

Document Object Model (DOM)

JavaScript



What is the Document Object Model?

The [DOM \(Document Object Model\)](#) is an API for HTML and XML documents that allows us to target, read, and manipulate elements and text throughout them.

This quickly became essential in web once JavaScript was introduced, as dynamic page features nearly always involve manipulation of a page's contents.

DOM representations of a webpage are most notably composed of nodes representing elements, and nodes representing text inside of those elements.

There are also nodes representing comments and other data (see the [full specification](#) for details.)

Often there are empty nodes preceding and succeeding each text node for easy injection of additional elements or text contents inside of a given element.

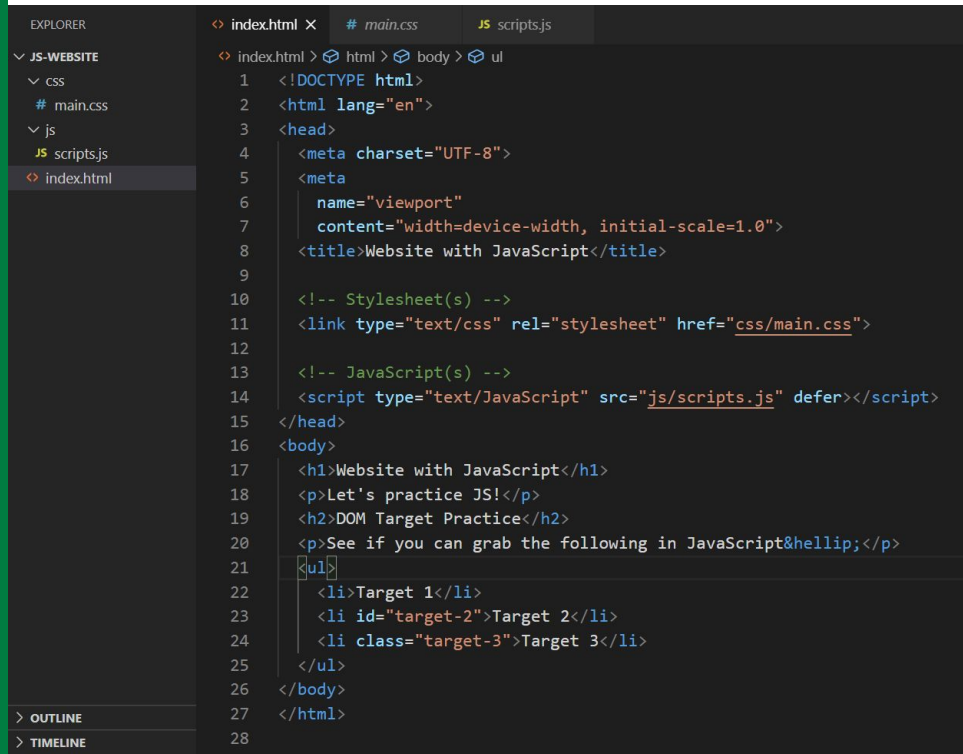
This offers a lot of flexibility to us as developers, and gives us very control over what we can target and how we can manipulate the page.

Let's give this a shot!

Set Up a Web Page

We'll need a page that we can run some experiments on!

Create a basic website that we can use some practice scripts inside of. Ensure you include a css/main.css and js/scripts.js file we can work with in our experiments!



The screenshot shows a code editor with a dark theme. On the left is the 'EXPLORER' sidebar showing a file tree for 'JS-WEBSITE' with folders 'css' and 'js', and files 'main.css', 'scripts.js', and 'index.html'. The 'index.html' file is selected and open in the main editor. The editor has tabs for 'index.html', 'main.css', and 'scripts.js'. The code in 'index.html' is as follows:

```
<?index.html X # main.css JS scripts.js>
<?index.html>
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta
6     name="viewport"
7     content="width=device-width, initial-scale=1.0">
8   <title>Website with JavaScript</title>
9
10  <!-- Stylesheet(s) -->
11  <link type="text/css" rel="stylesheet" href="css/main.css">
12
13  <!-- JavaScript(s) -->
14  <script type="text/JavaScript" src="js/scripts.js" defer></script>
15 </head>
16 <body>
17   <h1>Website with JavaScript</h1>
18   <p>Let's practice JS!</p>
19   <h2>DOM Target Practice</h2>
20   <p>See if you can grab the following in JavaScript&hellip;</p>
21   <ul>
22     <li>Target 1</li>
23     <li id="target-2">Target 2</li>
24     <li class="target-3">Target 3</li>
25   </ul>
26 </body>
27 </html>
28
```

JavaScript's Query Selector Method

JavaScript's DOM implementation provides us with two incredibly powerful methods for retrieving elements in a web page:

- [document.querySelector\(\)](#)
- [document.querySelectorAll\(\)](#)

With the `querySelector` method, you pass in a string containing a selector ([very similar to how we target elements in the CSS language](#).) If you'd like to practice CSS selectors, try out the [CSS Diner!](#)

If an element exists in the page that matches your selector, it will be returned as an object we can use.

It is important to note that the `querySelector` method only returns the first matching element.

If you'd like to retrieve an array of all matching elements, you'll have to use the `querySelectorAll` method. Aside from that difference, these two methods operate in much the same way.

Remember: JavaScript is case-sensitive! When using any keywords, variables, functions, methods, or properties be careful to always spell correctly and use the exact capitalization described in the documentation.

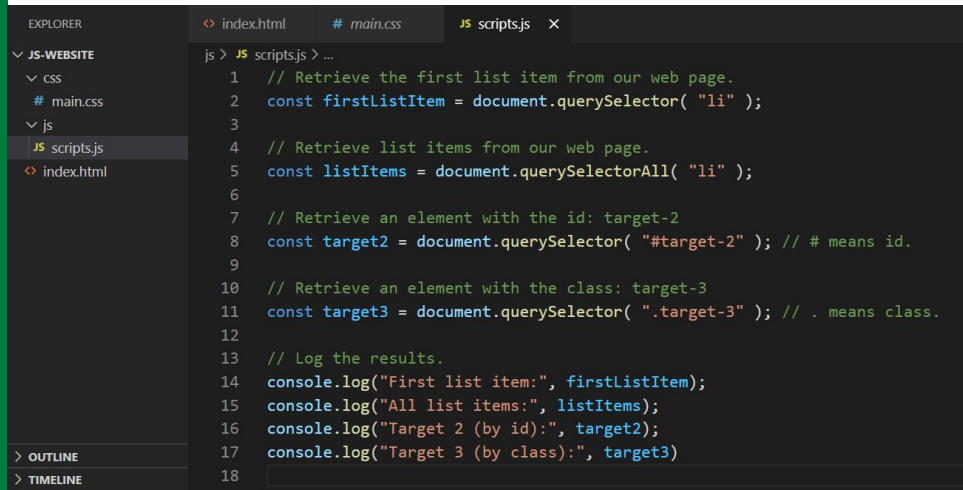
Let's Try Targeting a Few Elements

Using the `querySelector` and `querySelectorAll` methods, we can retrieve elements from the page.

Edit your `scripts.js` file and give these methods a try. Retrieve some elements from the page, and log them to check your result.

See if you were able to find them using JavaScript, or if you end up with a null or undefined value.

If you already have your page opened in a browser, make sure you refresh it after saving new changes in your code!



```
js > JS scripts.js > ...
1 // Retrieve the first list item from our web page.
2 const firstListItem = document.querySelector( "li" );
3
4 // Retrieve list items from our web page.
5 const listItems = document.querySelectorAll( "li" );
6
7 // Retrieve an element with the id: target-2
8 const target2 = document.querySelector( "#target-2" ); // # means id.
9
10 // Retrieve an element with the class: target-3
11 const target3 = document.querySelector( ".target-3" ); // . means class.
12
13 // Log the results.
14 console.log("First list item:", firstListItem);
15 console.log("All list items:", listItems);
16 console.log("Target 2 (by id):", target2);
17 console.log("Target 3 (by class):", target3)
18
```

Check out the Fish You've Caught!

If everything went according to plan, you should have output from each of the four console logs (or more, if you tried a few of your own.)

It can be helpful to note most browser developer tools will highlight the returned element if you hover over the result in your console. This can be a lifesaver when you need to know if you're really targeting the precise element you wanted to!

The reason we often store elements in variables, is to make it easy for us to perform operations on them later in our program.

Website with JavaScript

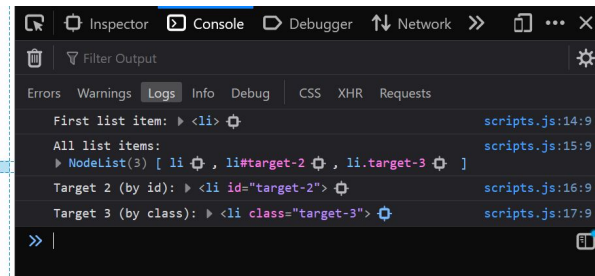
Let's practice JS!

DOM Target Practice

See if you can grab the following in JavaScript...

- Target 1
- Target 2
- Target 3

target3 : 431 x 18



If you see undefined as a result, ensure your variable name matches the name in your log.

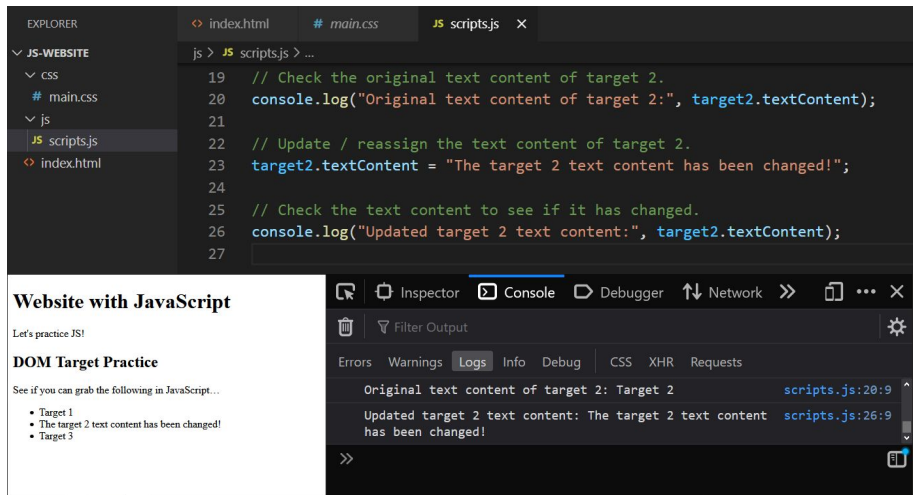
If you see null as a result, double-check your selector and the page contents to ensure there's something on the page that would be a match for it.

Updating the Text Contents of an Element

It's not uncommon for a JavaScript program to update the text contents in an element. Be it a timer, or a status output, a to-do's contents changing, or anything else you can think of—it is something that happens a lot in web applications with a sophisticated front-end.

For a simple change in an element's text contents, we can access and reassign the [textContent](#) property of an element object.

Let's give it a shot: add a few new lines to your scripts.js file...



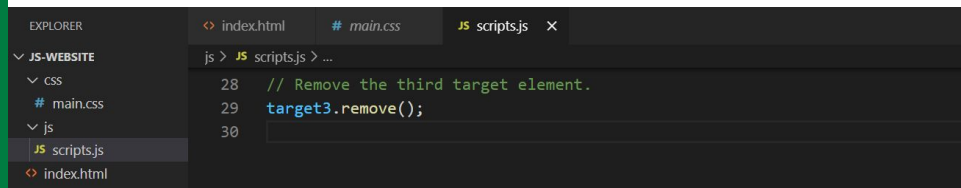
Once you've saved your changes, refresh the page.
Wow, brand new text content!

Removing an Element / Node from the DOM

The DOM node `remove()` method.

You can remove an element or node by executing its `remove()` method.

Add another line to your `scripts.js` file, save, refresh, and... viola: target 3 is gone!



The screenshot shows a code editor with a dark theme. On the left, the 'EXPLORER' sidebar shows a file structure for 'JS-WEBSITE' with folders for 'css', 'js', and 'index.html'. The 'js' folder is expanded, showing 'scripts.js'. The main editor area has tabs for 'index.html', 'main.css', and 'scripts.js'. The 'scripts.js' tab is active, showing a JavaScript file with line numbers 28, 29, and 30. The code at line 29 is `target3.remove();`, which is highlighted in blue. The comment above it says '// Remove the third target element.'

Website with JavaScript

Let's practice JS!

DOM Target Practice

See if you can grab the following in JavaScript...

- Target 1
- The target 2 text content has been changed!

Creating and Populating an Element

JavaScript affords us the [document.createElement\(\)](#) method for creating new elements. We pass in the name of the element as a string into this method, so it knows which type of element we want to make.

Note that when we create an element, it is in memory but not yet visible in the web page—that involves a different step.

To add text to this element, we can use the [document.createTextNode\(\)](#) method, passing into it text we'd like stored in the node.

Neither of these automatically add the HTML element node, or the text node, into anything. The web page doesn't know where to put these until you tell it where to put them!

That's where a node's [append\(\)](#) method comes in. We can use this method to place the text node into our new element, and to place that element into our web page.

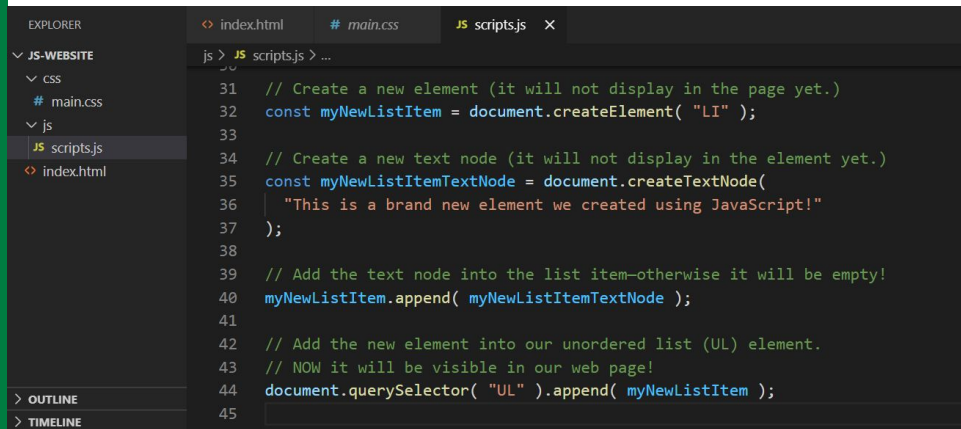
If you'd like for an addition to be placed before any existing content or elements inside of the target element, [prepend\(\)](#) is also available.

Adding a List Item

Using `document.createElement()`, `document.createTextNode()`, and a node's `append()` method.

Add a few more lines to your `scripts.js` file. Save and refresh the page.

If done correctly, we'll see a new list item! This one was created by us using JavaScript.



```
EXPLORER
  < index.html
  # main.css
  JS scripts.js
  < index.html

JS scripts.js
31 // Create a new element (it will not display in the page yet.)
32 const myNewListItem = document.createElement( "LI" );
33
34 // Create a new text node (it will not display in the element yet.)
35 const myNewListItemTextNode = document.createTextNode(
36   "This is a brand new element we created using JavaScript!"
37 );
38
39 // Add the text node into the list item—otherwise it will be empty!
40 myNewListItem.append( myNewListItemTextNode );
41
42 // Add the new element into our unordered list (UL) element.
43 // NOW it will be visible in our web page!
44 document.querySelector( "UL" ).append( myNewListItem );
45
```

Website with JavaScript

Let's practice JS!

DOM Target Practice

See if you can grab the following in JavaScript...

- Target 1
- The target 2 text content has been changed!
- This is a brand new element we created using JavaScript!

Add an ID to an Element

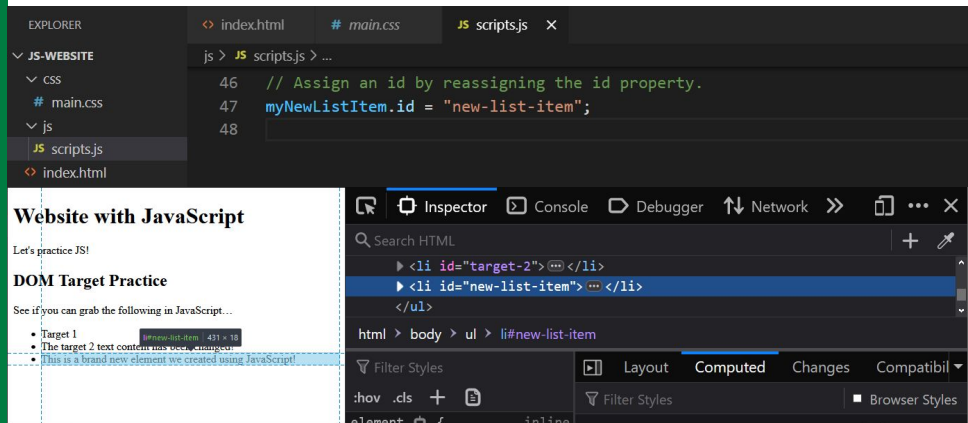
We can reassign the id property of a node to replace the ID currently assigned.

Let's add an ID to our new list item...

Assign the property its new value, save, and refresh your browser. Inspect the targeted element to confirm it has the ID you assigned!

If you don't see the expected result, check the console for errors.

Remember: elements can only have one ID (this is in contrast to how HTML element classes work.)



HTML Classes and JavaScript

To replace the entire class assignment for an element, you can use the [className](#) property (much like the id property.)

However, a newer—and often better—way to handle classes is via the [classList](#) property object. This property is great for accessing, adding, and removing singular class names at a time (in contrast to `className` containing the whole lot of classes in a single space delimited string.)

The `classList` property itself contains a list of each class assigned to the element.

The `classList` property also includes various methods for managing an element's class(es):

- **contains()**
true if argument string is a class name assigned to this element; false if it is not.
- **toggle()**
Adds the argument string as a class if it is not present; removes the class if it is present.
- **add()**
Adds argument string as a class name.
- **replace()**
Replaces class name assigned to the element matching the first argument string with the second argument string.
- **remove()**
Removes argument string class name.

Try Reading, Writing, and Removing Classes

Try out an element node's `classList` property's `contains()`, `add()`, `replace()`, and `remove()` methods.

The screenshot displays a web browser window with a dark theme. The top section shows the Explorer sidebar with a file tree containing `css`, `# main.css`, `js`, `JS scripts.js`, and `index.html`. The main editor area shows the `scripts.js` file with the following JavaScript code:

```
49 console.log( // Check current class name and list.
50   "Current class name and list (Log 1):",
51   myNewListItem.className,
52   myNewListItem.classList
53 );
54
55 // Add some classes to the element.
56 myNewListItem.classList.add( "my-first-js-class" );
57 myNewListItem.classList.add( "my-second-js-class" );
58
59 console.log( // Check current class name and list.
60   "Current class name and list (Log 2):",
61   myNewListItem.className,
62   myNewListItem.classList
63 );
64
65 console.log( // Check if class list contains specific class.
66   "Does element contain \"my-first-js-class\" class?",
67   myNewListItem.classList.contains( "my-first-js-class" )
68 );
69
70 // Remove a class.
71 myNewListItem.classList.remove( "my-second-js-class" );
72
73 // Replace a class.
74 myNewListItem.classList.replace( "my-first-js-class", "a-new-class" );
75
76 console.log( // Check current class name and list.
77   "Current class name and list (Log 3):",
78   myNewListItem.className,
79   myNewListItem.classList
80 );
81
```

The bottom section of the browser shows the "Website with JavaScript" page. It has a heading "DOM Target Practice" and a paragraph: "See if you can grab the following in JavaScript...". Below this is a list of three bullet points:

- Target 1
- The target 2 text content has been changed!
- This is a brand new element we created using JavaScript!

The right sidebar of the browser shows the "Inspector" and "Console" tabs. The "Console" tab is active, displaying the following log messages:

- Log 1: Current class name and list (Log 1): <empty string> DOMTokenList [] (scripts.js:49:9)
- Log 2: Current class name and list (Log 2): my-first-js-class my-second-js-class DOMTokenList ["my-first-js-class", "my-second-js-class"] (scripts.js:59:9)
- Log 3: Does element contain "my-first-js-class" class? true (scripts.js:65:9)
- Log 4: Current class name and list (Log 3): a-new-class DOMTokenList ["a-new-class"] (scripts.js:76:9)

HTML Element innerHTML Property

A very powerful property inside of HTML element objects is the [innerHTML](#) property.

It can be thought of as similar to the `append()` method in that you can use it to add nodes and content to an element in your page. Despite a similar use case, it has some important differences, however.

Many favour this approach in comparison to `document.createElement()` and `append()`, as you can assign a string representation of HTML markup. This is much more familiar and similar to what you type directly into your HTML files.

Assigning a value to the `innerHTML` property will replace the contents of the element completely. It is possible, however, to preserve the original contents by use of concatenation or string interpolation.

A more dire consequence of reassigning this property's value is that it treats the contents of the target element as brand new.

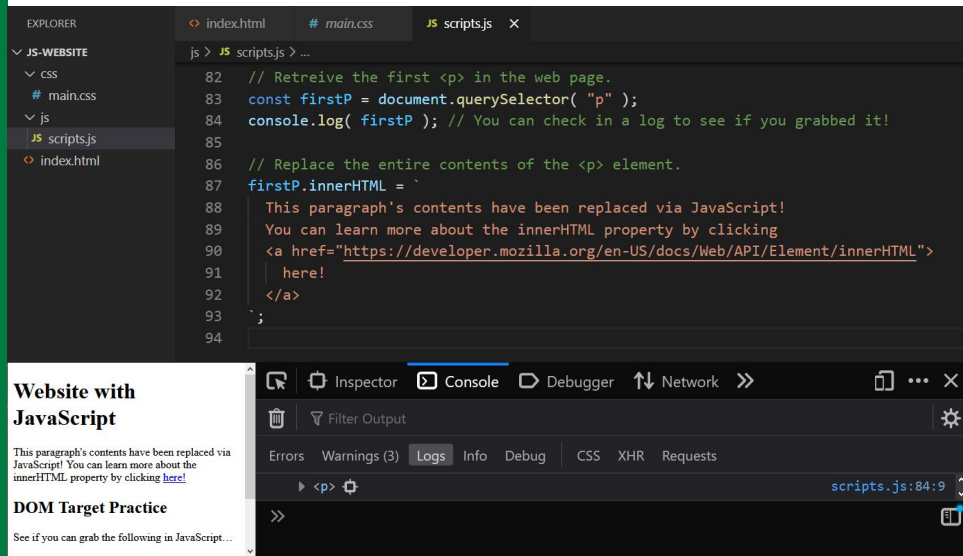
This means that any previous variables storing those child elements, event listeners, or other element instance-specific use cases will be destroyed in the process. Use it carefully.

Using innerHTML

Assign a valid HTML markup string to an element's innerHTML, it will replace its contents.

For multi-line strings, you can use backticks. Single and double quotes are also fine for such assignment, but will not reflect new-lines without use of `\n`.

Remember: using backticks also opens up the door for use of string interpolation via [template literals](#)!



insertAdjacentHTML() Method

If you need your HTML additions to be in a more specific place, consider an HTML element's [insertAdjacentHTML\(\)](#) method.

insertAdjacentHTML() is a very flexible method, in that it allows for injection of HTML code into [any of four places relative to your target element...](#)

https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentHTML#Visualization_of_position_names

Visualization of position names

```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  foo
  <!-- beforeend -->
</p>
<!-- afterend -->
```

Note: The beforebegin and afterend positions work only if the node is in the DOM tree and has a parent element.

This method has two parameters, the first being a “position” string. When passing in your first argument it must match one of:

- beforebegin
- afterbegin
- beforeend
- afterend

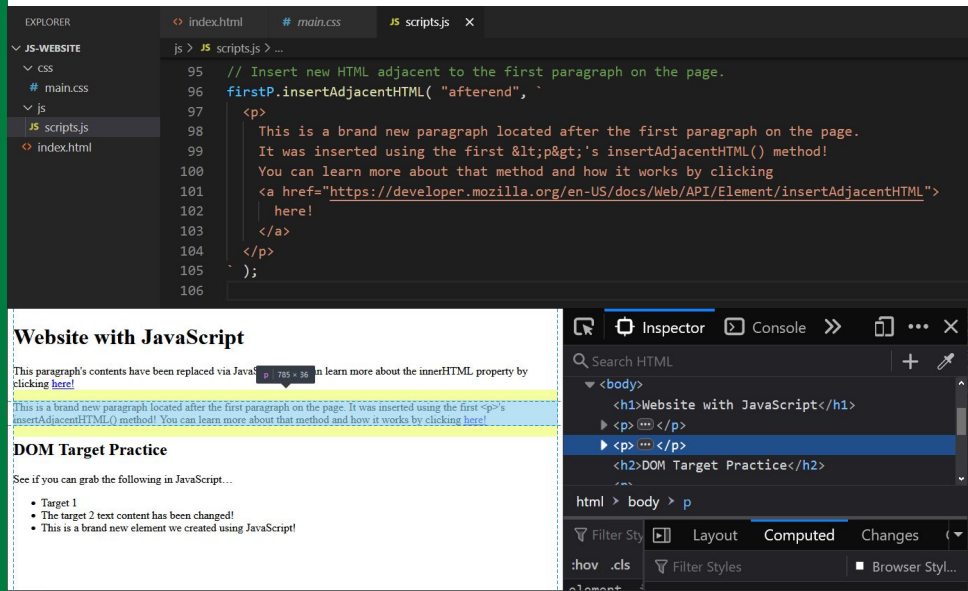
The second argument you will pass into this method is the HTML markup string (much like we had seen with the innerHTML property.)

Inserting an HTML Markup String into a More Specific Place

Let's give `insertAdjacentHTML()` a try.

Attempt inserting a new paragraph after the first paragraph on the page, using the element's `insertAdjacentHTML()` method.

Almost identical to this, but instead for use with node objects, are the `insertAdjacentElement()` and `insertAdjacentText()` methods.



More on JavaScript's DOM Implementation

There's plenty of more reading and practice to be done with the Document Object Model! Here are a few resources to get you started...

- [Document Object Model \(DOM\)](#)
- [The HTML DOM API](#)
- HTML Element [Properties](#) and [Methods](#)
- [The Document Class](#)
- document object [Properties](#) and [Methods](#)
- [W3Schools' JavaScript HTML DOM](#) Introduction
- [The DOM Living Standard](#) Official Specification