TANIA RASCIA

UNDERSTANDING THE DOM — DOCUMENT OBJECT MODEL

ISBN 978-0-9997730-9-3

Understanding the DOM — Document Object Model

Tania Rascia

DigitalOcean, New York City, New York, USA

2020-10

Understanding the DOM — Document Object Model

About DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale. It provides highly available, secure and scalable compute, storage and networking solutions that help developers build great software faster. Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available. For more information, please visit https://www.digitalocean.com or follow @digitalocean on Twitter.

Introduction

About this Book

JavaScript is the de facto programming language of the web, but the language itself does not include any built-in method for working with input/output (I/O), such as graphics display and sound. Instead, the web browser provides an API for accessing the HTML document in a tree structure known as the Document Object Model (DOM). The combination of JavaScript and the DOM is what allows us to create interactive, dynamic websites.

Many modern frameworks, such as React, Vue, and Svelte abstract away much of the DOM from the developer, but frameworks also use the DOM under the hood. The JavaScript library jQuery was also created to make working with the DOM easier, but the modern development practice is to work with the DOM directly. In order to be a proficient web developer, having a deep understanding of what the DOM is and how to work with it is essential. The goal of this book is to provide a base understanding of the DOM, as well as explore examples of the most common and useful methods for interacting with the DOM.

This book is based on the Understanding the DOM tutorial series found on the DigitalOcean Community. The topics that it covers include:

- The DOM and DOM tree structure
- How to access, traverse, and modify nodes and elements in the DOM
- How to modify attributes, classes, and styles in the DOM

- Use events to make interactive, dynamic websites

Each chapter is self-contained and can be followed independently of the others. However, if you are not yet familiar with the concept of the DOM and DOM tree, it is recommended that you read the introductory chapters first.

If you'd like to learn more about JavaScript, visit the DigitalOcean Community's JavaScript section. You can follow along with the How to Code in JavaScript series for a directed learning experience.

Introduction to the DOM

Written by Tania Rascia

The Document Object Model, usually referred to as the DOM, is an essential part of making websites interactive. It is an interface that allows a programming language to manipulate the content, structure, and style of a website. JavaScript is the client-side scripting language that connects to the DOM in an internet browser.

Almost any time a website performs an action, such as rotating between a slideshow of images, displaying an error when a user attempts to submit an incomplete form, or toggling a navigation menu, it is the result of JavaScript accessing and manipulating the DOM. In this article, we will learn what the DOM is, how to work with the `document` object, and the difference between HTML source code and the DOM.

Note: Although the DOM is language agnostic, or created to be independent from a particular programming language, throughout this resource we will focus on and refer to JavaScript's implementation of the HTML DOM.

Prerequisites

In order to effectively understand the DOM and how it relates to working with the web, it is necessary to have an existing knowledge of HTML and CSS. It is also beneficial to have familiarity with fundamental JavaScript syntax and code structure.

What is the DOM?

At the most basic level, a website consists of an HTML document. The browser that you use to view the website is a program that interprets HTML and CSS and renders the style, content, and structure into the page that you see.

In addition to parsing the style and structure of the HTML and CSS, the browser creates a representation of the document known as the Document Object Model. This model allows JavaScript to access the text content and elements of the website document as objects.

JavaScript is an interactive language, and it is easier to understand new concepts by doing. Let's create a very basic website. Create an `index.html` file and save it in a new project directory.

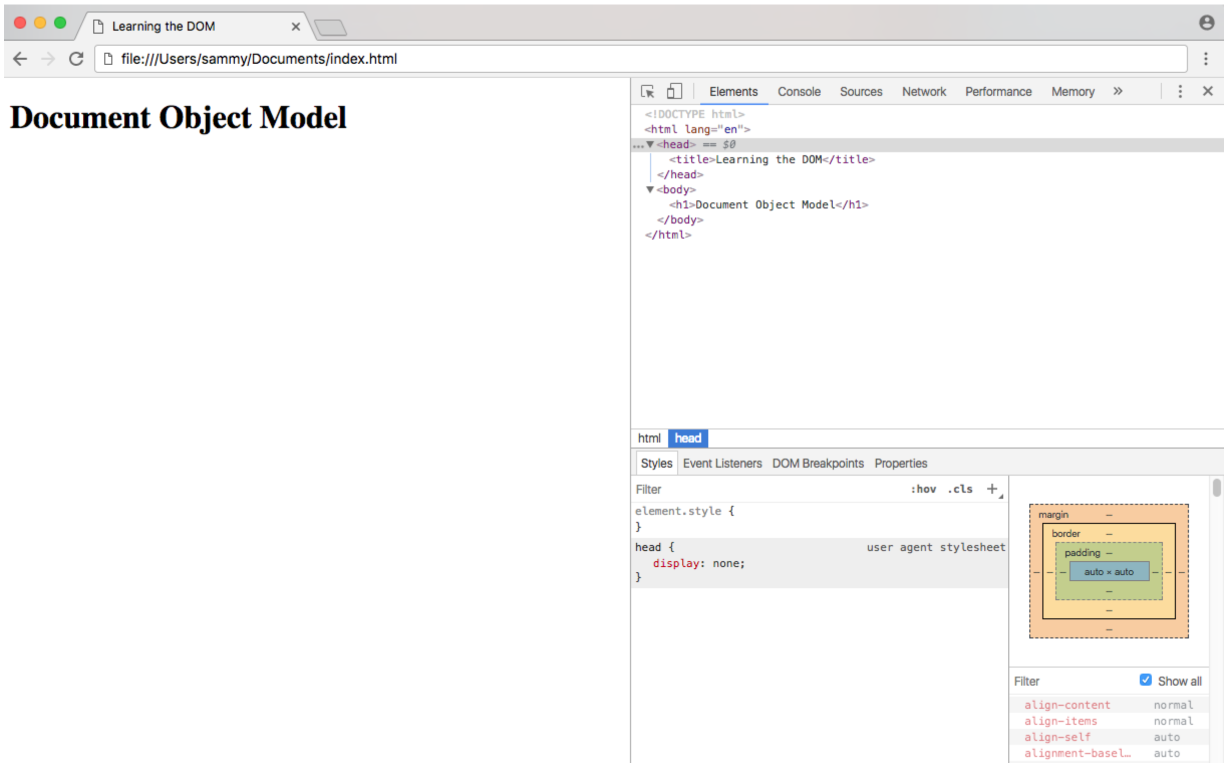
index.html

```html
<!DOCTYPE html>
<html lang="en">

  <head>
    <title>Learning the DOM</title>
  </head>

  <body>
    <h1>Document Object Model</h1>
  </body>

</html>
```

This code is the familiar HTML source of a new website skeleton. It contains the absolute most essential aspects of a website document — a doctype, and an `html` tag with the `head` and `body` nested inside.

For our purposes, we'll be using the Chrome browser, but you can receive similar output from other modern browser. Within Chrome, open up `index.html`. You'll see a plain website with our heading saying "Document Object Model". Right click anywhere on the page and select "Inspect". This will open up Developer Tools.

Under the Elements tab, you'll see the DOM.



```
<!DOCTYPE html>
<html lang="en">
...▼<head> == $0
    <title>Learning the DOM</title>
  </head>
 ▼<body>
    <h1>Document Object Model</h1>
  </body>
</html>
```

```
element.style {
}
head {                                    user agent stylesheet
  display: none;
}
```

```
align-content        normal
align-items          normal
align-self           auto
alignment-basel…     auto
```

DOM Example

In this case, when expanded, it looks exactly the same as the HTML source code we just wrote — a doctype, and the few other HTML tags that we added. Hovering over each element will highlight the respective element in the rendered website. Little arrows to the left of the HTML elements allow you to toggle the view of nested elements.

The Document Object

The `document` object is a built-in object that has many properties and methods that we can use to access and modify websites. In order to understand how to work with the DOM, you must understand how objects work in JavaScript. Review Understanding Objects in JavaScript if you don't feel comfortable with the concept of objects.

In Developer Tools on index.html, move to the Console tab. Type `document` into the console and press `ENTER`. You will see that what is output is the same as what you see in the Elements tab.

```
document;
```

Output

```
#document
<!DOCTYPE html>
<html lang="en">

  <head>
    <title>Learning the DOM</title>
  </head>

  <body>
    <h1>Document Object Model</h1>
  </body>

</html>
```

Typing `document` and otherwise working directly in the console is not something that you'll generally ever do outside of debugging, but it helps solidify exactly what the `document` object is and how to modify it, as we will discover below.

What is the Difference Between the DOM and HTML Source Code?

Currently, with this example, it seems that HTML source code and the DOM are the exact same thing. There are two instances in which the browser-generated DOM will be different than HTML source code:

- The DOM is modified by client-side JavaScript
- The browser automatically fixes errors in the source code

Let's demonstrate how the DOM can be modified by client-side JavaScript. Type the following into the console:

```
document.body;
```

The console will respond with this output:

Output

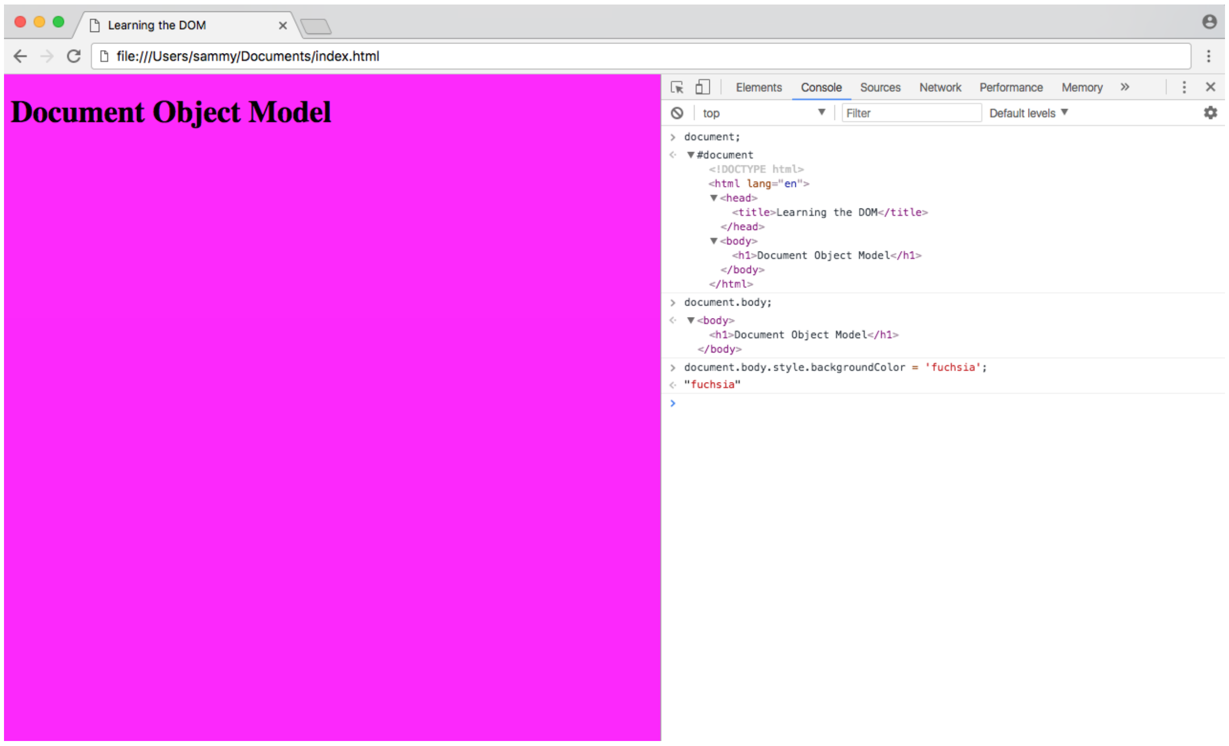
```
<body>
  <h1>Document Object Model</h1>
</body>
```

`document` is an object, `body` is a property of that object that we have accessed with dot notation. Submitting `document.body` to the console outputs the `body` element and everything inside of it.

In the console, we can change some of the live properties of the `body` object on this website. We'll edit the `style` attribute, changing the background color to `fuchsia`. Type this into the console:

```
document.body.style.backgroundColor = 'fuchsia';
```

After typing and submitting the above code, you'll see the live update to the site, as the background color changes.



```
> document;
<· ▼#document
    <!DOCTYPE html>
    <html lang="en">
    ▼<head>
        <title>Learning the DOM</title>
      </head>
    ▼<body>
        <h1>Document Object Model</h1>
      </body>
    </html>
> document.body;
<· ▼<body>
    <h1>Document Object Model</h1>
  </body>
> document.body.style.backgroundColor = 'fuchsia';
<· "fuchsia"
>
```

DOM Live Modification

Switching to the Elements tab, or typing `document.body` into the console again, you will see that the DOM has changed.

Output

```
<body style="background-color: fuchsia;">
  <h1>Document Object Model</h1>
</body>
```

Note: In order to change the `background-color` CSS property, we had to type `backgroundColor` in the JavaScript. Any hyphenated CSS property will be written in camelCase in JavaScript.

The JavaScript code we typed, assigning `fuchsia` to the background color of the `body`, is now a part of the DOM.

However, right click on the page and select “View Page Source”. You will notice that the source of the website does not contain the new style attribute we added via JavaScript. The source of a website will not change and will never be affected by client-side JavaScript. If you refresh the page, the new code we added in the console will disappear.

The other instance in which the DOM might have a different output than HTML source code is when there are errors in the source code. One common example of this is the `table` tag — a `tbody` tag is required inside a `table`, but developers often fail to include it in their HTML. The browser will automatically correct the error and add the `tbody`, modifying the DOM. The DOM will also fix tags that have not been closed.

Conclusion

In this tutorial, we defined the DOM, accessed the `document` object, used JavaScript and the console to update a property of the `document` object, and went over the difference between HTML source code and the DOM.

For more in-depth information on the DOM, review the Document Object Model (DOM) page on the Mozilla Developer Network.

In the next tutorial, we will review important HTML terminology, learn about the DOM tree, discover what nodes are, learn about the most common types of nodes, and begin creating interactive scripts with JavaScript.

Understanding the DOM Tree and Nodes

Written by Tania Rascia

The DOM is often referred to as the DOM tree, and consists of a tree of objects called nodes. In the Introduction to the DOM, we went over what the Document Object Model (DOM) is, how to access the `document` object and modify its properties with the console, and the difference between HTML source code and the DOM.

In this tutorial, we will review HTML terminology, which is essential to working with JavaScript and the DOM, and we will learn about the DOM tree, what nodes are, and how to identify the most common node types. Finally, we will move beyond the console and create a JavaScript program to interactively modify the DOM.

HTML Terminology

Understanding HTML and JavaScript terminology is essential to understanding how to work with the DOM. Let's briefly review some HTML terminology.

To begin, let's take a look at this HTML element.

```
<a href="index.html">Home</a>
```

Here we have an anchor element, which is a link to `index.html`.

- `a` is the tag
- `href` is the attribute

- `index.html` is the attribute value
- `Home` is the text.

Everything between the opening and closing tag combined make the entire HTML element.

We'll be working with the `index.html` from the previous tutorial:

index.html

```
<!DOCTYPE html>
<html lang="en">

  <head>
    <title>Learning the DOM</title>
  </head>

  <body>
    <h1>Document Object Model</h1>
  </body>

</html>
```

The simplest way to access an element with JavaScript is by the `id` attribute. Let's add the link we have above into our `index.html` file with an `id` of `nav`.

index.html

```
...
<body>
  <h1>Document Object Model</h1>
  <^><a id="nav" href="index.html">Home</a><^>
</body>
...
```

Load or reload the page in your browser window and look at the DOM to ensure that the code has been updated.

We're going to use the `getElementById()` method to access the entire element. In the console, type the following:

```
document.getElementById('nav');
```

Output

```
<a id="nav" href="index.html">Home</a>
```

We have retrieved the entire element using `getElementById()`. Now, instead of typing that object and method every time we want to access the `nav` link, we can place the element into a variable to work with it more easily.

```
let navLink = document.getElementById('nav');
```

The `navLink` variable contains our anchor element. From here, we can easily modify attributes and values. For example, we can change where the link goes by changing the `href` attribute:

```
navLink.href = 'https://www.wikipedia.org';
```

We can also change the text content by reassigning the `textContent` property:

```
navLink.textContent = 'Navigate to Wikipedia';
```

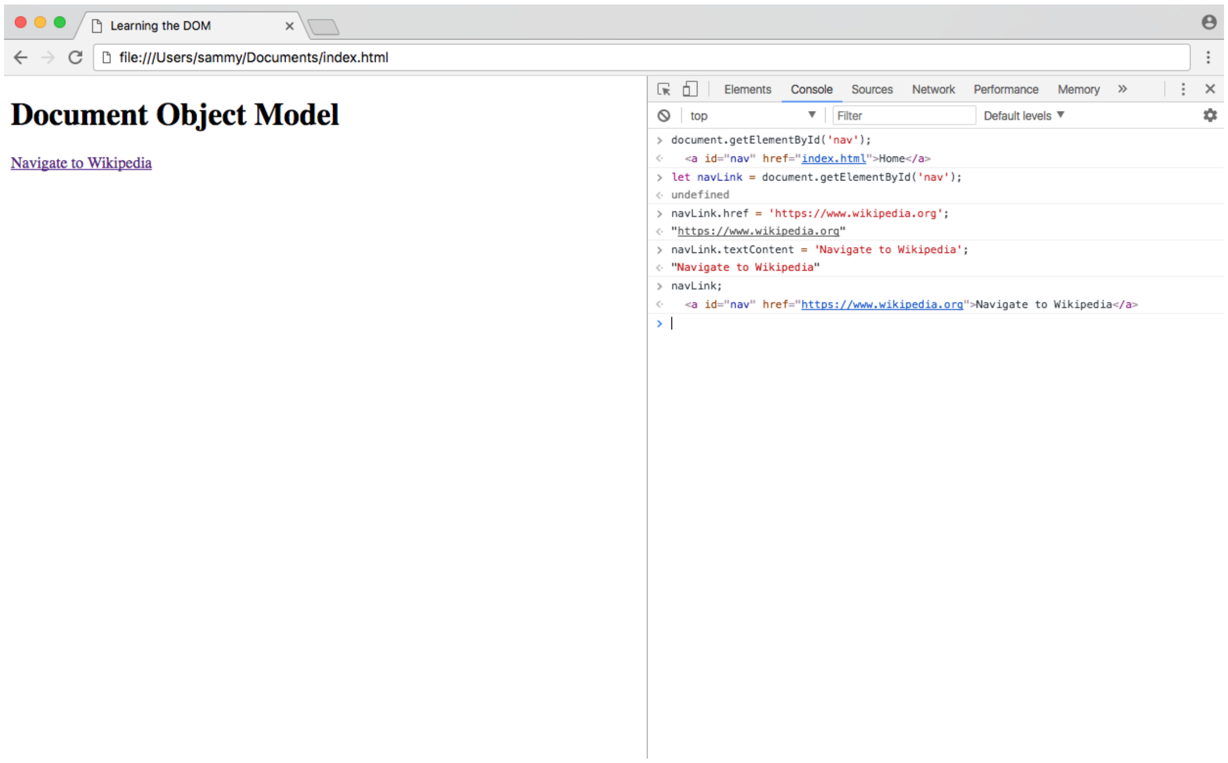
Now when we view our element, either in the console or by checking the Elements tag, we can see how the element has been updated.

```
navLink;
```

Output

```
<a id="nav"
href="https://www.wikipedia.org/">Navigate to
Wikipedia</a>
```

This is also reflected on the front-end of the website.



```
> document.getElementById('nav');
<   <a id="nav" href="index.html">Home</a>
> let navLink = document.getElementById('nav');
< undefined
> navLink.href = 'https://www.wikipedia.org';
< "https://www.wikipedia.org"
> navLink.textContent = 'Navigate to Wikipedia';
< "Navigate to Wikipedia"
> navLink;
<   <a id="nav" href="https://www.wikipedia.org">Navigate to Wikipedia</a>
> |
```

Updated Link via DOM

Refreshing the page will revert everything back to their original values.

At this point, you should understand how to use a `document` method to access an element, how to assign an element to a variable, and how to modify properties and values in the element.

The DOM Tree and Nodes

All items in the DOM are defined as nodes. There are many types of nodes, but there are three main ones that we work with most often:

- Element nodes
- Text nodes
- Comment nodes

When an HTML element is an item in the DOM, it is referred to as an element node. Any lone text outside of an element is a text node, and an HTML comment is a comment node. In addition to these three node types, the `document` itself is a document node, which is the root of all other nodes.

The DOM consists of a tree structure of nested nodes, which is often referred to as the DOM tree. You may be familiar with an ancestral family tree, which consists of parents, children, and siblings. The nodes in the DOM are also referred to as parents, children, and siblings, depending on their relation to other nodes.

To demonstrate, create a `nodes.html` file. We'll add text, comment, and element nodes.

nodes.html

```html
<!DOCTYPE html>
<html>

  <head>
    <title>Learning About Nodes</title>
  </head>

  <body>
    <h1>An element node</h1>
    <!-- a comment node -->
    A text node.
  </body>

</html>
```

The `html` element node is the parent node. `head` and `body` are siblings, children of `html`. `body` contains three child nodes, which are all siblings — the type of node does not change the level at which it is nested.

Note: When working with an HTML-generated DOM, the indentation of the HTML source code will create many empty text nodes, which won't be visible from the DevTools Elements tab. Read about Whitespace in the DOM

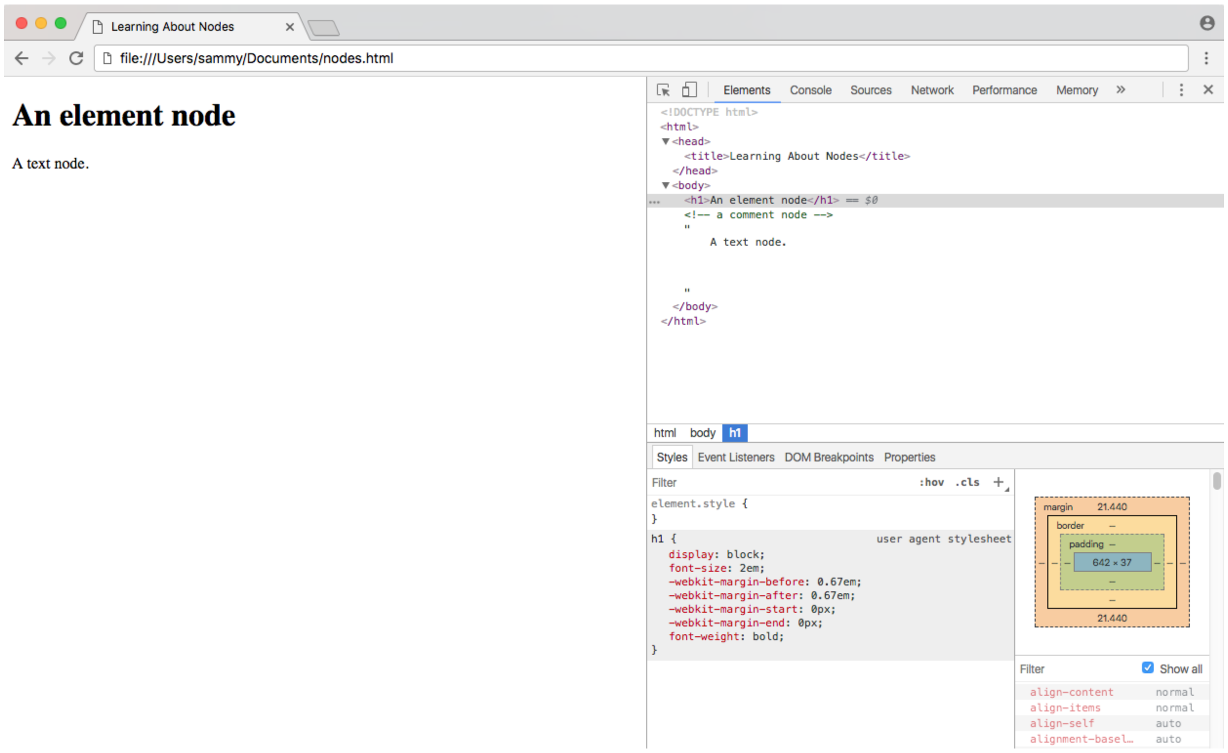
Identifying Node Type

Every node in a document has a node type, which is accessed through the `nodeType` property. The Mozilla Developer Network has an up-to-date list of all node type constants. Below is a chart of the most common node types that we are working with in this tutorial.

Node Type	Value	Example
`ELEMENT_NODE`	1	The `<body>` element
`TEXT_NODE`	3	Text that is not part of an element
`COMMENT_NODE`	8	`<!-- an HTML comment -->`

In the Elements tab of Developer Tools, you may notice that whenever you click on and highlight any line in the DOM the value of == `$0` will appear next to it. This is a very handy way to access the currently active element in Developer Tools by typing `$0`.

In the console of nodes.html, click on the first element in the `body`, which is an `h1` element.



An element node

A text node.

```
<!DOCTYPE html>
<html>
▼<head>
    <title>Learning About Nodes</title>
  </head>
▼<body>
    <h1>An element node</h1> == $0
    <!-- a comment node -->
    "
        A text node.


    "
  </body>
</html>
```

```
element.style {
}
h1 {                                   user agent stylesheet
   display: block;
   font-size: 2em;
   -webkit-margin-before: 0.67em;
   -webkit-margin-after: 0.67em;
   -webkit-margin-start: 0px;
   -webkit-margin-end: 0px;
   font-weight: bold;
}
```

```
align-content          normal
align-items            normal
align-self             auto
alignment-basel…       auto
```

DOM Node Type

In the console, get the node type of the currently selected node with the `nodeType` property.

```
$0.nodeType;
```

Output

```
1
```

With the `h1` element selected, you would see `1` as the output, which we can see correlates to `ELEMENT_NODE`. Do the same for the text and the comment, and they will output `3` and `8` respectively.

When you know how to access an element, you can see the node type without highlighting the element in the DOM.

```
document.body.nodeType;
```

Output

```
1
```

In addition to `nodeType`, you can also use the `nodeValue` property to get the value of a text or comment node, and `nodeName` to get the tag name of an element.

Modifying the DOM with Events

Up until now, we've only seen how to modify the DOM in the console, which we have seen is temporary; every time the page is refreshed, the changes are lost. In the Introduction to the DOM tutorial, we used the console to update the background color of the body. We can combine what we've learned throughout this tutorial to create an interactive button that does this when clicked.

Let's go back to our `index.html` file and add a `button` element with an `id`. We'll also add a link to a new file in a new `js` directory `js/scripts.js`.

index.html

```
<!DOCTYPE html>
<html lang="en">

  <head>
    <title>Learning the DOM</title>
  </head>

  <body>
    <h1>Document Object Model</h1>
    <button id="changeBackground">Change
Background Color</button>

    <script src="scripts.js"></script>
  </body>

</html>
```

An event in JavaScript is an action the user has taken. When the user hovers their mouse over an element, or clicks on an element, or presses a specific key on the keyboard, these are all types of events. In this particular case, we want our button to listen and be ready to perform an action when the user clicks on it. We can do this by adding an event listener to our button.

Create `scripts.js` and save it in the new `js` directory. Within the file, we'll first find the `button` element and assign it to a variable.

js/scripts.js

```
let button =
document.getElementById('changeBackground');
```

Using the `addEventListener()` method, we will tell the button to listen for a click, and perform a function once clicked.

js/scripts.js

```
...
button.addEventListener('click', () => {
  // action will go here
});
```

Finally, inside of the function, we will write the same code from the previous tutorial to change the background color to `fuchsia`.

js/scripts.js

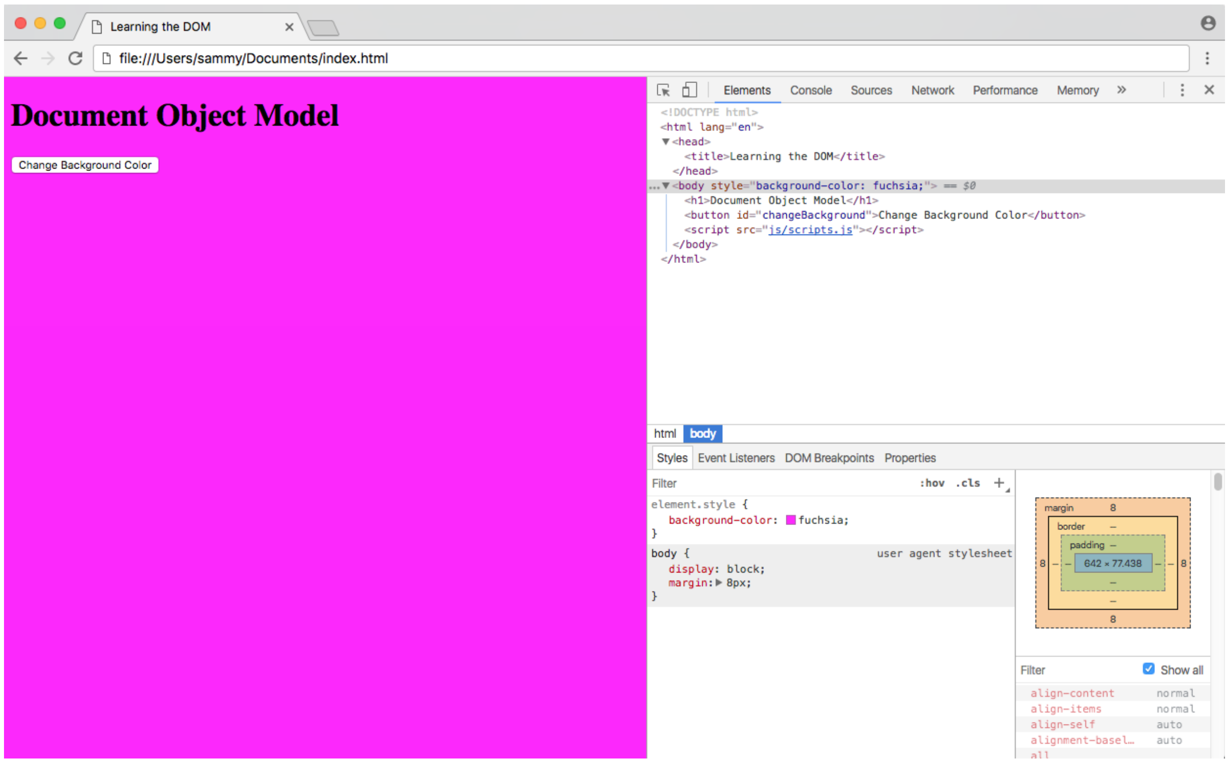
```
...
document.body.style.backgroundColor = 'fuchsia';
```

Here is our entire script:

js/scripts.js

```
let button =
document.getElementById('changeBackground');

button.addEventListener('click', () => {
  document.body.style.backgroundColor = 'fuchsia';
});
```

Once you save this file, refresh `index.html` in the browser. Click the button, and the event will fire.



```
<!DOCTYPE html>
<html lang="en">
▼<head>
    <title>Learning the DOM</title>
  </head>
…▼<body style="background-color: fuchsia;"> == $0
    <h1>Document Object Model</h1>
    <button id="changeBackground">Change Background Color</button>
    <script src="js/scripts.js"></script>
  </body>
</html>
```

```
element.style {
   background-color: fuchsia;
}
body {                                    user agent stylesheet
   display: block;
   margin:▶ 8px;
}
```

```
align-content          normal
align-items            normal
align-self             auto
alignment-basel…       auto
all
```

Modify Background with Events

The background color of the page has changed to fuchsia due to the JavaScript event.

Conclusion

In this tutorial, we reviewed terminology that will allow us to understand and modify the DOM. We learned how the DOM is structured as a tree of nodes that will usually be HTML elements, text, or comments, and we created a script that would allow a user to modify a website without having to manually type code into the developer console.

How To Access Elements in the DOM

Written by Tania Rascia

In Understanding the DOM Tree and Nodes, we went over how the DOM is structured as a tree of objects called nodes, and that nodes can be text, comments, or elements. Usually when we access content in the DOM, it will be through an HTML element node.

In order to be proficient at accessing elements in the DOM, it is necessary to have a working knowledge of CSS selectors, syntax and terminology as well as an understanding of HTML elements. In this tutorial, we will go over several ways to access elements in the DOM: by ID, class, tag, and query selectors.

Overview

Here is a table overview of the five methods we will cover in this tutorial.

Gets	Selector Syntax	Method
ID	`#demo`	`getElementById()`
Class	`.demo`	`getElementsByClassName()`
Tag	`demo`	`getElementsByTagName()`
Selector (single)		`querySelector()`
Selector (all)		`querySelectorAll()`

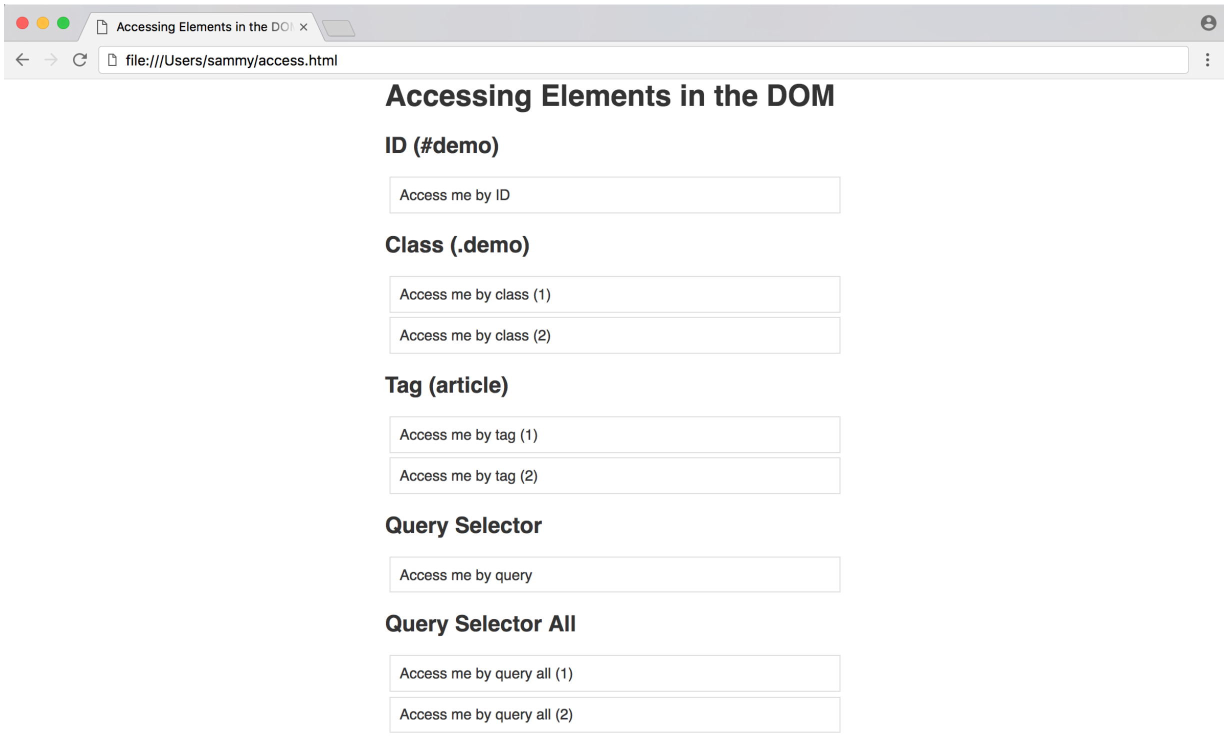
It is important when studying the DOM to type the examples on your own computer to ensure that you are understanding and retaining the information you learn.

You can save this HTML file, `access.html`, to your own project to work through the examples along with this article. If you are unsure how to work with JavaScript and HTML locally, review our How To Add JavaScript to HTML tutorial.

access.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">

  <title>Accessing Elements in the DOM</title>

  <style>
    html { font-family: sans-serif; color: #333; }
    body { max-width: 500px; margin: 0 auto;
padding: 0 15px; }
    div, article { padding: 10px; margin: 5px;
border: 1px solid #dedede; }
  </style>

</head>

<body>

  <h1>Accessing Elements in the DOM</h1>

  <h2>ID (#demo)</h2>
  <div id="demo">Access me by ID</div>
  <h2>Class (.demo)</h2>
  <div class="demo">Access me by class (1)</div>
  <div class="demo">Access me by class (2)</div>

  <h2>Tag (article)</h2>
  <article>Access me by tag (1)</article>
  <article>Access me by tag (2)</article>

  <h2>Query Selector</h2>
  <div id="demo-query">Access me by query</div>

  <h2>Query Selector All</h2>
  <div class="demo-query-all">Access me by query
all (1)</div>
  <div class="demo-query-all">Access me by query
all (2)</div>

</body>

</html>
```

In this HTML file, we have many elements that we will access with different `document` methods. When we render the file in a browser, it will look similar to this:


Accessing Elements in the DOM

file:///Users/sammy/access.html

Accessing Elements in the DOM

ID (#demo)

Access me by ID

Class (.demo)

Access me by class (1)

Access me by class (2)

Tag (article)

Access me by tag (1)

Access me by tag (2)

Query Selector

Access me by query

Query Selector All

Access me by query all (1)

Access me by query all (2)

Browser rendering of access.html page

We'll be using the different methods that we outlined in the Overview above to access the available elements in the file.

Accessing Elements by ID

The easiest way to access a single element in the DOM is by its unique ID. We can grab an element by ID with the `getElementById()` method of the `document` object.

```
document.getElementById();
```

In order to be accessed by ID, the HTML element must have an `id` attribute. We have a `div` element with an ID of `demo`.

```html
<div id="demo">Access me by ID</div>
```

In the Console, let’s get the element and assign it to the `demoId` variable.

```js
const demoId = document.getElementById('demo');
```

Logging `demoId` to the console will return our entire HTML element.

```js
console.log(demoId);
```

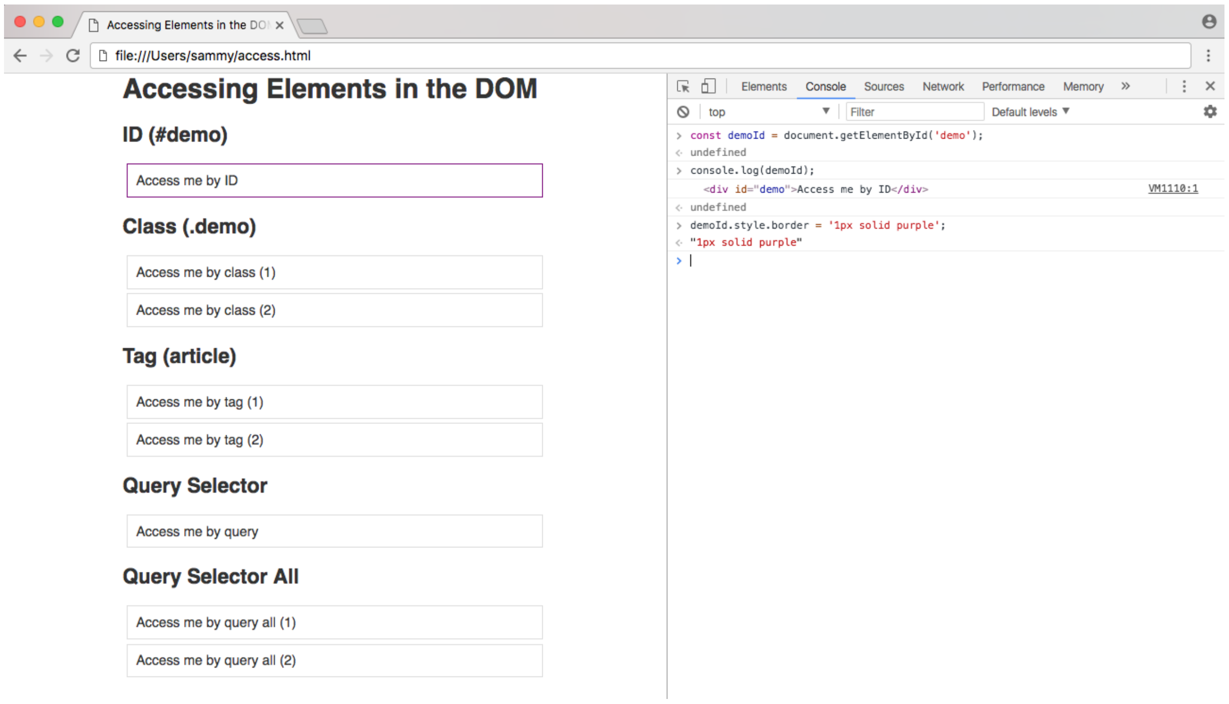
Output

```
<div id="demo">Access me by ID</div>
```

We can be sure we’re accessing the correct element by changing the `border` property to `purple`.

```js
demoId.style.border = '1px solid purple';
```

Once we do so, our live page will look like this:



Accessing Elements in the DOM

ID (#demo)

Access me by ID

Class (.demo)

Access me by class (1)

Access me by class (2)

Tag (article)

Access me by tag (1)

Access me by tag (2)

Query Selector

Access me by query

Query Selector All

Access me by query all (1)

Access me by query all (2)

```
> const demoId = document.getElementById('demo');
<· undefined
> console.log(demoId);
    <div id="demo">Access me by ID</div>                    VM1110:1
<· undefined
> demoId.style.border = '1px solid purple';
<· "1px solid purple"
> |
```

Browser rendering of ID element styling

Accessing an element by ID is an effective way to get an element quickly in the DOM. However, it has drawbacks; an ID must always be unique to the page, and therefore you will only ever be able to access a single element at a time with the `getElementById()` method. If you wanted to add a function to many elements throughout the page, your code would quickly become repititious.

Accessing Elements by Class

The [class](#) attribute is used to access one or more specific elements in the DOM. We can get all the elements with a given class name with the [`getElementsByClassName()`](#) method.

```
document.getElementsByClassName();
```

Now we want to access more than one element, and in our example we have two elements with a `demo` class.

```
<div class="demo">Access me by class (1)</div>
<div class="demo">Access me by class (2)</div>
```

Let’s access our elements in the Console and put them in a variable called `demoClass`.

```
const demoClass =
document.getElementsByClassName('demo');
```

At this point, you might think you can modify the elements the same way you did with the ID example. However, if we try to run the following code and change the `border` property of the class demo elements to orange, we will get an error.

```
demoClass.style.border = '1px solid orange';
```

Output

```
Uncaught TypeError: Cannot set property 'border'
of undefined
```

The reason this doesn’t work is because instead of just getting one element, we have an array-like object of elements.

```
console.log(demoClass);
```

Output

```
(2) [div.demo, div.demo]
```

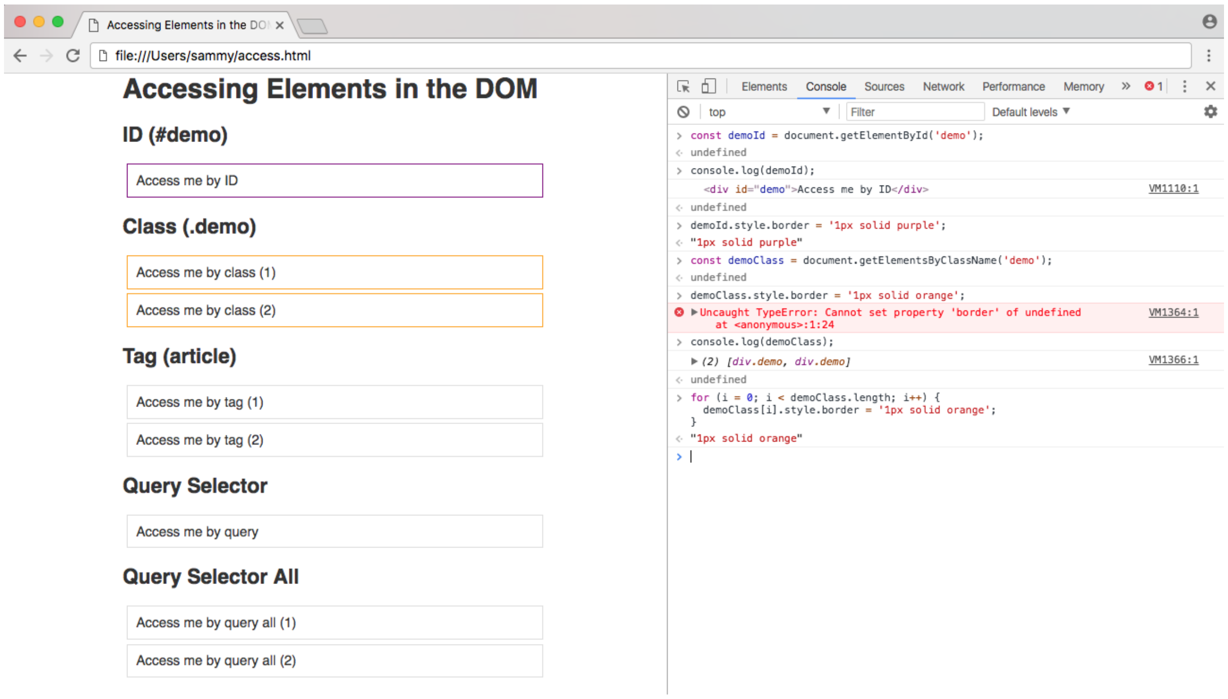
JavaScript arrays must be accessed with an index number. We can therefore change the first element of this array by using an index of `0`.

```
demoClass[0].style.border = '1px solid orange';
```

Generally when accessing elements by class, we want to apply a change to all the elements in the document with that particular class, not just one. We can do this by creating a `for` loop, and looping through every item in the array.

```
for (i = 0; i < demoClass.length; i++) {
  demoClass[i].style.border = '1px solid orange';
}
```

When we run this code, our live page will be rendered like this:



Accessing Elements in the DOM

ID (#demo)

Access me by ID

Class (.demo)

Access me by class (1)

Access me by class (2)

Tag (article)

Access me by tag (1)

Access me by tag (2)

Query Selector

Access me by query

Query Selector All

Access me by query all (1)

Access me by query all (2)

```
> const demoId = document.getElementById('demo');
<· undefined
> console.log(demoId);
    <div id="demo">Access me by ID</div>                              VM1110:1
<· undefined
> demoId.style.border = '1px solid purple';
<· "1px solid purple"
> const demoClass = document.getElementsByClassName('demo');
<· undefined
> demoClass.style.border = '1px solid orange';
⊗ ▶Uncaught TypeError: Cannot set property 'border' of undefined      VM1364:1
      at <anonymous>:1:24
> console.log(demoClass);
  ▶(2) [div.demo, div.demo]                                           VM1366:1
<· undefined
> for (i = 0; i < demoClass.length; i++) {
    demoClass[i].style.border = '1px solid orange';
  }
<· "1px solid orange"
> |
```

Browser rendering of class element styling

We have now selected every element on the page that has a `demo` class, and changed the `border` property to `orange`.

Accessing Elements by Tag

A less specific way to access multiple elements on the page would be by its HTML tag name. We access an element by tag with the `getElementsByTagName()` method.

```
document.getElementsByTagName();
```

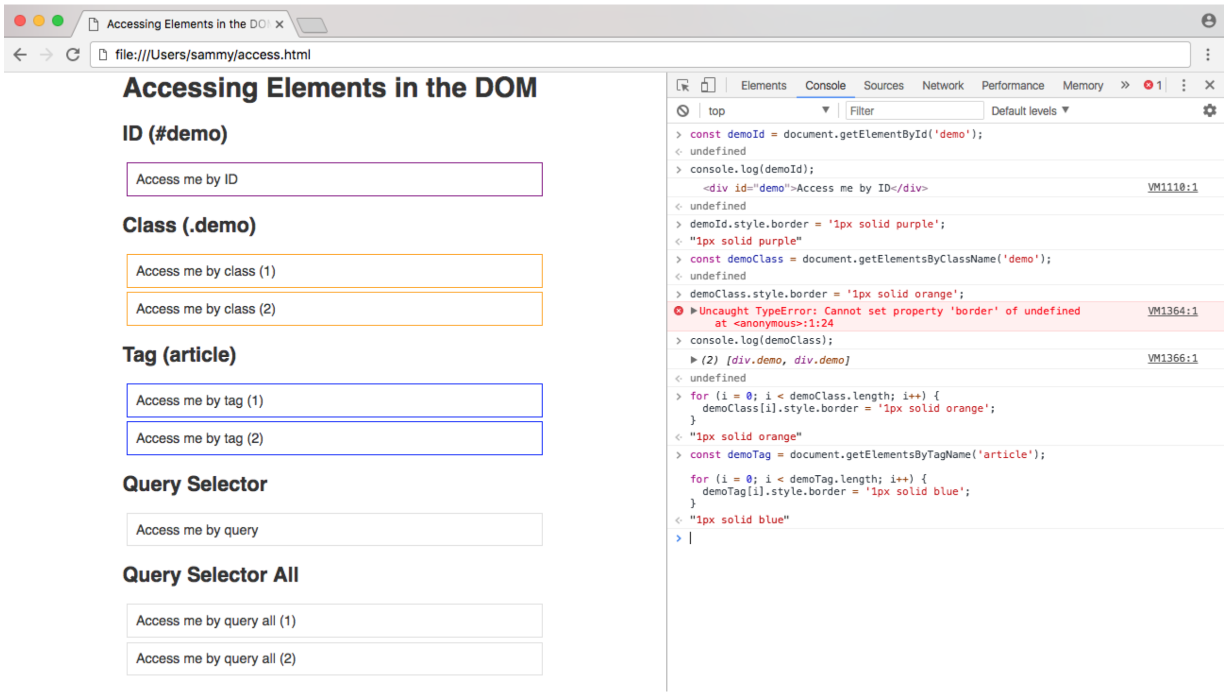
For our tag example, we're using `article` elements.

```
<article>Access me by tag (1)</article>
<article>Access me by tag (2)</article>
```

Just like accessing an element by its class, `getElementsByTagName()` will return an array-like object of elements, and we can modify every tag in the document with a `for` loop.

```
const demoTag =
document.getElementsByTagName('article');

for (i = 0; i < demoTag.length; i++) {
  demoTag[i].style.border = '1px solid blue';
}
```

Upon running the code, the live page will be modified like so:



```
> const demoId = document.getElementById('demo');
<· undefined
> console.log(demoId);
    <div id="demo">Access me by ID</div>                          VM1110:1
<· undefined
> demoId.style.border = '1px solid purple';
<· "1px solid purple"
> const demoClass = document.getElementsByClassName('demo');
<· undefined
> demoClass.style.border = '1px solid orange';
✖ ▶Uncaught TypeError: Cannot set property 'border' of undefined  VM1364:1
      at <anonymous>:1:24
> console.log(demoClass);
  ▶(2) [div.demo, div.demo]                                       VM1366:1
<· undefined
> for (i = 0; i < demoClass.length; i++) {
    demoClass[i].style.border = '1px solid orange';
  }
<· "1px solid orange"
> const demoTag = document.getElementsByTagName('article');

  for (i = 0; i < demoTag.length; i++) {
    demoTag[i].style.border = '1px solid blue';
  }
<· "1px solid blue"
>
```

Browser rendering of tag element styling

The loop changed the `border` property of all `article` elements to `blue`.

Query Selectors

If you have any experience with the jQuery API, you may be familiar with jQuery’s method of accessing the DOM with CSS selectors.

```
$('#demo'); // returns the demo ID element in jQuery
```

We can do the same in plain JavaScript with the `querySelector()` and `querySelectorAll()` methods.

```
document.querySelector();
document.querySelectorAll();
```

To access a single element, we will use the `querySelector()` method. In our HTML file, we have a `demo-query` element

```
<div id="demo-query">Access me by query</div>
```

The selector for an `id` attribute is the hash symbol (#). We can assign the element with the `demo-query` id to the `demoQuery` variable.

```
const demoQuery = document.querySelector('#demo-
query');
```

In the case of a selector with multiple elements, such as a class or a tag, `querySelector()` will return the first element that matches the query. We can use the `querySelectorAll()` method to collect all the elements that match a specific query.

In our example file, we have two elements with the `demo-query-all` class applied to them.

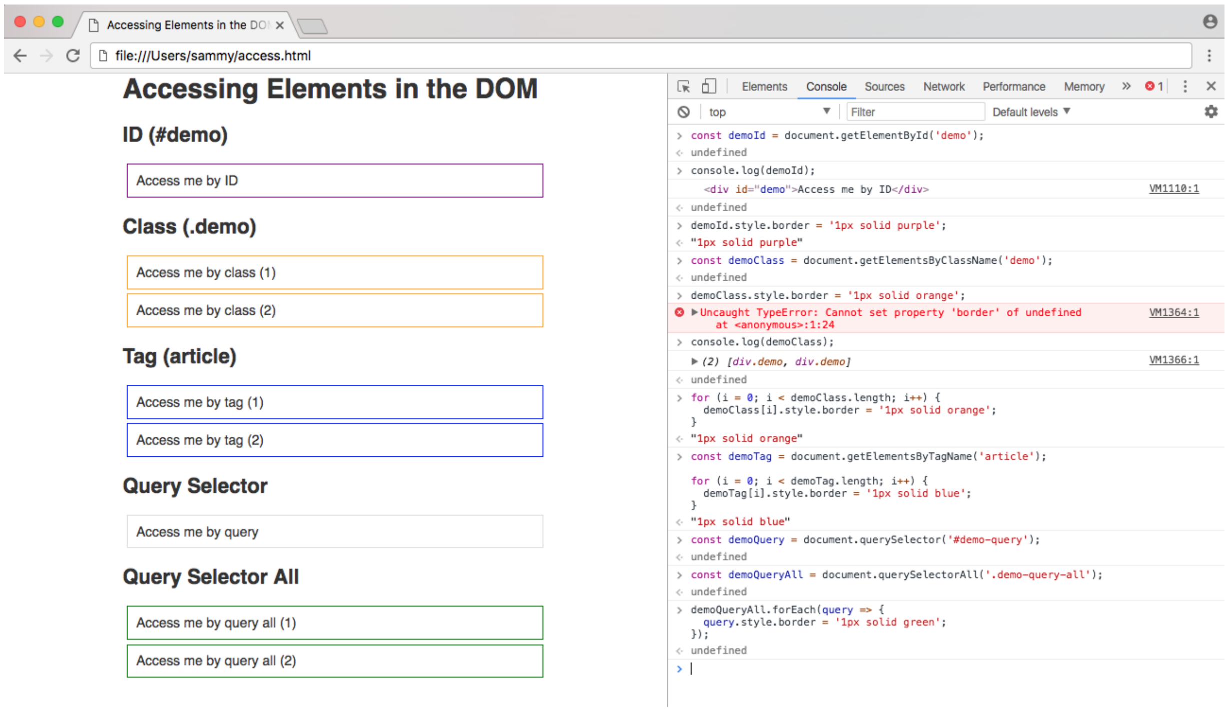
```
<div class="demo-query-all">Access me by query all
(1)</div>
<div class="demo-query-all">Access me by query all
(2)</div>
```

The selector for a `class` attribute is a period or full stop (`.`), so we can access the class with `.demo-query-all`.

```
const demoQueryAll =
document.querySelectorAll('.demo-query-all');
```

Using the `forEach()` method, we can apply the color `green` to the `border` property of all matching elements.

```
demoQueryAll.forEach(query => {
  query.style.border = '1px solid green';
});
```



Accessing Elements in the DOM

ID (#demo)

Access me by ID

Class (.demo)

Access me by class (1)

Access me by class (2)

Tag (article)

Access me by tag (1)

Access me by tag (2)

Query Selector

Access me by query

Query Selector All

Access me by query all (1)

Access me by query all (2)

```
> const demoId = document.getElementById('demo');
< undefined
> console.log(demoId);
    <div id="demo">Access me by ID</div>                          VM1110:1
< undefined
> demoId.style.border = '1px solid purple';
< "1px solid purple"
> const demoClass = document.getElementsByClassName('demo');
< undefined
> demoClass.style.border = '1px solid orange';
✖ ▶Uncaught TypeError: Cannot set property 'border' of undefined   VM1364:1
      at <anonymous>:1:24
> console.log(demoClass);
  ▶ (2) [div.demo, div.demo]                                        VM1366:1
< undefined
> for (i = 0; i < demoClass.length; i++) {
    demoClass[i].style.border = '1px solid orange';
  }
< "1px solid orange"
> const demoTag = document.getElementsByTagName('article');

  for (i = 0; i < demoTag.length; i++) {
    demoTag[i].style.border = '1px solid blue';
  }
< "1px solid blue"
> const demoQuery = document.querySelector('#demo-query');
< undefined
> const demoQueryAll = document.querySelectorAll('.demo-query-all');
< undefined
> demoQueryAll.forEach(query => {
    query.style.border = '1px solid green';
  });
< undefined
> |
```

Browser rendering of querySelector() styling

With `querySelector()`, comma-separated values function as an OR operator. For example, `querySelector('div, article')` will match `div` or `article`, whichever appears first in the document. With `querySelectorAll()`, comma-separated values function as an AND operator, and `querySelectorAll('div, article')` will match all `div` and `article` values in the document.

Using the query selector methods is extremely powerful, as you can access any element or group of elements in the DOM the same way you would in a CSS file. For a complete list of selectors, review CSS Selectors on the Mozilla Developer Network.

Complete JavaScript Code

Below is the complete script of the work we did above. You can use it to access all the elements on our example page. Save the file as `access.js` and load it in to the HTML file right before the closing `body` tag.

access.js

```
// Assign all elements
const demoId = document.getElementById('demo');
const demoClass =
document.getElementsByClassName('demo');
const demoTag =
document.getElementsByTagName('article');
const demoQuery = document.querySelector('#demo-
query');
const demoQueryAll =
document.querySelectorAll('.demo-query-all');

// Change border of ID demo to purple
demoId.style.border = '1px solid purple';

// Change border of class demo to orange
for (i = 0; i < demoClass.length; i++) {
  demoClass[i].style.border = '1px solid orange';
}

// Change border of tag demo to blue
for (i = 0; i < demoTag.length; i++) {
  demoTag[i].style.border = '1px solid blue';
}

// Change border of ID demo-query to red
demoQuery.style.border = '1px solid red';
```

```
// Change border of class query-all to green
demoQueryAll.forEach(query => {
  query.style.border = '1px solid green';
});
```

Your final HTML file will look like this:

access.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">

  <title>Accessing Elements in the DOM</title>

  <style>
    html { font-family: sans-serif; color: #333; }
    body { max-width: 500px; margin: 0 auto;
padding: 0 15px; }
    div, article { padding: 10px; margin: 5px;
border: 1px solid #dedede; }
  </style>

</head>

<body>

  <h1>Accessing Elements in the DOM</h1>

  <h2>ID (#demo)</h2>
  <div id="demo">Access me by ID</div>
  <h2>Class (.demo)</h2>
  <div class="demo">Access me by class (1)</div>
  <div class="demo">Access me by class (2)</div>

  <h2>Tag (article)</h2>
  <article>Access me by tag (1)</article>
  <article>Access me by tag (2)</article>

  <h2>Query Selector</h2>
  <div id="demo-query">Access me by query</div>

  <h2>Query Selector All</h2>
  <div class="demo-query-all">Access me by query
all (1)</div>
  <div class="demo-query-all">Access me by query
all (2)</div>

  <script src="access.js"></script>

</body>

</html>
```

You can continue to work on these template files to make additional changes by accessing HTML elements.

Conclusion

In this tutorial, we went over 5 ways to access HTML elements in the DOM — by ID, by class, by HTML tag name, and by selector. The method you will use to get an element or group of elements will depend on browser support and how many elements you will be manipulating. You should now feel confident to access any HTML element in a document with JavaScript through the DOM.

How To Traverse the DOM

Written by Tania Rascia

The previous tutorial in this series, How to Access Elements in the DOM, covers how to use the built-in methods of the `document` object to access HTML elements by ID, class, tag name, and query selectors. We know that the DOM is structured as a tree of nodes with the `document` node at the root and every other node (including elements, comments, and text nodes) as the various branches.

Often, you will want to move through the DOM without specifying each and every element beforehand. Learning how to navigate up and down the DOM tree and move from branch to branch is essential to understanding how to work with JavaScript and HTML.

In this tutorial, we will go over how to traverse the DOM (also known as walking or navigating the DOM) with parent, child, and sibling properties.

Setup

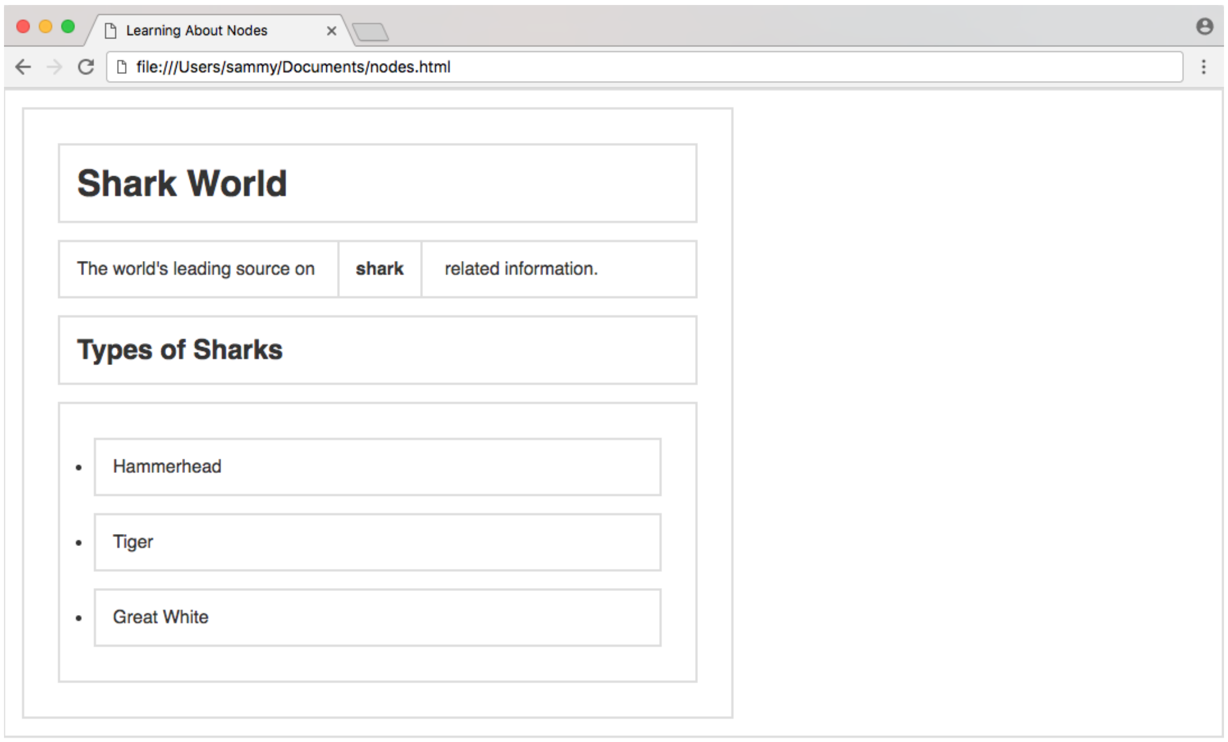
To begin, we will create a new file called `nodes.html` comprised of the following code.

nodes.html

```html
<!DOCTYPE html>
<html>

<head>
  <title>Learning About Nodes</title>

  <style>
    * { border: 2px solid #dedede; padding: 15px;
margin: 15px; }
    html { margin: 0; padding: 0; }
    body { max-width: 600px; font-family: sans-
serif; color: #333; }
  </style>
</head>

<body>
  <h1>Shark World</h1>
  <p>The world's leading source on
<strong>shark</strong> related information.</p>
  <h2>Types of Sharks</h2>
  <ul>
    <li>Hammerhead</li>
    <li>Tiger</li>
    <li>Great White</li>
  </ul>
</body>
```

```
<script>
  const h1 = document.getElementsByTagName('h1')
[0];
  const p = document.getElementsByTagName('p')[0];
  const ul = document.getElementsByTagName('ul')
[0];
</script>

</html>
```

When we load the file in a web browser, we'll see rendering that looks like the following screenshot.



Shark World

The world's leading source on **shark** related information.

Types of Sharks

- Hammerhead
- Tiger
- Great White

nodes.html page

In this example website, we have an HTML document with a few elements. Some basic CSS has been added in a `style` tag to make each element obviously visible, and a few variables have been created in the `script` for ease of access of a few elements. Since there is only one of each `h1`, `p`, and `ul`, we can access the first index on each respective `getElementsByTagName` property.

Root Nodes

The `document` object is the root of every node in the DOM. This object is actually a property of the `window` object, which is the global, top-level object representing a tab in the browser. The `window` object has access to such information as the toolbar, height and width of the window, prompts, and alerts. The `document` consists of what is inside of the inner `window`.

Below is a chart consisting of the root elements that every document will contain. Even if a blank HTML file is loaded into a browser, these three nodes will be added and parsed into the DOM.

Property	Node	Node Type
`document`	`#document`	`DOCUMENT_NODE`
`document.documentElement`	`html`	`ELEMENT_NODE`
`document.head`	`head`	`ELEMENT_NODE`
`document.body`	`body`	`ELEMENT_NODE`

Since the `html`, `head`, and `body` elements are so common, they have their own properties on the `document`.

Open the Console in DevTools and test each of these four properties by submitting them and viewing the output. You can also test `h1`, `p`, and `ul` which will return the elements due to the variables we added in the `script` tag.

Parent Nodes

The nodes in the DOM are referred to as parents, children, and siblings, depending on their relation to other nodes. The parent of any node is the node that is one level above it, or closer to the `document` in the DOM hierarchy. There are two properties to get the parent — `parentNode` and `parentElement`.

Property	Gets
`parentNode`	Parent Node
`parentElement`	Parent Element Node

In our `nodes.html` example:

- `html` is the parent of `head`, `body`, and `script`.
- `body` is the parent of `h1`, `h2`, `p` and `ul`, but not `li`, since `li` is two levels down from `body`.

We can test what the parent of our `p` element is with the `parentNode` property. This `p` variable comes from our custom `document.getElementsByTagName('p')[0]` declaration.

```
p.parentNode;
```

Output

```
▶ <body>...</body>
```

The parent of `p` is `body`, but how can we get the grandparent, which is two levels above? We can do so by chaining properties together.

```
p.parentNode.parentNode;
```

Output

```
▶ <html>...</html>
```

Using `parentNode` twice, we retrieved the grandparent of `p`.

There are properties to retrieve the parent of a node, but only one small difference between them, as demonstrated in this snippet below.

```
// Assign html object to html variable
const html = document.documentElement;

console.log(html.parentNode); // > #document
console.log(html.parentElement); // > null
```

The parent of almost any node is an element node, as text and comments cannot be parents to other nodes. However, the parent of `html` is a document node, so `parentElement` returns `null`. Generally, `parentNode` is more commonly used when traversing the DOM.

Children Nodes

The children of a node are the nodes that are one level below it. Any nodes beyond one level of nesting are usually referred to as descendants.

PROPERTY	GETS
`childNodes`	Child Nodes
`firstChild`	First Child Node
`lastChild`	Last Child Node
`children`	Element Child Nodes
`firstElementChild`	First Child Element Node
`lastElementChild`	Last Child Element Node

The `childNodes` property will return a live list of every child of a node. You might expect the `ul` element to get three `li` elements. Let's test what it retrieves.

```
ul.childNodes;
```

Output

```
▶ (7) [text, li, text, li, text, li, text]
```

In addition to the three `li` elements, it also gets four text nodes. This is because we wrote our own HTML (it was not generated by JavaScript) and the indentation between elements is counted in the DOM as text nodes. This is not intuitive, as the Elements tab of DevTools strips out white space nodes.

If we attempted to change the background color of the first child node using the `firstChild` property, it would fail because the first node is text.

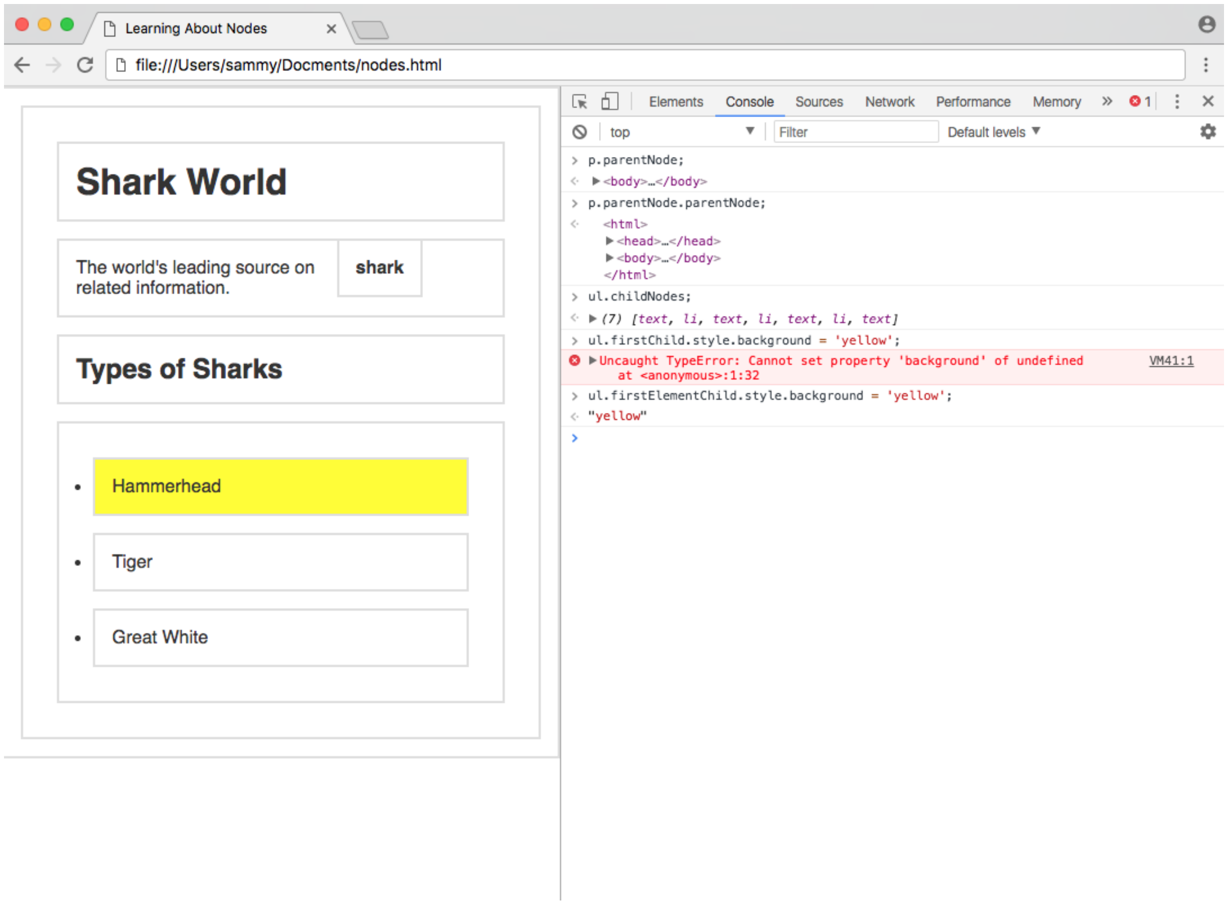
```
ul.firstChild.style.background = 'yellow';
[secondary_label Output]
Uncaught TypeError: Cannot set property
'background' of undefined
```

The `children`, `firstElementChild` and `lastElementChild` properties exist in these types of situations to retrieve only the element nodes. `ul.children` will only return the three `li` elements.

Using `firstElementChild`, we can change the background color of the first `li` in the `ul`.

```
ul.firstElementChild.style.background = 'yellow';
```

When you run the code above, your webpage will be updated to modify the background color.



```
> p.parentNode;
< ▶<body>…</body>
> p.parentNode.parentNode;
<   <html>
    ▶<head>…</head>
    ▶<body>…</body>
    </html>
> ul.childNodes;
< ▶(7) [text, li, text, li, text, li, text]
> ul.firstChild.style.background = 'yellow';
✖ ▶Uncaught TypeError: Cannot set property 'background' of undefined      VM41:1
     at <anonymous>:1:32
> ul.firstElementChild.style.background = 'yellow';
< "yellow"
>
```

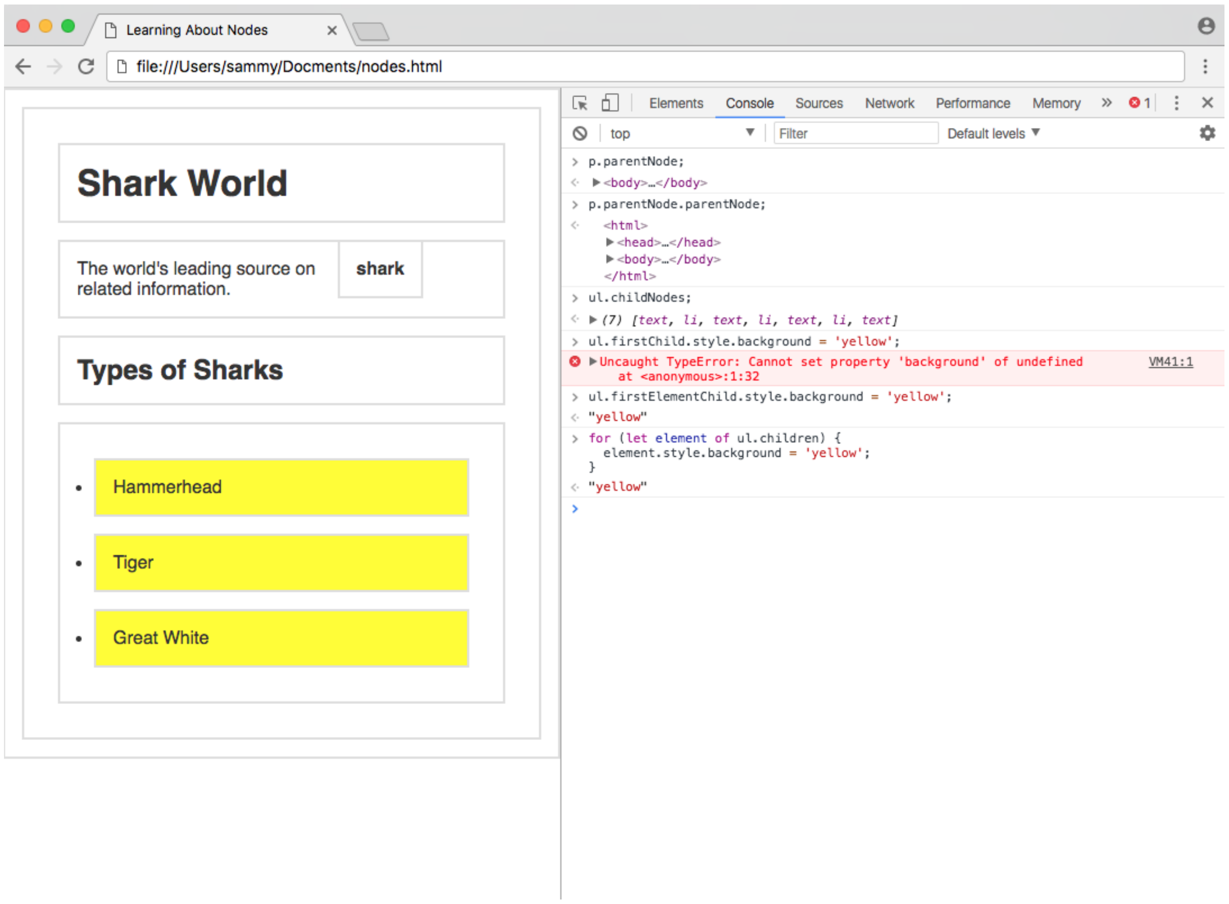
firstElementChild.style.background modification

When doing basic DOM manipulation such as in this example, the element-specific properties are extremely helpful. In JavaScript-generated web apps, the properties that select all nodes are more likely to be used, as white-space newlines and indentation will not exist in this case.

A `for...of` loop can be used to iterate through all `children` elements.

```
for (let element of ul.children) {
  element.style.background = 'yellow';
}
```

Now, each child element will have a yellow background.



```
> p.parentNode;
<· ▶<body>…</body>
> p.parentNode.parentNode;
<·  <html>
     ▶<head>…</head>
     ▶<body>…</body>
    </html>
> ul.childNodes;
<· ▶(7) [text, li, text, li, text, li, text]
> ul.firstChild.style.background = 'yellow';
✖ ▶Uncaught TypeError: Cannot set property 'background' of undefined    VM41:1
      at <anonymous>:1:32
> ul.firstElementChild.style.background = 'yellow';
<· "yellow"
> for (let element of ul.children) {
    element.style.background = 'yellow';
  }
<· "yellow"
>
```

children elements modification

Since our `p` element has both text and elements inside of it, the `childNodes` property is helpful for accessing that information.

```
for (let element of p.childNodes) {
  console.log(element);
}
```

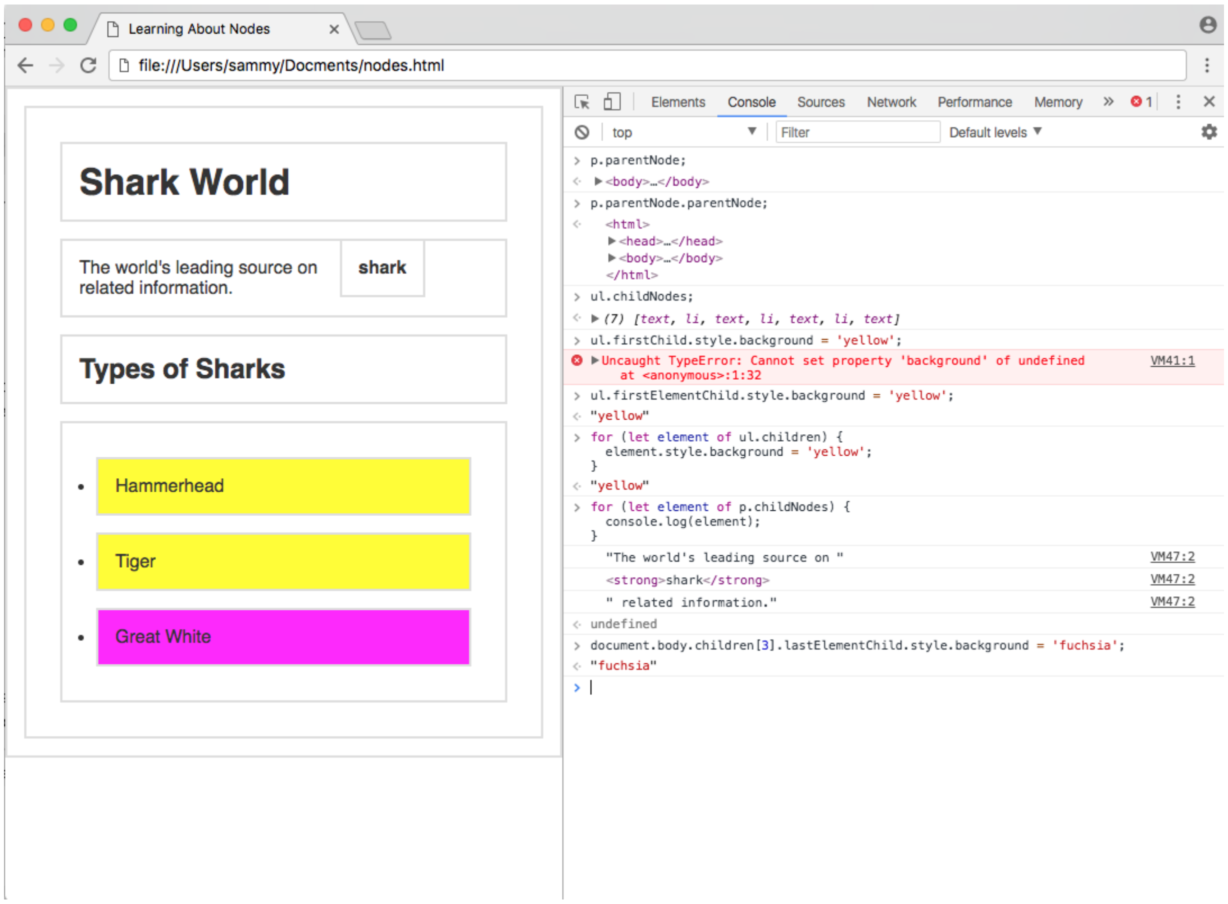
Output

```
"The world's leading source on "
<strong>shark</strong>
" related information."
```

childNodes and children do not return arrays with all the Array properties and methods, but they appear and behave similarly to JavaScript arrays. You can access nodes by index number, or find their length property.

```
document.body.children[3].lastElementChild.style.b
ackground = 'fuchsia';
```

The above code will find the last element child (li) of the fourth child element (ul) of body and apply a style.



```
> p.parentNode;
<· ▶<body>…</body>
> p.parentNode.parentNode;
<·   <html>
     ▶<head>…</head>
     ▶<body>…</body>
     </html>
> ul.childNodes;
<· ▶(7) [text, li, text, li, text, li, text]
> ul.firstChild.style.background = 'yellow';
✖ ▶Uncaught TypeError: Cannot set property 'background' of undefined      VM41:1
      at <anonymous>:1:32
> ul.firstElementChild.style.background = 'yellow';
<· "yellow"
> for (let element of ul.children) {
    element.style.background = 'yellow';
  }
<· "yellow"
> for (let element of p.childNodes) {
    console.log(element);
  }
    "The world's leading source on "                                      VM47:2
    <strong>shark</strong>                                                VM47:2
    " related information."                                               VM47:2
<· undefined
> document.body.children[3].lastElementChild.style.background = 'fuchsia';
<· "fuchsia"
> |
```

last child element modification

Using parent and child properties, you can retrieve any node in the DOM.

Sibling Nodes

The siblings of a node are any node on the same tree level in the DOM. Siblings do not have to be the same type of node - text, element, and comment nodes can all be siblings.

PROPERTY	GETS
`previousSibling`	Previous Sibling Node
`nextSibling`	Next Sibling Node
`previousElementSibling`	Previous Sibling Element Node
`nextElementSibling`	Next Sibling Element Node

Sibling properties work the same way as the children nodes, in that there is a set of properties to traverse all nodes, and a set of properties for only element nodes. `previousSibling` and `nextSibling` will get the next node that immediately precedes or follows the specified node, and `previousElementSibling` and `nextElementSibling` will only get element nodes.

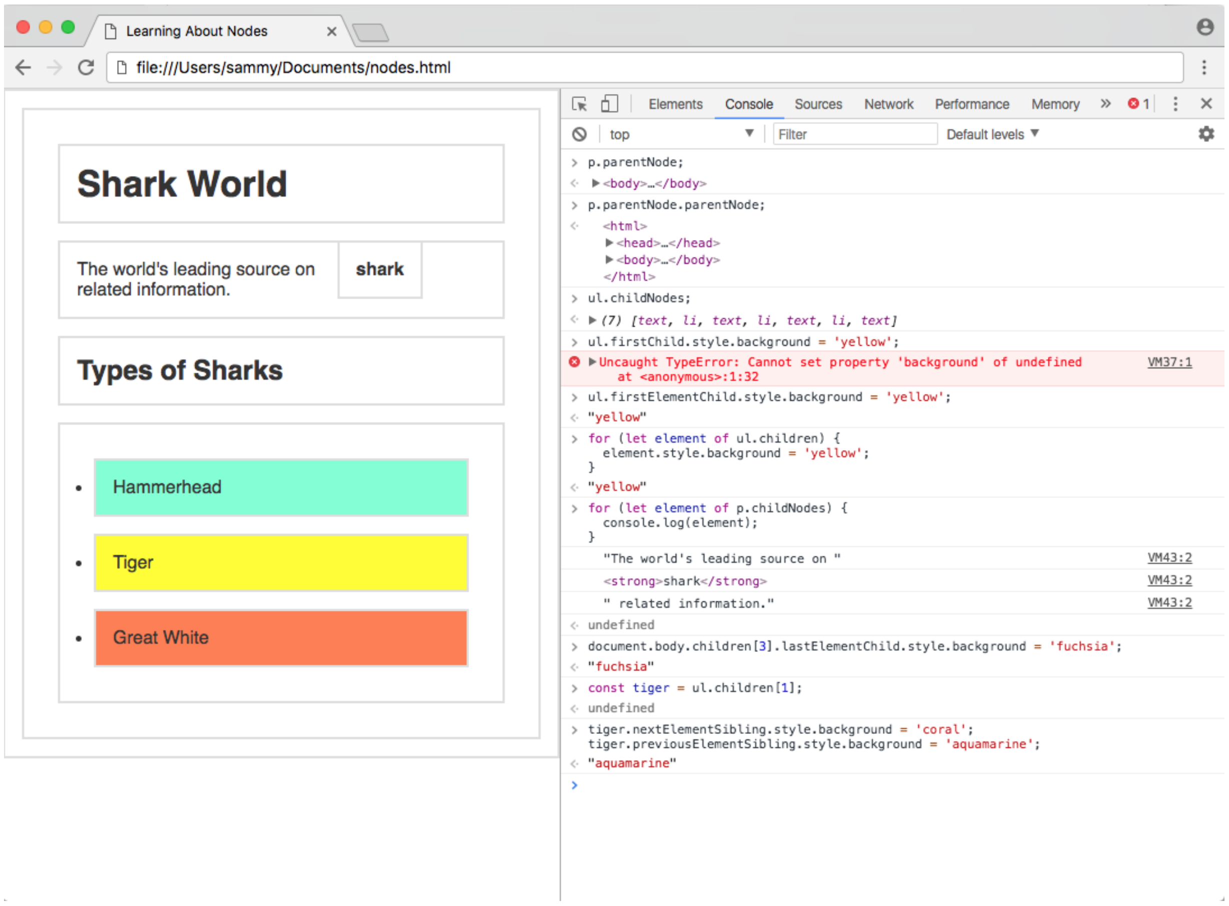
In our `nodes.html` example, let's select the middle element of `ul`.

```
const tiger = ul.children[1];
```

Since we created our DOM from scratch and not as a JavaScript web app, we will need to use the element sibling properties to access the previous and next element nodes, as there is white space in the DOM.

```
tiger.nextElementSibling.style.background =
'coral';
tiger.previousElementSibling.style.background =
'aquamarine';
```

Running this code should have applied `coral` to the background of `Hammerhead` and `aquamarine` to the background of `Great White`.



```
> p.parentNode;
<· ▶<body>…</body>
> p.parentNode.parentNode;
<·   <html>
    ▶<head>…</head>
    ▶<body>…</body>
    </html>
> ul.childNodes;
<· ▶(7) [text, li, text, li, text, li, text]
> ul.firstChild.style.background = 'yellow';
⊗ ▶Uncaught TypeError: Cannot set property 'background' of undefined      VM37:1
      at <anonymous>:1:32
> ul.firstElementChild.style.background = 'yellow';
<· "yellow"
> for (let element of ul.children) {
    element.style.background = 'yellow';
  }
<· "yellow"
> for (let element of p.childNodes) {
    console.log(element);
  }
    "The world's leading source on "                                  VM43:2
    <strong>shark</strong>                                            VM43:2
    " related information."                                           VM43:2
<· undefined
> document.body.children[3].lastElementChild.style.background = 'fuchsia';
<· "fuchsia"
> const tiger = ul.children[1];
<· undefined
> tiger.nextElementSibling.style.background = 'coral';
  tiger.previousElementSibling.style.background = 'aquamarine';
<· "aquamarine"
>
```

sibling element modification

Sibling properties can be chained together, just like parent and node properties.

Conclusion

In this tutorial, we covered how to access the root nodes of every HTML document and how to walk the DOM tree through parent, child, and sibling properties.

With what you learned in How to Access Elements in the DOM and this tutorial, you should be able to confidently access any node in the DOM of any website.

How To Make Changes to the DOM

Written by Tania Rascia

In the previous two installments of the Understanding the DOM series, we learned How To Access Elements in the DOM and How To Traverse the DOM. Using this knowledge, a developer can use classes, tags, ids, and selectors to find any node in the DOM, and use parent, child, and sibling properties to find relative nodes.

The next step to becoming more fully proficient with the DOM is to learn how to add, change, replace, and remove nodes. A to-do list application is one practical example of a JavaScript program in which you would need to be able to create, modify, and remove elements in the DOM.

In this tutorial, we will go over how to create new nodes and insert them into the DOM, replace existing nodes, and remove nodes.

Creating New Nodes

In a static website, elements are added to the page by directly writing HTML in an `.html` file. In a dynamic web app, elements and text are often added with JavaScript. The `createElement()` and `createTextNode()` methods are used to create new nodes in the DOM.

Property/Method	Description
`createElement()`	Create a new element node
`createTextNode()`	Create a new text node
`node.textContent`	Get or set the text content of an element node
`node.innerHTML`	Get or set the HTML content of an element

To begin, let's create an `index.html` file and save it in a new project directory.

index.html

```
<!DOCTYPE html>
<html lang="en">

  <head>
    <title>Learning the DOM</title>
  </head>

  <body>
    <h1>Document Object Model</h1>
  </body>

</html>
```

Right click anywhere on the page and select "Inspect" to open up Developer Tools, then navigate to the Console.

We will use `createElement()` on the `document` object to create a new `p` element.

```
const paragraph = document.createElement('p');
```

We've created a new `p` element, which we can test out in the Console.

```
console.log(paragraph)
```

Output

```
<p></p>
```

The `paragraph` variable outputs an empty `p` element, which is not very useful without any text. In order to add text to the element, we'll set the `textContent` property.

```
paragraph.textContent = "I'm a brand new
paragraph.";
console.log(paragraph)
```

Output

```
<p>I'm a brand new paragraph.</p>
```

A combination of `createElement()` and `textContent` creates a complete element node.

An alternate method of setting the content of the element is with the `innerHTML` property, which allows you to add HTML as well as text to an element.

```
paragraph.innerHTML = "I'm a paragraph with
<strong>bold</strong> text.";
```

Note: While this will work and is a common method of adding content to an element, there is a possible cross-site scripting (XSS) risk associated with using the `innerHTML` method, as inline JavaScript can be added to an element. Therefore, it is recommended to use `textContent` instead, which will strip out HTML tags.

It is also possible to create a text node with the `createTextNode()` method.

```
const text = document.createTextNode("I'm a new
text node.");
console.log(text)
```

Output

```
"I'm a new text node."
```

With these methods, we've created new elements and text nodes, but they are not visible on the front end of a website until they've been inserted into the document.

Inserting Nodes into the DOM

In order to see the new text nodes and elements we create on the front end, we will need to insert them into the `document`. The methods `appendChild()` and `insertBefore()` are used to add items to the beginning, middle, or end of a parent element, and `replaceChild()` is used to replace an old node with a new node.

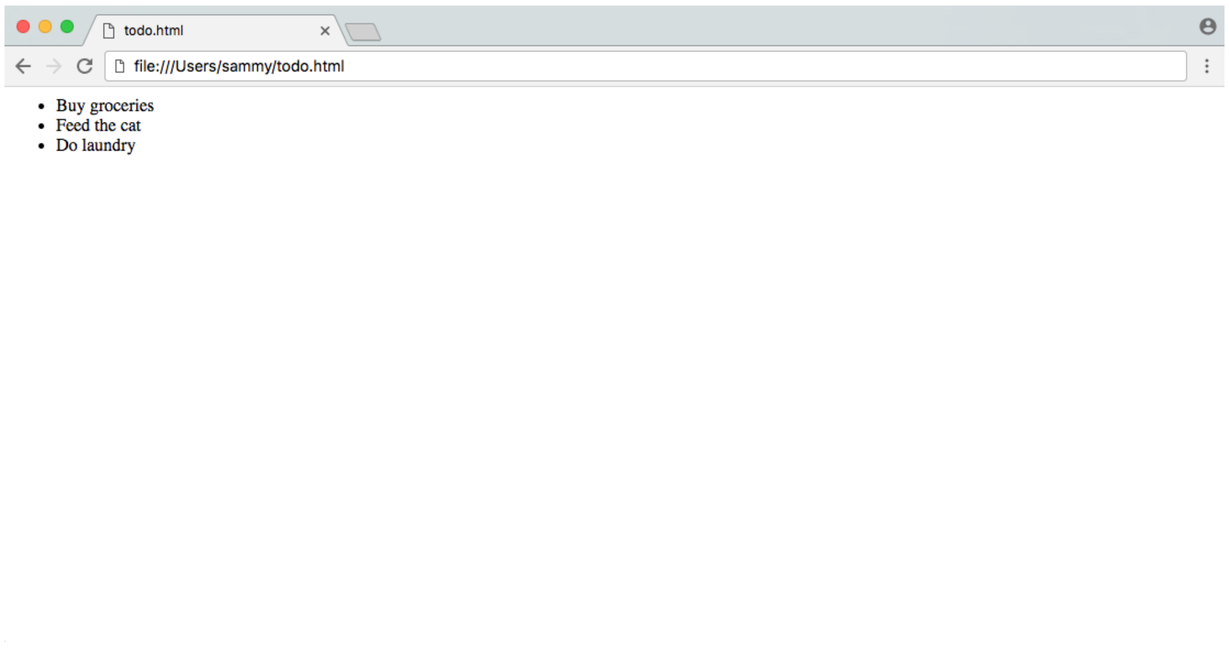
Property/Method	Description
node.appendChild()	Add a node as the last child of a parent element
node.insertBefore()	Insert a node into the parent element before a specified sibling node
node.replaceChild()	Replace an existing node with a new node

To practice these methods, let's create a to-do list in HTML:

todo.html

```
<ul>
  <li>Buy groceries</li>
  <li>Feed the cat</li>
  <li>Do laundry</li>
</ul>
```

When you load your page in the browser, it will look like this:



- Buy groceries
- Feed the cat
- Do laundry

DOM Screenshot 1

In order to add a new item to the end of the to-do list, we have to create the element and add text to it first, as we did in the "Creating New Nodes" section above.

```
// To-do list ul element
const todoList = document.querySelector('ul');

// Create new to-do
const newTodo = document.createElement('li');
newTodo.textContent = 'Do homework';
```

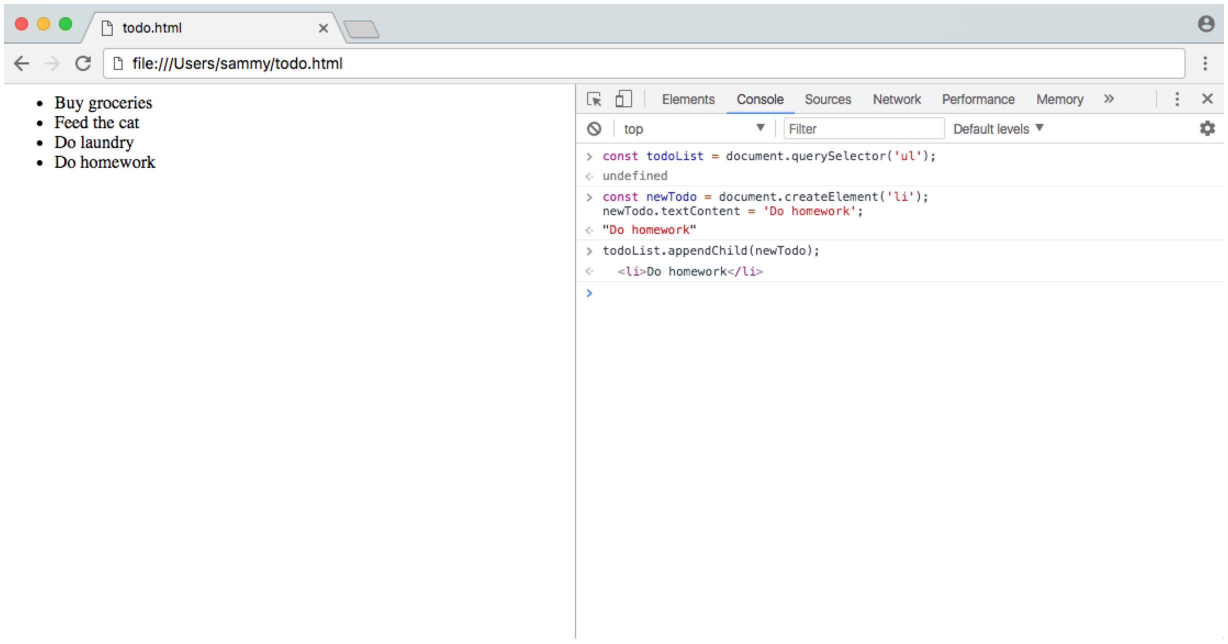
Now that we have a complete element for our new to-do, we can add it to the end of the list with `appendChild()`.

```
// Add new todo to the end of the list
todoList.appendChild(newTodo);
```

You can see the new `li` element has been appended to the end of the `ul`.

todo.html

```html
<ul>
  <li>Buy groceries</li>
  <li>Feed the cat</li>
  <li>Do laundry</li>
  <li>Do homework</li>
</ul>
```



- Buy groceries
- Feed the cat
- Do laundry
- Do homework

```
> const todoList = document.querySelector('ul');
< undefined
> const newTodo = document.createElement('li');
  newTodo.textContent = 'Do homework';
< "Do homework"
> todoList.appendChild(newTodo);
<   <li>Do homework</li>
>
```

DOM Screenshot 2

Maybe we have a higher priority task to do, and we want to add it to the beginning of the list. We'll have to create another element, as `createElement()` only creates one element and cannot be reused.

```
// Create new to-do
const anotherTodo = document.createElement('li');
anotherTodo.textContent = 'Pay bills';
```

We can add it to the beginning of the list using `insertBefore()`. This method takes two arguments — the first is the new child node to be added, and the second is the sibling node that will immediately follow the new node. In other words, you're inserting the new node before the next sibling node. This will look similar to the following pseudocode:

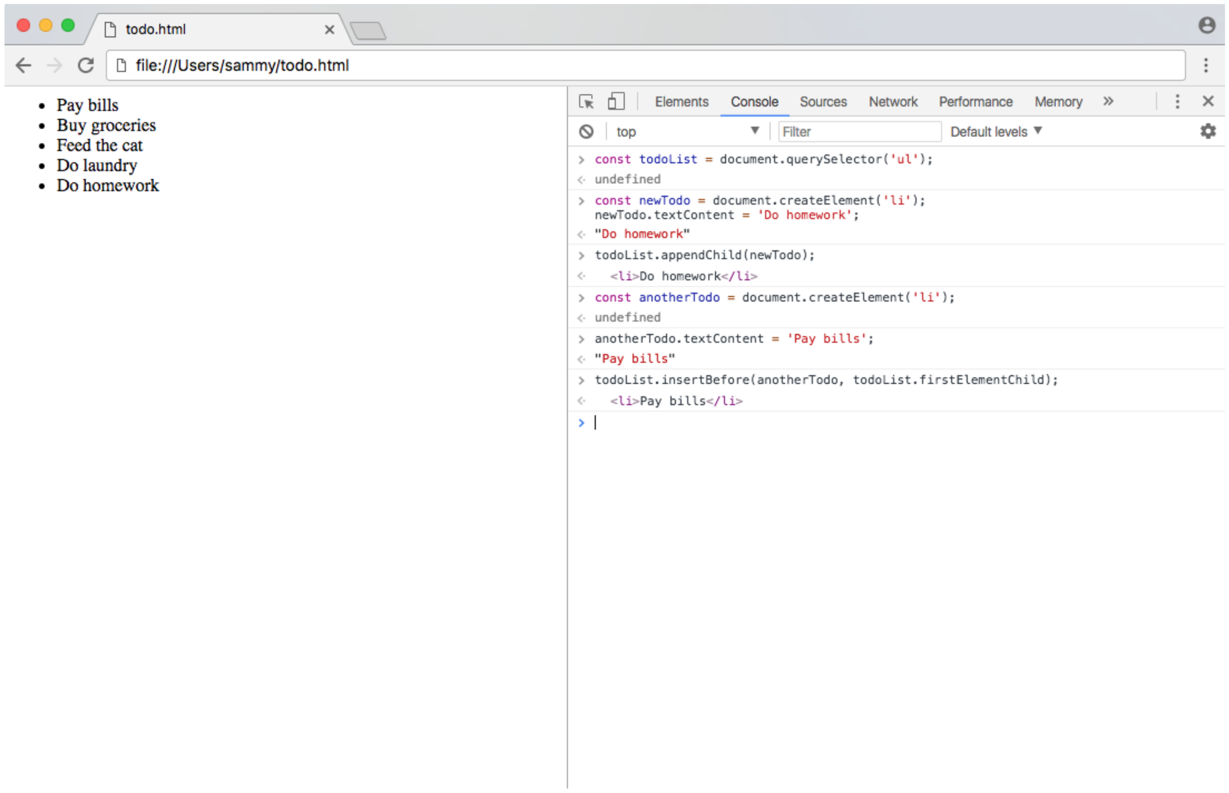
```
parentNode.insertBefore(newNode, nextSibling);
```

For our to-do list example, we'll add the new `anotherTodo` element before the first element child of the list, which is currently the `Buy groceries` list item.

```
// Add new to-do to the beginning of the list
todoList.insertBefore(anotherTodo,
todoList.firstElementChild);
```

todo.html

```html
<ul>
  <li>Pay bills</li>
  <li>Buy groceries</li>
  <li>Feed the cat</li>
  <li>Do laundry</li>
  <li>Do homework</li>
</ul>
```



- Pay bills
- Buy groceries
- Feed the cat
- Do laundry
- Do homework

```
> const todoList = document.querySelector('ul');
< undefined
> const newTodo = document.createElement('li');
  newTodo.textContent = 'Do homework';
< "Do homework"
> todoList.appendChild(newTodo);
<   <li>Do homework</li>
> const anotherTodo = document.createElement('li');
< undefined
> anotherTodo.textContent = 'Pay bills';
< "Pay bills"
> todoList.insertBefore(anotherTodo, todoList.firstElementChild);
<   <li>Pay bills</li>
> |
```

DOM Screenshot 3

The new node has successfully been added at the beginning of the list. Now we know how to add a node to a parent element. The next thing we may want to do is replace an existing node with a new node.

We'll modify an existing to-do to demonstrate how to replace a node. The first step of creating a new element remains the same.

```
const modifiedTodo = document.createElement('li');
modifiedTodo.textContent = 'Feed the dog';
```

Like `insertBefore()`, `replaceChild()` takes two arguments — the new node, and the node to be replaced, as shown in the pseudocode below.

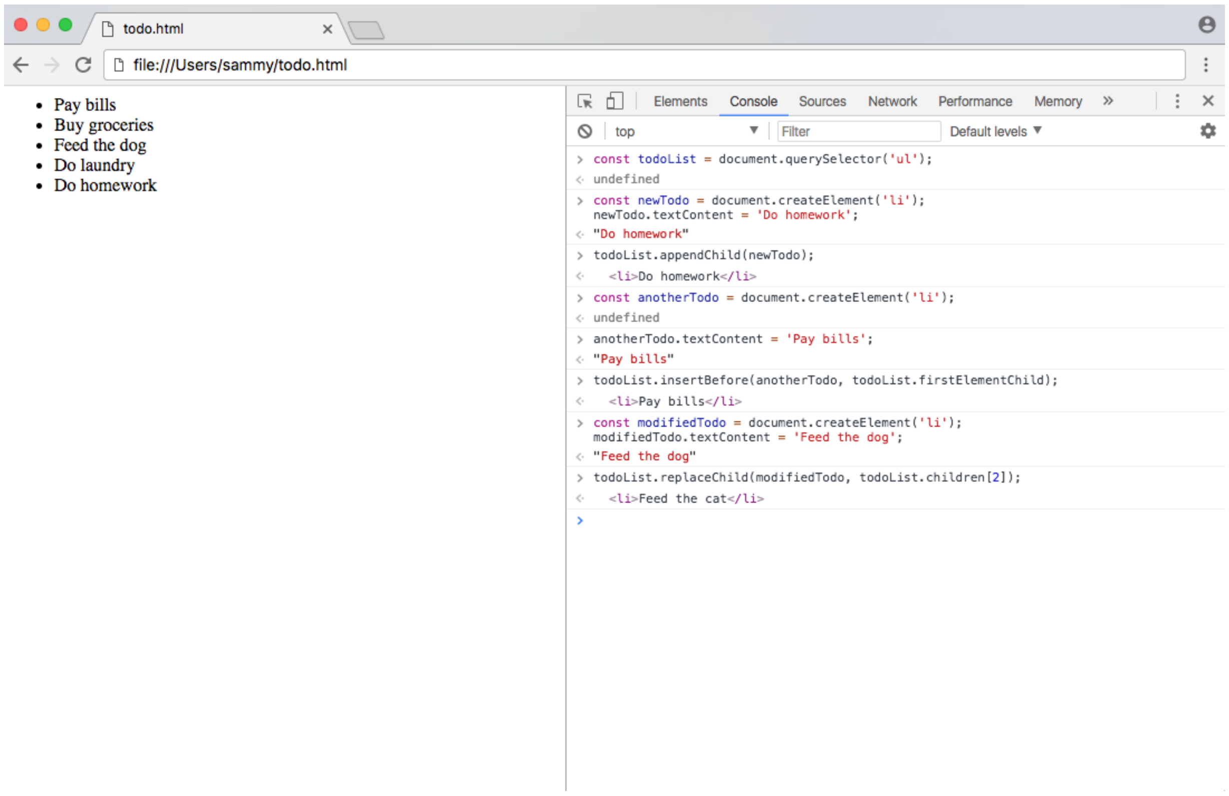
```
parentNode.replaceChild(newNode, oldNode);
```

We will replace the third element child of the list with the modified to-do.

```
// Replace existing to-do with modified to-do
todoList.replaceChild(modifiedTodo,
todoList.children[2]);
```

todo.html

```
<ul>
  <li>Pay bills</li>
  <li>Buy groceries</li>
  <li>Feed the dog</li>
  <li>Do laundry</li>
  <li>Do homework</li>
</ul>
```



- Pay bills
- Buy groceries
- Feed the dog
- Do laundry
- Do homework

```
> const todoList = document.querySelector('ul');
<· undefined
> const newTodo = document.createElement('li');
  newTodo.textContent = 'Do homework';
<· "Do homework"
> todoList.appendChild(newTodo);
<·   <li>Do homework</li>
> const anotherTodo = document.createElement('li');
<· undefined
> anotherTodo.textContent = 'Pay bills';
<· "Pay bills"
> todoList.insertBefore(anotherTodo, todoList.firstElementChild);
<·   <li>Pay bills</li>
> const modifiedTodo = document.createElement('li');
  modifiedTodo.textContent = 'Feed the dog';
<· "Feed the dog"
> todoList.replaceChild(modifiedTodo, todoList.children[2]);
<·   <li>Feed the cat</li>
>
```

DOM Screenshot 4

With a combination of `appendChild()`, `insertBefore()`, and `replaceChild()`, you can insert nodes and elements anywhere in the DOM.

Removing Nodes from the DOM

Now we know how to create elements, add them to the DOM, and modify existing elements. The final step is to learn to remove existing nodes from the DOM. Child nodes can be removed from a parent with `removeChild()`, and a node itself can be removed with `remove()`.

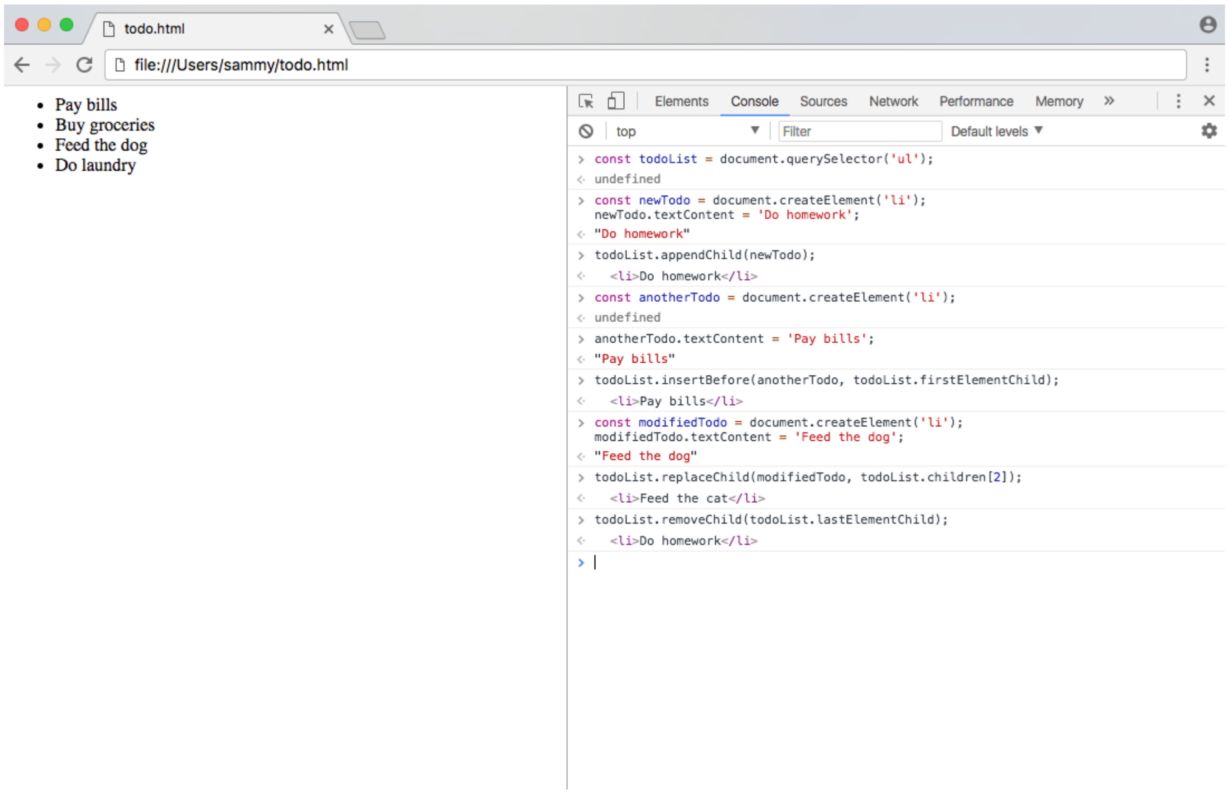
Method	Description
`node.removeChild()`	Remove child node
`node.remove()`	Remove node

Using the to-do example above, we'll want to delete items after they've been completed. If you completed your homework, you can remove the `Do homework` item, which happens to be the last child of the list, with `removeChild()`.

```
todoList.removeChild(todoList.lastElementChild);
```

todo.html

```html
<ul>
  <li>Pay bills</li>
  <li>Buy groceries</li>
  <li>Feed the dog</li>
  <li>Do laundry</li>
</ul>
```



- Pay bills
- Buy groceries
- Feed the dog
- Do laundry

```
> const todoList = document.querySelector('ul');
< undefined
> const newTodo = document.createElement('li');
  newTodo.textContent = 'Do homework';
< "Do homework"
> todoList.appendChild(newTodo);
<   <li>Do homework</li>
> const anotherTodo = document.createElement('li');
< undefined
> anotherTodo.textContent = 'Pay bills';
< "Pay bills"
> todoList.insertBefore(anotherTodo, todoList.firstElementChild);
<   <li>Pay bills</li>
> const modifiedTodo = document.createElement('li');
  modifiedTodo.textContent = 'Feed the dog';
< "Feed the dog"
> todoList.replaceChild(modifiedTodo, todoList.children[2]);
<   <li>Feed the cat</li>
> todoList.removeChild(todoList.lastElementChild);
<   <li>Do homework</li>
> |
```

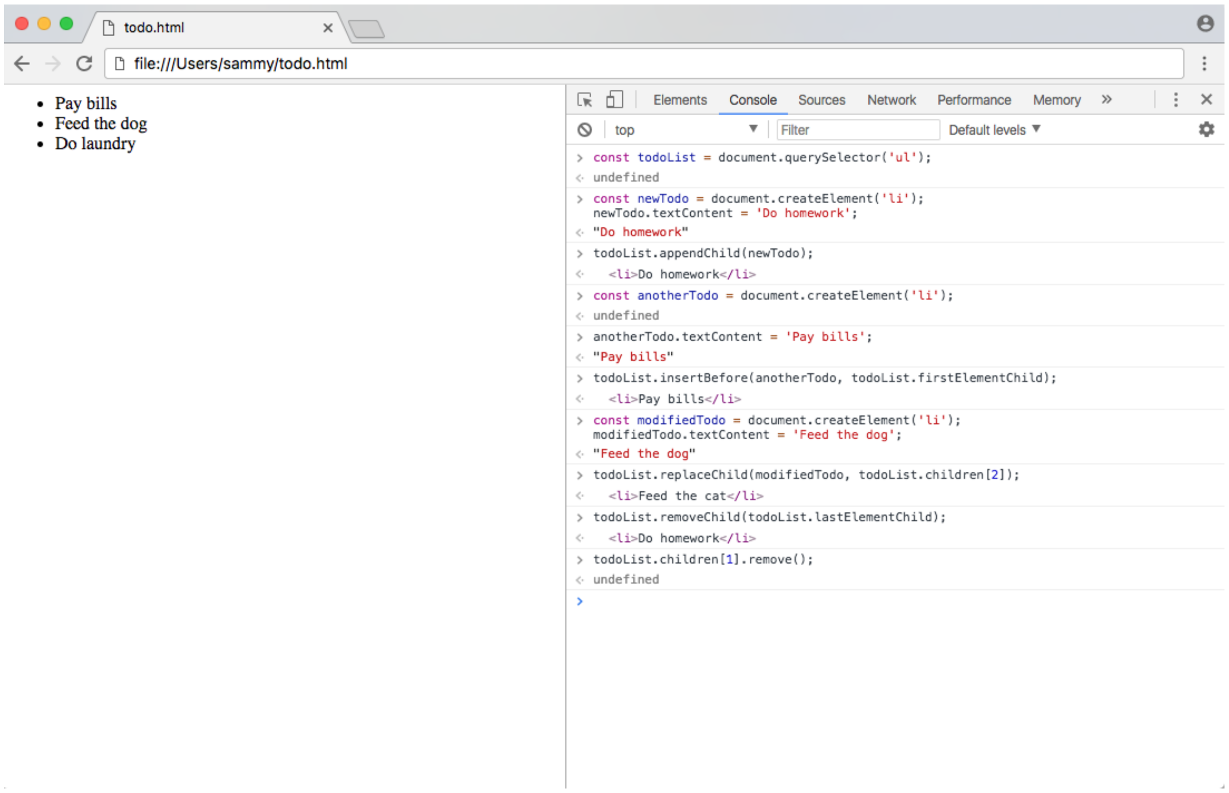
DOM Screenshot 5

Another method could be to remove the node itself, using the `remove()` method directly on the node.

```
// Remove second element child from todoList
todoList.children[1].remove();
```

todo.html

```
<ul>
  <li>Pay bills</li>
  <li>Feed the dog</li>
  <li>Do laundry</li>
</ul>
```



- Pay bills
- Feed the dog
- Do laundry

```
> const todoList = document.querySelector('ul');
< undefined
> const newTodo = document.createElement('li');
  newTodo.textContent = 'Do homework';
< "Do homework"
> todoList.appendChild(newTodo);
<   <li>Do homework</li>
> const anotherTodo = document.createElement('li');
< undefined
> anotherTodo.textContent = 'Pay bills';
< "Pay bills"
> todoList.insertBefore(anotherTodo, todoList.firstElementChild);
<   <li>Pay bills</li>
> const modifiedTodo = document.createElement('li');
  modifiedTodo.textContent = 'Feed the dog';
< "Feed the dog"
> todoList.replaceChild(modifiedTodo, todoList.children[2]);
<   <li>Feed the cat</li>
> todoList.removeChild(todoList.lastElementChild);
<   <li>Do homework</li>
> todoList.children[1].remove();
< undefined
>
```

DOM Screenshot 6

Between `removeChild()` and `remove()`, you can remove any node from the DOM. Another method you may see for removing child elements from the DOM is setting the `innerHTML` property of a parent element to an empty string (`""`). This is not the preferred method because it is less explicit, but you might see it in existing code.

Conclusion

In this tutorial, we learned how to use JavaScript to create new nodes and elements and insert them into the DOM, and replace and remove existing nodes and elements.

At this point in the Understanding the DOM series you know how to access any element in the DOM, walk through any node in the DOM, and modify the DOM itself. You can now feel confident in creating basic front-end web apps with JavaScript.

How To Modify Attributes, Classes, and Styles in the DOM

Written by Tania Rascia

In the previous tutorial in this series, "How To Make Changes to the DOM," we covered how to create, insert, replace, and remove elements from the Document Object Model (DOM) with built-in methods. By increasing your proficiency in manipulating the DOM, you are better able to utilize JavaScript's interactive capabilities and modify web elements.

In this tutorial, we will learn how to further alter the DOM by modifying styles, classes, and other attributes of HTML element nodes. This will give you a greater understanding of how to manipulate essential elements within the DOM.

Review of Selecting Elements

Until recently, a popular JavaScript library called jQuery was most often used to select and modify elements in the DOM. jQuery simplified the process of selecting one or more elements and applying changes to all of them at the same time. In "How To Access Elements in the DOM," we reviewed the DOM methods for grabbing and working with nodes in vanilla JavaScript.

To review, `document.querySelector()` and `document.getElementById()` are the methods that are used to access a single element. Using a `div` with an `id` attribute in the example below, we can access that element either way.

```
<div id="demo-id">Demo ID</div>
```

The `querySelector()` method is more robust in that it can select an element on the page by any type of selector.

```
// Both methods will return a single element
const demoId = document.querySelector('#demo-id');
```

Accessing a single element, we can easily update a part of the element such as the text inside.

```
// Change the text of one element
demoId.textContent = 'Demo ID text updated.';
```

However, when accessing multiple elements by a common selector, such as a specific class, we have to loop through all the elements in the list. In the code below, we have two `div` elements with a common class value.

```
<div class="demo-class">Demo Class 1</div>
<div class="demo-class">Demo Class 2</div>
```

We'll use `querySelectorAll()` to grab all elements with `demo-class` applied to them, and `forEach()` to loop through them and apply a change. It is also possible to access a specific element with `querySelectorAll()` the same way you would with an array — by using bracket notation.

```
// Get a NodeList of all .demo elements
const demoClasses = document.querySelectorAll('.demo-
class');

// Change the text of multiple elements with a loop
demoClasses.forEach(element => {
  element.textContent = 'All demo classes updated.';
});

// Access the first element in the NodeList
demoClasses[0];
```

This is one of the most important differences to be aware of when progressing from jQuery to vanilla JavaScript. Many examples of modifying elements will not explain the process of applying those methods and properties to multiple elements.

The properties and methods in this article will often be attached to event listeners in order to respond to clicks, hovers, or other triggers.

Note: The methods `getElementsByClassName()` and `getElementsByTagName()` will return HTML collections which do not have access to the `forEach()` method that `querySelectorAll()` has. In these cases, you will need to use a standard `for` loop to iterate through the collection.

Modifying Attributes

Attributes are values that contain additional information about HTML elements. They usually come in name/value pairs, and may be essential depending on the element.

Some of the most common HTML attributes are the `src` attribute of an `img` tag, the `href` of an `a` tag, `class`, `id`, and `style`. For a full list of HTML attributes, view the attribute list on the Mozilla Developer Network. Custom elements that are not part of the HTML standard will be prepended with `data-`.

In JavaScript, we have four methods for modifying element attributes:

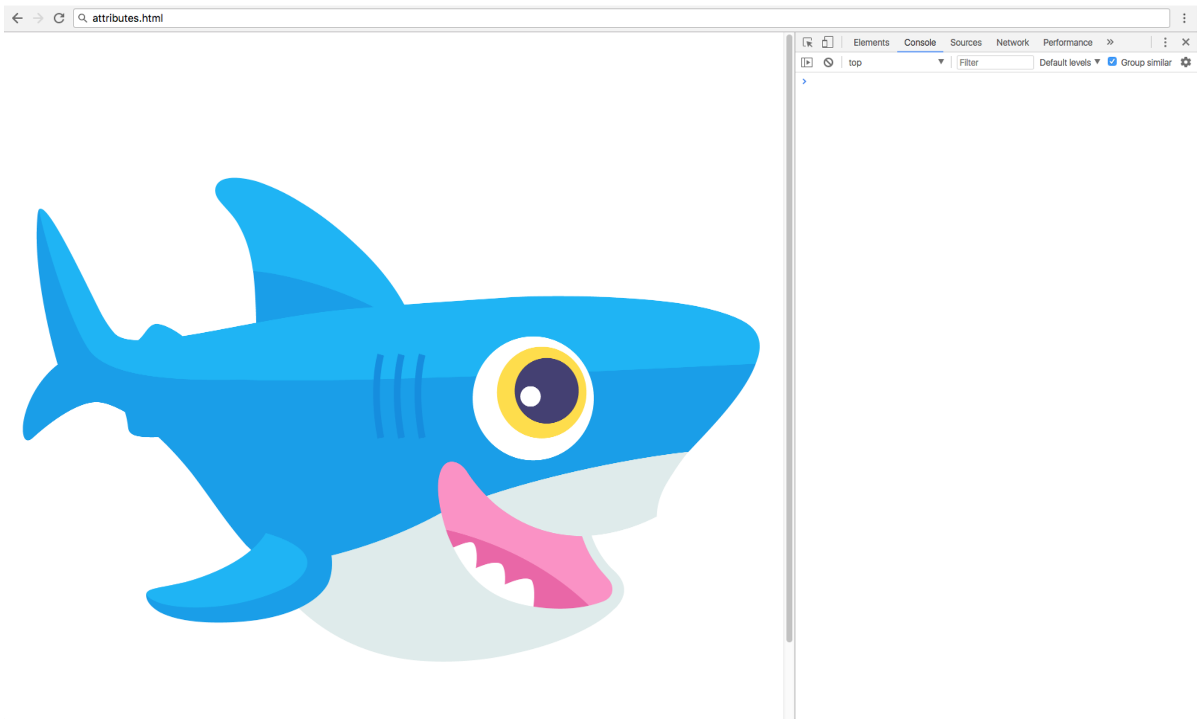
Method	Description	Example
`hasAttribute()`	Returns a `true` or `false` boolean	`element.hasAttribute('href');`
`getAttribute()`	Returns the value of a specified attribute or `null`	`element.getAttribute('href');`
`setAttribute()`	Adds or updates value of a specified attribute	`element.setAttribute('href', 'index.html');`
`removeAttribute()`	Removes an attribute from an element	`element.removeAttribute('href');`

Let's create a new HTML file with an `img` tag with one attribute. We'll link to a public image available via a URL, but you can swap it out for an alternate local image if you're working offline.

attributes.html

```
<!DOCTYPE html>
<html lang="en">
<body>

    <img src="https://js-
tutorials.nyc3.digitaloceanspaces.com/shark.png">

</body>

</html>
```

When you load the above HTML file into a modern web browser and open the built-in Developer Console, you should see something like this:



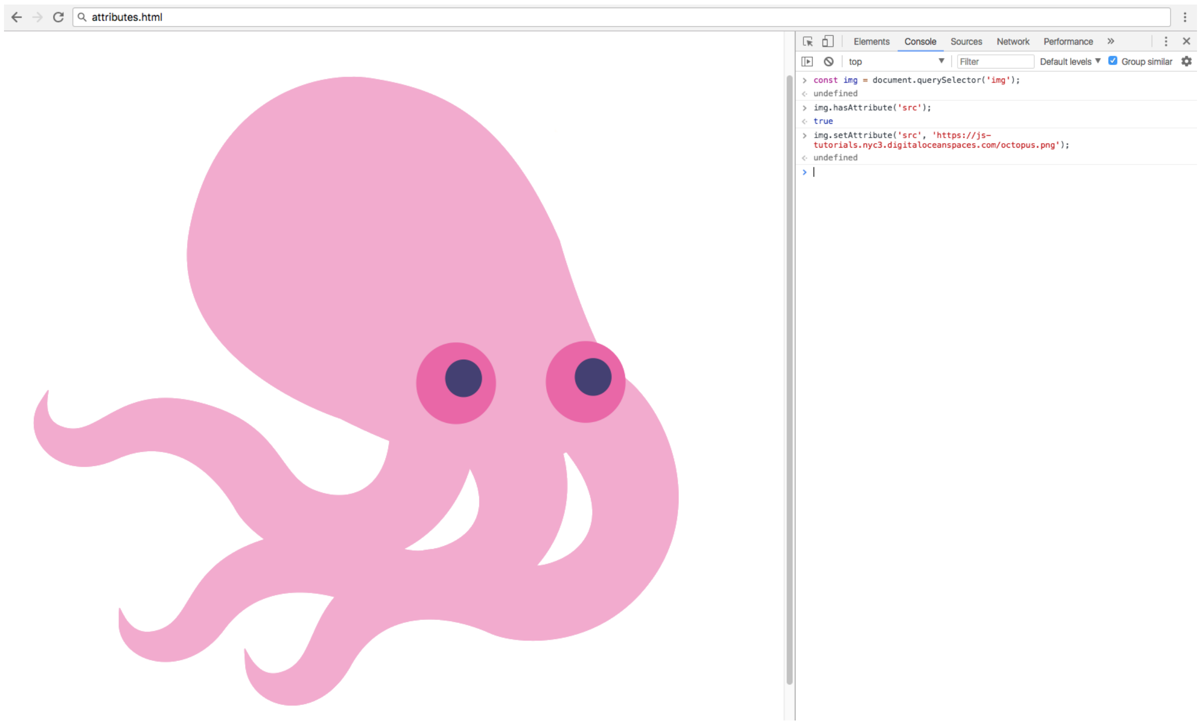
First rendering of classes.html

Now, we can test all the attribute methods on the fly.

```
// Assign image element
const img = document.querySelector('img');

img.hasAttribute('src');                    // returns
true
img.getAttribute('src');                    // returns
"...shark.png"
img.removeAttribute('src');                 // remove the
src attribute and value
```

At this point, you will have removed the `src` attribute and value associated with `img`, but you can reset that attribute and assign the value to an alternate image with `img.setAttribute()`:

```
img.setAttribute('src', 'https://js-
tutorials.nyc3.digitaloceanspaces.com/octopus.png');
```



```
> const img = document.querySelector('img');
<· undefined
> img.hasAttribute('src');
<· true
> img.setAttribute('src', 'https://js-
  tutorials.nyc3.digitaloceanspaces.com/octopus.png');
<· undefined
>
```

Second rendering of classes.html

Finally, we can modify the attribute directly by assigning a new value to the attribute as a property of the element, setting the `src` back to the `shark.png` file

```
img.src = 'https://js-
tutorials.nyc3.digitaloceanspaces.com/shark.png';
```

Any attribute can be edited this way as well as with the above methods.

The `hasAttribute()` and `getAttribute()` methods are usually used with conditional statements, and the `setAttribute()` and `removeAttribute()` methods are used to directly modify the DOM.

Modifying Classes

The class attribute corresponds to CSS class selectors. This is not to be confused with ES6 classes, a special type of JavaScript function.

CSS classes are used to apply styles to multiple elements, unlike IDs which can only exist once per page. In JavaScript, we have the `className` and `classList` properties to work with the class attribute.

Method/Property	Description	Example
`className`	Gets or sets class value	`element.className;`
`classList.add()`	Adds one or more class values	`element.classList.add('active');`
`classList.toggle()`	Toggles a class on or off	`element.classList.toggle('active');`
`classList.contains()`	Checks if class value exists	`element.classList.contains('active');`
`classList.replace()`	Replace an existing class value with a new class value	`element.classList.replace('old', 'new');`
`classList.remove()`	Remove a class value	`element.classList.remove('active');`

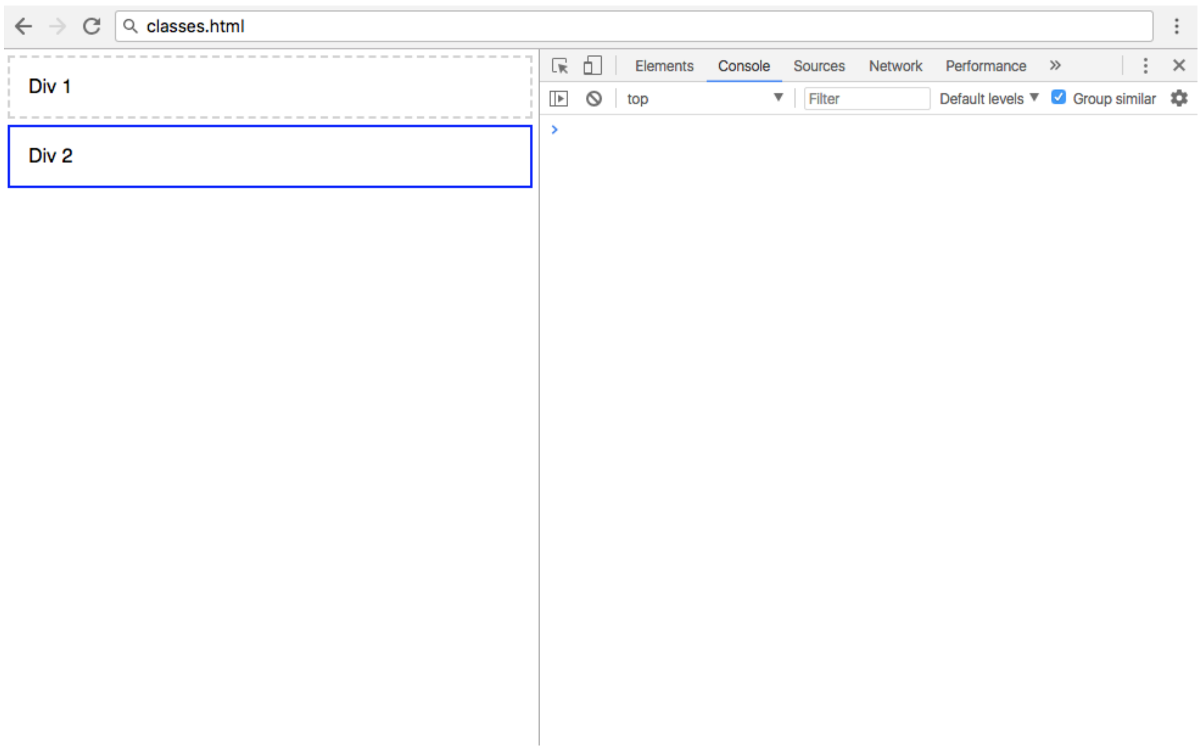
We'll make another HTML file to work with the class methods, with two elements and a few classes.

classes.html

```
<!DOCTYPE html>
<html lang="en">

<style>
    body {
        max-width: 600px;
        margin: 0 auto;
        font-family: sans-serif;
    }
    .active {
        border: 2px solid blue;
    }

    .warning {
        border: 2px solid red;
    }

    .hidden {
        display: none;
    }

    div {
        border: 2px dashed lightgray;
        padding: 15px;
        margin: 5px;
    }
</style>
```

```
<body>

    <div>Div 1</div>
    <div class="active">Div 2</div>

</body>

</html>
```

When you open the `classes.html` file into a web browser, you should receive a rendering that looks similar to the following:

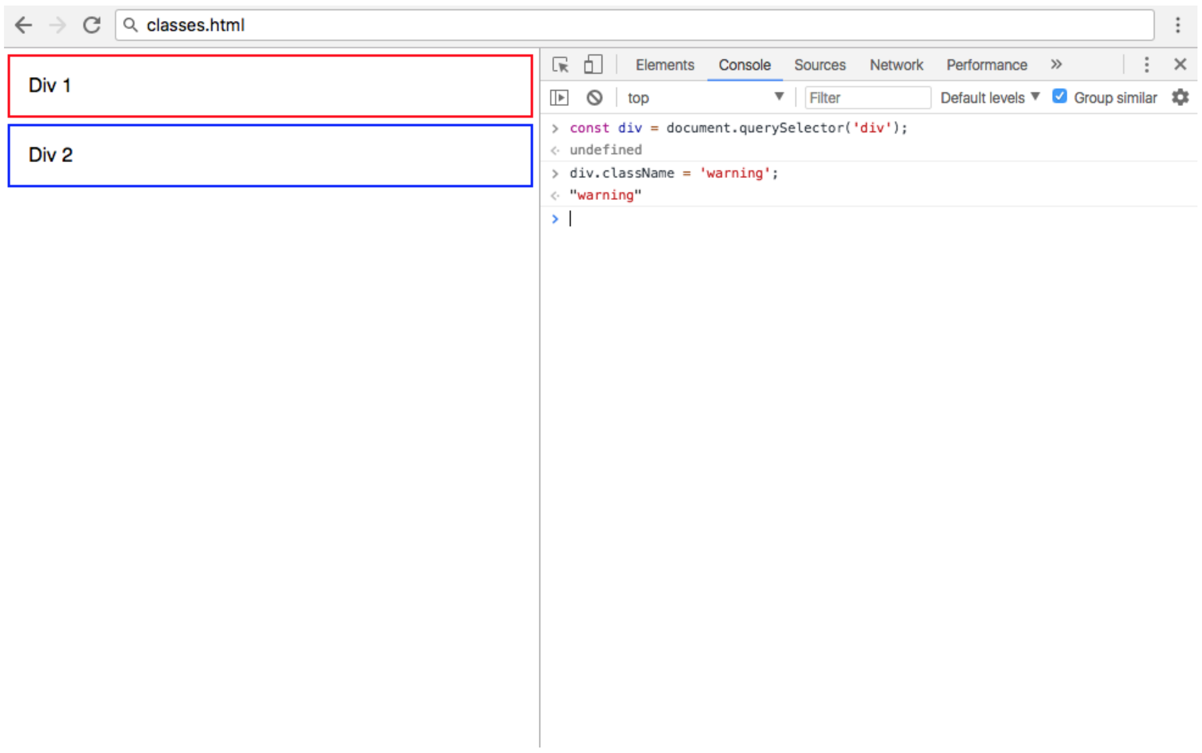


First rendering of classes.html

The `className` property was introduced to prevent conflicts with the `class` keyword found in JavaScript and other languages that have access to the DOM. You can use `className` to assign a value directly to the class.

```
// Select the first div
const div = document.querySelector('div');

// Assign the warning class to the first div
div.className = 'warning';
```

We have assigned the `warning` class defined in the CSS values of `classes.html` to the first `div`. You'll receive the following output:



```
> const div = document.querySelector('div');
< undefined
> div.className = 'warning';
< "warning"
>
```

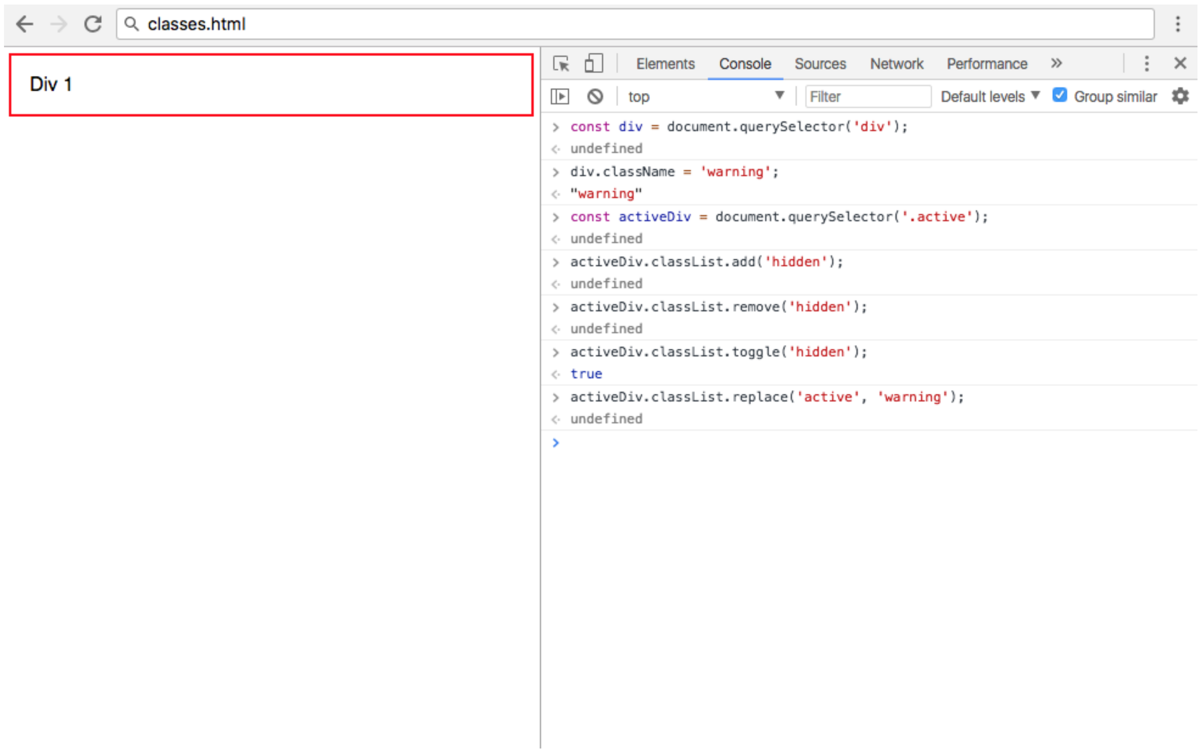
Second rendering of classes.html

Note that if any classes already exist on the element, this will override them. You can add multiple space delimited classes using the `className` property, or use it without assignment operators to get the value of the class on the element.

The other way to modify classes is via the [`classList`]() property, which comes with a few helpful methods. These methods are similar to the jQuery `addClass`, `removeClass`, and `toggleClass` methods.

```
// Select the second div by class name
const activeDiv = document.querySelector('.active');

activeDiv.classList.add('hidden');                  //
Add the hidden class
activeDiv.classList.remove('hidden');               //
Remove the hidden class
activeDiv.classList.toggle('hidden');               //
Switch between hidden true and false
activeDiv.classList.replace('active', 'warning'); //
Replace active class with warning class
```

After performing the above methods, your web page will look like this:



```
> const div = document.querySelector('div');
< undefined
> div.className = 'warning';
< "warning"
> const activeDiv = document.querySelector('.active');
< undefined
> activeDiv.classList.add('hidden');
< undefined
> activeDiv.classList.remove('hidden');
< undefined
> activeDiv.classList.toggle('hidden');
< true
> activeDiv.classList.replace('active', 'warning');
< undefined
>
```

Final rendering of classes.html

Unlike in the `className` example, using `classList.add()` will add a new class to the list of existing classes. You can also add multiple classes as comma-separated strings. It is also possible to use `setAttribute` to modify the class of an element.

Modifying Styles

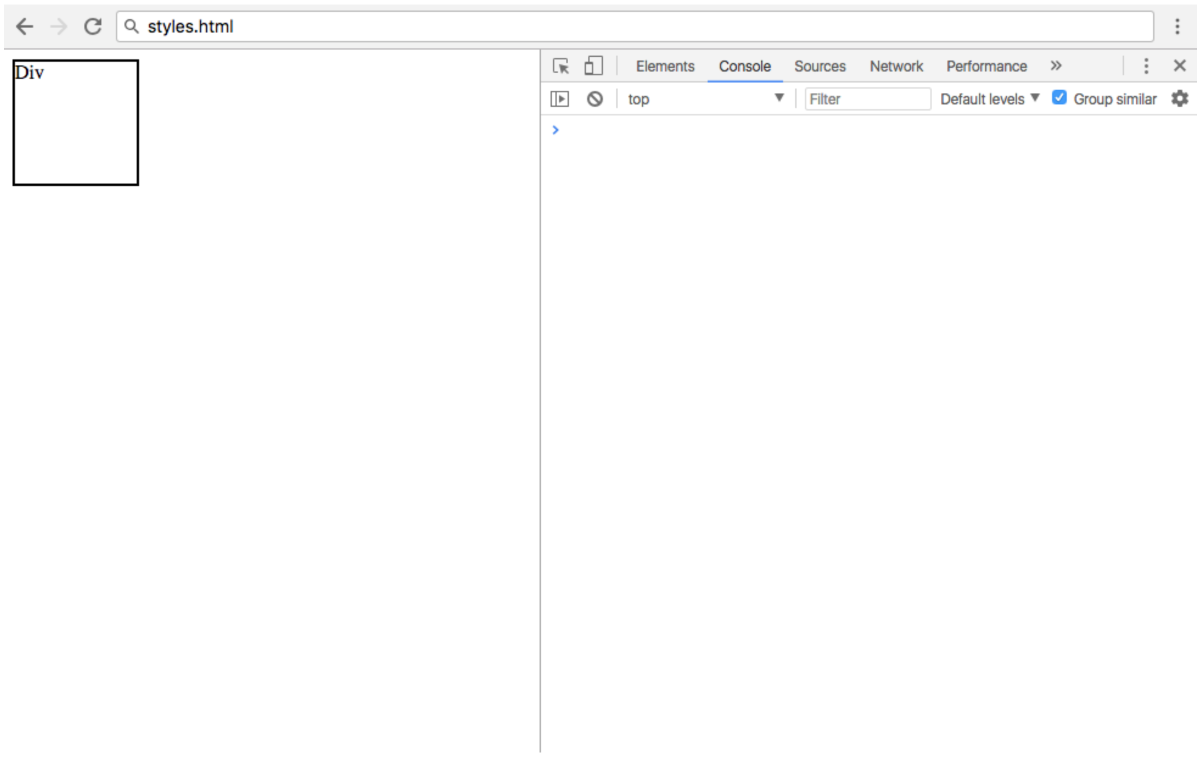
The style property repesents the inline styles on an HTML element. Often, styles will be applied to elements via a stylesheet as we have done previously in this article, but sometimes we have to add or edit an inline style directly.

We will make a short example to demonstrate editing styles with JavaScript. Below is a new HTML file with a `div` that has some inline styles applied to display a square.

styles.html

```
<!DOCTYPE html>
<html lang="en">

<body>

    <div style="height: 100px;
                width: 100px;
                border: 2px solid black;">Div</div>

</body>

</html>
```

When opened in a web browser, the `styles.html` will look something like this:



First rendering of styles.html

One option to edit the styles is with `setAttribute()`.

```
// Select div
const div = document.querySelector('div');

// Apply style to div
div.setAttribute('style', 'text-align: center');
```

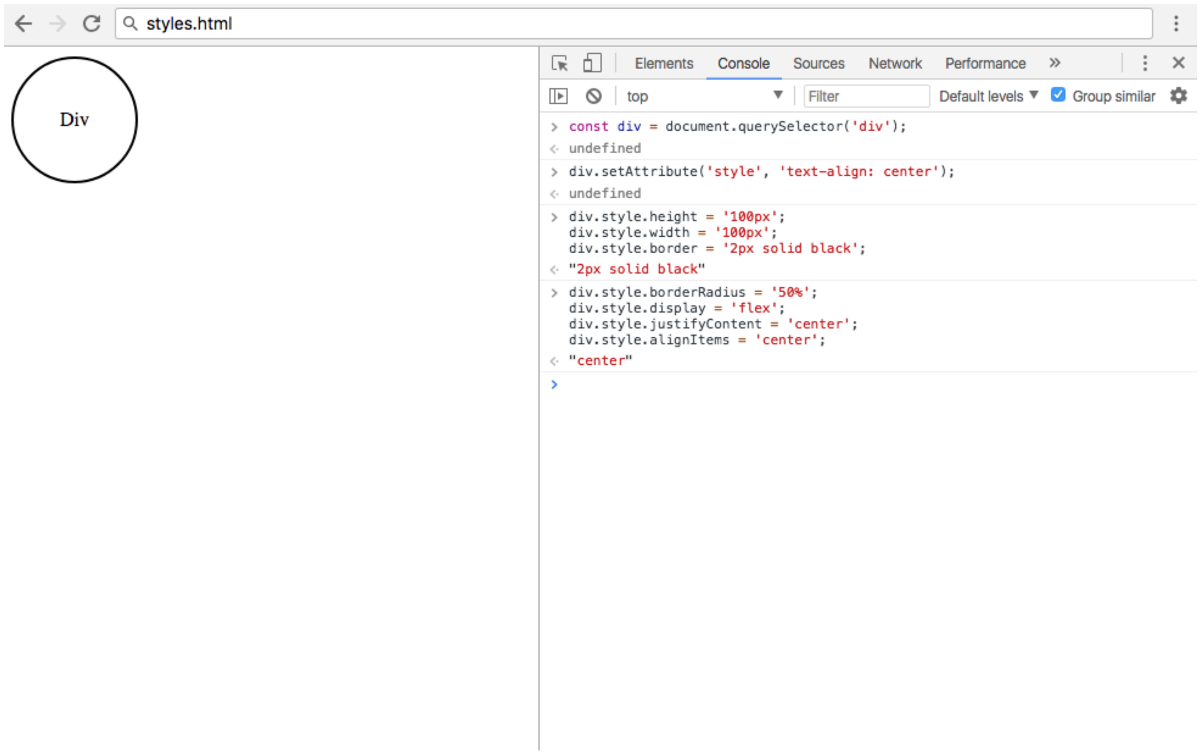
However, this will remove all existing inline styles from the element. Since this is likely not the intended effect, it is better to use the `style` attribute directly

```
div.style.height = '100px';
div.style.width = '100px';
div.style.border = '2px solid black';
```

CSS properties are written in kebab-case, which is lowercase words separated by dashes. It is important to note that kebab-case CSS properties cannot be used on the JavaScript style property. Instead, they will be replaced with their camelCase equivalent, which is when the first word is lowercase, and all subsequent words are capitalized. In other words, instead of `text-align` we will use `textAlign` for the JavaScript style property.

```
// Make div into a circle and vertically center the
text
div.style.borderRadius = '50%';
div.style.display = 'flex';
div.style.justifyContent = 'center';
div.style.alignItems = 'center';
```

After completing the above style modifications, your final rendering of `styles.html` will show a circle:



```
> const div = document.querySelector('div');
< undefined
> div.setAttribute('style', 'text-align: center');
< undefined
> div.style.height = '100px';
  div.style.width = '100px';
  div.style.border = '2px solid black';
< "2px solid black"
> div.style.borderRadius = '50%';
  div.style.display = 'flex';
  div.style.justifyContent = 'center';
  div.style.alignItems = 'center';
< "center"
>
```

Final rendering of styles.html

If many stylistic changes are to be applied to an element, the best course of action is to apply the styles to a class and add a new class. However, there are some cases in which modifying the inline style attribute will be necessary or more straightforward.

Conclusion

HTML elements often have additional information assigned to them in the form of attributes. Attributes may consist of name/value pairs, and a few of the most common attributes are `class` and `style`.

In this tutorial, we learned how to access, modify, and remove attributes on an HTML element in the DOM using plain JavaScript. We also learned how to add, remove, toggle, and replace CSS classes on an element, and how to edit inline CSS styles. For additional reading, check out the documentation on attributes on the Mozilla Developer Network.

Understanding Events in JavaScript

Written by Tania Rascia

In the Understanding the DOM series, we have discussed the DOM tree and how to access, traverse, add and remove, and modify nodes and elements using the Developer Tools Console.

Although at this point we can now make almost any change we want to the DOM, from a user perspective it is not very helpful because we have only manually triggered changes. By learning about events, we will understand how to tie everything together to make interactive websites.

Events are actions that take place in the browser that can be initiated by either the user or the browser itself. Below are a few examples of common events that can happen on a website:

- The page finishes loading
- The user clicks a button
- The user hovers over a dropdown
- The user submits a form
- The user presses a key on their keyboard

By coding JavaScript responses that execute upon an event, developers can display messages to users, validate data, react to a button click, and many other actions.

In this article, we will go over event handlers, event listeners, and event objects. We'll also go over three different ways to write code to handle events, and a few of the most common events. By learning about events, you'll be able to make a more interactive web experience for end users.

Event Handlers and Event Listeners

When a user clicks a button or presses a key, an event is fired. These are called a click event or a keypress event, respectively.

An event handler is a JavaScript function that runs when an event fires.

An event listener attaches a responsive interface to an element, which allows that particular element to wait and "listen" for the given event to fire.

There are three ways to assign events to elements: - Inline event handlers - Event handler properties - Event listeners

We will go over all three methods to ensure that you are familiar with each way an event can be triggered, then discuss the pros and cons of each method.

Inline Event Handler Attributes

To begin learning about event handlers, we'll first consider the inline event handler. Let's start with a very basic example that consists of a `button` element and a `p` element. We want the user to click the `button` to change the text content of the `p`.

Let's begin with an HTML page with a button in the body. We'll be referencing a JavaScript file that we'll add code to in a bit.

events.html

```html
<!DOCTYPE html>
<html lang="en-US">

<head>
    <title>Events</title>
</head>

<body>

  <!-- Add button -->
  <button>Click me</button>

  <p>Try to change me.</p>

</body>

<!-- Reference JavaScript file -->
<script src="js/events.js"></script>

</html>
```

Directly on the `button`, we will add an attribute called `onclick`. The attribute value will be a function we create called `changeText()`.

events.html

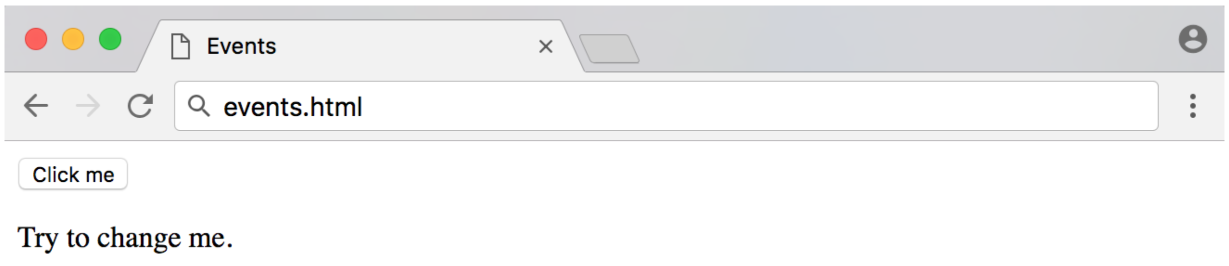
```
<!DOCTYPE html>
<html lang="en-US">

<head>
    <title>Events</title>
</head>

<body>

    <button onclick="changeText()">Click
me</button>

    <p>Try to change me.</p>

</body>

<script src="js/events.js"></script>

</html>
```

Let’s create our `events.js` file, which we placed in the `js/` directory here. Within it, we will create the `changeText()` function, which will modify the `textContent` of the `p` element.

js/events.js

```
// Function to modify the text content of the
paragraph
const changeText = () => {
    const p = document.querySelector('p');

    p.textContent = "I changed because of an
inline event handler.";
}
```

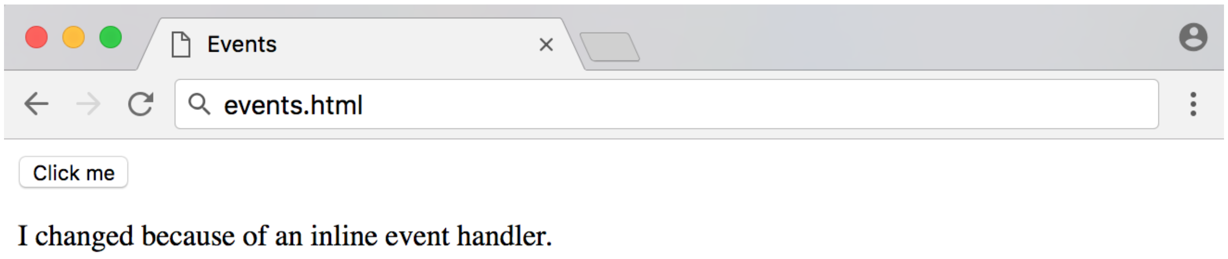
When you first load the `events.html`, you’ll see a page that looks like this:



Try to change me.

First rendering of events.html

However, when either you or another user clicks on the button, the text of the `p` tag will change from `Try to change me.` to `I changed because of an inline event handler.`:



I changed because of an inline event handler.

First response to event on events.html rendering

Inline event handlers are a straightforward way to begin understanding events, but they generally should not be used beyond testing and educational purposes.

You can compare inline event handlers to inline CSS styles on an HTML element. It is much more practical to maintain a separate stylesheet of classes than create inline styles on every element, just as it is more feasible to maintain JavaScript that is handled entirely through a separate script file than add handlers to every element.

Event Handler Properties

The next step up from an inline event handler is the event handler property. This works very similarly to an inline handler, except we're setting the property of an element in JavaScript instead of the attribute in the HTML.

The setup will be the same here, except we no longer include the `onclick="changeText()"` in the markup:

events.html

```
...
<body>

    <button>Click me</button>

    <p>I will change.</p>

</body>
...
```

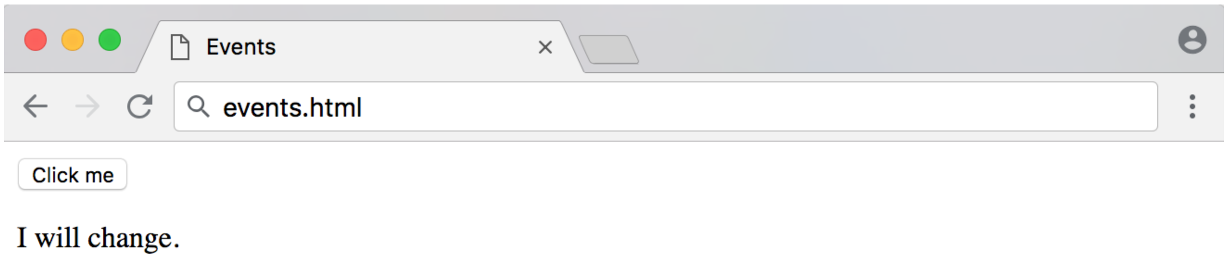
Our function will remain similar as well, except now we need to access the `button` element in the JavaScript. We can simply access `onclick` just as we would access `style` or `id` or any other element property, then assign the function reference.

js/events.js

```
// Function to modify the text content of the
paragraph
const changeText = () => {
    const p = document.querySelector('p');

    p.textContent = "I changed because of an event
handler property.";
}

// Add event handler as a property of the button
element
const button = document.querySelector('button');
button.onclick = changeText;
```

Note: Event handlers do not follow the camelCase convention that most JavaScript code adheres to. Notice that the code is `onclick`, not `onClick`.

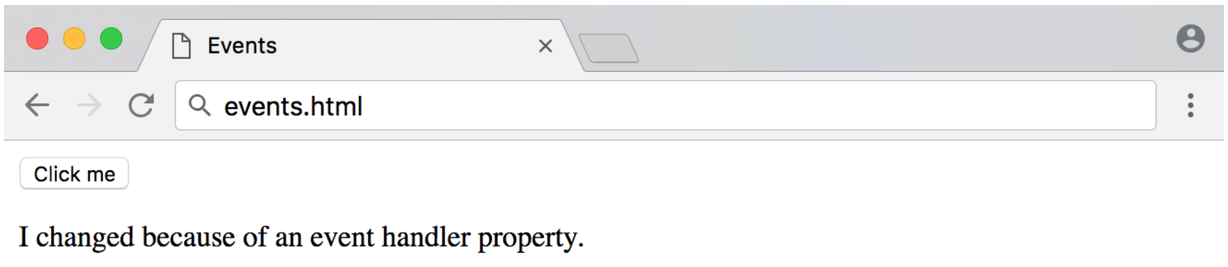
When you first load the page, the browser will display the following:



I will change.

Initial load of events.html with events handler

Now when you click the button, it will have a similar effect as before:



I changed because of an event handler property.

Response with events handler of events.html

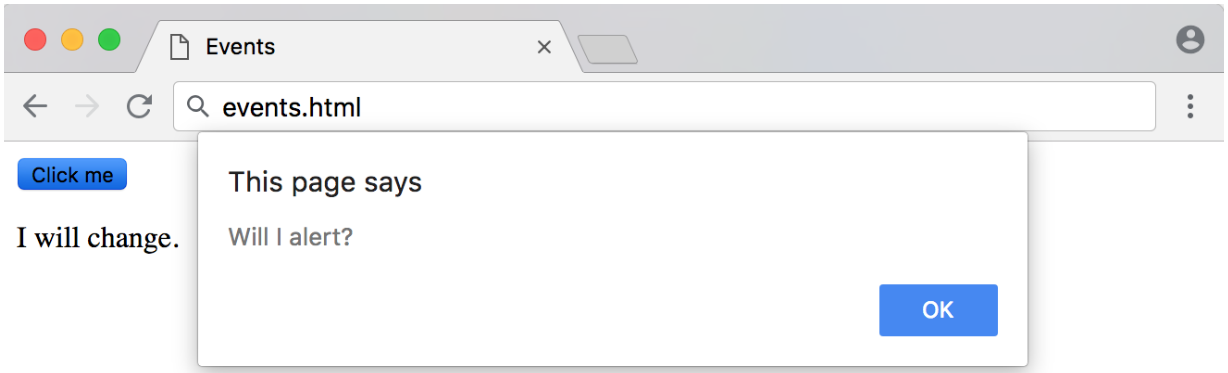
Note that when passing a function reference to the `onclick` property, we do not include parentheses, as we are not invoking the function in that moment, but only passing a reference to it.

The event handler property is slightly more maintainable than the inline handler, but it still suffers from some of the same hurdles. For example, trying to set multiple, separate `onclick` properties will cause all but the last one to be overwritten, as demonstrated below.

js/events.js

```
const p = document.querySelector('p');
const button = document.querySelector('button');

const changeText = () => {
    p.textContent = "Will I change?";
}

const alertText = () => {
    alert('Will I alert?');
}

// Events can be overwritten
button.onclick = changeText;
button.onclick = alertText;
```

In the above example, the `button` click would only display an alert, and not change the `p` text, since the `alert()` code was the last one added to the property.



Final response via events handler of events.html

With an understanding of both inline event handlers and event handler properties, let's move onto event listeners.

Event Listeners

The latest addition to JavaScript event handlers are event listeners. An event listener watches for an event on an element. Instead of assigning the event directly to a property on the element, we will use the `addEventListener()` method to listen for the event.

`addEventListener()` takes two mandatory parameters — the event it is to be listening for, and the listener callback function.

The HTML for our event listener will be the same as the previous example.

events.html

```
...
    <button>Click me</button>

    <p>I will change.</p>
...
```

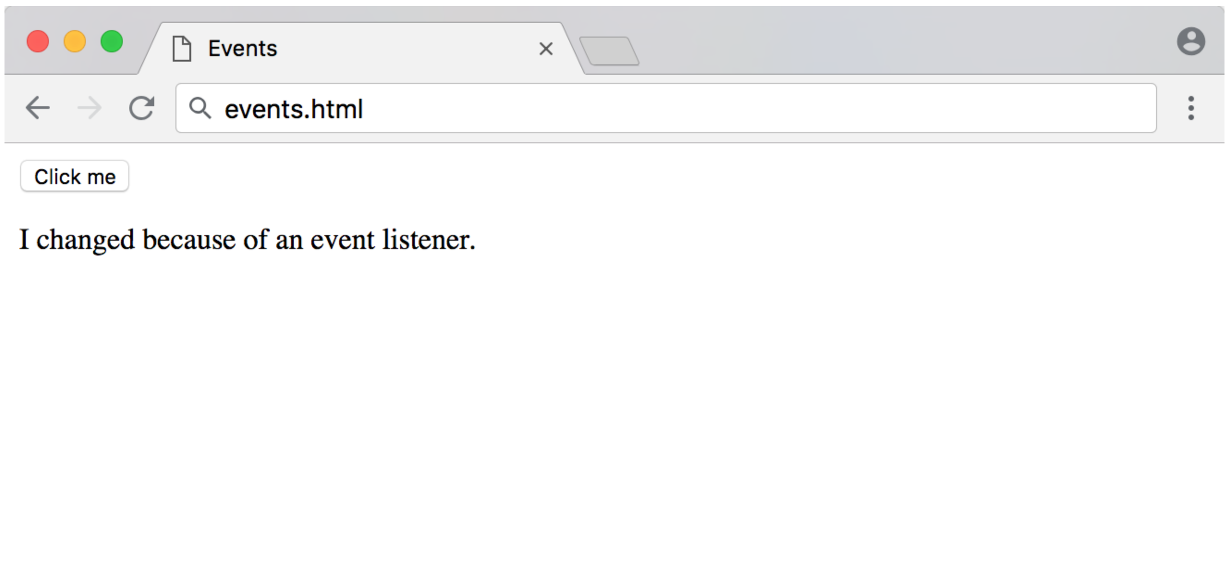
We will still be using the same `changeText()` function as before. We'll attach the `addEventListener()` method to the button.

js/events.js

```
// Function to modify the text content of the
paragraph
const changeText = () => {
    const p = document.querySelector('p');

    p.textContent = "I changed because of an event
listener.";
}

// Listen for click event
const button = document.querySelector('button');
button.addEventListener('click', changeText);
```

Notice that with the first two methods, a click event was referred to as `onclick`, but with event listeners it is referred to as `click`. Every event listener drops the `on` from the word. In the next section, we will look at more examples of other types of events.

When you reload the page with the JavaScript code above, you'll receive the following output:



I changed because of an event listener.

Event listener response of events.html

At first look, event listeners seem very similar to event handler properties, but they have a few advantages. We can set multiple event listeners on the same element, as demonstrated in the example below.

js/events.js

```
const p = document.querySelector('p');
const button = document.querySelector('button');

const changeText = () => {
    p.textContent = "Will I change?";
}

const alertText = () => {
    alert('Will I alert?');
}

// Multiple listeners can be added to the same
event and element
button.addEventListener('click', changeText);
button.addEventListener('click', alertText);
```

In this example, both events will fire, providing the user with both an alert and modified text once clicking out of the alert.

Often, anonymous functions will be used instead of a function reference on an event listener. Anonymous functions are functions that are not named.

```
// An anonymous function on an event listener
button.addEventListener('click', () => {
    p.textContent = "Will I change?";
});
```

It is also possible to use the `removeEventListener()` function to remove one or all events from an element.

```
// Remove alert function from button element
button.removeEventListener('click', alertText);
```

Furthermore, you can use `addEventListener()` on the `document` and `window` object.

Event listeners are currently the most common and preferred way to handle events in JavaScript.

Common Events

We have learned about inline event handlers, event handler properties, and event listeners using the click event, but there are many more events in JavaScript. We will go over a few of the most common events below.

Mouse Events

Mouse events are among the most frequently used events. They refer to events that involve clicking buttons on the mouse or hovering and moving the mouse pointer. These events also correspond to the equivalent action on a touch device.

Event	Description
`click`	Fires when the mouse is pressed and released on an element
`dblclick`	Fires when an element is clicked twice
`mouseenter`	Fires when a pointer enters an element
`mouseleave`	Fires when a pointer leaves an element
`mousemove`	Fires every time a pointer moves inside an element

A `click` is a compound event that is comprised of combined `mousedown` and `mouseup` events, which fire when the mouse button is pressed down or lifted, respectively.

Using `mouseenter` and `mouseleave` in tandem recreates a hover effect that lasts as long as a mouse pointer is on the element.

Form Events

Form events are actions that pertain to forms, such as `input` elements being selected or unselected, and forms being submitted.

Event	Description
`submit`	Fires when a form is submitted
`focus`	Fires when an element (such as an input) receives focus
`blur`	Fires when an element loses focus

Focus is achieved when an element is selected, for example, through a mouse click or navigating to it via the `TAB` key.

JavaScript is often used to submit forms and send the values through to a backend language. The advantage of using JavaScript to send forms is that it does not require a page reload to submit the form, and JavaScript can be used to validate required input fields.

Keyboard Events

Keyboard events are used for handling keyboard actions, such as pressing a key, lifting a key, and holding down a key.

Event	Description
`keydown`	Fires once when a key is pressed
`keyup`	Fires once when a key is released
`keypress`	Fires continuously while a key is pressed

Although they look similar, `keydown` and `keypress` events do not access all the exact same keys. While `keydown` will acknowledge every key that is pressed, `keypress` will omit keys that do not produce a character, such as `SHIFT`, `ALT`, or `DELETE`.

Keyboard events have specific properties for accessing individual keys.

If a parameter, known as an `event` object, is passed through to the event listener, we can access more information about the action that took place. Three properties that pertain to keyboard objects include `keyCode`, `key`, and `code`.

For example, if the user presses the letter `a` key on their keyboard, the following properties pertaining to that key will surface:

PROPERTY	DESCRIPTION	EXAMPLE
`keyCode`	A number pertaining to the key	65
`key`	Represents the character name	a
`code`	Represents the physical key being pressed	KeyA

To show how to gather that information via the JavaScript Console, we can write the following lines of code.

```
// Test the keyCode, key, and code properties
document.addEventListener('keydown', event => {
    console.log('key: ' + event.keyCode);
    console.log('key: ' + event.key);
    console.log('code: ' + event.code);
});
```

Once we press `ENTER` on the Console, we can now press a key on the keyboard, in this example, we'll press `a`.

Output

```
keyCode: 65
key: a
code: KeyA
```

The `keyCode` property is a number that pertains to the key that has been pressed. The `key` property is the name of the character, which can change — for example, pressing `a` with `SHIFT` would result in a `key` of `A`. The `code` property represents the physical key on the keyboard.

Note that `keyCode` is in the process of being deprecated and it is preferable to use `code` in new projects.

To learn more, you can view the complete list of events on the Mozilla Developer Network.

Event Objects

The `Event` object consists of properties and methods that all events can access. In addition to the generic `Event` object, each type of event has its own extensions, such as `KeyboardEvent` and `MouseEvent`.

The `Event` object is passed through a listener function as a parameter. It is usually written as `event` or `e`. We can access the `code` property of the `keydown` event to replicate the keyboard controls of a PC game.

To try it out, create a basic HTML file with `<p>` tags and load it into a browser.

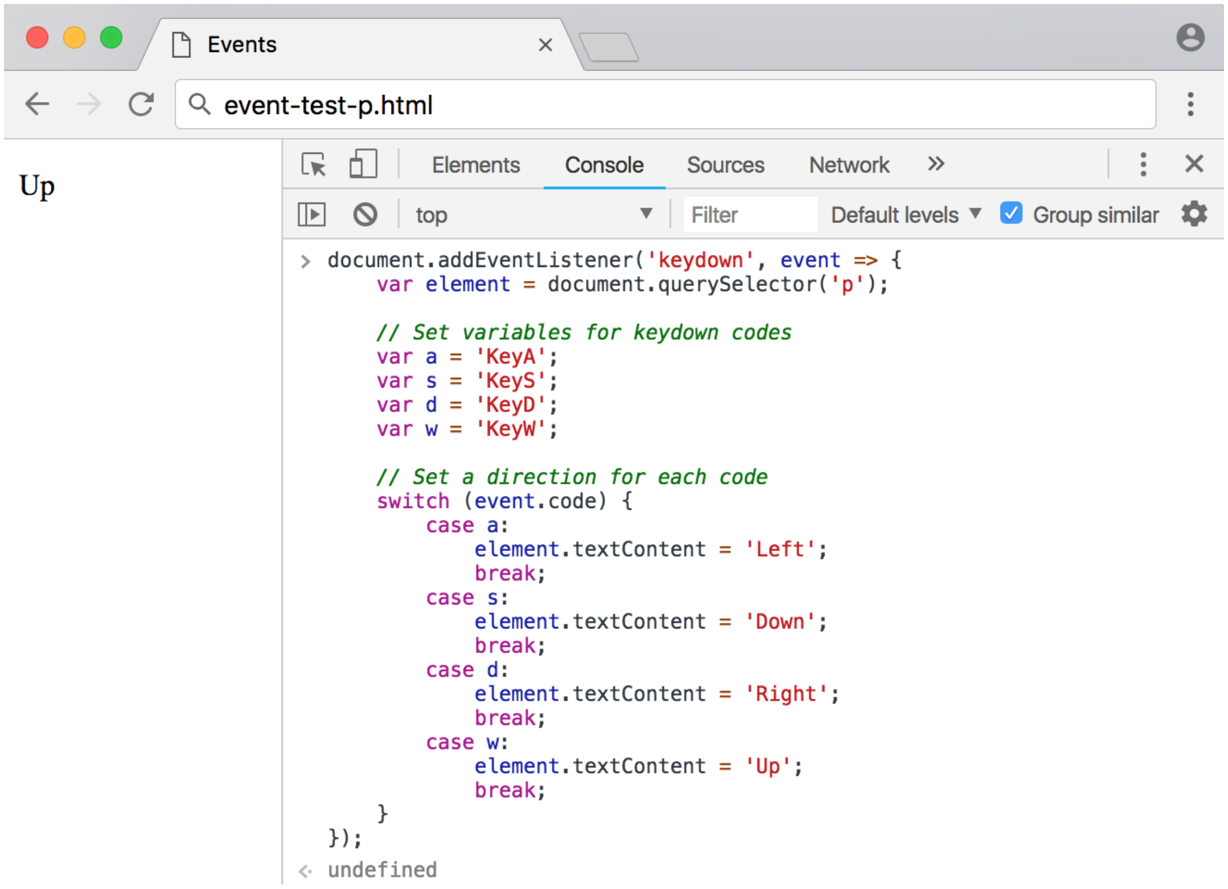
event-test-p.html

```
<!DOCTYPE html>
<html lang="en-US">
<head>
    <title>Events</title>
</head>
<body>

  <p></p>

</body>
</html>
```

Then, type the following JavaScript code into your browser's Developer Console.

```
});


// Pass an event through to a listener
document.addEventListener('keydown', event => {
    var element = document.querySelector('p');

    // Set variables for keydown codes
    var a = 'KeyA';
    var s = 'KeyS';
    var d = 'KeyD';
    var w = 'KeyW';

    // Set a direction for each code
    switch (event.code) {
        case a:
            element.textContent = 'Left';
            break;
        case s:
            element.textContent = 'Down';
            break;
        case d:
            element.textContent = 'Right';
            break;
        case w:
            element.textContent = 'Up';
            break;
    }
});
```

When you press one of the keys — `a`, `s`, `d`, or `w` — you'll see output similar to the following:



```
> document.addEventListener('keydown', event => {
      var element = document.querySelector('p');

      // Set variables for keydown codes
      var a = 'KeyA';
      var s = 'KeyS';
      var d = 'KeyD';
      var w = 'KeyW';

      // Set a direction for each code
      switch (event.code) {
          case a:
              element.textContent = 'Left';
              break;
          case s:
              element.textContent = 'Down';
              break;
          case d:
              element.textContent = 'Right';
              break;
          case w:
              element.textContent = 'Up';
              break;
      }
  });
< undefined
```

First event object example

From here, you can continue to develop how the browser will respond and to the user pressing those keys, and can create a more dynamic website.

Next, we'll go over one of the most frequently used event properties: the `target` property. In the following example, we have three `div` elements inside one `section`.

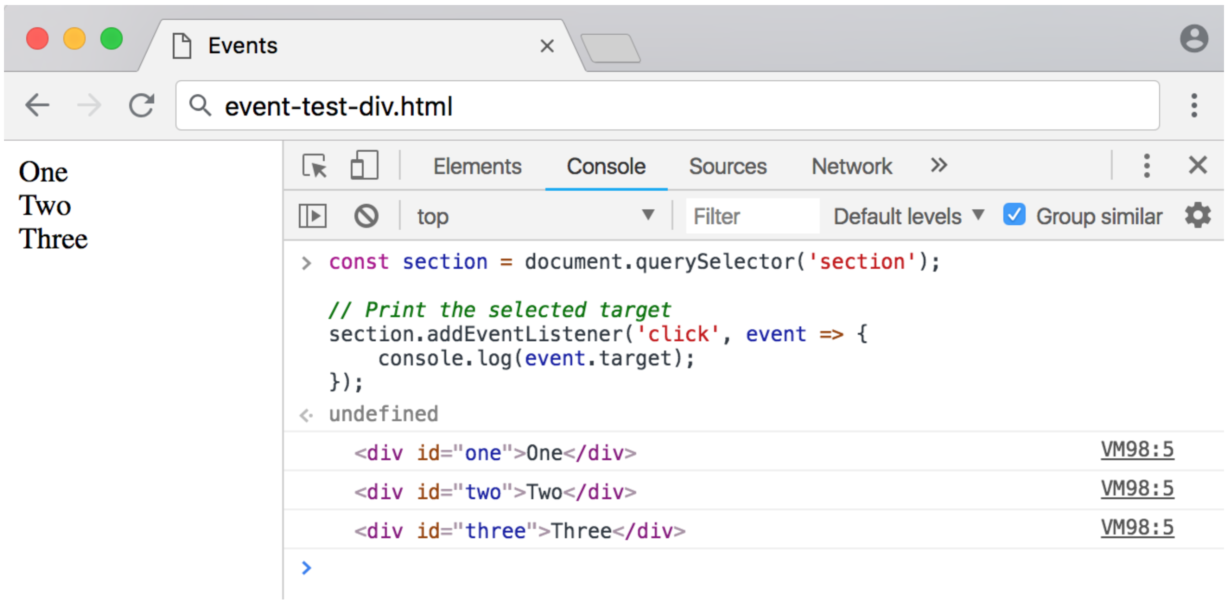
event-test-div.html

```
<!DOCTYPE html>
<html lang="en-US">
<head>
    <title>Events</title>
</head>
<body>

  <section>
    <div id="one">One</div>
    <div id="two">Two</div>
    <div id="three">Three</div>
  </section>

</body>
</html>
```

Using `event.target` with JavaScript in our browser's Developer Console, we can place one event listener on the outer `section` element and get the most deeply nested element.

```
const section = document.querySelector('section');

// Print the selected target
section.addEventListener('click', event => {
    console.log(event.target);
});
```

Clicking on any one of those elements will return output of the relevant specific element to the Console using `event.target`. This is extremely useful, as it allows you to place only one event listener that can be used to access many nested elements.



```
> const section = document.querySelector('section');

  // Print the selected target
  section.addEventListener('click', event => {
      console.log(event.target);
  });
<· undefined
    <div id="one">One</div>                                   VM98:5
    <div id="two">Two</div>                                   VM98:5
    <div id="three">Three</div>                               VM98:5
>
```

Second event object example

With the `Event` object, we can set up responses related to all events, including generic events and more specific extensions.

Conclusion

Events are actions that take place on a website, such as clicking, hovering, submitting a form, loading a page, or pressing a key on the keyboard. JavaScript becomes truly interactive and dynamic when we are able to make websites respond to actions the user has taken.

In this tutorial, we learned what events are, examples of common events, the difference between event handlers and event listeners, and how to access the `Event` object. Using this knowledge, you will be able to begin making dynamic websites and applications.