the mouse coordinates relative to the browser window (or iframe window):

```
let op = document.getElementById('op');

function showCoords(e) {
  op.innerHTML = 'clientX value: ' +
                 e.clientX + '<br>' +
                 'clientY value: ' +
                 e.clientY;
}

document.onclick = function (e) {
  showCoords(e);
};
```

Here is a demo with the body expanded vertically so you can test it after scrolling down a bit.

Using the `clientX` and `clientY` properties on the `event` object, I'm displaying the coordinates of the mouse on the screen when the mouse was clicked. And take note that this will be regardless of whether the page has been scrolled. For example, even if you scroll down on the page, clicking the immediate top-left corner will always return "0, 0".

But what if you want the coordinates on the page, relative to scrolling? You can replace the `clientX` and `clientY` properties with `pageX` and `pageY`. Here is a demo that again allows scrolling.

Finally, you can get the mouse coordinates relative to the entire screen by swapping out the two properties again and this time using `screenX` and `screenY`. Here is the result.

# Array.every()

In ES5 there were added a number of new Array methods, including `Array.prototype.every()`. With this method you can check all the items inside an array for a condition that's defined in a callback function.

The method will return `true` if all items meet the condition, but will return `false` as soon as it finds one item in the array that doesn't meet the specified condition.

Here is a simple example:

```
let set1 = [2, 4, 6, 8, 10],
    set2 = [2, 4, 5, 8, 10];

console.log(set1.every(function (a) {
  return a % 2 === 0;
})); // true

console.log(set2.every(function (a) {
  return a % 2 === 0;
})); // false
```

Try it on CodePen

Here I have two numeric arrays defined. I log `every()` method call for each of the arrays. Notice the function that's passed as an argument into `every()`. That's the function that defines the condition to be met.

In this condition, I'm checking to ensure all the numbers in the array are even numbers, using the modulus operator (%). As shown in the trailing comment for each log, the first array passes the test, but the second one doesn't (it has a "5" in it).

As you can see, this is a simple way to loop through all the items for a specific check, without incorporating too much complex logic.

Other things to note:

- `every()` doesn't alter the original array.
- It takes an optional second argument where you tell it what value to use as `this` when executing the callback function.
- Supported in IE9+, so it's pretty safe to use and you can find a polyfill for IE8 and below.
- More info on MDN

# Function.call() and Function.apply()

*This tip was contributed by Derick Bailey of WatchMeCode.net*

The `.call()` and `.apply()` methods exist on all functions — that's right, every JavaScript function has functions attached to it. The purpose of these methods is to let you execute a function with a specified context and also to let you supply parameters to the function in two different ways.

## Using .call()

When using `.call()` on a function, the first parameter will be the context or the `this` variable. Any additional parameters passed in will be passed as arguments to the original function.

```
function doStuff(b) {
  return this.a + b;
}

let myContext = { a: 1 };
let result = doStuff.call(myContext, 2);
console.log(result); // 3
```

Try it on CodePen

You can supply any number of arguments to the `.call()` method, and all of them will be passed to the original function after setting the context.

```
doStuff.call(myContext, 1, 2, [... n]);
```

The `.call()` method is most often used when you need to dynamically call a function with a given context and a set of values that you already have as variables.

## Using .apply()

The use of `.apply()` is essentially the same, except you pass in an array as the second parameter and that array gets passed through to the original object as the list of arguments.

```
function doStuff(b, c){
  return this.a + b + c;
}

let myContext = { a: 1 },
    result = doStuff.apply(myContext, [2, 3]);
console.log(result); // 6
```

Try it on CodePen

If you try to pass more than 2 parameters to the `.apply()` method, you'll get an error.

The `.apply()` method is most often used when you need to dynamically call a function with a given context and a set of values in an array, such as splitting the arguments object from another function.

# Function.bind()

*This tip was contributed by Derick Bailey of WatchMeCode.net*

The `.call()` and `.apply()` methods allow you to specify the context of a function as well as any parameters needed for that function. There's one more method that let's you change the context for a function: The `.bind()` method.

The `.bind()` method exists on all JavaScript functions, the same as `.call()` and `.apply()` (available since ES5). The major difference between `.bind()` and `.call()` / `.apply()`, is that `.bind()` will not invoke the function in question. Instead, it returns a new function to you. Invoking this new function will in turn invoke the original function with the context that was specified in the `.bind()` call.

In other words, `.bind()` allows you to set the context of a function without invoking it.

```
function doStuff (b, c) {
  return this.a + b + c;
}

// bind the function to a context
let myContext = { a: 1 },
    boundStuff = doStuff.bind(myContext);

// call the bound function with additional params
let result = boundStuff(2, 3);
console.log(result); //=> 6
```

Try it on CodePen

In this code, the original `doStuff()` function is "bound" to the `myContext` object as the context. This results in a new function referenced in the `boundStuff` variable. Now when the `boundStuff` function is invoked — and no matter how it is invoked — the `this` variable of the original function will always be set to the `myContext` object.

It's important to note that this is true only for invoking the `boundStuff` function, though. If you invoke the `doStuff()` method directly, you will still be able to set the context as needed.

# More on Function.bind()

*This tip was contributed by Derick Bailey of WatchMeCode.net*

Previously I showed you how to use `.bind()` to bind the context of a function to an object of your choice. Using the `.bind()` call also allows you to specify a list of parameters for the original function call to use. This is commonly referred to as *partial function application*, since you're applying a partial list of parameters to the function for later use.

## Partially Apply a Function

To use partial application, you need first to bind the function to a context. After that, passing in additional parameters to the `.bind()` call withholds those parameters and applies them when the bound function is called.

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

let addOne = add.bind(null, 1),
    result = addOne(2);
console.log(result); // 3
```

Try it on CodePen

In the above code, a generic `add()` function is set up to add two parameters. Just below that, the `add()` function is bound to a null context (since this is never used in the function) and has a value of `1` stored as the first parameter. The result of this `.bind()` call is a function that takes one parameter and applies it to the original function along with the specified context and parameter passed to the `.bind()` call.

Later, when the `addOne()` method is called, a single parameter is passed to it. Since the `.bind()` call provided the first parameter for the original function, calling `addOne(2)` passes the value of `2` to the `b` parameter of the original `add()` function. The result is then logged, with the expected value of `3`.

## Partial Application vs Currying?

Having seen partial function application, I've got a homework assignment for you. First off, watch out for people calling this technique "currying". While currying and partial function application can

look similar on the surface, they are not the same thing. Your homework assignment, then, is to look into currying a function in JavaScript.

Do a bit of research to figure out what currying is and how it differs from partial function application. Then see if you can write a small library to do function currying in JavaScript.

# The arguments Object

If you write JavaScript functions, then you know that you can pass arguments to them, and the values of the arguments have the ability to get used inside the function in some way.

While it's common to refer to the values by their variable names, you also have the option to access the arguments using the `arguments` parameter. Here's a really silly and contrived example:

```
function doSomething(zero, one, two) {
  console.log(arguments[1]); // "one"
}

doSomething('zero', 'one', 'two');
```

Try it on CodePen

As you can see, in this case it would be perfectly fine to just use the parameter name itself (one). But you can probably see how the alternative (i.e. using the `arguments` object) might be useful.

JavaScript doesn't throw an error when a function is called with too many arguments, so you might work around this by looping through the arguments, or even slicing up the `arguments` object, rather than dealing with the actual parameters themselves.

And although the `arguments` object looks and acts like an array, it's not an array. So it doesn't have any array properties or methods attached to it (`push()`, `pop()`, `slice()`, etc.) other than `length`.

And, of course, you can only reference the `arguments` object inside of a function body, otherwise you'll get an error.

# Child Nodes

There are dozens of methods and properties that let you manipulate and get info from the DOM. Let's look at four not too well known ones that have had support since IE6 but, because they were implemented with bugs in old IE, they've only been able to be useful since IE9.

Here's the HTML I'll use:

```
<ul class="mylist">
  <li class="one">Papayas</li>
  <li>Lemons</li>
  <li><span>Oranges</span></li>
  <li>Frankfurters</li>
  <li class="five">Eggs</li>
</ul>
```

And now we'll get some info from these elements using the following:

```
myList = document.querySelector('.mylist');

console.log(myList.childElementCount);
console.log(myList.children[2].innerHTML);
console.log(myList.firstElementChild.className);
console.log(myList.lastElementChild.className);
```

Try it on CodePen

Here I'm using `childElementCount` to find out how many child elements the queried element has. Then I'm using the `children` collection to get the `innerHTML` content of the third list item. I then get the class name of the first and last elements in the collection using `firstElementChild` and `lastElementChild`.

As mentioned, you can use these methods comfortably in IE9+. The main bug that occurred with these in IE6-8 was the fact that those browsers would erroneously include comment nodes as part of the results.

The main advantage of using these as opposed to the similar `childNodes`, `firstChild`, `lastChild`, etc., is the fact that these deal only with element nodes. For example if you were to use the `childNodes` property on the list element shown above, you would find there are actually 11 nodes instead of 5, because it also counts white space nodes.

More info on these on MDN.

# Augmenting Types

Here's a little tidbit from Crockford's JavaScript: The Good Parts on augmenting (adding to) the basic types in JavaScript.

Adding a method to `Object.prototype` makes the new method available to all objects. This is the same idea when adding methods to functions, arrays, strings, etc. So I can make a method available to all functions like so:

```
Function.prototype.method = function (name, func) {
  this.prototype[name] = func;
  return this;
};
```

With this, I'm able to hide the name of the `prototype` property. In *The Good Parts*, Crockford uses a few examples to demonstrate how this works, including adding an `integer()` method to `Number.prototype`:

```
Number.method('integer', function () {
  return Math[this < 0 ? 'ceil' : 'floor'](this);
});

console.log((-20 / 3).integer()); // -6
```

Here I'm calling the newly created `integer()` method on the number that results from the calculation inside the parentheses, giving me the closest integer to the original decimal number. As mentioned, this method is available to all functions, so you could do something like this:

```
console.log(function () {
  return -5.45 + 60.34;
}().integer()); // 54
```

As shown in the comment on the last line, this will print out "54". (See CodePen demo)

Crockford also mentions a few concerns with using such a technique, so if you get a chance, be sure to check out that part of the book for more info on this technique.

# Configuration Data

In his excellent book Maintainable JavaScript, Nicholas Zakas recommends externalizing configuration data. For many experienced developers, this is probably old hat. But let's look at the example he uses, so we can see exactly what this means and why it's beneficial to your applications.

```
function validate(value) {
  if (!value) {
    console.log('Invalid value');
    location.href = '/errors/invalid.php';
  }
}

function toggleSelected(element) {
  if (hasClass(element, 'selected')) {
    removeClass(element, 'selected');
  } else {
    addClass(element, 'selected');
  }
}
```

There are three pieces of data in this code that are "configuration data":

1. The string "Invalid value"
2. The URL "/errors/invalid.php", and
3. The CSS class name "selected", which is used three times.

All of these are configuration values that could change at any time.

Yes, you could change them in each of their hard-coded instances, but naturally that would not be practical, especially if there are lots of places where they appear and if they are in separate files or modules or even separate projects that have common modules.

Here is a better way to write the same piece of code:

```
let config = {
  MSG_INVALID_VALUE: 'Invalid value',
  URL_INVALID:       '/errors/invalid.php',
  CSS_SELECTED:      'selected'
};

function validate(value) {
  if (!value) {
    console.log(config.MSG_INVALID_VALUE);
    location.href = config.URL_INVALID;
  }
}

function toggleSelected(element) {
  if (hasClass(element, config.CSS_SELECTED)) {
    removeClass(element, config.CSS_SELECTED);
  } else {
    addClass(element, config.CSS_SELECTED);
  }
}
```

Although this does technically create more code if used only once, the benefits are great if these values are reused many times. The `config` object is where all such data now exists, and each of the property names has its own prefix to categorize the data.

As Zakas points out, the actual method you use and the naming conventions are a matter of preference, but the important thing is, you've removed the hard-coded values from the functions and replaced them with placeholders that need to be changed only in a single place in your code, should the need arise.

# Switch Fall-Throughs

You might be aware that the coding style for a JavaScript `switch` statement is a topic of varying opinions. One of the debated topics has to do with something called *falling through*. This is in reference to the `case` statements that are used in a `switch` construct, as shown here:

```
switch(example) {
  case 'one':
  case 'two':
    // do something...
    break;

  case 'three':
    // do something...

  default:
    // do something...
}
```

Let's assume the only cases in that switch that will have executing code will be where it says "do something..." Notice a couple of things in the code:

- There is no code executed for case "one"
- There is no "break" for case "one"
- There is no "break" for case "three"

Personally, I don't prefer code like that, so I will follow Crockford's and JSLint's suggestions to include `break` statements for each case that's not an obvious fall-through. But some people enjoy this kind of thing and they'll actually use it as a feature, rather than viewing it as a 'bad part' of JavaScript.

To understand what the lack of `break` statements in this example does, check out this CodePen demo. Notice what happens when you change the variable to one of four options:

- "one" - The code for "two" executes, because "one" falls through into "two";
- "two" - only "two" executes, which is the same result as "one";
- "three" - both "three" and "default" execute, because "three" falls through into "default".
- "four" (or anything else) - Only the "default" executes.

In his book Maintainable JavaScript, Nicholas Zakas recommends that if you allow fall-throughs like this, that you should comment your code to indicate it. Also, because the fall-through in "one" is obvious (i.e. there's no code to execute), you don't have to comment that type of fall-through.

# Interacting with a Live DOM

What do you do if you want to remove all of a specific type of HTML element from a page with JavaScript? For example, let's say you wanted to mimic something that an ad-blocker might do, removing all `<object>` elements.

Simple, you might say, just use jQuery!

```
$('object').remove();
```

It's true, that would be simple. But how would you do this with plain JavaScript? Well, I'm sure there are a few ways to accomplish this, but I immediately thought of doing this:

```
let spans = document.getElementsByTagName('span'),
length = spans.length, i;

for (i = 0; i < length; i++) {
  spans[i].parentNode.removeChild(spans[i]);
}
```

Not too bad, right? First I collect all `<object>` elements in an array, then I get the length of the collection. I then use a `for` loop to loop through all the elements, removing them from the DOM. But this doesn't work.

Here is the corrected `for` loop:

```
for (i = 0; i < l; i++) {
  objs[0].parentNode.removeChild(objs[0]);
}
```

Try it on CodePen

Notice, I'm still using the `i` variable to start the count, and I'm still using the `l` (length) variable to define how many elements to remove. But the difference is, I'm not using the `i` variable inside the loop to represent which element to remove.

As soon as I remove element "0" in the index of elements, element "1" then becomes the new element "0", and this continues with each removal. That's why the original code won't work, because I don't want to increment the index. It's a fairly small correction, but it does the trick, correctly removing each of the elements in order.

Like I said, there are probably other ways to do this, and I really have no idea what the performance issues with this technique would be if you're removing a large amount of elements, but I guess this is just a small reminder that even the simplest of tasks can be somewhat complicated by interacting with a live DOM.

# The NodeIterator API

All browsers, including IE9 and higher, support a lesser-known way for traversing the DOM: The `NodeIterator` interface. Let's look at a brief example of how it's used.

Let's say I have this HTML:

```
<div id="wrap">
  <ul>
    <li>Apples</li>
    <li>Oranges</li>
    <li>Peaches</li>
    <li>Bananas</li>
    <li>Mangoes</li>
  </ul>
</div>

<div id="wrap-2">
  <p>Paragraph <span>example</span>.</p>
</div>
```

I can iterate over all elements starting from the `#wrap` element like this:

```
let div = document.getElementById('wrap'),
    op = document.querySelector('output'),
    iterator = document.createNodeIterator(
      div, NodeFilter.SHOW_ELEMENT, null, false
    ),
    node = iterator.nextNode();

while (node !== null) {
  op.innerHTML += node.tagName + ' ';
  node = iterator.nextNode();
}
```

Try it on CodePen

Here I'm using the `createNodeIterator()` method, which takes four arguments. These are:

- The root element from which you want to start searching.

- A “bitmask” to determine which nodes to visit. In this case, I’m specifying element nodes. I could also specify “SHOW_ALL” which would also show text nodes and white space nodes, or even “SHOW_TEXT”, which shows only text nodes.
- A function that acts as a custom filter. I didn’t include a filter, so I just used “null”.
- A Boolean value that has no effect on HTML pages, so I could just leave this out.

The example code also uses the `nextNode()` method, and there’s also an opposite `previousNode()`.

There’s lots I could do with this, creating filters and using various options in the second argument. For more info on the accepted values for the second argument, check out the NodeIterator article on MDN.

# TreeWalker

In a previous tip I briefly introduced the `NodeIterator` DOM traversal interface. There's another, more advanced traversal method that I'll introduce here. It's called the `TreeWalker` interface and it too is supported in all browsers including IE9+.

`TreeWalker` has the same functionality as `NodeIterator` (including `previousNode()` and `nextNode()`), but adds some useful methods:

- `parentNode()` - Moves to the current node's parent.
- `firstChild()` / `lastChild()` - Moves to the first or last child of the current node.
- `nextSibling()` / `previousSibling()` - Moves to the next or previous sibling of the current node.

Using the same HTML as `NodeIterator` (which used a `<div>` wrapping an unordered list), here's an example:

```
let el = document.getElementById('wrap'),
    op = document.querySelector('output'),
    walker = document.createTreeWalker(
      el, NodeFilter.SHOW_ELEMENT, null, false
    ),
    node = walker.firstChild();

while (node !== null) {
  op.innerHTML += node.tagName + ' ';
  node = walker.parentNode();
}
```

Try it on CodePen

Here I'm using the `firstChild()` method to move to the `<ul>` element from the original `#wrap <div>`. Then, once inside the loop that runs through the elements, I'm moving right to the parent node using the `parentNode()` method. I could also instead use `firstChild()` or even `lastChild()` to quickly jump into the list items themselves.

You can see the display of the accessed elements on the page, which lists "UL" and "DIV", since those are the two elements I jumped to using `TreeWalker`.

MDN has more detailed info on their TreeWalker article.

# Array.some()

In another tip I briefly introduced `Array.prototype.every()`, which lets you check all the values in an array against a condition. The method returns `true` if all values meet the condition, and `false` if at least one does not; hence "every".

You also have the option to use a similar method: `Array.prototype.some()`. It's somewhat of a misnomer but it basically does the reverse of what `every()` does. Here is an example:

```
let a = [1, 3, 5, 7, 9],
    b = [1, 3, 4, 8, 9];

console.log(a.some(function (c) {
  return c % 2 === 0; // false
}));

console.log(b.some(function (c) {
  return c % 2 === 0; // true
}));
```

Try it on CodePen

Similar to the examples with `every()`, I'm using a modulus operator function as the condition to check if the arrays contain "some" even numbers. And by "some", this means "at least one".

The first example returns `false` because all the numbers in the first array are odd. The second example returns `true` because it has two even numbers, hence "some" of the array items are even numbers.

As is the case with `every()`, `some()` lets you provide an optional second argument where you tell it what value to use as `this` when executing the function.

It should also be noted that in the case of both methods, they will stop iterating over the values as soon as they determine what value to return. So in the second `console.log` example shown above, the script wouldn't get to the "8" in the array because it will return `true` as soon as it gets to the "4", which is an even number, thus meeting the required condition and then exiting the iteration.

`Array.prototype.some()` is supported in IE9+, and there is a polyfill shown on MDN's page.

# Avoiding Null Comparisons

In Maintainable JavaScript, Nicholas Zakas recommends avoiding null comparisons. That is, testing the value of a variable against `null`. This type of test can be used in a few cases, but generally should be avoided as it doesn't give you enough information about the value to let you proceed further.

Instead of testing against `null`, here are his recommendations for testing for the existence of different types of values in JavaScript:

- Use `typeof` to detect primitive values (strings, numbers, Boolean, and `undefined`)
- Use `instanceof` to detect specific object types (dates, RegExp, errors, etc.)
- Use `typeof` to detect functions (but not in IE8)
- Use ES5's `isArray()` or a custom `isArray()` method to detect arrays.
- Use the `in` operator to detect properties

All of the above could be expanded on, and he does so nicely in Chapter `8` of his book, so check it out if you haven't already.

# Array.map()

The `Array.prototype.map()` method is supported everywhere including IE9+. It's a simple and easy to understand method that adds a little extra to array manipulation. Here's some example code:

```
let myArray = [5, 20, 34, 45, 65, 98, 123],
    mapArray;

mapArray = myArray.map(function (a) {
  return a*a;
});

console.log(mapArray);
// new array:
// [25, 400, 1156, 2025, 4225, 9604, 15129]
```

Try it on CodePen

Here I have an array of random numbers, then I call the `map()` method on my array, creating a new array called `mapArray`. The `map()` method passes each of the original array's elements into the supplied function, and returns a new array containing the elements as modified by the function. In this case, I'm multiplying each element by itself.

The returned array, as mentioned, is a new array, and, unless the function itself does something to manipulate the original array, it does not affect the original array's elements. If the original array had missing elements, the new array would mirror this, having the same length and same missing elements. Also, if the number of elements in the original array changes after `map()` is invoked, the function will not act on those new values; it will act only on the elements of the original array, before `map()` was invoked.

As with other methods, `map()` takes an optional second argument that defines the value of `this` when executing the function.

Be sure to check out the MDN article, which has some useful examples along with a polyfill for IE8.

# trim()

A while back, I wrote an article outlining various cross-browser JavaScript string methods. One that I didn't mention, due to lower levels of browser support, was the ECMAScript 5 `trim()` method, shown in the example below:

```
let a = '  example  ';
console.log(a.trim()); // output: "example"
```

Just like the `trim()` function in PHP, JavaScript's `trim()` returns the value of the string with any white space removed from the beginning and end.

And as a bonus, MDN's reference offers a polyfill for IE8 and lower, should you need deeper browser support:

```
if (!String.prototype.trim) {
  String.prototype.trim = function () {
    return this.replace(/^\s+|\s+$/g, '');
  };
}
```

Another nice reminder of a simple little method that you can use that might not immediately come to mind simply because for a long time this feature wasn't supported natively by many in-use browsers.

# Mouse Event Properties

As discussed previously, when you create an instance of a mouse event, there are a number of properties exposed. For example:

```
let btn = document.querySelector('button'),
    op = document.querySelectorAll('output');

btn.addEventListener('mousedown', function (e) {
  op[0].innerHTML = '<p><b>ALT key:</b> ' + e.altKey + '</p>';
  op[1].innerHTML = '<p><b>CTRL Key:</b> ' + e.ctrlKey + '</p>';
  op[2].innerHTML = '<p><b>SHIFT Key:</b> ' + e.shiftKey + '</p>';
  op[3].innerHTML = '<p><b>Which button:</b> ' + e.button + '</p>';
}, false);
```

Try it on CodePen

Here I'm checking to see if the `ALT`, `CTRL`, or `SHIFT` key was pressed while the mouse event fired. Click the button using any mouse button (left, right, or middle) and look at the result output. Try holding down the `CTRL`, `ALT`, or `SHIFT` key, or a combination of them to see the output change.

The results are displayed by accessing the various properties on the mouse event instance (`e.altKey`, `e.ctrlKey`, etc). I'm also displaying which mouse button was pressed (identified by a number).

MDN has a full list of the readable properties for mouse events. You can also use `click`, `dblclick`, `mouseup`, and others, if they are applicable to what you're trying to read.

# Function.length

Previously I discussed the `arguments` object that's available in any function body. Using the `length` property on the `arguments` object lets you see the number of arguments that actually got passed in.

But here's another similar use of the `length` property:

```
function doSomething (a, b) {
  console.log(arguments.length); // 4
  // function body...
}

doSomething(1, 2, 3, 4);
console.log(doSomething.length); // 2
```

Try it on CodePen

Notice inside the function body, I'm logging the length of the `arguments` object. Because I called the `doSomething()` function with four arguments, the log produces `4`.

But at the bottom, outside the function, I'm also logging the `length` value of the function itself. Here the log output is "2". What's this?

The `length` property on a function tells you how many parameters the function is expecting. In this case, the function is declared with two parameters (a, b). But because JavaScript doesn't really care if we actually pass in the correct number of arguments, I'm passing in four values instead of two.

So the `length` property can be used in both of these ways — not only to find out the size of the `arguments` object (the values passed in) but also to get the number of parameters in the function definition.

Function.length on MDN

# Array.reduce() and Array.reduceRight()

All browsers, including IE9+, support the `Array.reduce()` and `Array.reduceRight()` methods. Here's an example of `Array.reduce()`, using an array of numbers:

```
let myArray   = [2, 4, 6, 8, 10],
    myReduced = myArray.reduce(function (a, b) {
      return a + b;
    }, 0);

console.log(myReduced); // 30
```

Try it on CodePen

What this does is "reduce" the original array to a single value. The first argument in `reduce()` is a function that performs the reduction operation. The second argument is an initial value that you want to pass into the `reduce()` function, which becomes the "a" variable in the example above.

In the example above, to end up with the result of "30", the basic logic goes like this:

- On first iteration, the function adds 0 + 2 (the second argument + the first array element)
- On second iteration, the function adds 2 + 4 (result of previous iteration + second array element)
- On third iteration, the function adds 6 + 6
- And so on...

So in that same example, if the second argument was "10" instead of "0", then I'd end up with a final return of "40".

The `Array.reduceRight()` method does exactly the same thing, except it goes through the array values from right to left (starting with the last item). Here are some notes on `Array.reduce()` and `reduceRight()`:

- The second argument is optional. If it's omitted, the function simply begins with the first two values of the array.
- If you call one of these methods on an array with a single element and with no second argument, it returns the single element.
- The final "reduced" value is no longer an array. In the case above, it's of type "number".

- These methods are not limited to math computations; they can reduce any kind of array and can use any kind of logic in the function to produce the final value.
- There is no optional argument to define the value of `this` when invoking the function, but you can optionally use `Function.bind()` if you need to.
- The callback function takes two further arguments: The third is the current index of the current array element, and the fourth is the array itself. So you can modify the original array inside the function if you want.

See Array.reduce() on MDN for more info.

# Number to String Conversion

JavaScript lets you convert any number to a string using the `toString()` method, but it also has three other number-to-string methods that give you a little more control over the string that results.

First there's `Number.prototype.toFixed()`:

```
let num = 6723.765128;

console.log(num.toFixed(2)); // "6723.77"
console.log(num.toFixed()); // "6724"
console.log(num.toFixed(5)); // "6723.76513"
```

This converts the number to a string with a specified number of digits after the decimal point. Notice also the rounding that takes place. When the argument is omitted, it assumes zero digits after the decimal, with rounding.

Then there's `Number.prototype.toExponential()`:

```
let num = 6723.765128;

console.log(num.toExponential()); // "6.723765128e+3"
console.log(num.toExponential(4)); // "6.7238e+3"
console.log(num.toExponential(10)); // "6.7237651280e+3"
```

Here the number is converted to a string in exponential notation. It always ends up with one digit before the decimal point, and the argument defines how many digits there will be after the decimal point. Omitting the argument will use as many digits as needed to display the number uniquely.

Finally, there's `Number.prototype.toPrecision()`:

```
let num = 6723.765128;

console.log(num.toPrecision(5)); // 6723.8
console.log(num.toPrecision(2)); // 6.7e+3
console.log(num.toPrecision(11)); // 6723.7651280
```

In this case, the number is converted to a string with the specified number of significant digits, meaning the total digits before and after the decimal. So in the first example, the last 5 digits are stripped off and it's rounded. The second example converts to an exponent to display the integer

portion (before the decimal) because the argument didn't provide enough digits. Also, a trailing "0" is added to the last example, to ensure it's 11 digits in total.

These methods have been around since ES3, so all three are supported in every browser, even back to IE6. Try them out in this CodePen demo.

# compareDocumentPosition()

There's a somewhat complex but useful DOM method called `compareDocumentPosition()` that lets you compare the position of a specific node in a document with the position of another node.

Suppose I have the following HTML:

```
<div id="one">
  <div id="two">
  </div>
</div>
```

Assuming these elements are inside a regular HTML page, I can run the following checks using `compareDocumentPosition()`:

```
let one = document.getElementById('one'),
    two = document.getElementById('two'),
    head = document.getElementsByTagName('head')[0],
    body = document.getElementsByTagName('body')[0];

if (one.compareDocumentPosition(two) === 20) {
  console.log('one contains two');
}

if (head.compareDocumentPosition(body) === 4) {
  console.log('head precedes body');
}
```

Try it on CodePen

As you can see, the `compareDocumentPosition()` method is called on one node and then another node is passed in as the only argument. The return is compared to an integer, which is something called a bitmask. A bitmask, as explained by John Resig, is "a way of storing multiple points of data within a single number".

In the above example, I'm testing the position of the "one" element in comparison to the "two" element, but I'm also checking the position of the `<head>` element in comparison to the `<body>`. This can be done with any elements, these are just examples.

The possible return values for this method are shown in the list below, with the numbers representing the values you'd test against:

- Node.DOCUMENT_POSITION_DISCONNECTED: 1
- Node.DOCUMENT_POSITION_PRECEDING: 2
- Node.DOCUMENT_POSITION_FOLLOWING: 4
- Node.DOCUMENT_POSITION_CONTAINS: 8
- Node.DOCUMENT_POSITION_CONTAINED_BY: 16
- Node.DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC: 32

You'll notice in the code example above, I checked for a value of "20", which is not in the list above. This is because a node that contains another node also precedes that node, so it gets a value of 16 + 4 (a combination of two of the above).

It's kind of bizarre, to be honest, but to fully understand this method, you just have to fiddle around with the values and do a number of different tests. I think this method could come in handy if you want to check if some user-generated or dynamically-generated HTML is well-formed, or if it complies with some set of standards. This is an old method, so browser support is excellent, including old IE.

More info: * Comparing Document Position by John Resig * Node.compareDocumentPosition() on MDN

# null vs. undefined

In JavaScript, you are most certainly familiar with the values `null` and `undefined`. Let's briefly consider the facts and quirks of these two values to demonstrate the differences between them. First here is a summary of facts on `null`:

- `null` is a language keyword used to indicate the absence of a value, or empty value.
- `null` can be used to indicate "no value" for numbers, strings, and objects.
- Oddly, using `typeof` on `null` returns the string "object". According to David Flanagan, this is supposed to indicate that `null` is a special object value indicating "no object". However, MDN's reference says this is a bug in ECMAScript.

The `undefined` value, on the other hand, can be described as follows:

- `undefined` is not a language keyword but a predefined global variable. Thus: `undefined === window.undefined; // true`
- A variable that has not been initialized has a value of `undefined`.
- An object property or array element that doesn't exist has a value of `undefined`.
- A function that has no return value will always return `undefined`.
- Function parameters for which no argument is supplied will equal `undefined`.
- In ES3 `undefined` can be set to a new value, but since ES5 this is no longer possible.
- Using `typeof` on `undefined` returns "undefined".

When comparing `null` and `undefined`, you'll notice the following:

```
1  null == undefined; // true
2  null === undefined; // false
```

As you can see, in order to differentiate between them, you have to use the strict equality operator (triple-equals), which is usually recommended for all comparisons anyhow.

Generally, `undefined` is considered an unexpected value, similar to an error, whereas `null` is usually expected. So if you have to assign the absence of value to something, you're almost always better off using `null` rather than `undefined`.

- undefined on MDN
- null on MDN

# Node.isEqualNode()

You can compare two nodes in an HTML document using the `isEqualNode()` method, supported in IE9+. The method is called on an existing node, and it's passed another node as the only argument.

I'll run some tests using the following HTML:

```
<section id="container">
  <!-- test comment -->
  <!-- test comment -->
  <!-- test comment extra -->
  <!-- test comment extra-->
  <div class="one" id="two" data-extra="three">Testing</div>
  <div class="one" id="two" data-other="three">Testing</div>
  <div id="two" class="one" data-extra="three">Testing</div>
  <div class="one" id="two" data-extra="three">Testing </div>
</section>
```

Inside the `section` element are some HTML comments and I've included three `div` elements with some text and various attributes on each. Notice the differences between the comment nodes as well as the differences between the element nodes. Now let's run some tests using `isEqualNode()`:

```
let divs = document.getElementsByTagName('div'),
    myNodes = document.getElementById('container').childNodes;

console.log(divs[0].isEqualNode(divs[1])); // false
console.log(divs[0].isEqualNode(divs[2])); // true
console.log(divs[1].isEqualNode(divs[2])); // false
console.log(divs[2].isEqualNode(divs[3])); // false
console.log(myNodes[1].isEqualNode(myNodes[3])); // true
console.log(myNodes[3].isEqualNode(myNodes[5])); // false
console.log(myNodes[5].isEqualNode(myNodes[7])); // false
```

Try it on CodePen

The first collection holds all the `div` elements, while the next collection holds all nodes, using the `childNodes` property. Let's look at the results of the tests on the `div` elements:

- `div` elements 0 and 1 are not equal because one of the `data-*` attributes is different, even though the values are the same.

- `div` elements 0 and 2 are equal, which is odd because they don't look equal. Note, however, that for all practical purposes, they are equal. The only difference is the order of the attributes, which doesn't matter to `isEqualNode()`.
- `div` elements 1 and 2 are not equal for the same reason as 0 vs. 1.
- `div` elements 2 and 3 are not equal, which is due to the single extra space in the text of the 3 element, before the closing tag.

Now what about the comment nodes?

- The comment nodes are collected using `childNodes`, which collects all nodes, including text nodes (white space), elements, and comments, so I have to reference the comments by skipping numbers in the indexes.
- The first two comment nodes (1 and 3) are the same.
- The 3 and 5 nodes are clearly different, having different text.
- The last two nodes are also different but only because of the missing space at the tail end of the last comment.

More info on MDN:

- Node.isEqualNode()
- Node.childNodes

# window.getSelection()

Here's a method you may not have seen before. It's the `window.getSelection()` method and it lets you get the contents of the selected text on the page.

Here's an example that displays the selected text on the page in response to a `mouseup` event on the document:

```
let op = document.querySelector('output');

document.addEventListener('mouseup', function () {
  op.innerHTML = window.getSelection();
}, false);
```

Try it on CodePen

In this example you can see that the `getSelection()` method is called on `window`, and then the resulting text is displayed inside the `output` element (referenced as `op`).

The selection that's returned, however, is not a string; it's an object with a whole slew of its own properties. So if you try to manipulate the selected content as a string object, you'll get an error. To do any string manipulation, you have to first convert the object to a string, then perform your string manipulation. To see the properties this method exposes on the selection object, try displaying the object in your browser's console and then you can examine the property tree.

`window.getSelection()` is available in all browsers including IE9+. For old IE support, you can check out this StackOverflow thread, which discusses some IE-only workarounds.

# getElementsByClassName()

The `getElementsByClassName()` method works in IE9+, so it's pretty safe to use in almost all projects. Here are some quick notes on this method and how you can use it.

It takes a single argument, which is the class name of the element(s) you want to target:

```
let myEls = document.getElementsByClassName('example');
```

The single argument can be more than one class, separated by spaces:

```
let myEls = document.getElementsByClassName('example other next');
```

But note that this will not target all elements that contain any one of these class names. Each element collected has to contain all three classes, at the very least. For example, if I have the following HTML:

```
<div class="example other next"></div>
<div class="other next"></div>
<div class="example"></div>
<div class="other"></div>
<div class="next"></div>
```

Note the console result if I apply to that the following JavaScript:

```
let myEls = document.getElementsByClassName('example other');
console.log(myEls.length); // 1
```

Try it on CodePen

You might think initially that the length of the `myEls` collection would be "4", but that's not the case because I'm specifically targeting elements that have both classes. As you can see, although the first element in the HTML also contains a third class ("next"), it still matches because it has both of the classes I want. I could even switch the order of the classes in the HTML, and it would still work as long as both "example" and "other" were present on the element.

And just like `getElementsByTagName()`, this method can also be applied to a specific element instead of the whole document, which forces the collection to occur in a nested portion of the document:

```
let el = document.getElementById('container'),
    myEls = el.getElementsByClassName('example');
```

This will return a collection of elements with a class of “example” that are children of an element with an ID of “container”.

MDN’s reference mentions also that you can “use methods of `Array.prototype` on any `HTMLCollection` by passing the `HTMLCollection` as the method’s `this` value.” You can see the example they use on that page, which allows you to treat the collection like a real array.

# Method Lookups

Because of the problems inherent in using JavaScript's `switch` statement, many developers and reference guides will recommend avoiding `switch` constructs and instead using something called a *method lookup* (also referred to as a *lookup table* or even *dispatch table*). I'm going to convert a `switch` statement to a method lookup, so you can see how this is done. Here's my `switch`:

```
function doSomething(condition) {
  switch (condition) {
    case 'one':
      console.log('one');
    break;

    case 'two':
      console.log('two');
    break;

    case 'three':
      console.log('three');
    break;

    default:
      console.log('default');
    break;
  }
}
```

View on CodePen

All of this is inside a `doSomething()` function, with a condition passed in. The possible values are 'cased', to define the logged value. This is cleaner than a messy `if-else` construct, but can we clean it up even more?

Here's basically the same thing in the form of a method lookup:

```
function doSomething (condition) {
  let stuff = {
    'one': function () {
      console.log('one');
    },

    'two': function () {
      console.log('two');
    },

    'three': function () {
      console.log('three');
    }
  };

  if (typeof stuff[condition] !== 'function') {
    console.log('default');
  }

  return stuff[condition]();
}
```

View on CodePen

In the demo, you can see four logs in the console, one for each valid condition and then another one to demonstrate the default condition when the argument doesn't match one of the methods.

You can see why people like this technique. It's clean, elegant, and seems a little more sophisticated without being more complex. I've only scratched the surface on this, so here are a few good resources to read up more on this subject:

- Don't Use Switch from the book *Programming JavaScript Applications* by Eric Elliot.
- Using Dispatch Tables to Avoid Conditionals in JavaScript by Josh Clanton

# The dataset Object

You've likely heard of `data-*` attributes in HTML. There are ways to access these attributes using JavaScript with some more traditional DOM methods, but you can also use the `dataset` property, available on all HTML elements and supported in all browsers including IE11+.

Here's a snippet of HTML, which is a single `div` with multiple `data-*` attributes:

```
<div
  id="myData"
  data-name="Foghorn Leghorn"
  data-type="Rooster"
  data-company="Looney Tunes"
  data-birth="1946"
  data-author-name="Robert McKimson"
></div>
```

You can see that the element has multiple `data-*` attributes, along with the ID I'll use to access it. Here's my JavaScript:

```
let myData = document.getElementById('myData');

console.log(myData.dataset.name); // "Foghorn Leghorn"
console.log(myData.dataset.type); // "Rooster"
console.log(myData.dataset.company); // "Looney Tunes"
console.log(myData.dataset.birth); // "1946"
console.log(myData.dataset.authorName); // "Robert McKimson"

myData.dataset.authorName = "McKimson, Robert";

console.log(myData.dataset.authorName); // "McKimson, Robert"
```

Try it on CodePen

The `dataset` property is not difficult to use. For each element, you're given a `DOMStringMap` object that looks like this when logged out (last line in the demo):

```
[object DOMStringMap] {
  authorName: "McKimson, Robert",
  birth: "1946",
  company: "Looney Tunes",
  name: "Foghorn Leghorn",
  type: "Rooster"
}
```

As shown in the prior code snippet and demo, you access the different `data-*` properties by converting the custom part of the name to a property name, camel-cased where necessary. So "data-name" becomes "dataset.name", "data-type" becomes "dataset.type", "data-author-name" becomes "dataset.authorName", and so on.

As shown in the last part of the code example, the `dataset` property is both readable and writeable (the `data-author-name` property was changed).

# Encoding/Decoding URIs

JavaScript has a few different methods available for dealing with URL (or URI) strings, and strings that are intended to be portions of a URI. These strings can be encoded or decoded, to ensure they are valid in the context in which they're going to be used.

Here's a simple example demonstrating the `encodeURI()` and `decodeURI()` methods:

```
let uri = 'http://example.com/archive link.html#top';

uri = encodeURI(uri);
console.log(uri);
// http://example.com/archive%20link.html#top

uri = decodeURI(uri);
console.log(uri);
// http://example.com/archive link.html#top
```

Here the URI string has, technically, and invalid character: a space. To remedy this, I encode it, allowing the illegal character to be escaped. Then I can convert it back by decoding it.

Notice that the `encodeURI()` method didn't touch any of the other characters, including the slashes and the colon. These are ignored because they are understood by the method to be meaningful in a URI.

Notice, however, that the URI contains a pound/hash sign (#), which is also ignored. If you need to encode/decode a hash symbol, you would use `encodeURIComponent()` and its opposite method:

```
let uri = 'http://example.com/archive link.html',
    uriCom = '#top';

uriCom = encodeURIComponent(uriCom);
console.log(uri + uriCom);
// http://example.com/archive link.html%23top

uriCom = decodeURIComponent(uriCom);
console.log(uri + uriCom);
// http://example.com/archive link.html#top
```

In this case, I'm encoding/decoding only the portion of the URI that gets appended to the end, which is how these latter two methods should be used.

View on CodePen

Some points to keep in mind:

- These methods replace the now deprecated `escape()` and `unescape()` methods.
- `encodeURI()` ignores ASCII letters and digits and certain ASCII punctuation characters (hyphen, underscore, tilde, etc.)
- As mentioned, `encodeURI()` also ignores the characters that can be commonly found in URIs, like the slash, question mark, at-sign, etc.
- I'm not entirely sure, but I believe browser support for these goes back to at least IE9, and maybe earlier.

For more info, here are some relevant links:

- encodeURI() on MDN
- encodeURIComponent on MDN

# Enumerable vs. Non-enumerable Properties

I've looked around for a number of definitions for enumerable vs. non-enumerable properties in JavaScript. The best I found is David Flanagan's in JavaScript: The Definitive Guide. He says: "Built-in methods that objects inherit are not enumerable, but the properties that your code adds to objects are enumerable."

The following code snippet demonstrates that objects in JavaScript can have both kinds of properties, accessible with different methods, and we can add a non-enumerable property if needed:

```
myObj = {
  color: 'blue',
  size: 'small',
  height: 10
};

console.log(Object.getOwnPropertyNames(myObj));
// ["color", "size", "height"]

Object.defineProperty(myObj, 'name', {
  value: 'ObjectName',
  enumerable: false
});

console.log(Object.getOwnPropertyNames(myObj));
// ["color", "size", "height", "name"]

console.log(Object.keys(myObj));
// ["color", "size", "height"]
```

View on CodePen

To summarize, the code above does the following:

- Creates an object with three custom properties/values
- Lists the properties in the console
- Defines a new property that's non-enumerable
- Lists the properties again

- Lists them yet again, this time with a method that doesn't have access to non-enumerable properties

Three different methods are used in this example: `Object.getOwnPropertyNames()`, `Object.defineProperty()`, and `Object.keys()`. The `Object.defineProperty()` method lets me set my own custom property, in which I can also define whether the property is enumerable.

The `Object.getOwnPropertyNames()` method is used twice, to demonstrate that it displays my custom property. Notice, however, that when we I `Object.keys()`, the non-enumerable property is not in the output.

All three of these methods were introduced in ES5, and thus are available in IE9+. The only exception is `defineProperty()`, which is available in IE8 but only on DOM elements, not on custom objects.

Much more info on MDN:

- Object.defineProperty()
- Object.getOwnPropertyNames()
- Object.keys()

# elementFromPoint()

There's an interesting DOM method called `elementFromPoint()` that returns an element that is the topmost element from the coordinates passed to the method.

Let's say I have a row of block elements on the page that are each about 100px tall with some bottom margin separating them. Here's how I might use this method:

```
// logs text of targeted element
console.log(document.elementFromPoint(50, 50).innerText);

// changes color of text of element
document.elementFromPoint(50, 50).style.color = 'lightblue';
```

The first example displays the inner text content of the element that's topmost from the coordinates "50, 50" (meaning 50px from top and left of the window). The second example targets the same element but this time the `color` property of that element is changed.

Every time I change the coordinates, I have the potential to get a different element as a result. As the name implies, only a single element is returned.

Here's a CodePen demo that uses `elementFromPoint()` to display the text content of the selected element, using different buttons to target different elements via custom coordinates.

The coordinates could be outside the document, in which case the result would be `null`. Notice also, if you fiddle with the coordinate values (or use different sized windows) in that demo, at certain pixels, the nearest element will be the `body` element. This can happen if the coordinate point is in one of the margin areas. You might have to shrink the window's width down to see the boxes on the page qualify for any of the button clicks.

This method is supported all the way back to IE5.5 and you read more about it on MDN.

# scrollTop and scrollLeft

With two simple JavaScript properties, you can get or set the distance in the non-visible parts of a scrolled element: the `scrollTop` and `scrollLeft` properties.

Let's say you have an element set to `overflow: auto`, and an element inside it forces the parent to have scrollbars. After scrolling the element, you can find out where exactly the scroll position is, vertically or horizontally, which basically tells you how much distance is out of view in the scrolled element. Here's some basic code that displays both values after a button is clicked:

```
btn.addEventListener('click', function () {
  console.log(box.scrollTop, box.scrollLeft);
}, false);
```

You can view a full example in this CodePen demo. Scroll the box, then push the button, then scroll it again, and push the button again, etc. You'll see the values change with each push of the button, as long as you adjust at least one of the scrollbar positions.

As mentioned, you can also set the `scrollTop` and `scrollLeft` distance with these properties by simply attaching them to an element and setting a value:

```
box.scrollTop = 400;
box.scrollLeft = 500;
```

Try it out in this demo which allows you to change the values using to form fields.

Some notes on these properties:

- If the element can't be scrolled, the values are 0
- You can set the value to a negative number, but that will just set it to 0
- Similarly, if you set one of them to a value higher than is possible to be scrolled, it will simply choose the highest possible value instead

These properties were originally a Microsoft creation, so they are supported in all browsers, including back to IE6.

# appendData()

The `appendData()` method is one of many lesser-known features for manipulating text nodes in the DOM. There are many alternatives to this method (for example, simply using `+=` along with `innerHTML` will often do what you want), but `appendData()` deals strictly with text content in the DOM node specified.

For example, say I have the following HTML:

```
<p>Example text #1 <button>ONE</button></p>
<p>Example text #2 <button>TWO</button></p>
```

Here I have two paragraphs, each with a text node, then a button element. I'll attach click events to each of the buttons to "append" text content using `appendData()`:

```
let pars = document.querySelectorAll('p'),
    btns = document.querySelectorAll('button'),
    text;

btns[0].addEventListener('click', function () {
  text = pars[0].firstChild;
  text.appendData('TEXT ONE ADDED ');
}, false);

btns[1].addEventListener('click', function () {
  text = pars[1].firstChild;
  text.appendData('TEXT TWO ADDED ');
}, false);
```

Try it on CodePen

Note here that for each button click, I first store the text by accessing the text node using `firstChild`. The first child of each paragraph is the text. In a real application, I might have to first test to see if the the targeted node is in fact a text node. Note also that the text node I'm altering will continue to be a single text node, which helps because this would usually be what I want.

Support for this method is IE9+ so it's pretty safe to use in most projects.

# deleteData() and insertData()

Previously, I discussed the `appendData()` method for manipulating text nodes, so let's look at two more methods for dealing with text nodes in the DOM: `deleteData()` and `insertData()`.

Here's the HTML I'll work from:

```
<p>But haven't the sweetest idea <button>DELETE</button></p>
<p>To me buzz is <button>INSERT</button></p>
```

I'll use the two methods to do two simple things with this HTML: (1) Remove the word "idea" from the first paragarph and (2) insert the word "onomatopoeia" in the second. Here's the JavaScript (assuming I've grabbed references to the targeted collections, `btns` and `pars`):

```
btns[0].addEventListener('click', function () {
  text = pars[0].firstChild;
  text.deleteData(25, 4);
}, false);

btns[1].addEventListener('click', function () {
  text = pars[1].firstChild;
  text.insertData(14, 'onomatopoeia ');
}, false);
```

Try it on CodePen

As with the previous demo, this code on its own is fairly mundane, other than to demonstrate how the methods are used.

`deleteData()` takes two arguments: the starting point of the deletion, and the number of characters to delete. The `insertData()` method takes the insertion point as the first argument and the text to be inserted as the second.

Things to be aware of:

- These are text nodes I'm manipulating, so anything you "insert" will appear as text. This means if you add HTML to the inserted text, it will be inserted as entities to appear on the page.
- Arguments that take an index will throw an error if the index doesn't make sense (i.e. "30" in a text node that's only 20 characters long).
- You'll get unexpected results when dealing with certain funky characters. See this demo for example. If you're up for it, Mathias Bynens explains the concept. Both methods supported in IE9+.

# replaceData() and substringData()

I've already covered some different methods to manipulate text nodes. Let's look at two more. They are the `replaceData()` and `substringData()` methods. As the names suggest, the `replaceData()` method replaces a portion of the text node, and the the `substringData()` method returns a specified portion of the text node.

Here's the HTML I'll use:

```
<p>Yes we have no bananas <button>DELETE</button></p>
<p>To be or not <button>INSERT</button></p>
```

And here's the JavaScript incorporating the two new methods (again, `btns` and `pars` represent collections that reference the targeted elements):

```
btns[0].addEventListener('click', function () {
  text = pars[0].firstChild;
  text.replaceData(15, 7, 'papayas ');
}, false);

btns[1].addEventListener('click', function () {
  text = pars[1].firstChild;
  text.insertData(13, text.substringData(0, 6));
}, false);
```

Try it on CodePen

When the first button is clicked, I "replace" a portion of the data in the targeted text node using `replaceData()`, which takes three arguments: the starting point of the text I want to replace, the number of characters to replace, and the data to insert in its place.

The `substringData()` method is used in this example along with the `insertData()` method. The `substringData()` method takes two arguments: the start and end points of the data I want to extract from the text node. Then I use `insertData()` to insert the substring into the same text node where I took the substring from. So in this case, the text "To be or not" becomes "To be or not To be".

If that's confusing, just fiddle around with the demo. You probably noticed that these DOM methods that manipulate text nodes are very similar to some string methods in JavaScript.

# Manipulating HTML Comment Nodes

Previously I've demonstrated five different methods that can be used to manipulate text nodes in different ways. Interestingly, every one of these methods can also be used to manipulate the text in comment nodes.

To demonstrate, here's the HTML I'll manipulate:

```
<section>
  <!-- Comment node: -->
  <!-- Comment node: -->
  <!-- Comment node: -->
  <!-- Comment node: -->
  <!-- Comment node: -->
</section>
```

Notice there are five comment nodes inside the `section` element. I'll do a separate manipulation on each one.

```
let sc = document.querySelector('section'),
    c1 = sc.childNodes[1],
    c2 = sc.childNodes[3],
    c3 = sc.childNodes[5],
    c4 = sc.childNodes[7],
    c5 = sc.childNodes[9];

c1.appendData('1 ');
c2.deleteData(1, 13);
c3.insertData(15, 'inserted ');
c4.replaceData(1, 13, 'new text ');
sc.innerHTML += c5.substringData(9, 4);
```

CodePen demo

Clearly this isn't efficient code. It would be better to loop through the elements looking for comment nodes by testing that a given node's nodeType is equal to "8". In this case, I'm just selecting the nodes I know are comment nodes, which is every other node, accounting for white space nodes.

Of course, you won't see any of these comments visibly, since comments exist only in the source. But you can view the result of the above actions by inspecting the page with your browser's developer tools or using some other method to view the generated source.

The HTML from above, after the code is run will look like this:

```
<section>
  <!-- Comment node: 1 -->
  <!--  -->
  <!-- Comment node: inserted -->
  <!-- new text  -->
  <!-- Comment node: -->
  node
</section>
```

Notice that I've moved some of the text from one of the comments to the actual HTML, so that text ("node") will be visible on the page.

# Array.forEach()

If you're like me, you're probably still accustomed to writing your loops using a traditional `for` loop. But I keep having to remind myself that, in many cases, the ES5+ `array.forEach()` method is often the best choice.

The `forEach()` method is supported in IE9+ and is similar to jQuery's `each()`. Here's an example of its use:

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

arr.forEach(function (v, i, a) {
  console.log(`${i} :  ${v} - from array: [${a}]`);
});
```

View on CodePen

That's a fairly contrived example, but it's to help understand how this feature works. Here's a bullet rundown to explain it:

- The lone required argument is a callback function to be executed once on each array item.
- The function passes three arguments (current element, index, and the array itself).
- `forEach()` takes an optional second argument that allows you to define the value of `this` in the function.
- If the array is modified, the third argument will reflect the changes, however, the loop will visit only the elements that were part of the array before execution began.

In the example above, I'm outputting the index, the current value of the array item, then I'm stating where the item came from by outputting the full array (which doesn't get modified in this case), just to demonstrate the basics.

The real power of this method becomes clear when you are modifying the array on the fly or when you are providing a custom value for the `this` keyword. And of course, the syntax is much nicer than a standard loop.

Also, note that in my example, I'm using an anonymous function inside the `forEach()` method, but you're probably better off doing something like the following:

```
function doSomething(a, b, c) {
  // code here...
}

let arr = ['one', 'two', 'three', 'four'];

arr.forEach(doSomething);
```

This allows your function to be reused elsewhere and makes the whole thing much cleaner.

MDN has lots more detailed info and examples, as well as a polyfill for older browsers.

# The text Property on Script Elements

I recently discovered the `text` property that is available on any `script` element in an HTML page. I'm surprised I haven't come across this before, and maybe you're in the same boat.

Let's say I have the following HTML:

```
<script class="sc">
  Example text inside a script element.
</script>

<script class="sc" src="//code.jquery.com/jquery-1.9.1.min.js"></script>

<script class="sc" src="//code.jquery.com/jquery-1.9.1.min.js">
  <!-- comment test -->
  <p>Example text inside a script element.</p>
</script>
```

Here I have three `script` elements – two with a `src` attribute set. I've also added a class to each, so I can reference them as a collection.

Notice that two of the `script` elements have some content included, the last one also including an HTML comment and paragraph tags. The `text` property for `script` elements allows me to grab the contents of the `script` element, similar to how the `textContent` property works on regular elements. In his Definitive Guide, David Flanagan says:

> "This makes a script element an ideal place to embed arbitrary textual data for use by your application."

I've heard of developers using hidden `textarea` elements for this sort of thing, but not a `script` element.

So my code to grab the text might look like this:

```
let scripts = document.querySelectorAll('.sc');

console.log(scripts[0].text);
console.log(scripts[1].text);
console.log(scripts[2].text);
```

Of course, I could store those values wherever I want, but in this case I'm just outputting them to the console. See this demo to test it out, and notice that in that demo, I'm also setting the text content for the second `script` element, then outputting that after I've set it, showing that the `text` property is readable and writable.

Here's a description of some of my findings regarding this element:

- Flanagan seems to imply that the `text` property is available only on `script` elements that don't have a `src` attribute set. My demo disproves that in the three main browsers, however what he seems to be implying is that it would only make sense to do this sort of thing if the element has no source defined.
- If you plan to use a `script` element to store data, Flanagan recommends that you set the type attribute to something like "text/x-custom-data", to tell the browser's JavaScript interpreter not to try to execute the script. The demo I've linked to does this, to prevent the browser from throwing an error.
- The spec says this property "must return a concatenation of the contents of all the Text nodes that are children of the `script` element (ignoring any other nodes such as comments or elements), in tree order", however, all 3 browsers return the paragraph element in my demo and Chrome also returns the comment.
- Bugs aside, I believe this property is supported in all browsers, even back to old versions of IE.

# More Weekly Tips!

If you liked this e-book, be sure to subscribe to my weekly newsletter Web Tools Weekly. 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.

# 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, speaks at conferences, and does editing and writing for various web and print publications.