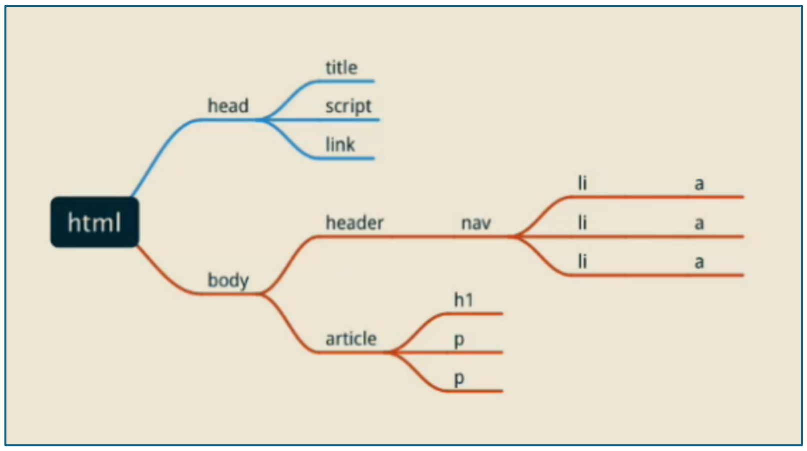
Document Object Model

Because JavaScript works hand in hand with HTML, you need to know how to interact with HTML via JavaScript code. The DOM (Document Object Model) is the interface that allows JavaScript to interact with HTML.



Every element in the HTML document is a node in a tree structure, with parent nodes, child nodes, and sibling nodes. For example, in this figure, the HTML is parent to the head and body areas on the page and then the body is a parent to its children, header and article, which are siblings and so on.

These elements model how the browser presents the page to the user. JavaScript can access, change, add, or delete elements in HTML by interacting with the DOM.

You can reference or select these elements using JavaScript with the following methods. The code shows the elements being selected and then assigned to a constant.

- querySelector() selects the *first* element that matches the specified CSS selector.

```
const element = document.querySelector("#myIdName");
```

In this example it finds the element with an id of 'myIdName'. There should be only one element with that id name and therefore the constant element will be assigned that one element.

- querySelectorAll() selects *all* elements that match the specified CSS selector and returns a node list instead of a single element.

```
const elements = document.querySelectorAll(".myClassName");
```

In this example it finds all elements with a class of ‘myClassName’. There could be multiple elements with this class name so in this case the constant elements could be assigned a list of elements.

A useful aspect of both querySelector methods is that they use the same CSS selectors, with dots for classes, hashtags for IDs, and element names.

- getElementById() selects the first element with the specified ID attribute.

```
const element = document.getElementById("myElementId");
```

Notice here that you don’t have to use the hashtag, ‘#’ because the method name getElementById already references that you are looking for an ID.

- getElementsByClassName() selects all elements with the specified class name and returns multiple elements.

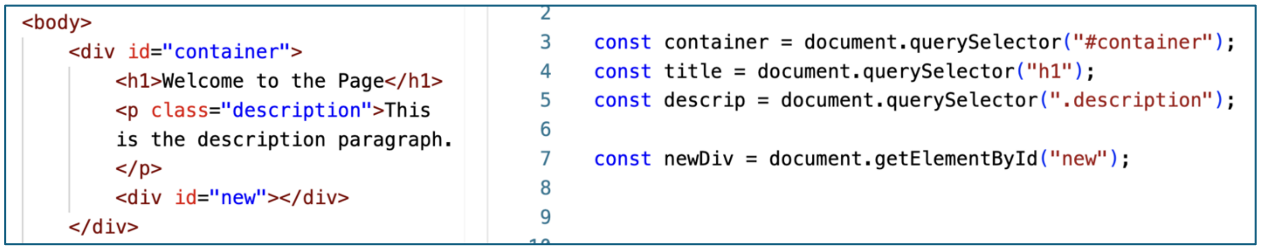
```
const elements = document.getElementsByClassName("myClassName");
```

Again, you don’t have to use a ‘.’ for class since the method name refers to class name already.

- getElementsByTagName() selects all elements with a specific tag name (like div, p, etc.) and returns multiple elements.

```
const elements = document.getElementsByTagName("p");
```

In the figure below, the first three statements use querySelector to select an element with the ID "container", an h1 element, and an element with the class "description". The last statement uses getElementById. Each of these four elements is then assigned to a constant.



```
<body>
    <div id="container">
        <h1>Welcome to the Page</h1>
        <p class="description">This
        is the description paragraph.
        </p>
        <div id="new"></div>
    </div>
```

```
const container = document.querySelector("#container");
const title = document.querySelector("h1");
const descrip = document.querySelector(".description");

const newDiv = document.getElementById("new");
```

Once you selected the elements, you can use different properties to change, add, or delete information about the element.

For example, to change the content of an element in JavaScript, you can use these properties:

- .innerHTML sets or gets the HTML content inside an element.

```
element.innerHTML = "<p>New content</p>";
```

In this example the element will have its content replaced with this new paragraph.

- .innerText sets or gets the visible text inside an element.

```
element.innerText = "New text content";
```

- .textContent is similar to innerText but includes any hidden elements.

```
element.textContent = "New text content";
```

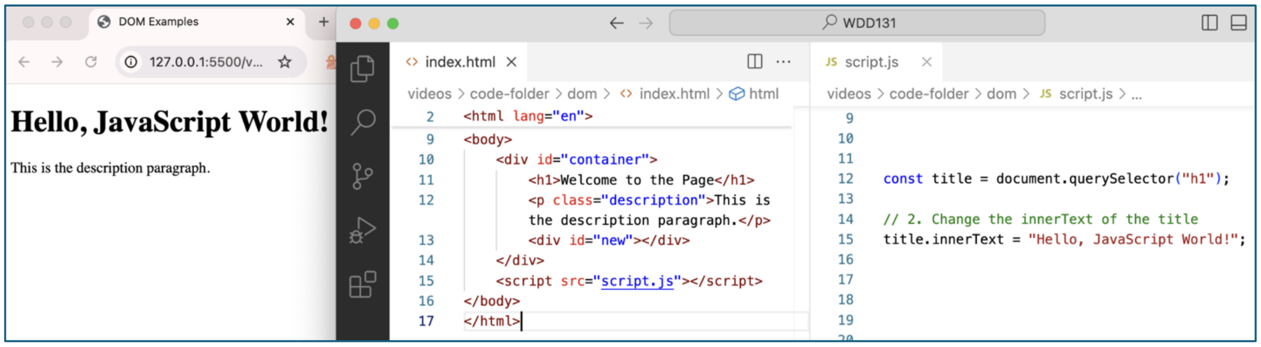
- .value is used with form input or textarea elements to set or get the input values.

```
inputElement.value = "New input value";
```

Often you will use the .value property to get the value from the user's input instead of setting a value. Like this:

```
let userInput = inputElement.value;
```

We can change the text in our title using innerHTML. Where it originally said, "Welcome to the Page" in HTML, it now says "Hello, JavaScript World!".


DOM Examples
127.0.0.1:5500/v...
WDD131
index.html
videos > code-folder > dom > index.html > html
script.js
videos > code-folder > dom > script.js > ...

Hello, JavaScript World!

This is the description paragraph.

```
<html lang="en">
<body>
    <div id="container">
        <h1>Welcome to the Page</h1>
        <p class="description">This is
        the description paragraph.</p>
        <div id="new"></div>
    </div>
    <script src="script.js"></script>
</body>
</html>
```

```


const title = document.querySelector("h1");

// 2. Change the innerText of the title
title.innerText = "Hello, JavaScript World!";
```

In addition to changing the content of an element, JavaScript allows you to change how the element is styled.

The style property can be followed by any CSS property in camelCase, which is then assigned a value.

- .style

```
element.style.color = "blue";
element.style.backgroundColor = "yellow";
element.style.fontSize = "20px";
```

You can also set or get a class name on elements to change how they are styled. className will replace any existing classes.

- .className

```
element.className = "newClass";
```

That element will now be assigned a new class called ‘newClass’ which will automatically bring in any CSS rules that are assigned to that class.

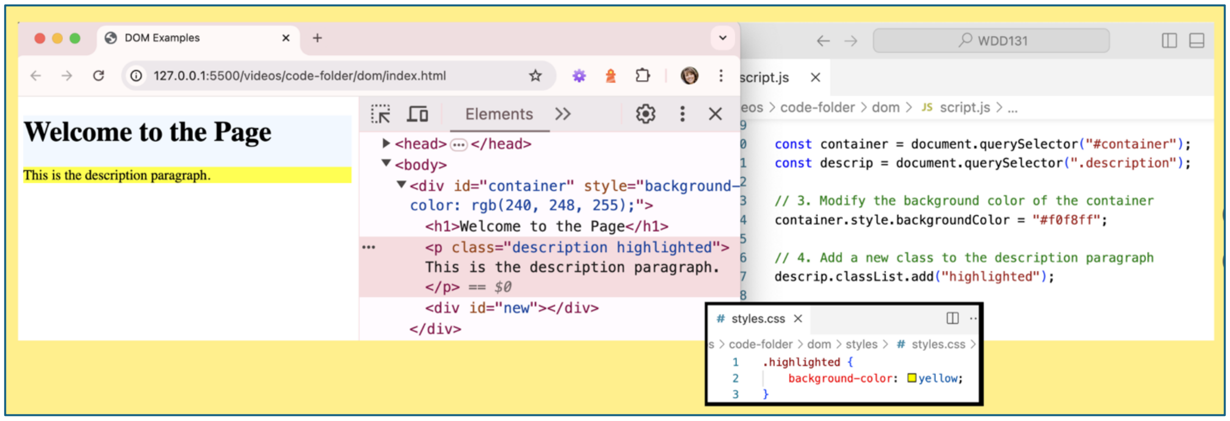
A more flexible way to work with classes is by using classList which allows adding, removing, toggling, and checking classes with the add, remove, or toggle methods.

- .classList()

```
element.classList.add("newClass");
element.classList.remove("oldClass");
element.classList.toggle("active");
```

Toggle will add the class if it’s not already added or will remove it if it’s already added.

You can use style.backgroundColor to assign a light blue to the container. You can also add a class to an element. If you already have CSS that styles any element with the class "highlighted" to have a yellow background color, you can add that class to the description paragraph using classList.add.


DOM Examples
127.0.0.1:5500/videos/code-folder/dom/index.html
Welcome to the Page
This is the description paragraph.
Elements
WDD131
script.js
eos > code-folder > dom > JS script.js > ...
styles.css
s > code-folder > dom > styles > # styles.css >

```
▶<head>…</head>
▼<body>
  ▼<div id="container" style="background-color: rgb(240, 248, 255);">
      <h1>Welcome to the Page</h1>
      <p class="description highlighted">
      This is the description paragraph.
      </p> == $0
      <div id="new"></div>
    </div>
```

```
const container = document.querySelector("#container");
const descrip = document.querySelector(".description");

// 3. Modify the background color of the container
container.style.backgroundColor = "#f0f8ff";

// 4. Add a new class to the description paragraph
descrip.classList.add("highlighted");
```

```
.highlighted {
    background-color: yellow;
}
```

You can even create elements that don't exist in the HTML using createElement( ).

```
const newParagraph = document.createElement("p");
```

And then give the new element content

```
newParagraph.textContent = "This is new paragraph content.";
```

- .appendChild( )

```
const container = document.getElementById("container");
```

And then add it to the HTML with appendChild()

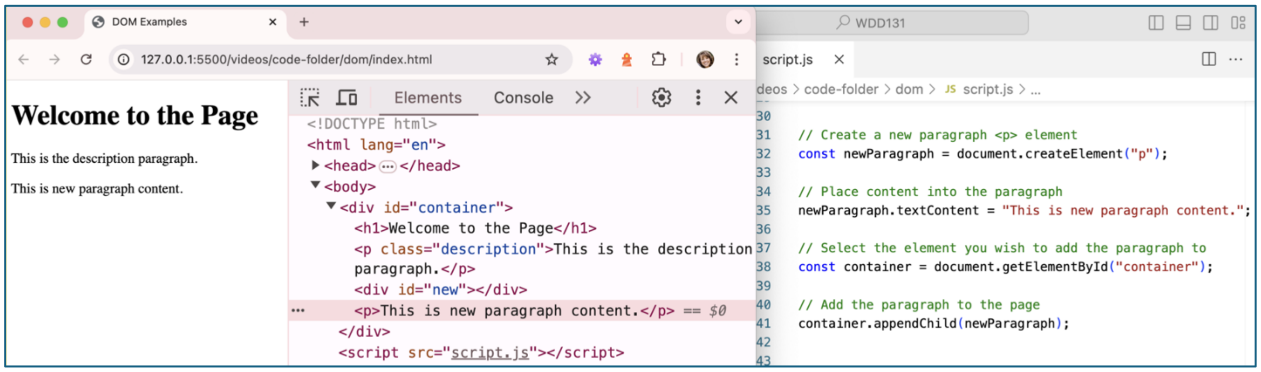
```
container.appendChild(newParagraph);
```

- .remove( )

```
newParagraph.remove();
```

You can even remove an element as well with remove().

Here a new paragraph (<p>) element has been created using createElement(). It then gets content with textContent and is added after the other elements in the container div to the page with appendChild().


DOM Examples
127.0.0.1:5500/videos/code-folder/dom/index.html
Elements
Console
WDD131
script.js
deos > code-folder > dom > script.js > ...

Welcome to the Page

This is the description paragraph.

This is new paragraph content.

```
<!DOCTYPE html>
<html lang="en">
  <head>…</head>
  <body>
    <div id="container">
      <h1>Welcome to the Page</h1>
      <p class="description">This is the description
      paragraph.</p>
      <div id="new"></div>
      <p>This is new paragraph content.</p> == $0
    </div>
    <script src="script.js"></script>
```

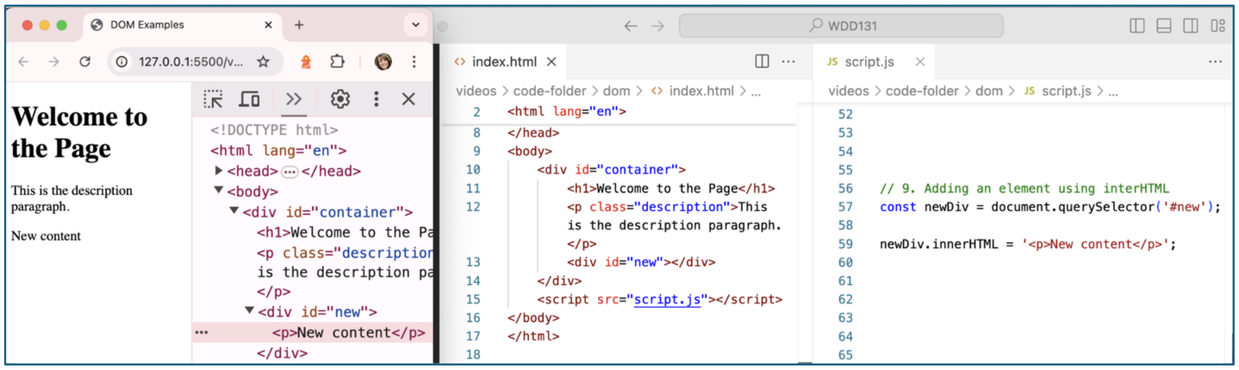
```
// Create a new paragraph <p> element
const newParagraph = document.createElement("p");

// Place content into the paragraph
newParagraph.textContent = "This is new paragraph content.";

// Select the element you wish to add the paragraph to
const container = document.getElementById("container");

// Add the paragraph to the page
container.appendChild(newParagraph);
```

Another way you will see a new element added along with the content is to use innerHTML. This creates a new HTML paragraph inside the container element and gives it content at the same time.



Welcome to the Page

This is the description paragraph.

New content

```
<!DOCTYPE html>
<html lang="en">
  <head>…</head>
  <body>
    <div id="container">
      <h1>Welcome to the Pa
      <p class="description
      is the description pa
      </p>
    <div id="new">
      <p>New content</p>
    </div>
```

```
<html lang="en">
</head>
<body>
    <div id="container">
        <h1>Welcome to the Page</h1>
        <p class="description">This
        is the description paragraph.
        </p>
        <div id="new"></div>
    </div>
    <script src="script.js"></script>
</body>
</html>
```

```
// 9. Adding an element using interHTML
const newDiv = document.querySelector('#new');

newDiv.innerHTML = '<p>New content</p>';
```

The first example using appendChild is generally safer and more efficient for adding elements, but it takes more code to implement. While innerHTML is quick and simple, it comes with trade-offs in terms of security and performance. innerHTML also replaces everything inside the container, whereas appendChild only adds the new content to the end of what is already there.

You can add an attribute to an element using the setAttribute() method. For example, if you wanted to add a new src (source) attribute to an image element, you can assign it this way in JavaScript.

- .setAttribute( )

```
element.setAttribute("src", "new-image.jpg");
```

You can even retrieve and remove attributes as well.

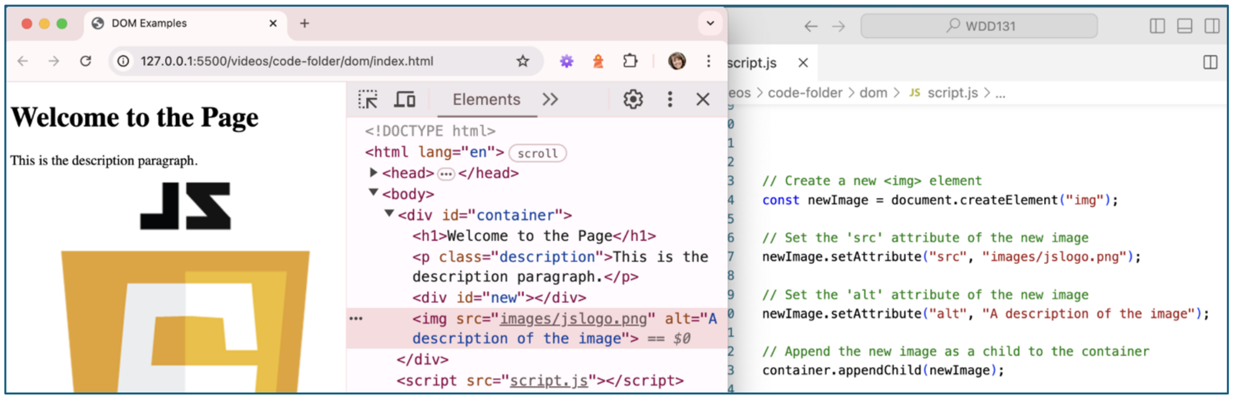
- .getAttribute( )

```
let src = element.getAttribute("src");
```

- .removeAttribute( )

```
element.removeAttribute("disabled");
```

In the figure below, a new image has been created with createElement and then we have added two attributes with setAttribute: first, the src (source) attribute of a JS logo we already have in our images folder, and then an alt (alternate text) description. Lastly, we need to add it to the page with appendChild.


DOM Examples
127.0.0.1:5500/videos/code-folder/dom/index.html
Welcome to the Page
This is the description paragraph.
Elements
WDD131
script.js
eos > code-folder > dom > script.js > ...

```
<!DOCTYPE html>
<html lang="en"> scroll
 ▶<head> ··· </head>
 ▼<body>
   ▼<div id="container">
      <h1>Welcome to the Page</h1>
      <p class="description">This is the
      description paragraph.</p>
      <div id="new"></div>
      <img src="images/jslogo.png" alt="A
      description of the image"> == $0
    </div>
    <script src="script.js"></script>
```

```
// Create a new <img> element
const newImage = document.createElement("img");

// Set the 'src' attribute of the new image
newImage.setAttribute("src", "images/jslogo.png");

// Set the 'alt' attribute of the new image
newImage.setAttribute("alt", "A description of the image");

// Append the new image as a child to the container
container.appendChild(newImage);
```

So, there are just a few of the DOM properties and methods demonstrated that you could use to retrieve or get the output you want to the user, using JavaScript to interact with the DOM of the HTML.