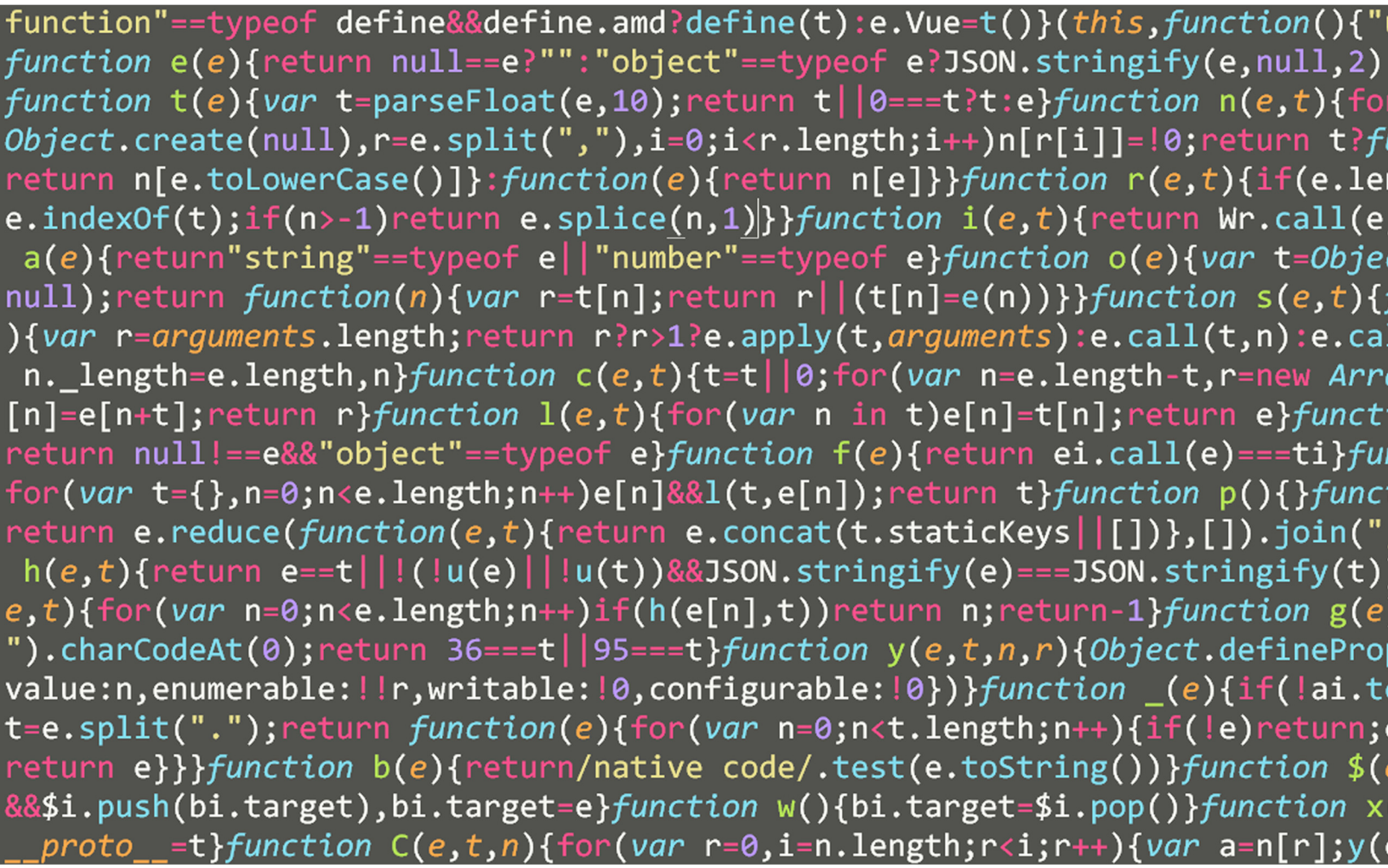
JavaScript & DOM

Tips, Tricks, and Techniques

No. 2



Louis Lazaris

Quick Tips for all Levels of Developers

JavaScript & DOM Tips, Tricks, and Techniques (Volume 2)

Quick Tips for all Levels of Developers

Louis Lazaris

This book is for sale at http://leanpub.com/javascriptdom2

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, similar to what you're going to read in this book.

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 James Hibbard for his two contributions to this second 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!

offsetParent

The `offsetTop` and `offsetLeft` properties are read-only properties available on every element on a page. These tell you the distance of an element's top-left corner from its nearest `offsetParent`. The `offsetParent` is determined by positioning in CSS and potentially due to the presence of an HTML table cell. If that's too theoretical and confusing, maybe an example will help.

Suppose I have two `div` elements and a table with a single cell. Each `div` has a nested `div` and so does the table cell. I want to figure out the `offsetTop` or `offsetLeft` values of those three child elements. Here's how the JavaScript might look:

```
child.innerHTML = child.offsetLeft;
child2.innerHTML = child2.offsetLeft;
t_child.innerHTML = t_child.offsetLeft;

child.innerHTML += '<br>' + child.offsetParent.id;
child2.innerHTML += '<br>' + child2.offsetParent.id;
t_child.innerHTML += '<br>' + t_child.offsetParent.id;
```

I'm using the above snippet of code in this CodePen demo. The "child" variables represent each of the child divs, including the one inside the table cell. In the CSS, I've varied the `position` property for the three parent elements, so you can see different results. Try changing the `position` value of any of the elements in the CSS to see the offset values change.

The first three lines of code above print out the left offsets in pixels, and the last three lines identify the element that qualifies as the "offsetParent". You can see, due to the different position values, the `offsetParent` varies – first it's the natural parent, then it's the `body` element, then it's the table cell.

Here's a summary with some further points:

- Without any relative positioning in the CSS, the `offsetParent` is almost always the `body` element.
- The exception is when an element is inside a table cell, in which case the cell is the `offsetParent`.
- If an element is set to `display: none`, its `offsetParent` value is `null`.
- `offsetLeft` and `offsetTop` are always relative to whatever element qualifies as the `offsetParent`.
- In WebKit and IE, an element set to `position: fixed` will have a `null offsetParent` value, but in Firefox this is not the case. This bug was reported back in 2008, but still isn't fixed as of this writing.
- See more on all three properties in the spec.

Best to fiddle with these values in the demo, and also test with different window sizes, to get a better grasp of how these work.

window.scrollBy()

Here's a DOM method that's really simple, is supported in all browsers including old IE, but doesn't seem to have been added to the spec until recently: `window.scrollBy()`.

Let's look at some example code. Let's assume I have a really long page of content with two button elements:

```
let btn = document.querySelector('button'),
    btn2 = document.querySelectorAll('button')[1];

btn.addEventListener('click', function () {
  window.scrollBy(0, window.innerHeight);
}, false);

btn2.addEventListener('click', function () {
  window.scrollBy(0, -(window.innerHeight));
}, false);
```

Try it on CodePen.

If you click the first button, the page scrolls down. If you click the second button, it scrolls up. If there's no room to scroll, it will do nothing. The `scrollBy()` method takes two arguments, a number that's a pixel value to scroll horizontally (x-axis) and one to scroll vertically (y-axis, which is what you'll normally use).

In this case, I'm using the value of the window's `innerHeight` property to allow the buttons to scroll up and down exactly one page. As you can see in the second event listener, the arguments can be negative, to scroll either up (for vertical) and left (for horizontal).

You can find some details on `scrollBy()` in the CSSOM View spec and on the Dottoro reference.

cloneNode(deep)

You might know you can create nodes in the DOM using `createElement()`. But don't forget that you can also clone existing elements using the `cloneNode()` method, along with its optional argument. Let's quickly review how `cloneNode()` works, what the lone argument is for, and some things to be aware of.

Here's some example HTML:

```
<div class="box">
  <p>Example text.</p>
</div>
```

I can clone the parent `.box` element in two ways, shown below:

```
let box = document.querySelector('.box'),
    boxClone = box.cloneNode(),
    boxClone2 = box.cloneNode(true);

document.body.appendChild(boxClone);
document.body.appendChild(boxClone2);
```

Try it on CodePen

Notice the second line clones the box without the optional argument. This tells the browser to clone and subsequently append only the parent element, without its children. But if I specify the Boolean `deep` parameter, I tell the browser to clone the element along with its children. So if you look at the demo, you'll see that the two appended elements are different: One is empty, but retains its styles attached to the "box" class, while the other has the styles in addition to the inner paragraph content.

So keep in mind that if you're creating elements, it might be much easier to use `cloneNode()`. This way you can take an existing, already styled element and insert that, instead of an element that you have to apply various attributes to. It all depends on what we're trying to accomplish.

Some further notes on `cloneNode()`:

- Remember, if you clone an element that has an ID, you will also be cloning the ID, causing the document to become invalid. So if you clone such a node, you should also modify the ID before appending the new node.
- You could use a value of "false" just to be explicit about not including the children, which I suppose might be easier to read, but this would be superfluous.

- Interestingly, according to Peter-Paul Koch, the default value should be "true" and he claims Firefox supports this, however, I don't see this happening in Firefox, besides the fact that this would be awful for backwards compatibility and should not change in browsers in the future. (It's possible his page is out of date, or maybe I'm misunderstanding.)
- On void elements (e.g. `img`, `meta`, etc.), the argument makes no difference, since these elements can't have children.
- Regarding what's copied, MDN explains: "Cloning a node copies all of its attributes and their values, including intrinsic (in–line) listeners. It does not copy event listeners added using `addEventListener()` or those assigned to element properties (e.g. `node.onclick = fn`)."

importNode()

In addition to the `cloneNode()` method that I discussed previously, there's also a method called `importNode()` that lets you clone a node from an external document (e.g. one embedded in an iframe in the current document). For example, I could have HTML that looks like this in a file called import.html:

```
<div id="example">
  <p>This is some example text to import.</p>
</div>
```

Then I have the following HTML in my primary document:

```
<iframe src="import.html"></iframe>

<div id="container"></div>
```

A script running on the latter document can use `importNode()` like this:

```
let iframe = document.getElementsByTagName('iframe')[0],
    frameDoc = iframe.contentWindow.document,
    frameNode = frameDoc.getElementById('example'),
    imported = document.importNode(frameNode, true);

document.getElementById('container').appendChild(imported);
```

You can view this in action in this CodePen, which imports from another CodePen demo, to keep the origin of the imported node the same.

Grabbing nodes from external documents in this way doesn't seem to be difficult. From what I can see, `importNode()` isn't always necessary but apparently there are `ownerDocument` issues that are in play, which you can read about in this FAQ by the W3C.

Some notes:

- As shown, the first argument for `importNode()` is the node being imported. The 2nd argument is the "deep" Boolean, indicating if you want a "deep" copy or just the parent (same as with `cloneNode()`, discussed previously).
- You should always include the "deep" argument because, evidently, the default value has changed, so you'll get mixed results in old vs. new browsers if you omit the second argument.

- The import action doesn't remove or otherwise affect the original node.
- Supported in IE9+.

See MDN's article for a little more info.

A Note About appendChild() and insertBefore()

Here's a really quick tip about using `appendChild()` and `insertBefore()`: If you use either of these methods on an element that's already been added to the document, the element will be removed from its place and put into the new place.

To demonstrate, here's some HTML:

```
<div class="box1"></div>
<div class="box2"></div>
```

Then, in the CSS, I have this:

```
.box1 div {
  width: 100px;
  height: 100px;
  background: firebrick;
}

.box2 div {
  width: 100px;
  height: 100px;
  background: orange;
}
```

Here the different background colors ensure that I can visually differentiate between a `div` inserted into `box1` compared to a `div` inserted into `box2`. Now I can test it out:

```
let b1 = document.querySelector('.box1'),
    b2 = document.querySelector('.box2'),
    el = document.createElement('div'),
    btn = document.querySelector('button');

b1.appendChild(el);
btn.addEventListener('click', function () {
  b2.appendChild(el);
}, false);
```

Try it on CodePen.

Use the button in the demo to “re-append” the already appended element.

This might seem pretty trivial, but with some planning you can see how it might come in handy. If you know you have to add and remove an element, then eventually add the same element again, this saves you the trouble of removing it. Just add it again wherever you want, and it will automatically be removed, as shown above. It all depends on what you’re trying to do, but in some instances this might save you a step when manipulating the DOM.

getClientRects()

In a previous tip, I discussed the `getBoundingClientRect()` method, which exposes some interesting properties. Mainly this is used to check the position or dimensions of a block-level element on the page.

But there could be a rare case where I want to get the full width of an inline element. That is, the width of the element as if it were not broken up into multiple lines. If I use `getBoundingClientRect()`, I'll get the full width of the element as it appears on the page, but that might not be what I want.

Instead, I can use `getClientRects()`, a related method that exposes the same properties as `getBoundingClientRect()`, but in this case the method will return as many CSS border boxes as the element contains, with properties for each. A block element will only contain one border box, but an inline element that spans multiple lines can contain two or more.

So, using `getClientRects()` I can construct a function that gets the expanded width of the inline element, as if it were stretched out on a single line:

```
function getInlineWidth(el) {
  final = 0;
  for (i of el.getClientRects()) {
    final += i.width;
  }
  op[0].innerHTML = final;
}
```

View on CodePen

In the demo page, click one of the two buttons to display the width of each of the two example `span` elements in the paragraph. Notice that the values change slightly if you adjust the frame width. This seems to be caused by the way the browser is wrapping the words, with slight variation in their position on the page.

As mentioned, `getClientRects()` returns a collection of all the border boxes associated with that element. So if the `span` element wraps onto four lines, there will be four boxes. In the function, I'm looping through each of the elements and adding up the widths of all the boxes to get the full width.

There probably will not be many cases where you'll need to get such a value, but this is one way to do it. I'm not sure if there is another way, but that's what I've come up with.

See the `getClientRects()` article on MDN for more info.

Array.indexOf()

If you've done any kind of string manipulation in JavaScript, then you've likely used `indexOf()`. Since ES5, `indexOf()` is also available for use on arrays.

Here's an example, with the outputs to the right of each log:

```
let myArray = ['one', 'two', 3, 'four', 'five'];

console.log(myArray.indexOf('two')); // 1
console.log(myArray.indexOf('four')); // 3
console.log(myArray.indexOf('six')); // -1
console.log(myArray.indexOf('3')); // -1
console.log(myArray.indexOf(3)); // 2
console.log(myArray.indexOf('one', 1)); // -1
```

View on CodePen

As with `String.indexOf()`, the return value is either the index of the queried object within the array or else `-1` if the object is not found.

Here are some notes on this method:

- The search and comparison is done using strict equality. You can see this in the two examples that search for "3" and 3. Only one matches because it's a number in the array, not a string.
- Just like `String.indexOf()`, you can provide a second optional argument to define where to start searching. The last line in the code above can't find "one" because it starts searching at the second slot in the array (1). The default, of course, is to start searching at 0.
- Remember that this method returns as soon as it finds the item being searched for, so this wouldn't be good to use if you have to find all elements – unless you utilize the second argument to allow a function to keep searching (or something else along those lines).
- This is supported everywhere, including IE9+, as are most other ES5 features. MDN has a polyfill if necessary.
- There's also `Array.lastIndexOf()`, which starts searching at the end of the array, with the same browser support.

script.onload

You've likely heard of using the `onload` event handler on the `window` object, but you can also use it on a `script` element. So you can do something like this:

```
let myScript = document.createElement('script');
myScript.src = 'https://code.jquery.com/jquery-1.9.1.min.js';

myScript.addEventListener('load', function() {
  console.log('jQuery loaded.');
}, false);

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

View on CodePen

If you're creating and inserting script elements dynamically, this could come in handy to check if any given script has loaded.

And, of course, you can also do this using the old-school single line event handler:

```
myScript.onload = function () {
  // do something...
};
```

Document.createComment()

The `Document` interface allows for the creation of all sorts of stuff you might not normally think could be created. And, admittedly, many of them are probably of little use. Nonetheless, one example is the `Document.createComment()` method. This does exactly what the name says — it creates an HTML comment that you can then insert into the page wherever you want.

Again, I'm not entirely convinced this is of much use, but crazier things have been found to be of value. Here's an example:

```
let comment = document.createComment('comment text');
console.log(comment.textContent); // produces: "comment text"
document.body.appendChild(comment);
```

View on CodePen

The method itself returns the full comment object that was created, demonstrated in the line that logs its info. I'm using the `textContent` property to display the text I inserted into the comment. I could also check the `nodeType`, reference its parent node (although that would be `null` at this point), or do all sorts of other things I could do with any HTML element.

The last line is where I actually add the comment to the page. You can inspect the live DOM in the page in the demo to see the inserted comment. You have the option to use `appendChild()`, as I did, or maybe `insertBefore()` or something else. It's just like inserting any other node.

Support for this goes back to IE6, so it's safe to use. There's a little more info on `createComment()` on the Dottoro reference.

previousElementSibling / nextElementSibling

In a previous tip I discussed:

- `firstElementChild`
- `lastElementChild`

Which are DOM properties that let you select elements (excluding text nodes and comment nodes) based on being first/last elements in their parent.

Similarly, here are two other properties:

- `nextElementSibling`
- `previousElementSibling`

The meanings should be clear from the names; you're accessing either the previous or next element (siblings) in relation to the specified element.

For example if I have two `section` elements with differing types of content, I could do the following checks (after caching the elements in their respective variables):

```
let s1 = document.getElementById('s1'),
    s2 = document.getElementById('s2');

console.log(s1.nextElementSibling.nodeName); // Node name of s2
console.log(s2.previousElementSibling.offsetWidth); // width of s1
```

View on CodePen

By reviewing the demo you can see the HTML in each element and see the results. Pretty straightforward. Just a simple way to select the "next" or "previous" element in relation to the specified element. If the previous or next sibling doesn't exist, the property will return `null`.

Like `firstElementChild` and `lastElementChild`, these properties are supported everywhere including IE9+.

Getting Image Widths

There are a few different DOM features that let you get the width of an image via JavaScript. Three of them you've probably seen before but the fourth might be new to you.

First of all, let's say my image is in the HTML like this:

```
<img src="https://placekitten.com/g/400/600" width="300" height="450">
```

Notice that the image is grabbed from a placeholder service (placekitten), and I've specified the size of image that I want. On top of that, however, I've changed the dimensions of the image, scaling it down proportionally. This will illustrate one of the features I'm going to use in my JavaScript.

But before I do that, I'll also change the image's size in the CSS by one pixel both ways:

```
img {
  width: 299px;
  height: 449px;
}
```

So now I have three different width and height values, all of which can be gleaned from the DOM. If I have a variable called `myImg` that holds a reference to our image, I can obtain the width in different ways:

```
let c = getComputedStyle(myImg).getPropertyValue('width'), // 299px
    w = myImg.width, // 299
    a = myImg.getAttribute('width'), // 300
    n = myImg.naturalWidth; // 400
```

Here's a CodePen that uses all of these techniques and displays them on the page. The same results are shown in comments in the code above.

Here's a description of each one:

- The `getComputedStyle()` method lets you get any property value, and this is the final computed style, whether it's in the CSS or not. Note that if I used ems or another unit, it would still spit out the value in pixels.
- The second method is accessing the `width` property of the image directly, which basically gets the same result, but without the unit. Again, this is the computed width, not the original or that in the HTML.

- The `getAttribute()` method allows you to get exactly what's in the HTML. This was overridden in my CSS, but I can access the original width with this.
- Finally, I have probably the least-known technique, using the `naturalWidth` property. This gets the original width of the element, regardless of what's in the HTML or the CSS. There's also the complementary `naturalHeight` to go along with that.

So that just about covers every possible way to get the dimensions of any image on a web page with JavaScript.

The void Operator for IIFEs

There are a lot of sources discussing the `void` operator in JavaScript and I'm sure we've all seen it. It comes in handy in bookmarklets, in `href` URLS (though it's not recommended to do that), and for ensuring that you're returning `undefined` and not a shadowed version of `undefined`.

But there's another interesting use pointed out on MDN and in a comment on an article on `void` (see references below). It looks like this:

```
void function() {
  // function body here...
}();
```

View on CodePen

MDN explains: "When using an immediately-invoked function expression, void can be used to force the function keyword to be treated as an expression instead of a declaration."

From what I understand, this is basically an alternative to the usual IIFE methods, but with arguably a simpler syntax – potentially making it easier to read and understand what's happening at a glance. But interestingly, in *JavaScript: The Good Parts*, Douglas Crockford says `void` in JavaScript is confusing and not consistent with how `void` is used in other languages, so he recommends to avoid it (no pun intended).

Nonetheless, good to know what it does and how it can be used, so check out the resources below for more:

- void on MDN
- The void operator in JavaScript on 2ality (See first comment by Jonathan Buchanan)
- JavaScript's void Operator on A Drip of JavaScript

Function as Second Argument for String.replace()

JavaScript's `String.replace()` method is pretty powerful when used with regular expressions. As in the following example, you can pass a regular expression as the first argument to determine what part of the string should be replaced:

```
let myString = 'EXAMPLEstring',
    myNewString = myString.replace(/[A-Z]/g, '0');

console.log(myNewString); // "0000000string"
```

In the above example, I'm replacing each uppercase letter with a zero, and the result of the log is in the comment in the code.

But maybe you didn't know that the `replace()` method can take as its second argument a function instead of the string you want to insert in place of the found uppercase letters. For example:

```
function replaceFunc (a, b, c) {
  console.log(a, b, c); // will log for each match
  return a.toLowerCase();
}

let myOtherString = myString.replace(/[A-Z]/g, replaceFunc);
console.log(myOtherString); // "examplestring"
```

Try it on CodePen

Notice the `replaceFunc()` function that's called as the second argument. This allows you to not only find a specific match, but to manipulate each match and then return a new value after dealing with the found substring. In this case, I'm just pointlessly converting each matched uppercase letter to lowercase, which results in the entire string being logged in lowercase on the final line. There are much more powerful things you can do here, but this should serve to demonstrate how this feature works.

Here are some notes on this technique:

- Notice my function expects three arguments. This could be more, depending on how many "capture" groups I've included in my regular expression. I don't have any, so the function doesn't expect any more than three arguments. The capture groups would be arguments 2, 3, 4, etc.

- The first argument (a) is always the full text of the match.
- Since I have no capture groups, the second argument (b) is the zero-based index of the match within the string.
- The final argument (c) is the full text of the string being searched.
- As shown in the demo, the match is global (using the "g" identifier in the RegEx), and three arguments are displayed in each of the 7 logs.

This feature certainly adds a lot of power to the `replace()` method. There's more info on this on the MDN page.

DOM Collections

When working with web content, sometimes you'll want to collect specific elements into an object and then manipulate them in some way. You can use loops with conditionals or similar techniques, but sometimes the simplest method is to simply use a DOM property:

```
let d = document;
console.log(
  d.links,
  d.anchors,
  d.styleSheets,
  d.embeds,
  d.forms,
  d.images,
  d.plugins,
  d.scripts
);
```

View on CodePen

You might be familiar with some of these already (e.g. `document.forms` is pretty well known) but notice you can also easily collect images, embeds, plugins, scripts, and more. The demo logs the length of each collection, but you can access any property on those elements using these features.

Some things to note:

- The `links` property collects both `<a>` elements and `<area>` elements; basically anything that has an `href` attribute, but not an element on which `href` is not valid (e.g. a `<div>`).
- The `embeds` and `plugins` properties were added in the HTML5 DOM spec but were originally only in Microsoft's documentation.
- The `anchors` property collects all `<a>` elements that have a `name` attribute, which is now a deprecated method for doing in-page anchors. This one is not included in the HTML5 DOM spec, but it will still work for compatibility reasons.

For more info, see the Document Interface in the HTML5 spec as well as the Document object on MDN.

Colors with getComputedStyle()

In previous issues, I've mentioned using `getComputedStyle()` and `getPropertyValue()` to obtain computed style information on any element on the page, regardless of where those styles have been defined (a stylesheet, by the browser, inline styles, etc).

If you're doing this, one thing you'll have to deal with is the fact that color values obtained in this manner are always returned in `rgb()` format, rather than the more commonly used hex format.

The following would get the background color of a `#box` element:

```
let box = document.getElementById('box'),
    clr = window.getComputedStyle(box).getPropertyValue('background-color');

console.log(clr); // "rgb(68, 68, 68)"
```

Try it on CodePen

Looks to me like all browsers return the same `rgb()` format. But interestingly, according to an older Smashing Magazine article, at that time Firefox seemed to be the only browser to return colors in `rgb()` instead of hex.

Personally, I don't think it matters either way. The most important thing is that all browsers handle it the same way. Then you can deal with the returned value in a standard manner, without the need to convert it for other browsers.

Couple things to note:

- If you define the value as `rgba()`, the returned result will, of course, be `rgba()`.
- If the `rgba()` value has a transparency value of "1" (i.e. "fully opaque"), you'll get `rgb()` as the value with `getComputedStyle()`.
- If you define the color in `hsl()` or `hsla()`, you get the same kind of results: `rgba()` if there is a transparency setting, but `rgb()` if there is no transparency, or if it's set to fully opaque.

event.target vs. event.currentTarget

When an event is triggered on an element, two properties help to identify the element: the `target` property and the `currentTarget` property. Assuming I have a `.parent` element with a `.child` element inside it, look at the following code:

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

parent.addEventListener('click',function(e) {
  op.innerHTML = `The target element is: ${e.target.className}<br>The current target\
 is: ${e.currentTarget.className}`;
},false);
```

Try it on CodePen

In the demo, instead of logging the results, I'm adding them to the page. Notice that there is a different result when you click on the parent element compared to clicking on the child. This is a simple way to demonstrate the difference between `target` and `currentTarget`. To explain:

- `event.target` is the element that triggered the event. In this case, it would be whichever element you clicked on. So the result would log either "parent" or "child", depending on where you click.
- `event.currentTarget` is the element on which the event listener is being processed. So in this case, it will always be the "parent" element, even if you click on the "child" element.

More on this on MDN's currentTarget article.

preventDefault()

The `preventDefault()` method is used to keep elements on the page from firing the default event associated with a particular action. The two most common uses for `preventDefault()` are to prevent a URL from being visited when a link is clicked, or for preventing form submission. This was formerly done with `return false`, but that's now deprecated.

But `preventDefault()` can be used to intercept the default event on any action that has an associated default event. Three examples are shown in the code below:

```
let radio = document.querySelector('input'),
    checkbox = document.querySelectorAll('input')[1],
    textarea = document.querySelector('textarea');

radio.addEventListener('click', function (e) {
  e.preventDefault();
}, false);

checkbox.addEventListener('click', function (e) {
  e.preventDefault();
}, false);

textarea.addEventListener('keypress', function (e) {
  e.preventDefault();
}, false);
```

Try it on CodePen

As you can see in the code and the demo, the first example is a radio button. Even though it's not disabled with the HTML `disabled` attribute, it's not able to be selected because the default event (i.e. selecting it) is prevented from occurring.

The checkbox is the same, except it starts out checked, so you can't uncheck it unless you disable JavaScript. Finally, the `textarea` element is essentially disabled because the default action that occurs on `keypress` is to type whatever key is pressed. But that too is prevented. You can, however, paste text in, which is not prevented.

In the case of user input, this is useful if you want to prevent only certain keys from being pressed. MDN gives an example script where a user cannot type any uppercase characters, so the `preventDefault()` method fires on `keypress` if an uppercase letter is typed.

Using Array.splice()

If you're not too familiar with JavaScript's `Array.prototype.splice()` method (notice that's *splice*, not *slice*), here are three different operations you can perform with it when manipulating arrays:

To add an array to an item at a given index:

```
let fruits = ['apple', 'orange', 'banana', 'pear'];
fruits.splice(3, 0, 'kiwi');
console.log(fruits);
// ["apple", "orange", "banana", "kiwi", "pear"]
```

By using `0` for the second argument, I'm telling the JavaScript engine to delete zero items from the array, but to add the specified array item into the third slot (between "banana" and "pear". The key in this case is the `0` value.

To remove an item located at a given index:

```
let cars = ['porsche', 'ferrari', 'lambo', 'maserati'];
cars.splice(2, 1);
console.log(cars);
// ["porsche", "ferrari", "maserati"]
```

You'll notice two differences from the previous code. The second argument is greater than zero (which defines how many items to remove) and there's also no third argument. This means I'm removing an item but I'm not adding. So the item at index 2 gets removed and nothing takes its place. The key here is the missing third argument, which is optional.

To replace an item with one or more items at a given index:

```
let planets = ['mars', 'saturn', 'jupiter', 'pluto'];
planets.splice(3, 1, 'uranus', 'mercury');
console.log(planets);
// ["mars", "saturn", "jupiter", "uranus", "mercury"]
```

In this last example, I've somewhat combined the previous two. The second argument defines one item to be removed from index 3, but I'm adding two new items at the same spot.

That's the `splice()` method in a nutshell. See MDN's docs for more info and here's a CodePen demo to play around with these examples.

The form.submit() and form.reset() Methods

Here's something a little quirky with regards to forms. As you probably know, when a form is submitted, the `submit` event is fired. This allows developers to use the `onsubmit` event handler to check when a form is submitted and temporarily intercept the submission while data is validated. The same is also true of the reset button on forms (which you should rarely, if ever, use); the reset action can be temporarily stopped using the `onreset` handler.

But maybe you didn't know that both these events can be mimicked using the `submit()` and `reset()` methods, respectively. Something like this:

```
document.querySelector('form').reset();
document.querySelector('form').submit();
```

In this case, the first form found on the page will be reset, and then submitted.

But what's strange about these two methods is that neither of them triggers the `submit` or `reset` events. Look at the following example code. This assumes I have a form with a submit button and then a separate unrelated button outside of the form:

```
let myForm = document.querySelector('form'),
    myBtn = document.querySelector('button');

myForm.addEventListener('submit', function (e) {
  console.log('submitted');
  e.preventDefault();
}, false);

myBtn.addEventListener('click', function () {
  myForm.submit();
}, false);
```

Try it on CodePen

Try first hitting the "SUBMIT" button in the demo. Notice the log outputs the fact that the `submit` event has been fired. The other button is the separate non-form element. In the code, I'm triggering the `submit()` method on the form when that button is clicked. Notice that the form submits (shown by the fact that the page reloads) but the console doesn't display the "submitted" message. This is because the `submit` event was not fired, even though the `submit()`method was.

To be completely honest, I'm not 100% sure why this happens but it seems to have been in place for quite some time in old browsers so my guess is the behaviour is still present for legacy reasons. Some notes I uncovered:

- While MDN explains it correctly, the DOM 2 spec incorrectly says that the behaviour is the same.
- The updated WHATWG spec has a full "Form Submission Algorithm" that makes more of a distinction but it's as clear as mud in my opinion.
- This problem has caused developers to file related bugs with the jQuery team, although one commenter seemed to say that jQuery made a correction in a later version to deal with this.

Regardless, it's good to know that there's a difference between the `submit()` method and the natural way that forms are submitted, in case you're considering using `submit()`.

Converting an Array-like Object to an Array

There might be other ways to do this but here I'll quickly demonstrate three different ways to convert an array-like object (e.g. an element collection) into a true array, which will allow me to use native array methods on it.

The array-like object will be a collection of the list items on a page:

```
let list = document.getElementsByTagName('li');
console.log(Array.isArray(list)); // false
```

As you can see from the log, my list object is not an array (yet).

I'll convert it to an array as follows:

```
let listArray = Array.prototype.slice.call(list);
console.log(Array.isArray(listArray)); // true
```

That's one way to do it, utilizing `slice()` and `call()` and passing in the list object. Now the `isArray()` method returns `true`.

I can also use a function, like this:

```
function toArray(obj) {
  let array = [];
  for (let i = obj.length >>> 0; i--;) {
    array[i] = obj[i];
  }
  return array;
}

let listArray2 = toArray(list);
console.log(Array.isArray(listArray2)); // true
```

Same result as previous, with a little more code.

Finally, if you don't need old IE support, it's possible to use the ES6 `Array.from()` method, like this:

```
let listArray3 = Array.from(list);
console.log(Array.isArray(listArray3)); // true
```

This method works in Firefox 32+ and Chrome 45+, but you can use this polyfill if you need support in earlier versions or in IE. Here's a CodePen demo with the above examples.

document.designMode

There's a property called `designMode` on the `document` object that allows you to make an HTML document editable, the same way that an element can be editable using the `contenteditable` attribute.

I'm going to use this property on an `iframe` element. Here's the HTML to display my iframe:

```
<iframe src="https://s.codepen.io/impressivewebs/debug/rRXmEe/nqAwvGEGZadr" id="myFr\
ame"></iframe>
```

I could apply this to the parent document too, but this will make things more clear for the purposes of some demo code. Here's how I'll access the iframe's document to apply this property:

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

btn.addEventListener('click', function () {
  myFrame.contentDocument.body.innerHTML = 'Click and edit the contents of this fram\
e. Use ctrl-b and ctrl-i to add bold or italic.';
  myFrame.contentDocument.designMode = 'on';
}, false);
```

Try it on CodePen

In the demo, I'm using a `window.onload` handler to ensure the iframe is ready before I start to manipulate it. I'm also using `myFrame.contentDocument` to access the frame's document (which would be equivalent to the `document` object in the parent).

The `designMode` property is then set to `on`. This value can be set to either "on" or "off". After the `iframe` becomes editable, you can then click inside it to change its contents.

Note also in some browsers, if you use CTRL-B or CTRL-I, just like a rich text editor, the text will become bold or italic. If you choose to access that content via JavaScript after it's edited (or inspect it in your developer tools), you'll see that the formatting and HTML will still be intact.

There's lots more I could write about this, as this does have quite a few quirks and browser differences. You can view the resources below for further info:

- document.designMode on MDN
- designMode on WHATWG
- Rich-Text Editing in Mozilla on MDN

JSON.parse() and JSON.stringify()

`JSON.parse()` and `JSON.stringify()`, both introduced in ES5, are supported in all modern browsers and even back to IE8. In brief, `JSON.parse()` parses a string as JSON and `JSON.stringify()` converts an object to a string. Here's an example:

```
let team = {
  "BlueJays": {
    "wins": 92,
    "losses": 67,
    "league": "American",
    "division": "East",
    "year": 2015
  }
};

strTeam = JSON.stringify(team);
console.log(strTeam);

console.log(JSON.parse(strTeam));
```

CodePen demo

The first log spits out the whole object as a string, all on one line, while the second log, if you view the console, will display the object like it was originally, but converted from the string.

`JSON.stringify()` also allows two optional parameters: a "replacer" function and a "space" to insert into the output for readability. For example, assuming the same object as above, let's use the following function along with a tab character as the "space":

```
function replacer(key, value) {
  if (typeof value === "string") {
    return undefined;
  }
  return value;
}

strTeam = JSON.stringify(team, replacer, '\t');
console.log(strTeam);
/*
```

```
{
  "BlueJays": {
    "wins": 92,
    "losses": 67,
    "year": 2015
  }
}
*/
```

CodePen demo

The replacer function, in this case, filters out values that are strings and the tab character helps the format of the output so it's more readable. You can use any string or number for the final parameter. If it's a number, this represents the number of spaces to use for each level of indent. If it's a string (as I used above), the string is inserted for each indent.

For more info on these useful methods, see:

- JSON.parse() on MDN
- JSON.stringify() on MDN

Property Access on Strings

Here's a ridiculously simple tip using strings that was introduced in ES5. You likely know that you can access individual items of an array using square brackets. For example:

```
let myArray = ['item0', 'item1', 'item2', 'item3'];
console.log(myArray[2]); // "item2"
```

That's simple enough. But what about treating a string like an array-like object, where the string's characters are different "items"? You could use the `charAt()` method:

```
let myString = 'example string';
console.log(myString.charAt(3)); // "m"
```

But you can also accomplish the same thing using square brackets directly on a string:

```
console.log('my example string'[7]); // "p"
```

Just like in an array, the square brackets indicate the position of the character you want.

Try it on CodePen

But there's limited use for this. MDN says

> "For character access using bracket notation, attempting to delete or assign a value to these properties will not succeed. The properties involved are neither writable nor configurable."

This technique is supported in IE8+, so it's safe to use if you think it's useful.

The Screen API

The global `window` object gives you access to a `Screen` interface, or `screen` object, that lets you access various values that provide info on the current window. Here is what the object looks like if you log it out on CodePen or in the Dev tools using `window.screen`:

```
[object Screen] {
  availHeight: 1040,
  availLeft: 0,
  availTop: 0,
  availWidth: 1920,
  colorDepth: 24,
  height: 1080,
  orientation: [object ScreenOrientation] {
    ...
  },
  pixelDepth: 24,
  width: 1920
}
```

I've abbreviated the "orientation" section, but as you can see, you get values that include information about the height and width of the window, the color depth, and pixel depth. These would be accessible via `window.screen.availWidth`, `window.screen.pixelDepth`, etc. (of course, the `window` object is global, so you could omit it).

Any of this info could be used to provide specific functionality or a different experience based on some user interaction. This is similar to using media queries, but instead the info is accessibile via JavaScript using a clean API.

If you try this in different browsers, you'll get some different results, but many of the primary properties are available on all modern browsers.

More info:

- The Screen API on MDN
- The Screen Interface on W3C's CSSOM View Module

The `disabled` Attribute for Stylesheets and Scripts

Here's another obscure feature: stylesheets and script elements have a Boolean `disabled` attribute that can be queried or defined. This can work as a really simple way to disable a specific stylesheet or script.

For example, if I referenced a stylesheet and script using two variables called `mySheet` and `myScript`, I could disable them both like this:

```
mySheet.disabled = true;
myScript.disabled = true;
```

And I could create a stylesheet toggle button using something like the following:

```
let sheet = document.getElementById('boot'),
    btn = document.querySelector('.btn');

btn.addEventListener('click', function () {
  sheet.disabled = !sheet.disabled;
  console.log(sheet.disabled);
}, false);
```

Try it on JS Bin

Visit the demo page and you'll notice the "toggle" button at the top. Below the toggle button are a bunch of Bootstrap-styled HTML elements. The Bootstrap CSS file is toggled on and off using the `disabled` attribute on the stylesheet element.

Seems like this could be a really simple way to add some kind of style-switcher or to conditionally add/remove stylesheets or script elements. The `disabled` attribute is discussed as part of the StyleSheet interface of the CSSOM on the W3C's site.

I couldn't find any definitive browser support listings for this, but it works in the latest Chrome, Firefox, IE11, and even in Safari 5.x on my Windows machine, so there does seem to be legacy support (which makes sense since it was part of the DOM 2 spec)

useCapture with addEventListener()

The `addEventListener()` method takes a third argument that is a Boolean that's almost always set to `false`. I usually write it without thinking about it, always setting it to `false`. So what is this third argument?

It's an optional argument (but required for legacy support) and it's called `useCapture`. In brief, this argument lets you define whether to use event *bubbling* or event *capturing*. So if a parent and a child element have an event handler with the same event, which one will fire first? There's a great StackOverflow answer that explains it nicely:

- With bubbling, the event is first captured and handled by the innermost element and then propagated to outer elements.
- With capturing, the event is first captured by the outermost element and propagated to the inner elements.

To use capturing with `addEventListener()`, your code would look like this:

```
1  el.addEventListener('click', doSomething, true);
```

Notice the third argument set to `true` rather than the usual `false`.

You can see the difference if you view this CodePen demo. Notice there are two parent/child combos. Click the parent, then the child for each one. The output will display which event gets fired first. In the 2nd nested set, you can see in the code that the listener has the third argument set to `true`, thus causing the output message to be reversed when the inner element is clicked.

That's basically it in a nutshell. I haven't seen too many practical use cases for implementing `useCapture`, but here are two articles that explain the whole concept a little more deeply:

- Event order by PPK
- Bubbling and capturing by Ilya Kantor

event.detail

The UIEvent interface lets you get the value of a `detail` property. You can access this property like so:

```
btn.addEventListener('click', function (e) {
  console.log(e.detail);
}, false);
```

The value depends on the event and, of course, requires that you pass the event object into the function.

- For `click` or `dblclick` events, the `detail` value is the current click count.
- For `mousedown` or `mouseup` events, MDN says the value is supposed to be `1` plus the current click count. But I don't see this behavior; seems to be the same as `click`/`dblclick`.
- For other UI events, the value is `0`.

The problem with this property is that no two browsers seem to treat it the same way and the spec is pretty vague about what type of info it should provide. You can try it out using this CodePen demo.

Here are my browser findings:

- WebKit browsers keep counting clicks unlimited, whereas Firefox only counts up to 3. Firefox seems to be limiting it to a "triple-click" whereas WebKit leaves things open for a quintuple-click or more.
- IE11 has has inconsistent behaviour. In my demo, clicks don't get counted until you mouse over the "mouse over" button, which is just plain bizarre. `mousedown` and `mouseup` events do get counted though.
- Double click is the same everywhere except IE11 which doesn't seem to provide any info.
- Mouse over correctly produces `0` everywhere except IE11 which (as mentioned) just prints out the value of the clicks on other buttons. This happens even if the `mouseover` event is the only one on the page (i.e. you could click anywhere and the `mouseover` will produce the number of clicks).

There's a little more info in the spec, but not much. Your best bet is probably the Dottoro reference which discusses how the value appears for the different types of events.

HTML Elements as Global Variables

Here's a simple fact about global variables and IDs in JavaScript: Every time you add an `id` attribute to an element in an HTML page, that `id` becomes a global variable (that is, a property of the `Window` object) accessible in your scripts on that page. For example, let's say you have the following HTML:

```
<section id="mySection" class="home">
  <div id="one"><p>This is div one</p></div>
  <div id="two"><p>This is div two</p></div>
  <div id="three"><p>This is div three</p></div>
  <div id="four" data-name="fourDiv"><p>This is div four</p></div>
  <div id="innerHeight"><p>This is div five</p></div>
</section>
```

You can access any of those HTML elements by simply using the values of the `id` attributes:

```
console.log(mySection.className); // "home"
console.log(one.textContent); // "This is div one"
console.log(two.innerHTML); // "<p>This is div two</p>"
console.log(three.id); // "three"
console.log(four.dataset.name); // "fourDiv"
console.log(innerHeight); // already global
```

Here it is on CodePen

Notice that the script doesn't access the elements using `getElementById()`, `querySelector()`, or anything else. I'm simply referring to the `id` attributes directly. All of these could also be written as *window.id* if you wanted to be more explicit so the code isn't confusing.

But, as you'll notice with the last log in that demo, this technique can cause problems. If a global variable already exists with the same keyword as the `id` value chosen, it won't work, but will instead reference the original global variable. In this case, the final example references `innerHeight` which is already a valid property of the `Window` object and so it just spits out the inner height value of the window. Other examples that can cause conflicts include `history`, `location`, `navigator`, `innerWidth`, and much more.

So I would chalk up this little tip to "good to know" but probably not something you'll want to use, unless you're being really careful with your naming of `id` attributes (e.g. you use a very specific prefix to namespace all the values).

The scrollHeight Property

Every HTML element has a read-only `scrollHeight` property that gives you the height of the element's content including non-visible content that's clipped vertically because of `overflow` set to `hidden` or `auto` in the CSS.

To access you would do this:

```
let shValue = el.scrollHeight;
```

The JavaScript alone is not that interesting. It needs to be combined (as mentioned) with some CSS so you can see the values that result. This CodePen demo has four `div` elements with varying heights and content, a few clipped with hidden/auto overflow.

You'll see that the fourth `div` has a height that's larger than the content, but the `scrollHeight` value is merely the height of the content (i.e. it's smaller than the defined height). Meanwhile, in the other instances, the `scrollHeight` value is always larger than the height of the element due to the overflow. So it's the content that determines `scrollHeight`, not the height set in the CSS or elsewhere. Padding is included in the height, margins are not, and (from what I can tell) setting the `box-sizing` to border-box doesn't affect the returned value.

MDN's page on the subject has an interesting use case with a live demo. In the demo, when the user scrolls a `div` to the bottom, it's assumed the content has been read. So with `scrollHeight`, you can (quite superficially) determine if the user has read the content. This might be used on a terms of service page. If there's no scroll, then you know the user hasn't read it all (although there's obviously no guarantee, this is just one way to attempt this).

As always, the best thing about this is that it's cross-browser, supported everywhere including back to IE8.

classList.toggle() with Boolean Force

You're probably familiar with the `classList` API for manipulating CSS classes in a more jQuery-like fashion (which is much easier than the tedious `className` method). One of the features available for `classList` is the `toggle()` method, which (much like jQuery's `.toggleClass()`), lets you toggle a specified class.

But you might not know that `classList.toggle()` takes a second, optional argument. The DOM spec refers to this as an *optional Boolean force*. This doesn't mean it has to be set to either `true` or `false`, but it has to *evaluate* to either `true` or `false`.

To demonstrate, I'll use the following HTML:

```
<section class="one two three four">
```

Then I'll manipulate those classes with some JavaScript:

```
let m = document.querySelector('section'),
    i = 8;
m.classList.toggle('one'); // false
m.classList.toggle('two', true); // true
m.classList.toggle('three', false); // false
m.classList.toggle('four', i < 10); // true
m.classList.toggle('five'); // true
m.classList.toggle('four', i > 10); // false
```

View CodePen demo

You can see that the `toggle()` method always returns a Boolean. This represents whether or not the class that was toggled is now present. So in the first toggle line, I know the `one` class existed. When I toggle it, that means I'm removing it, so it returns `false`. For the fifth toggle, I'm toggling a class that doesn't exist, so it's added. Thus, it returns `true`.

The other four examples use the optional "force" Boolean. If this is present and it evaluates to `true`, it will force the class to be added, not removed. If it's `false`, the class will only be removed, not added. You might think: Why use this instead of `classList.add()` or `classList.remove()`? Well, for one, those methods return "undefined", whereas `toggle()` returns a Boolean, which might be useful.

The other advantage is that you can put in a conditional (as I've done for two of the lines above) that evaluates to a Boolean, to allow the script to decide whether to toggle or not.

If that's confusing, test it out in the demo or read more:

- The classList API on HTML5 Doctor
- DOMTokenList Interface spec
- classList on MDN

The only drawback to this second argument is the lack of IE support, so you'll get different log results in that demo in IE. It is supported in Edge however.

outerHTML

Here's a DOM feature that seems to have gone under the radar: the `outerHTML` property. This one may have gone unnoticed for some time by many developers because it wasn't supported in Firefox until version 11. So those who had to support Firefox versions 3 to 4 (prior to Firefox becoming an auto-updating browser) couldn't use it.

According to MDN this propery "gets the serialized HTML fragment describing the element including its descendants." It's basically the same as `innerHTML`, except it also grabs the parent itself, not just what's inside the parent. So with the following HTML:

```
<div id="parent">
  <p>Some text for parent.</p>
  <div id="child">
    <div id="grandChild">
      <p>Some text for grandchild.</p>
    </div>
    <p>Some text for child.</p>
  </div><!-- #child -->
</div><!-- #parent -->
```

Assuming I have references to `child` and `grandChild`, I can use `outerHTML` like so:

```
console.log(child.outerHTML);

/* will log:
<div id="child">
  <div id="grandChild">
    <p>Some text for grandchild.</p>
  </div>
  <p>Some text for child.</p>
</div>
*/

grandChild.outerHTML = '<p>New paragraph</p>';
console.log(document.getElementById('grandChild')); // null
```

View on CodePen

You can see that the log of the `outerHTML` of the child is not a reference to the object, but a string representation of the content, including the markup. You can use this property on any element and you can get the `outerHTML` of the root element, but you can't change the root element; this will throw an error.

Notice in the demo that I'm replacing the content of the `grandChild` element using `outerHTML` so it logs `null` when I try to reference it on the last line.

The beforeprint and afterprint Events

Here are two DOM events that I think will be quite useful to many: the `beforeprint` and `afterprint` events.

In MDN's docs, `beforeprint` is explained: "The `beforeprint` event is fired when the associated document is about to be printed or previewed for printing." The `afterprint` event, on the other hand, "is fired after the associated document has started printing or the print preview has been closed."

It's notable that these events trigger whether the browser's print functionality is triggered or just the print preview. MSDN's documentation, which is no longer online, had an interesting use case. It provided some code and explained: "This example uses the `onbeforeprint` to make all hidden sections of the document visible just before the document prints. The `onafterprint` event is processed after the document prints to return the document to its original state."

Here's how that would look in a code example using both events:

```
window.addEventListener('beforeprint', function () {
  document.body.innerHTML = 'this is before printing';
}, false);

window.addEventListener('afterprint', function () {
  document.body.innerHTML = 'this is after printing';
}, false);
```

Try it on CodePen

Another possible use case might be to track how many times a user has printed or previewed for printing. This way, you don't have to have an actual print button that checks for click events, and which wouldn't tell the whole story. These events will be triggered regardless of the way the print functionality is used.

Browser support for these is good but it seems that Safari is the only browser that doesn't support them.

Radio Buttons and the change Event

If a user clicks to toggle a checkbox on a web page, as you probably know, this will fire both a `click` event and a `change` event. The `change` event will only fire if the value is changed. For example, if part of your script prevents the change (like a `preventDefault()` call), then that will prevent the `change` event from firing. The `change` event, of course, will fire whether the checkbox is being checked or unchecked. As long as it's changing, the event will fire.

Radio buttons are similar, but you might say they respond in a somewhat unintuitive manner. As you know, a set of related radio buttons can only have one of the buttons "checked". So logic would tell us that if radio button #1 is checked and I click to check radio button #2, that should fire a change event on both radio buttons – because technically they both changed.

But this is not the case. Only the radio button that gets the checked state will fire a `change` event. You can test it out in this CodePen demo. Notice there are two sets of form elements, a set of checkboxes and a set of radio buttons. Each time you click a checkbox, you'll see the log output for each "change" that occurs.

The radio buttons behave basically the same way but the `change` event does not fire on a radio button that loses its selection when another one is selected. I don't think this is necessarily a problem, but it seems like it would be more logical if the `change` event fired on two radio buttons, rather than just one, because, as mentioned, technically they're both changing.

document.readyState

If you've done Ajax with vanilla JavaScript, then you're probably familiar with the `readyState` property of the `XMLHttpRequest` (XHR) object. The `document` object also has a `readyState` property. Unlike the XHR object's property, which returns the state as an integer, `document.readyState` returns one of three values:

loading Exactly what the name implies, this value means the document is still in the process of loading.

complete
Again, pretty clear. But there's no in-between here; this value means the document, including all resources (stylesheets, frames, etc.) has finished loading.

interactive This is one that could prove useful. This means the DOM is ready to be interacted with but resources have not completed downloading. So this state would fall somewhere in between the previous two.

When the `readyState` property changes, the `readystatechange` event fires on the `document` object. So you could theoretically do something like this:

```
document.onreadystatechange = function () {
  switch (document.readyState) {
    case 'loading':
      console.log('loading...');
      break;
    case 'interactive':
      console.log('DOM is ready...');
      break;
    case 'complete':
      console.log('Document complete...');
      break;
  }
};
```

View this example on CodePen here. You probably won't see the "loading" indicator because it seems that the JavaScript doesn't load in time for the message to appear, or else something else is going on that I don't understand.

As MDN points out, you can use `document.readyState` as an alternative to the `DOMContentLoaded` event (using the value `interactive`) or as an alternative to the `load` event (using `complete`). You might also want to check out this JS Tips article, which discusses the same basic concept, but more specifically how to do something similar to jQuery's `$(document).ready()` in plain JavaScript.

“splat” an Array in ES5 vs. ES6

Let’s say you have a function that takes a specific number of arguments, like this:

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

Somewhere in your application code, you might have an array that contains the values you want to pass into the function. Using the example of two separate arrays, you can pass those in as follows:

```
let ArrayOne = [8, 14, 432],
    ArrayTwo = [74, 4, 26];

console.log(myFunc.apply(null, ArrayOne)); // 454
console.log(myFunc.apply(null, ArrayTwo)); // 104
```

Here I’m using JavaScript’s `Function.prototype.apply()` method to “splat” the values of the array. The `null` value is the first argument for `apply()`, which is where you custom-define the value of `this`. Using `apply()` in this way prevents the need to do any kind of array manipulation; you just drop in the array.

But this becomes even easier if you use ES6’s spread syntax (which is represented by three dots):

```
console.log(myFunc(...ArrayOne)); // 454
console.log(myFunc(...ArrayTwo)); // 104
```

Here I’m simply prefixing the array with the spread operator and I’m able to accomplish the same thing.

You can view both versions in use in this CodePen.

Hat tip to Josh Branchaud’s TIL JavaScript tip for the ES5 version. You might also want to check out this Stack Overflow thread, which explains the term “splat”.

Spread Operator vs. Rest Parameters

The previous tip discussed how to "splat" an array in two different ways, using ES5 and ES6. The latter method used an ES6 feature called *spread syntax*. Let's quickly consider both spread syntax and rest parameters, which look very similar but do different things.

Here's another example that uses the spread syntax:

```
let myFruit = ['apples', 'oranges', 'peaches'],
    myNewFruit = ['kiwis', 'pairs', 'melons'];

myFruit.push(...myNewFruit);
console.log(myFruit);
// ["apples", "oranges", "peaches", "kiwis", "pairs", "melons"]
```

View on CodePen

There might be some confusion with regards to spread syntax and rest parameters, which is likely due to the fact that they use the same characters (three dots). As Addy Osmani explains, rest parameters are different in that they allow your functions to have variable number of arguments without using the `arguments` object.

The main advantage to using rest parameters over the `arguments` object is the fact that the items passed in as rest parameters are an array, so you can manipulate them using any valid array methods, which is unlike the non-array `arguments` object.

Here is an example that uses rest parameters:

```
function myFunc (a, ...b) {
  console.log(b.length);
  console.log(Array.isArray(b));
}

myFunc(3, 1, 2, 3);
myFunc(1, 2, 3, 5, 20, 34);
myFunc('one', 'two', 'three', 'four', 'five');
myFunc('one', 'two');
```

View on CodePen

The function is returning the length of `b`, which is variable depending on what we pass in. The `a` variable (the first thing passed in) is separate from the remaining arguments (this is where the word

"rest" comes from, the fact that it covers the "rest" of the arguments). This allows you to pass in as many arguments as you want using a simple syntax. And, unlike the `arguments` object, you have the added advantage of the use of array methods (my example checks to confirm it's an array using `isArray()`).

I hope that clarifies the difference between those two new and useful features. Be sure to mess around with the demos to better understand them.

insertAdjacentElement() and insertAdjacentText()

There have been two interesting additions to the DOM spec in recent years: the `insertAdjacentElement()` method and the `insertAdjacentText()` method. These methods are very similar to the `insertAdjacentHTML()` method, which I discussed in a previous tip. Below is some code to demonstrate how the new methods work.

Suppose I have the following HTML:

```
<section>
  <p id="p1">Example element with text.</p>
  <p id="p2">Example 2nd element.</p>
</section>
```

I'll do some manipulation using the two new methods:

```
let p1 = document.getElementById('p1'),
    p2 = document.getElementById('p2');

p1.insertAdjacentElement('beforebegin', p2);
p1.insertAdjacentText('beforeend', 'TEXT');
```

After running that code, the DOM will look like this:

```
<section>
  <p id="p2">Example 2nd element.</p>
  <p id="p1">Example element with text.TEXT</p>
</section>
```

Try it on CodePen

Use the button in the demo to trigger the action and notice the changes that take place on the page.

Notice the `p2` element was removed from its place and moved before the `p1` element. So the element is essentially moved, not just inserted (unless of course you are referencing an element not yet in the DOM). The `TEXT` snippet was inserted at the end of the `p1` element.

As with `insertAdjacentHTML()`, both of these methods take a string as the first argument that must be one of:

- `beforebegin`
- `afterbegin`
- `beforeend`
- `afterend`

The string argument is mandatory and it represents where, in relation to the element on which the method is applied, to place the inserted argument (before the element begins, before it ends, etc). The second argument is the element or text you want to insert. If you fiddle with it in the demo, you'll get a clearer picture.

These two new methods are nice additions because previously you could only insert a string of HTML as text. Now you can insert an element, an HTML string, or a string of text (which would include markup inserted as HTML entities).

The best part about these "new" methods? They're not really new. These have been supported as nonstandard features for some time in most browsers and all browsers currently support them, including IE11 and Edge.

- insertAdjacentElement on Dottoro
- insertAdjacentText on Dottoro
- Both methods on WHATWG DOM spec

DOMTokenList

The `classList` object, which I've discussed before, returns what's referred to as a `DOMTokenList`, which is a DOM interface that opens up the possibility of using various methods, like `add()`, `remove()`, and `toggle()`, commonly used with `classList`.

But `classList` is not the only DOM feature that returns a `DOMTokenList`. The following objects also return a `DOMTokenList`:

- `relList` for HTML `<link>` elements
- `relList` for anchors (hyperlink elements)
- `relList` for HTML `<area>` elements

A `relList` is a collection of the `rel` attribute values on these various elements. As of this writing, only Firefox, Chrome, and Edge support these. In each case, you'll have access to the same methods that are customarily used with `classList`. The full list of methods and properties includes:

- `length`
- `item()`
- `contains()`
- `add()`
- `remove()`
- `toggle()`
- `replace()`
- `supports()`
- `value`

As long as the browser supports the feature that returns the `DOMTokenList`, it should also support all of the above methods. More info:

- DOMTokenList Interface on WHATWG
- DOMTokenList on MDN
- Link Types on MDN

Detecting Shift/CTRL/ALT/Meta Keys

I discussed this before in relation to mouse events, but let's expand on this a little more with some further code examples.

If you want to find out what key the user has pressed on the keyboard, you can write a cross-browser function, like this:

```
function doWhichKey(e) {
  e = e || window.event;
  let charCode = e.keyCode || e.which;
  return String.fromCharCode(charCode);
}
```

When you call this function while passing in the event object, this will return the name of the key that was pressed. In such a case, I'd be passing in a keyboard event, which has a bunch of different methods and properties. A few interesting ones are:

- `e.shiftKey`
- `e.altKey` (Option key on Mac)
- `e.ctrlKey`
- `e.metaKey` (Windows/Command key)

Each of these read-only properties will return a Boolean to indicate if the specified key was active when the keyboard event was generated. So theoretically you should be able to detect SHIFT-L, CTRL-W, ALT-5, etc.

Unfortunately, because the operating system already has responses to some of these key combos (e.g. CTRL-P will print), you can't detect all of them, so you'd be better off with another solution. The `e.shiftKey` property is probably the only one that works well in this regard, because it doesn't seem to be hooked into any OS-generated events.

Thus, along with the function shown above, I could do something like this to display the combo pressed:

```
window.addEventListener('keypress', function (e) {
  if (e.shiftKey) {
    document.body.innerHTML += 'SHIFT-'+doWhichKey(e)+'<br>';
  }
}, false);
```

Try it on CodePen. Try changing `e.shiftKey` to `e.altKey` or `e.ctrlKey` in the JS panel to see how those work. As mentioned, `e.shiftKey` is probably the only useful one of the four due to OS shortcut conflicts

CSS.supports()

You've probably heard of using `@supports` in your CSS to perform feature queries, similar to what you might do with a tool like Modernizr or other feature detection scripts. Well, maybe you didn't know you can also do the same tests in your JavaScript using the `CSS.supports()` method. All browsers with the exception of IE11 and earlier have implemented `@supports` as well as `CSS.supports()`.

It's simple to use:

```
1 console.log(CSS.supports('display', 'flex')); // true
2 console.log(CSS.supports('object-fit', 'none')); // true
3 console.log(CSS.supports('color', 'currentColor')); // true
4 console.log(CSS.supports('hyphens', 'auto')); // false
```

Try it on CodePen in different browsers.

I ran that code in Chrome, so three of the four tests returned `true`. Chrome doesn't support CSS hyphenation, so it returned `false` for the last one.

Instead of passing in two strings as a property/value pair (as I did in the examples above) I also have the option to use a `supportCondition` argument, which would look like this:

```
1 CSS.supports('( animation: none ) or ( -webkit-animation: none )');
```

View on CodePen

So it's basically the same idea as using `@supports` in CSS. If you already know the syntax for `@supports`, then you know this too.

Expando Properties

Ever heard of an "expando" property? Many of you probably have. I just came across the term while reading Secrets of the JavaScript Ninja. MDN explains:

"Expando properties are properties added to DOM nodes with JavaScript but not part of the object's DOM specification." (source)

So basically the term can serve to differentiate between properties that are supposed to be on a DOM element and properties that you've added yourself, as is possible with any JavaScript object.

What's interesting is that Internet Explorer before IE9 had a sort of universal expando property that allowed you to define whether or not you can use "expandos" on elements in the document (e.g. `document.expando = true`). And using `element.expando` you could also test to see if a given property is an expando property.

Again, that's for old IE, so it's a fairly trivial legacy point at this stage. But keep in mind that IE's expando properties are not the same thing as the general term, which, as explained above, is just a generic term for custom properties on objects. More info in the links below:

- MSDN's expando property
- Expando property on Dottoro
- Expando on Stack Overflow

form.length / select.length

In JavaScript, the `length` property is widely used and available in a number of different ways. You can use it to check the length of a string, an array, or a NodeList. You can even use it on a function, as I discussed in a previous tip.

If you use `length` on something that doesn't accept the property, you'll usually get `undefined`. For example, if you check the length of an HTML element (like the `body` element):

```
console.log(document.body.length); // undefined
```

But you are able to use the `length` property on forms and select elements. The following code will display the length of a form and the length of the select element in the form:

```
console.log(document.forms[0].length);
console.log(document.getElementsByTagName('select')[0].length);
```

So what exactly does this `length` value represent? For the form itself, it tells you how many form elements it contains (including hidden form inputs). For the `select` element, it tells you how many `option` elements are inside the `select` element.

In this CodePen demo you can see a more complete example, along with attempts to check the length of the body and of a random `div` element.

If nothing else, this can serve as a quick way to check if a `form` or `select` contains a certain number of elements (as opposed to, say, looping through the form elements, or using some other more convoluted method). Of course, this would be most useful in dynamic apps where the number of elements changes for whatever reason.

Node.isSameNode()

Here's yet another little-known DOM method: The `Node.isSameNode()` method. Previously I've discussed a similar method: `Node.isEqualNode()`. This one is very similar except it will only check to see if two references point to the exact same node, rather than checking general equality.

Let's assume my HTML looks like this:

```
<section id="mySection">
  <div class="module"></div>
  <div class="module"></div>
</section>
```

All we have here is a `section` element along with two identical `.module` elements, though they are different elements. Let's do some checks with `isSameNode()` and `isEqualNode()`:

```
let mySection = document.getElementById('mySection'),
    otherSection = document.querySelector('section'),
    moduleOne = document.querySelectorAll('.module')[0],
    moduleTwo = document.querySelectorAll('.module')[1];

// using isSameNode
console.log(mySection.isSameNode(otherSection)); // true
console.log(moduleOne.isSameNode(moduleTwo)); // false

// using isEqualNode
console.log(mySection.isEqualNode(otherSection)); // true
console.log(moduleOne.isEqualNode(moduleTwo)); // true
```

Try it on CodePen

As you can see, I've grabbed a `section` element two different ways, so I have two references to it. That's the first comparison, which logs `true` using `isSameNode()`. But when I compare the two `div` modules, I get `false`, because they're not the same node.

On the other hand, when using `isEqualNode()`, both comparisons log `true`, because they're equal (i.e. same attributes and same content, which is no content), even though they're not the same node in the case of the modules.

It turns out, all major desktop browsers support this but Firefox removed support for it about 5 years ago and they've maintained that status, even though it is part of the DOM spec. Their reasoning for

removal seems pretty sound though; it's basically the same as doing a == check. So while this might be a nice-to-know sort of feature, you'd probably just be better off doing a simple == comparison instead, especially in light of lack of Firefox support.

Array.prototype.filter()

One day I was looking for the quickest and most efficient way to remove all empty items from an array. Doing a quick search, as usual, brings up a few Stack Overflow threads with multiple solutions. One that I liked uses the `Array.prototype.filter()` method, introduced in ES5 and supported by all browsers including IE9+.

The `Array.filter()` method, according to MDN, takes a callback function as its one mandatory argument, and "creates a new array with all elements that pass the test implemented by the provided function." In other words, it filters out the stuff you don't want. Here's an example:

```
let myArray = [47, 534, 8, 389, 144, 99],
    myNewArray;

function getLargeNumbers (a) {
  return a >= 100;
}

myNewArray = myArray.filter(getLargeNumbers);

console.log(myNewArray); // [534, 389, 144]
```

As you can see, the function simply returns the values in the array that are larger than 100. But what about my original problem where I wanted to remove empty items?

Well, this works like a charm:

```
let myArray = ['apple', 'orange', '', 'kiwi', 'pumpkin', 'grape', ''],
    myNewArray;

myNewArray = myArray.filter(Boolean);

console.log(myNewArray); // => ["apple", "orange", "kiwi", "pumpkin", "grape"]
```

View on CodePen

Notice that the two empty items are now removed. In this instance, I'm using `Boolean` as the lone argument. MDN explains that the Boolean object is "an object wrapper for a boolean value." So, in a nutshell, it will automatically filter out falsy values when used in this way. But beware of this, because it will filter out zero values, `false`, `null`, and `undefined`, in addition to empty slots. If for some reason you want any of those to remain in the array, then you'll have to use a custom function that does what you want.

The eventPhase Property

The `event` object has a property called `eventPhase`, which lets you see which phase of the event flow is currently being evaluated. Here's an example, used in a function triggered on a `click` event:

```
el.addEventListener('click', function (e) {
  console.log(e.eventPhase); // logs "2"
}, false);
```

The return value of the `eventPhase` property is an integer (as shown in the comment in the code) indicating the phase of the event flow. If you've researched event capturing, bubbling, and similar concepts, then the meanings of the events should be fairly straightforward. Here's a quick summary:

- 0 - No event is being processed
- 1 - Capturing phase (event is being propagated through the target's ancestor objects)
- 2 - The event has arrived at the event's target
- 3 - Bubbling phase (event is propagating back up through the target's ancestors in reverse order)

I've created a CodePen demo that demonstrates using a couple of `div` elements. When using the `addEventListener()` method, the last argument, normally omitted or set to `false`, can be set to `true`, which means you are registering the event for capture mode, like this:

```
el.addEventListener('click', function (e) {
  console.log(e.eventPhase); // logs "1"
}, true); // set to "true" for capture mode
```

This will log different results compared to the click event that doesn't use capture mode. Fiddle with the demo and visit the links below for more info:

- event.eventPhase on MDN
- Bubbling and Capturing on JavaScript.info
- Event order on QuirksMode.org

window.open() and window.opener

Although it's generally a bad choice to open new windows with JavaScript, you can do some interesting things with it, due to the power given to the different windows, particularly the source window. So if you're new to this sort of thing, here's a quick summary of some features you have at your disposal.

To open a new window you use the `window.open()` method:

```
window.open('example.html', 'windowName', 'location=no,status=no');
```

The first argument is the page to open, the second argument is a custom window name (of your choice, which you can use to access the window again), followed by a set of "features" separated by commas within the final argument quotes. If you really want to harness the power of opening new windows, however, you'll want to store a reference to the window that's opened:

```
let myWin = window.open('example.html', 'windowName', 'location=no');
```

Now with the `myWin` variable, you can do stuff like this:

```
myWin.document.body.style.background = 'red';
```

This is done from the original source window. As long as you're obeying cross-origin rules, you have full access to the child window from the source (or parent). Also, from the child page (in this case, example.html), you can do stuff to, or get info from the source window:

```
window.opener.document.body.style.background = 'orange';
```

Here I'm using the `window.opener` property to reference the original source window, changing the background of the source. And note that although you can close windows from the source window, you can't close the original source window via one of the spawned windows.

I'm sure that's pretty basic stuff for many of you, but if you want to examine a really cool chunk of code that manipulates browser windows, check out Browser Ball, a Chrome Experiment by Mark Mahoney. After you try it out, it's worth viewing and beautifying the source to see what's going on.

- window.open() on MDN
- window.opener on MDN

Attribute Collections as NamedNodeMaps

When using the `attributes` property to get the attributes of an HTML element, you're exposing the `NamedNodeMap` interface, which has a few funky little methods and properties of its own.

Let's assume the HTML element is a `section` element with two attributes:

```
<section class="one" id="two"></section>
```

I can use the `getNamedItem()` method to read the value of a specific attribute in the collection:

```
let s = document.querySelector('section');
console.log(s.attributes.getNamedItem('class').value); // "one"
console.log(s.attributes.getNamedItem('id').value); // "two"
```

There's also the `setNamedItem()` method, which can be used along with `createAttribute()` to add a new attribute to the element:

```
let myAttr = document.createAttribute('foo');
myAttr.value = 'bar';
s.attributes.setNamedItem(myAttr);
console.log(s.attributes.getNamedItem('foo').value); // "bar"
```

In this case I've created an attribute "foo" with a value of "bar" and this would be reflected in the generated DOM.

Finally, I can use the `item()` method to grab an attribute by index, and I can also remove an attribute with `removeNamedItem()`:

```
console.log(s.attributes.item(1).value); // "two"
s.attributes.removeNamedItem('foo');
console.log(s.attributes.length); // 2
```

You can find a full CodePen demo here.

In most cases, some of the more traditional methods for attribute manipulation will suffice, but this API has its own set of unique features that you might find useful on deeper examination.

Blank Lines

Something related to code maintenance that you may have neglected to think about is the use of blank lines. Opinions will vary wildly as to what's right and wrong in this area, but I like the suggestions provided by Nicholas Zakas in his book Maintainable JavaScript. He says, "Blank lines should be used to separate related lines of code from unrelated lines of code." To demonstrate, here is a chunk of code similar to the one used in his book:

```
if (one && two.length) {
  for (i = 0, l = two.length; i < l; ++i) {
    j = one[i];
    k = this.doSomething();
    if (this.something) {
      if (this.anotherThing) {
        doWhatever();
      } else {
        doAnotherWhatever();
      }
    }
  }
}
```

The code itself is meaningless and only theoretical. But we can improve our ability to scan this chunk of code by using blank lines. Here's the same code, improved:

```
if (one && two.length) {

  for (i = 0, l = two.length; i < l; ++i) {
    j = one[i];
    k = this.doSomething();

    if (this.something) {

      if (this.anotherThing) {
        doWhatever();
      } else {
        doAnotherWhatever();
      }
    }
```

```
15    }
16  }
```

You can instantly see the difference just by scanning each block. I like Zakas' literary analogy when he says, "Code should look like a series of paragraphs rather than one continuous blob of text." He recommends blank lines before flow control statements, between methods, before comment lines, and between logical sections inside methods.

I'm sure many of you do this sort of thing already, but I've certainly seen code that doesn't use blank lines too often, so this might be a good reminder for many of you who haven't started doing it, or those of you who are like me and don't necessarily have a standard system in place for using blank lines.

Disabling the Context Menu

If you're building a web app, it's good to be familiar with the different techniques to disable the right-click menu, AKA context menu. Of course, many would suggest this should never be done, but now that web apps are pushing more towards native behavior, I'm sure a lot of developers will be interested in doing this sort of thing.

To disable the context menu, the best technique is as follows:

```
window.addEventListener('contextmenu', function (e) {
  e.preventDefault();
}, false);
```

Try it on CodePen

Here I'm listening for the `contextmenu` event, then I use `event.preventDefault()` (like the old `return false`) to keep the menu from appearing. This disables the context menu whether it's accessed via mouse or keyboard.

You can do something similar by only looking for a mouse-based context menu trigger. Basically you do something like this:

```
document.addEventListener('mouseup', function (e) {
  if (e.button === 2) {
    console.log('right click enabled');
    e.preventDefault();
  }
}, false);
```

The problem with the latter method is that the context menu still appears. In this instance, I can't use `preventDefault()`, because the context menu is not the default behavior of the `mouseup` event; it's the default behavior of the `contextmenu` event.

The HTML spec includes a menu element that is supposed to make this technique easier and more semantic, but browser support is still poor. For now the safest and most cross-browser way is to use the `contextmenu` event. If you want a full tutorial on building a custom context menu with HTML, CSS, and JavaScript, this one by Nick Salloum is really good.

Queueing Items in an Array

If you're relatively new to JavaScript and have started working with arrays, or even if you've been working with them for a while, you might find this little reminder useful. There's a really simple way to quickly move the first item in an array to the back of the array, and vice-versa. Here's the code:

```
let myArray = ['one', 'two', 'three', 'four'];
console.log(myArray); // ["one", "two", "three", "four"]

myArray.push(myArray.shift());
console.log(myArray); // ["two", "three", "four", "one"]

myArray.unshift(myArray.pop());
console.log(myArray); // ["one", "two", "three", "four"]
```

View on CodePen

The array methods used here are fairly common features of the language: `push()`, `shift()`, `pop()`, and `unshift()`. But the key is relying on the return values of the methods.

After creating the array and logging its contents, I'm using `push()` to add an item to the end. But the item I'm adding is the returned item via `shift()`. The `shift()` method not only removes the first item, but it returns that item, thus providing an easy way to move the first item from beginning to end.

The counterparts to those two methods, `unshift()` and `pop()`, are doing something similar except it's the reverse. The `unshift()` method adds an item to the beginning and the item added is the one returned when we "pop" one off the end.

text.wholeText Property

In a previous tip I discussed the `splitText()` and `normalize()` methods, which can be used to manipulate text nodes. Here's a quick summary: Calling `normalize()` on an element with multiple adjacent child text nodes will "normalize" those text nodes into a single text node, so the node count will be only one. `splitText()` on the other hand lets you split a text node at a specified place, to create multiple text nodes.

However, if you want a quick way to treat multiple text nodes like a single text node, simply for the purpose of reading the text content, you can do so using the `wholeText` property, which is accessed via the text node interface. Below is some commented code, along with a demo so you can see how it works. I'll assume that the main element I'm accessing (referenced via `firstElementChild`) is just a simple paragraph of text, with no other HTML inside it:

```
let b = document.body;

// length of child nodes is "1"d
console.log(b.firstElementChild.childNodes.length);

// I split the child text nodes
b.firstElementChild.firstChild.splitText(15);

// Now the length is "2"
console.log(b.firstElementChild.childNodes.length);

// Read the text as a single text node using "wholeText"
console.log(b.firstElementChild.childNodes[0].wholeText);

// It's still 2 nodes
console.log(b.firstElementChild.childNodes.length);

// Then I can normalize as needed
b.firstElementChild.normalize();

// Back to "1" text node
console.log(b.firstElementChild.childNodes.length);
```

Try it on CodePen

The comments explain most of what's going on. The lone variable is just a reference to the `body` element, for brevity, but I could access the text nodes in any number of ways.

You can see the `wholeText` property used on the fifth line of code (not counting comments). As shown, you need to access it via one of the text nodes. As the spec explains, this property "must return a concatenation of the data of the contiguous Text nodes of the context object, in tree order."

This is similar to what happens when you use the more well-known `innerText` and `textContent` properties. The main difference being that this one is accessed on a text node that forms part of the full text (i.e. the "whole text") that you want to access. It should also be noted that `wholeText` also will preserve white space and only accesses text that is continuous, without any HTML elements interspersed (which of course would create multiple text nodes, separated by elements).

From my research, this lesser-known property is supported in all browsers, including back to IE9, so it seems to be fine to use in most projects.

Modern Cookie Limitations

Today there are much better ways than cookies of storing data client-side when building apps. That being said, all developers should be relatively familiar with how cookies work and what, if any, limitations there are in the different browsers.

If you read different sources online that discuss cookies, you'll notice that many of them describe the following general limitations for most browsers: 300 cookies total, 20 cookies per domain, and about 4 KB of data per cookie. This is also stated in David Flanagan's JavaScript: The Definitive Guide, the latest edition of which is now more than seven years old.

That data is generally taken from RFC2965, the older standard that defines HTTP cookies (dated October 2000). But that standard is long since obsolete, as indicated in RFC6265. In that latest document, it states the following minimum requirements for browsers:

- At least 4096 bytes per cookie
- At least 50 cookies per domain
- At least 3000 cookies total

Using this test page, you can try out the limits in different browsers. From my testing, I had the following results:

- Latest Chrome: Maximum 180 cookies per domain; 4096 bytes per cookie;
- Latest Firefox: Maximum 1000 cookies per domain; 4097 bytes per cookie;
- Edge: Maximum 180 cookies per domain; 5117 bytes per cookie

In previous tests I had done, the cookie count for Firefox was 150 but this time it was much higher and the page stated that it stopped counting at 1000, so the total is likely much higher or else unlimited.

As mentioned, better storage options are now available (mainly LocalStorage), and highly recommended for larger bits of data. But cookies will likely still be in use for small bits of data, so it's good to be familiar with the limitations in the different browsers, keeping in mind that many online sources still reflect older statistics that are no longer accurate.

Commenting Confusing Code

In his book Maintainable JavaScript, Nicholas Zakas gives some suggestions on commenting code. One of the things he talks about is commenting code that could look like an error to a future developer, but in fact is intentional. The example he provides is the use of an assignment operator in a while loop's control condition. He takes this example from the YUI library, which he worked on.

Looking through the YUI source (which is no longer actively maintained, so keep that in mind), there are actually three such instances where code is commented in this way. Here's an example:

```
1  if ((el = cache[docStamp])) { // assignment
2    return el;
3  }
```

The single-line comment clarifies what might be construed as an error. Normally you don't see assignment operators used in conditionals in this way. Without the comment, a future developer might think this is a mistake and try to fix it (e.g. changing equals to double-equals or triple-equals). At best, the developer might waste time testing to see if it is an error, even if it isn't immediately fixed.

By default, some linters will warn about code like this (e.g. JSLint does), so again this might prompt a developer to waste time trying to fix a problem that doesn't exist. Of course, code standards for a team inheriting a project might change, and this might be a required fix, but that's another point altogether.

With the comment in place, the potentially confusing code is made clearer. When deciding to include comments in code, always comment code that could be potentially confusing or that is generally difficult to understand at first glance.

Assigning Anonymous, Immediately Invoked Functions to Variables

Assuming you're still using ES5, here's a quick maintenance tip regarding immediately invoked anonymous functions that have a return value that you assign to a variable. Here's some example code:

```
var myVar = function () {
  // function body goes here...
  return {
    value: 'something'
  };
}();
```

This is fine, but the minor problem that occurs here with regards to maintaining this code is that, at a quick glance, it looks like a typical anonymous function assigned to a variable. But it's not just an anonymous function; it's also an immediately invoked one, as shown by the parentheses at the end.

To improve the readability of the code and make it immediately clear that this function is immediately invoked, you can do the following:

```
var myVar = (function () {
  // function body goes here...
  return {
    value: "something"
  };
}());
```

The change is subtle but it's significant for someone scanning the code later and trying to figure out what's going on. You can see I've added an extra set of parentheses around the anonymous function, which serves to visually differentiate this from just a standard anonymous function assigned to a variable. This doesn't change the way the code works, but only adds a few bytes.

I should mention, of course, that with later versions of ECMAScript, you might not need to do this. Usually functions are enclosed in this way to prevent variables from leaking globally, but in ES6+ we can use `let` and `const` to ensure variables maintain block scope. Wes Bos discusses that a little bit in this post and you can also read a little bit about IIFEs with arrow functions in this post by Jack Tarantino.

The storage Event

When using the Local Storage API, you can listen for a `storage` event that's triggered when a Local Storage item is modified. If you're new to the `storage` event, then there are a few things to take note of.

First, a `storage` event will fire only on other same-origin windows where the item was modified. So the following code will not trigger a `storage` event if there is only one window/tab open running the script:

```
localStorage.setItem('one','one-value');
console.log(localStorage.getItem('one'));

window.addEventListener('storage', function () {
  console.log('Storage item was updated.');
}, false);
```

Notice I'm listening for the `storage` event but if I run this code I won't see the logged message "Storage item was updated." You can try it on this CodePen. The message won't log. Again, this is because the `storage` event is specifically designed for use in apps that use multiple windows.

To see the message get logged, I have to have two versions of the same window open and then run the script on one of those windows. The other window will display the log message. You can try it by opening the demo, then opening the same demo in another window. Then go back to the original window and you'll see the log will be updated with the `storage` event message.

Another interesting thing to note is that the `storage` event will fire if the user manually deletes the storage item (or clears their Local Storage). This too apparently qualifies as modifying the item, even though it doesn't happen in a script.

Finally, if you set a `storage` item to the same value that it already has, this will not trigger a `storage` event. So the message would not log inside the listener function in the code above. In such a case, the browser does not consider the storage item "modified", even though technically you have accessed it and overwritten what was there.

String Creation Using fromCharCode()

The `String` object in JavaScript has two methods that allow you to produce a string from a sequence of Unicode values. The first method is `String.fromCharCode()`, and has full browser support. Here's how it looks in action:

```
console.log(
  String.fromCharCode(175,92,95,40,12484,41,95,47,175)
);

// Output: ¯\_(ツ)_/¯
```

This will output the famous shrugging ASCII character as shown in this CodePen demo.

The `String.fromCharCode()` method will return a string and you have to use it as a method of the global `String` object rather than on a string you've created yourself. So if your app receives Unicode values as input, this method can be used to produce output based on the values received.

As explained on MDN, in order to deal with all legal Unicode values up to 21 bits, this method won't be enough (though it's probably more than sufficient for most cases). ES6 has therefore introduced `String.fromCodePoint()` to fill this gap. In this CodePen demo you can see the different output when using a character from a high code point with each of the two methods. MDN's article on the topic also includes a polyfill to get support in IE and Safari, the only browsers that don't support the new method.

Template Literals

This quick tip comes from James Hibbard, one of SitePoint's JavaScript content editors.

In my opinion, one of the best (and probably most underrated) features of ES6 is the template literal. Template literals are multi-line strings that can include interpolated expressions and which have the potential to make your code much easier to read. To give you a better idea of what that means, let's look at an example.

Given a `company` object:

```
var company = {
  'name' : 'SitePoint',
  'url' : 'https://www.sitepoint.com/'
};
```

In ES5 I might have done something like this, to output the data contained in that object into a list:

```
var html = '<ul>\n';
html += ' <li>Name: ' + company.name + '<li>\n';
html += ' <li>URL: <a href="' + company.url + '">' + company.url
      + '</a><li>\n';
html += '</ul>';
```

However, with the new ES6 syntax, it becomes much more readable:

```
const {name, url} = company;
const html = `
<ul>
  <li>Name: ${name}<li>
  <li>URL: <a href="${url}">${url}</a><li>
</ul>
`;
```

As shown, template literals are declared using backticks. Additionally, anything inside `${}` is evaluated as an expression – this means I'm not limited to just variables, but anything that resolves to a value. So one of my list items could look like this:

```
const html = `
  <li>${ Object.keys(company).map(
    (value) => company[value]
  ).join(', ') }</li>
`;
```

View it on CodePen with the above example.

That's template literals in a nutshell. Check out MDN's article on the topic for more info.

Tagged Template Literals

This quick tip once again comes from James Hibbard, one of SitePoint's JavaScript content editors.

In a previous tip, I introduced template literals, an ES6 feature that adds support for multi-line strings and string interpolation to JavaScript. In this tip, I'd like to demonstrate a further powerful feature of template literals: Tagged templates. These enable you to transform the output of a template literal using a custom function. For example:

```
function upcase(strings, ...values) {
  return values.map(
    name => name[0].toUpperCase() + name.slice(1)
  ).join(' ') + strings[2];
}

const first = 'brendan';
const last = 'eich';

console.log(upcase`${first} ${last} is the creator of JavaScript!`);
// Brendan Eich is the creator of JavaScript!
```

Notice how the `upcase()` function is called on the last line. This function receives two arguments:

- `strings` — an array of string literals
- `values` — the processed substitution expressions

In my example these will be:

- `['', ' ', ' is the creator of JavaScript!']`
- `['brendan', 'eich']`

There are many potential use cases for tagged template literals. For example, you could use them to sanitize user input in templates, implement a basic form of internationalization, or correctly encode a URL before outputting it. This fewature is also often used to get the raw version of a string where character escape sequences have not yet been processed:

```
console.log(String.raw`\nBrendan Eich:\t\tCreator of JavaScript\n`);
// \nBrendan Eich:\tCreator of JavaScript\n
```

Here's a CodePen demo with the above examples.

The Range API

The DOM spec describes something called *DOM Ranges*, also referred to as the Range API, which allows you to select and manipulate chunks of data inside the DOM tree. Generally speaking, you can do this sort of thing with more commonly known DOM methods but this API gives you some very specific features to make this somewhat easier.

The spec uses an example similar to what I'll show you here. Let's say I have the following HTML:

```
<p>
  <img src="cat.jpg" alt="Cat"><br>
  Cats don't ride scooters. <em>But they do sit on them</em>.
</p>
```

I'll run the following JavaScript (assuming no white space between elements in the above HTML):

```
let range = document.createRange(),
    btn = document.querySelector('button'),
    p = document.querySelector('p'),
    em = document.querySelector('em'),
    start = p.childNodes[2],
    end = em.firstChild;

range.setStart(start, 0);
range.setEnd(end, 4);
range.extractContents();
```

After setting up the range container using the `createRange()` method, the key lines here are the final three where I'm using other methods of the Range API, namely `setStart()`, `setEnd()`, and `extractContents()`.

Using the start/end offsets I've defined which part of the document I want to set as my range, then I extract that portion from the page. In this case I've extracted the text "Cats don't ride scooters. " Upon extraction, the range gets placed into a `DocumentFragment`, from which I can deal with it accordingly. You can see how the range can extract text that is intermixed with elements.

In this CodePen demo I'm using a `click` event on a button to do the extraction, that way you can see exactly what happens on the page. I'm also logging the document fragment so you can examine it to see what exactly gets extracted (use the browser developer tools to view the fragment since CodePen can't handle certain types of logs).

For more info, see Ranges in the DOM spec or Range on MDN. Basic browser support goes back to IE9, but some features are not available in all browsers. Seems pretty safe to use if you're just doing some basic manipulation.

defaultChecked on Radio Buttons and Checkboxes

Here's an interesting DOM feature when working with forms that maybe you haven't seen before. Let's say you have a set of radio buttons or checkboxes in a form, like this:

```
<form id="form">
  <input type="radio" value="one" name="setOne">One
  <input type="radio" value="two" name="setOne" checked>Two
  <input type="radio" value="three" name="setOne">Three
</form>
```

There are three radio buttons here and the second one is selected by default. As you probably know, you can use the `checked` attribute to determine which of those is selected at any given time. So even if the user switches the selection to a different radio button, you can find out which one it is:

```
let myForm = document.getElementById('form');

for (i of myForm.setOne) {
  if (i.checked === true) {
    console.log(i.value);
    // logs "two"
  }
}
```

Even after the user switches the selected button, you can run the same type of script and get the newly selected value. Simple enough.

But what if you want to determine which radio button was selected by default when the page first loaded? You can use the `defaultChecked` property for that:

```
for (i of myForm.setOne) {
  if (i.defaultChecked === true) {
    console.log(i.value);
    // logs "two"
  }
}
```

Here's a CodePen demo with buttons that display the checked value vs the default checked value. Try changing the selected radio option and then use the buttons again.

So no matter what gets changed on the form as a result of user input, the `defaultChecked` property will always be the value of the one that was selected by default. The only strange behaviour with this feature is the fact that I can set the `defaultChecked` option with JavaScript and I can define it for more than one radio button in a set.

This means if I do the following on the same form:

```
myForm.setOne[0].defaultChecked = true;
```

I will then have two radio buttons that log `true` for `defaultChecked`. So I have to unset the other one if I want to have only one radio option have this attribute. This seems like strange behaviour, to be honest.

Using Negative Values with Array.prototype.slice()

Here's something I recently learned (or maybe I just forgot about it) regarding JavaScript's `Array.slice()` method. `Array.slice()` is a well-known feature that goes back to ES3 that allows you to "slice" a portion of an array using one or two arguments.

The first argument represents where to begin slicing in the array and, somewhat confusingly, the second argument represents the slot after the slice's ending position. Some simple examples:

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

console.log(array.slice(2)); // ["three", "four", "five"]
console.log(array.slice(2, 4)); // ["three", "four"]
console.log(array.slice(1, 3)); // ["two", "three"]
```

As you can see, if you include only one argument, the second argument is assumed to be the length of the array, so it will grab everything from the start point on.

But interestingly, you can also use negative numbers for either of the arguments. So the following does a bit of a bizarre extraction:

```
console.log(array.slice(-4, -1)); // ["two", "three", "four"]
```

In that case I'm extracting the middle three elements in the array using negative values for start/end. But the most useful technique using negative values is when you use a single argument. This allows you to extract the specified number of elements from the back of the array, but still in the order they appear inside the array:

```
console.log(array.slice(-4)); // ["two", "three", "four", "five"]
console.log(array.slice(-2)); // ["four", "five"]
```

CodePen with all examples

More info on Array.slice on MDN and, as mentioned, this is safe to use in all browsers.

getElementsByName()

Probably unbeknownst to many of you, there's actually a DOM method called `getElementsByName()` that has good overall browser support (more on that in a minute). It does exactly what its name suggests: it lets you collect elements using the `name` attribute present in the HTML.

Assuming, for example, that I have a form with a couple of different radio button collections, I could do the following to gather these sets into live `nodeList` collections:

```
let setOne = document.getElementsByName('setOne'),
    setTwo = document.getElementsByName('setTwo');

console.log(setOne.length, setTwo.length);
```

In this case I'm logging out the length of each collection, but I could do whatever I want with the elements, the same as I would be able to do with similar DOM methods.

There are probably lots of reasons that `getElementsByName()` has been relatively unknown until now and some of this is detailed on the Dottoro reference's article on the topic. As that page explains, in older versions of Opera and IE, this method returned elements that had the specified name in the `id` attribute. Also, IE was case sensitive for the value of the `name` attribute whereas the other browsers were not. From my testing, this method now seems to work the same in all browsers.

Since the `name` attribute in HTML has become obsolete on certain elements (e.g. links and images), probably the best use of it is to collect form inputs or maybe `meta` elements, which can use valid `name` attributes. I suppose it could also be used in a legacy document to collect old elements that use the `name` attribute.

Of course, even if a document is invalid because of a `name` attribute, you can still reference the element in the same way. Your page won't validate, but you'll have access to it and any other elements that use the `name` attribute.

In this CodePen demo I'm using `getElementsByName()` on two radio collections and I've also included a few other elements with the same `name` attributes as the first radio collection so you can see how the browser handles these.

Performance Testing with console.time() and console.timeEnd()

The book Secrets of the JavaScript Ninja is now in its second edition. If you're serious about making progress as a JavaScript developer, I highly recommend it. In the introductory sections, the authors introduce three primary best practices that are part of being an expert app developer. One of those is performance analysis.

To help with this, the book provides a simple example to demonstrate what they'll do in later examples to collect performance information. The example uses the console API to measure the time that it takes for a section of code to run. Here's an example similar to the one provided in the book:

```
console.time('My Operation');

for (let i=0;i<10000;i++) {
  // perform operation here...
}

console.timeEnd('My Operation');
```

I'm sure many of you already know this technique, but for those of you who have only used the `log()` command of the console API, this is a new one you can add to your repertoire. The gist of it is that you sandwich the code you want to speed test between `console.time()` and `console.timeEnd()` calls.

The string passed into `console.time()` is a label parameter that needs to be exactly the same as that passed into `console.timeEnd()`, because this is the identifier for this particular timer. When the code finishes running, the `console.timeEnd()` call will spit out the time of the operation in the console.

You can test out my example in this CodePen demo where I'm toggling a class on the `body` element 10,000 times. Make sure to open up the real console in your browser's developer tools rather than the CodePen one because CodePen doesn't display the time. In my testing in Chrome, I got varying results between 15ms and 25ms.

The book mentions that it's common to do something like this where you perform a single operation thousands or maybe even millions of times to measure something so that you can get a reliable measurement. For example, toggling a class once would take maybe a fraction of a millisecond at most and so that wouldn't be very helpful.

Dynamic Downloads with Data URIs and the download Attribute

Some time ago, HTML5 added the `download` attribute for anchors (`a` elements, or links). This feature indicates to the browser that when a link is clicked, instead of visiting the URL, the user should be prompted to download the requested file.

If the `download` attribute is specified with no value, the browser will attempt download the file specified in the `href` attribute (assuming it exists). If a value is applied to the `download` attribute, this will be the file name, which the user can then change when the OS's native download dialog appears.

If I combine the `download` attribute with some JavaScript, I have the ability to let the user edit the content of the file to be downloaded before it's downloaded (actually, before the file even exists). Basically, I'm generating the file on the fly using a data URI. Here's my HTML:

```
<textarea id="txt"></textarea>
<a href="nothing.html" id="link" download="code.html">Download HTML</a>
```

The user is given the option to type some HTML code into the `textarea` element. When they click the link, the following JavaScript will execute:

```
function downloadCode(link, code) {
  link.href = 'data:text/html;charset=utf-8,' + code;
}

let link = document.getElementById('link');

link.addEventListener('click', function (e) {
  downloadCode(this, txt.value);
}, false);
```

Try it on CodePen

The key part of the code is the single line inside the `downloadCode()` function. As mentioned, the browser will, by default, attempt to download whatever is specified in the `href` attribute. On this line, I'm changing the attribute to a data URI, specified with a content type of `text/html` and a UTF-8 character encoding. The content of the `href` attribute after the character encoding will be the content of the file that's created and downloaded.

This sort of thing is pretty common in apps that generate graphics on the fly (maybe via Canvas or SVG) and which allow users to download the graphics. With data URIs, you can pretty much let the user download any type of content, and it can be generated dynamically based on previous user input. You just have to specify the content type correctly.

Credit for this tip goes to an older article by Chris West and this tweet by Cody Lindley where he links to examples demonstrating downloading three types of content. You can also check out MDN's article for more details on the `download` attribute.

Math.abs() Behavior with Different Values

I'm sure many of you have seen the `Math.abs()` method, which is one of the many properties and methods available on JavaScript's built-in `Math` object. `Math.abs()` returns the absolute value of a number (hence, "abs"). So you pass in something and it will either give you zero, `NaN`, or the absolute number of what's passed in.

Unlike other methods, this is not a constructor. So you can't call `abs()` on an object you've created. In the code example below, you'll see the results you get when you try passing in different things:

```
Math.abs(.57);       // 0.57
Math.abs(4.65);      // 4.65
Math.abs('-1');      // 1
Math.abs(-2);        // 2
Math.abs(null);      // 0
Math.abs('');        // 0
Math.abs([]);        // 0
Math.abs([2]);       // 2
Math.abs(Infinity);  // Infinity
Math.abs(-Infinity); // Infinity
Math.abs(3,000);     // 3
Math.abs(3, 'd');    // 3
Math.abs([1,2]);     // NaN
Math.abs({});        // NaN
Math.abs('5abc');    // NaN
Math.abs();          // NaN
```

View on CodePen

You'll notice that fractions using decimals are returned intact, while strings that are numbers will return the number correctly. Meanwhile non-number strings and objects return `NaN`. Also notice that multi-item arrays with all numbers will return `NaN`, while a single item array will return the number. A somewhat strange behavior is the fact that empty strings or empty arrays will return zero.

Also notice that when I tried to enter a number with a comma (which I doubt would ever happen), the digits after the comma are treated as a second argument, which is ignored, returning the absolute value of the first argument. And the same happens with a string as the second argument.

The nodeValue property

There are many ways to read content from elements and nodes. Here's another one that allows you to read and write an individual node's value. It uses the `nodeValue` property of the `Node` interface.

The `nodeValue` of an element will always return `null`, whereas text nodes, `CDATA` sections, comment nodes, and even attribute nodes will return the text value of the node. To demonstrate its use, I'll use the following HTML:

```
<section class="home">
  <p>Example. <span>Example paragraph one.</span></p>
  <p>Example paragraph two.</p>
  <p>Example paragraph three.</p>
  <p>Example paragraph four.</p>
  <!-- comment text -->
</section>
```

Now I'll run the following JavaScript to read/write a few of the nodes:

```
let s = document.querySelector('section'),
    p1 = s.querySelector('p');

// reading node values
console.log(s.nodeValue); // null for all elements
console.log(s.attributes[0].nodeValue); // "home"
console.log(s.childNodes[7].nodeValue); // " comment text "

// changing nodeValue of first node inside first paragraph
p1.firstChild.nodeValue = 'inserted text<br>';
```

View on CodePen

Notice the second `console.log` is displaying the text of an attribute node. This would be the equivalent of using `getAttribute()`, with the difference being that `getAttribute()` acts on elements, whereas `nodeValue` can be applied to any type of node.

Also notice that I'm using `nodeValue` to read the contents of an HTML comment. This is one of many ways you can do this. This would be the equivalent of reading the `textContent` property or `data` property of the comment node. As you can see from the final line in that code example, I'm able to define the `nodeValue` of one of the text nodes, so this isn't read-only.

A few other things to note regarding defining the `nodeValue` property: Setting `nodeValue` will return the value that was set, and you cannot set the `nodeValue` when the `nodeValue` is `null` (unless of course you change it to `null`, which is the same as an empty string that's changeable later).

MDN's article on nodeValue has a chart that lists the different node types and what their `nodeValue` will return.

Using the debugger Statement

As you might already know, your browser's developer tools can be used to debug JavaScript by utilizing breakpoints. Breakpoints allow you to pause script execution as well as step into, out of, and over function calls. Breakpoints are added in your developer tools by means of line numbers where you indicate where you want script execution to be paused.

This is fine until you start changing your code. Now a defined breakpoint may no longer be in the place you want it to be. Take the following code as an example:

```
1  function doSomething() {
2    console.log('first log...');
3    console.log('last log...');
4  }
5
6  doSomething();
```

If I open this code in my browser's developer tools debugger, I can place a breakpoint before one of the lines inside the `doSomething()` function. Let's say I want to pause script execution before the "last log..." message. This would require that I place the breakpoint on that line. In this case, it would be line 3 of my code.

But what if I add the breakpoint, then I add another line of code before that line?

```
1  function doSomething() {
2    console.log('first log...');
3    console.log('second log...');
4    console.log('last log...');
5  }
6
7  doSomething();
```

If I refresh the page, the breakpoint will still be there but now the script will pause execution on the "second log..." message instead of the "last log..." message. Again, this is because the breakpoints are based on line numbers. The debugger is still stopping on line 3, but that's not what I want.

Enter JavaScript's `debugger` statement.

Instead of setting breakpoints directly in the developer tools, I can use the `debugger` statement to tell the developer tools where to pause execution:

```
function doSomething() {
  console.log('first log...');
  debugger;
  console.log('last log...');
}

doSomething();
```

Try it on CodePen

Now I can add as much code as I want prior to the `debugger` statement and the script will still pause in the right place, without any concerns over changing line numbers.

If you've been writing JavaScript for a while, I'm sure this tip is nothing new to you. But those new to debugging will certainly benefit from using this feature, which is part of the official ECMAScript spec. As expected, inserting a `debugger` statement will have no effect on your code if a debugger is not present (e.g. if the developer tools are not open).

innerHTML and HTML Entities

At the top of the MDN article for the `innerHTML` property, you'll find a note that indicates what happens when you use `innerHTML` to grab the content of an element that contains HTML entities like `>` or `&`. This is particularly noteworthy when those entities are not escaped.

For example, let's consider the following HTML:

```
<pre>> format C & celebrate</pre>
```

Notice inside the `pre` element there's a greater-than symbol (`>`), used here to indicate a mock command prompt. There's also an ampersand character (`&`). Neither of these is using its HTML entity equivalent, so they're a bit risky to use in this way. However, due to HTML parsing rules, they will still appear correctly on the page.

If I grab the content of the paragraph element using `innerHTML`, then insert it into another element using the `textContent` property, this is what will appear on the page:

```
&gt; format C &amp; celebrate
```

This is probably not what I want. To get around this, I would want to avoid using `innerHTML` if possible. Instead, I can use `textContent`, which would display the paragraph of text in the way I want it to appear, like this:

```
> format C & celebrate
```

You can see the difference in this CodePen demo where I'm displaying the contents of the paragraph inside three `output` elements. The first uses `innerHTML` to grab the `pre` content and the second uses `textContent`.

The third example uses `innerHTML` to grab the `pre` content but it also uses `innerHTML` instead of `textContent` to display it inside the new element. This works as desired. It will depend on what you're trying to do in any particular case, but it's good to keep in mind the differences.

Of course, if there were other HTML elements inside the `pre`, then using `textContent` may not be the right solution, because the HTML wouldn't be preserved. In that case, I'd probably have to grab all the nodes in the paragraph individually and only use `textContent` on the text nodes.

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.